Penetration Testing Logbook

Student:
Alessandro Buonerba

Module Leader:
Dr Anatolij Bezemskij

Computer Science (Cybersecurity)
Penetration Testing and Ethical Vulnerability Scanning
COMP-1671

Department of Computing & Mathematical Sciences
Liberal Arts & Sciences

University of Greenwich
London, United Kingdom

December 2021

CONTENTS

LIST OF FIGURES

1

LAB 1: PASSIVE ENUMERATION

Contrarily from active enumeration, passive enumeration is a technique that does not rely on explicit communication with a target system (Cooper, 2020). To perform a passive enumeration, a network monitor tool such as Wireshark is often used.

1.1 CONNECT TO FTP

The first part of the task is to connect to the FTP server and download the .pcap file with all the captured network traffic.

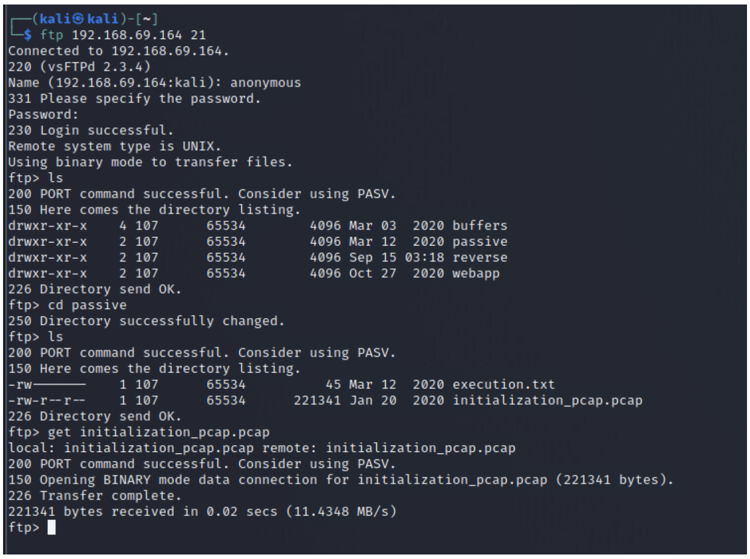


Figure 1.1: Connect to the FTP and get the .pcap file

Now that the file has been downloaded, it can be found in the home directory and we can start the analysis of the network traffic through Wireshark following the tasks assigned to this lab.

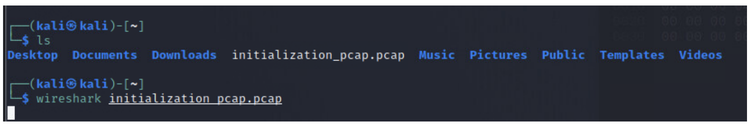
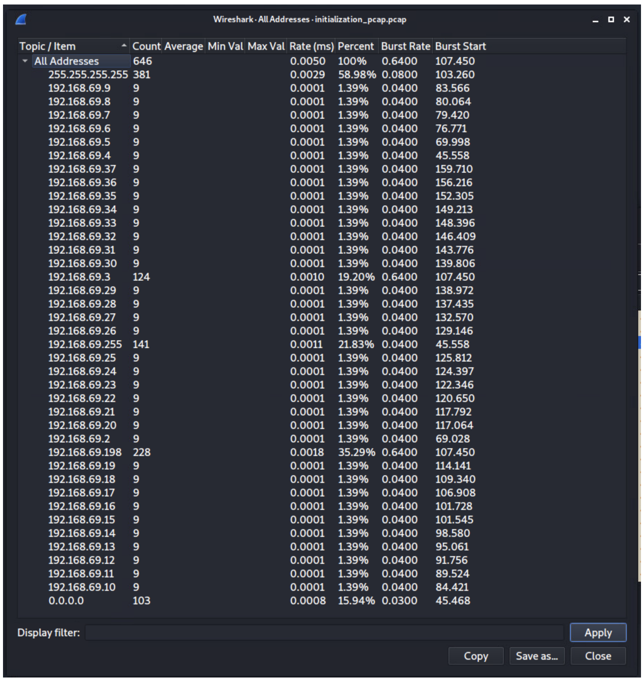


Figure 1.2: Open .pcap with Wireshark

1.2 find unique ipv4 addresses

The first tasks asks to find the unique IPs that are stored and captured. We can achieve that through the top menu, selecting statistics and IPv4 addresses. The result is shown in the figure below.



Wireshark · All Addresses · initialization_pcap.pcap

Topic / Item	Count	Average	Min Val	Max Val	Rate (ms)	Percent	Burst Rate	Burst Start
All Addresses	646				0.0050	100%	0.6400	107.450
255.255.255.255	381				0.0029	58.98%	0.0800	103.260
192.168.69.9	9				0.0001	1.39%	0.0400	83.566
192.168.69.8	9				0.0001	1.39%	0.0400	80.064
192.168.69.7	9				0.0001	1.39%	0.0400	79.420
192.168.69.6	9				0.0001	1.39%	0.0400	76.771
192.168.69.5	9				0.0001	1.39%	0.0400	69.998
192.168.69.4	9				0.0001	1.39%	0.0400	45.558
192.168.69.37	9				0.0001	1.39%	0.0400	159.710
192.168.69.36	9				0.0001	1.39%	0.0400	156.216
192.168.69.35	9				0.0001	1.39%	0.0400	152.305
192.168.69.34	9				0.0001	1.39%	0.0400	149.213
192.168.69.33	9				0.0001	1.39%	0.0400	148.396
192.168.69.32	9				0.0001	1.39%	0.0400	146.409
192.168.69.31	9				0.0001	1.39%	0.0400	143.776
192.168.69.30	9				0.0001	1.39%	0.0400	139.806
192.168.69.3	124				0.0010	19.20%	0.6400	107.450
192.168.69.29	9				0.0001	1.39%	0.0400	138.972
192.168.69.28	9				0.0001	1.39%	0.0400	137.435
192.168.69.27	9				0.0001	1.39%	0.0400	132.570
192.168.69.26	9				0.0001	1.39%	0.0400	129.146
192.168.69.255	141				0.0011	21.83%	0.0400	45.558
192.168.69.25	9				0.0001	1.39%	0.0400	125.812
192.168.69.24	9				0.0001	1.39%	0.0400	124.397
192.168.69.23	9				0.0001	1.39%	0.0400	122.346
192.168.69.22	9				0.0001	1.39%	0.0400	120.650
192.168.69.21	9				0.0001	1.39%	0.0400	117.792
192.168.69.20	9				0.0001	1.39%	0.0400	117.064
192.168.69.2	9				0.0001	1.39%	0.0400	69.028
192.168.69.198	228				0.0018	35.29%	0.6400	107.450
192.168.69.19	9				0.0001	1.39%	0.0400	114.141
192.168.69.18	9				0.0001	1.39%	0.0400	109.340
192.168.69.17	9				0.0001	1.39%	0.0400	106.908
192.168.69.16	9				0.0001	1.39%	0.0400	101.728
192.168.69.15	9				0.0001	1.39%	0.0400	101.545
192.168.69.14	9				0.0001	1.39%	0.0400	98.580
192.168.69.13	9				0.0001	1.39%	0.0400	95.061
192.168.69.12	9				0.0001	1.39%	0.0400	91.756
192.168.69.11	9				0.0001	1.39%	0.0400	89.524
192.168.69.10	9				0.0001	1.39%	0.0400	84.421
0.0.0.0	103				0.0008	15.94%	0.0300	45.468

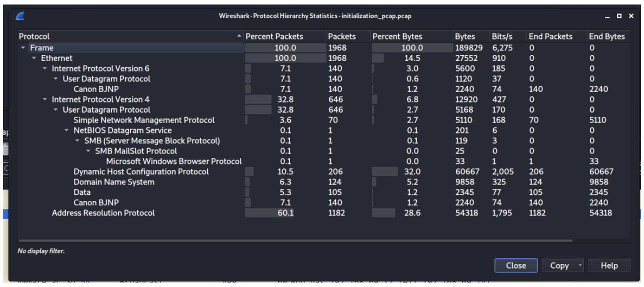
Display filter: Apply

Copy Save as… Close

Figure 1.3: Unique IPv4 addresses

1.3 application-layer protocols

The second task asks to find the application-layer protocols that are used in the captured network traffic. This can be displayed using the Protocol Hierarchy command. The result is shown in the figure below.



Wireshark · Protocol Hierarchy Statistics · initialization_pcap.pcap

Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes
Frame	100.0	1968	100.0	189829	6,275	0	0
Ethernet	100.0	1968	14.5	27552	910	0	0
Internet Protocol Version 6	7.1	140	3.0	5600	185	0	0
User Datagram Protocol	7.1	140	0.6	1120	37	0	0
Canon BJNP	7.1	140	1.2	2240	74	140	2240
Internet Protocol Version 4	32.8	646	6.8	12920	427	0	0
User Datagram Protocol	32.8	646	2.7	5168	170	0	0
Simple Network Management Protocol	3.6	70	2.7	5110	168	70	5110
NetBIOS Datagram Service	0.1	1	0.1	201	6	0	0
SMB (Server Message Block Protocol)	0.1	1	0.1	119	3	0	0
SMB MailSlot Protocol	0.1	1	0.0	25	0	0	0
Microsoft Windows Browser Protocol	0.1	1	0.0	33	1	1	33
Dynamic Host Configuration Protocol	10.5	206	32.0	60667	2,005	206	60667
Domain Name System	6.3	124	5.2	9858	325	124	9858
Data	5.3	105	1.2	2345	77	105	2345
Canon BJNP	7.1	140	1.2	2240	74	140	2240
Address Resolution Protocol	60.1	1182	28.6	54318	1,795	1182	54318

No display filter.

Close Copy Help

Figure 1.4: Protocol Hierarchy

1.4 NAME OF THE PROTOCOLS

The application-layer protocol are the following.

- SNMP (Single Network Management Protocol): responsible for the management of network devices, allows the communication between them independently of their spec (Scarpati, 2020).
- DNS (Domain Name System): responsible for the resolution of domain names to IP addresses (Insam, 2020).
- DHCP (Dynamic Host Configuration Protocol): responsible for the dynamic configuration of network devices. This protocol is used to automatically assign IPs to network devices (IBM, 2021).
- SMB (Server Message Block): responsible for the communication between shared devices such as printers on a network (Sheldon and Scarpati, 2020).

1.5 NETWORK DIAGRAM

This task will allow us to have a visual representation of the analysis of the network. Below the diagram with the active protocols and devices.

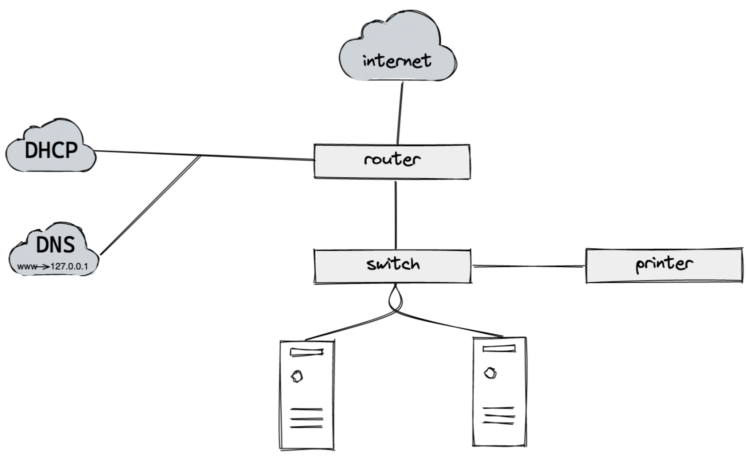


Figure 1.5: Network Diagram

Following the explanation of the protocols, further analysis portrays the use of internet protocol. The protocols in use are UDP, SNMP, DHCP and DNS, meaning computers and shared devices on the network. We can also certify using a BJNP protocol, meaning that the shared device on the network is a Canon printer.

1.6 DISCUSSION

The network traffic analysis suggests that a user uses the shared device since there is a BJNP protocol. There are also ACKs and NAKs portraying active communication between the devices of the network. Some of the UDP packets were broadcasting an std discovery all to find all the services on the network.

1.7 TCP DUMP

Following the instructions and the man page for the tcpdump command, I have been able to reproduce a one liner to output a number of unique MAC addresses in the provided and previously used .pcap file. Below a picture with the result.

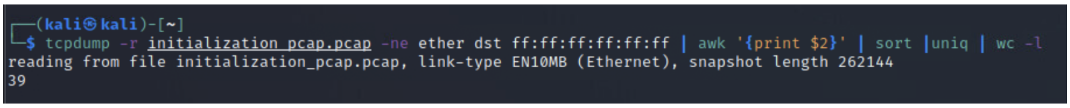


Figure 1.6: TCP Dump

The flag `-r` is used to read the file and the flag `-ne` before `ether dst` looks for ethernet destinations with the MAC address format specified right after it. The command `awk` is used to separate them while printing the second argument to get the second column. It will then sort and check for unique entries for then count everything with the last `wc -l` command

1.8 REFLECTION

This has been a very fun lab. I have learned a lot more about Wireshark and how to analyse a .pcap file. Even though I have never used `tcpdump`, there were manyexamples and exhaustive official documentation.

2 LAB 2: ACTIVE ENUMERATION

2.1 INTRODUCTION

Active enumeration is when a user programmatically gather informations on a system through the use of a set of predefined commands (Cooper, 2020). The most common set of informations that is usually gathered through enumeration are DNS, IPs, ports, and services.

2.2 FINAL OUTPUT

The software that has been written for this lab is a simple but effective nmap clone. Once executed it asks for a range of ip addresses and then starts to ping them incrementally starting from the first input till the end. While pinging, it first of all recognise if the machine is responding, and it then looks for open ports, mac addresses, dns and ttl. At the end of the scan, when the objects has been populated, it will then produce a final report with all the information gathered during the process. All the tools used to write the code is primitive and already included in python. The final output of the python program can be executed with `python action.py`.

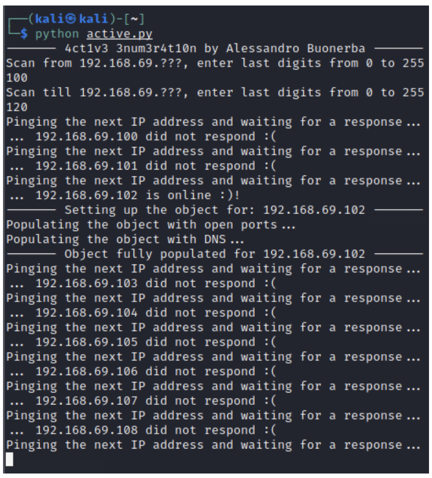


Figure 2.1: Executing the Custom Nmap Clone

The figure above, shows how the program asks for an input by the user. In order to know which IP to ping, it will ask for a range to scan. Once the user gives in the requested information, it will then start to scan and print an updated of what it is doing step by step. Once it finishes to scan the range, it will print the final report with a list of IPs and all the informations gathered during the previous step. At the end of the list it will also print a nice general summary report as shown below.

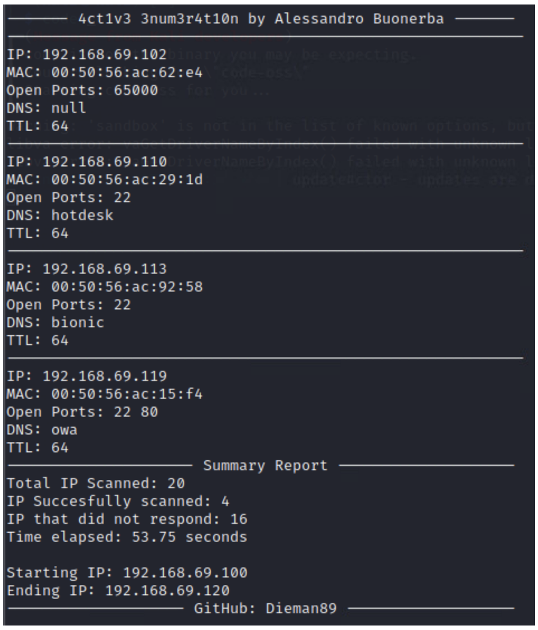


Figure 2.2: Results and Summary Report

2.3 PYTHON CODE

The code has been written with primitive tooling, meaning that all the packages imported were already installed in the VM and part of the python language. The figure below shows the imports of the packages used to accomplish the tasks. Here is a list of the modules used:

- `socket` provides access to the socket interface and is available on almost all modern platforms.
- `os` provides acccess to the miscellaneous operating system interfaces.
- `time` provides access to time-related functions.
- `platform` provides access to underlying platform-s identifying data.
- `subprocess` provides access to spawn processes and their input/output.
- `re` provides access to regular expression matching operations.

```python
import socket
import os
import time
import platform
import subprocess
import re

operative_system = platform.system()
ping_flag = 'n' if operative_system == 'Window' else 'c'
ports = [20, 22, 25, 53, 80, 587, 631, 3306, 10000, 65000]
ttl_grep = 'grep -o ttl=[0-9][0-9]*'
mac_grep = 'grep -o ..:..:..:..:..:..'
array = []
```

Figure 2.3: imports-declarations

There are also some variables that have been set globally in order to be used anywhere in the code. The operative system variable has been used to check system where the code is running as the ping command would have a different flag depending on this factor. An array of the most important ports is also declared, where initially the first iteration of the software would scan a large set of ports. The grep for TTL and MAC format are respectively used to extract them from other commands. The `ping_flag` and `ttl_grep` are used in the main code shown in Figure 2.13, `ports` is used in the code in Figure 2.6 while `mac_grep` is used in the code shown in figure 2.7.

```python
def regex_chars(str):
    return re.sub('\W+', '', str)

def arp(ip, grep):
    return os.popen('sudo arping -c 1 %s | %s' %(ip, grep)).read()

def ping(ip, flag, grep):
    return os.popen('ping -%s 1 %s | %s' %(flag, ip, grep)).read()
```

Figure 2.4: Regex, arping and ping

The function `regex_chars` takes a string as a parameter and returns a string that gets cleaned from all the extra characters that are not letters through regular expression. The regex function is called in the code shown in picture 2.5. The function `arp` takes two strings as parameters, one being `ip` and the other being `grep`. This function will basically run the arping command and will be called later in the Figure 2.7. The last function `ping` takes three strings as arguments same as the previous one, but with the exception of the additional flag that will injected in the command depending on which operative system the machine is running on. Also, this function is called in the main method shown in the Figure 2.13 Since the VM has only the old Python 2.7, the old `%` has been used to format with a specifier to say how the value should be go in. With Python >= 3.6, it is usually replaced with the more flexible `f-strings`.

```
def format_dns(obj):
    dns = os.popen('host -l %s' %(obj['ip'])).read()
    if 'not found' in str(dns):
        obj['dns'] = 'null'
    else:
        obj['dns'] = regex_chars(str(dns).split(' ')[4].rstrip())
```

Figure 2.5: Format DNS

The function `format_dns` takes a dictionary as a parameter, where has been created and partially populated in Figure 2.7. The IP will be the referenced key and its value used to perform the host command to find the dns name. Since the messages are printed in terminal in a very predefined format, the string will get splitted and transformed in an array where name of the dns gets picked and removed from any special characters, in this case a dot and the results gets populated in the dns key of the dictionary as a value. If the host is not found, it will push a null value instead.

```
def format_ports(obj):
    for port in ports:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ip = (obj['ip'], port)
        open_port = s.connect_ex(ip)
        if open_port == 0:
            obj['ports'].append(str(port) + ' ')
        s.close()
```

Figure 2.6: Format Ports

The function `format_ports` takes again a dictionary as parameter, similarly to the previous one, and loops through the array ports that are globally declared in figure 2.3. It creates a socket object that specify address family and socket type, respectively `IPv4` and `TCP`. Moving down, an ip variable will be created to reference the ip address and a port. This variable will then be used as a parameter to the `connect_ex` method from the socket object previously created and assigned again to another variable. If the last variable is `0`, it means that the operation has been successfull and the port is open. The last step is to append the open port to the dictionary after converting it to a string.

```
def format_arp(ip, array, ttl):
    print '------- Setting up the object for: %s -------' %ip
    arped = arp(ip, mac_grep)
    ttl_num = ttl.split('=')
    obj = {
        'ip': ip,
        'ttl': ttl_num[1].rstrip(),
        'arp': str(arped).rstrip(),
        'ports': [],
        'dns': '' }
    array.append(obj)
    print 'Populating the object with open ports...'
    format_ports(obj)
    print 'Populating the object with DNS...'
    format_dns(obj)
    print '------- Object for: %s fully populated -------' %ip
```

Figure 2.7: Format ARP

The function `format_arp` takes three parameters, the first one being an ip as a string, the second one an array and a TTL format string that will be previously grepped from the main before calling this function. This function will initialise the object structure of each IP that will then be pushed into the global array. It will also partially populate it with IP and TTL that will get passed from main and ARP that will contain the MAC address generated from the `arp` function. The `rstrip()` method is used to remove the extra characters that are not needed, such as whitespaces. The object will then be appended to the array, that will then be referenced in `format_ports` and `format_dns` functions. From now on the mutations will be done on the array level. Lastly, some print to the terminal will be done here to keep the user updated on the progress.

```
def printer(arr):
    for item in arr:
        open_ports = ''
        for port in item['ports']:
            open_ports += str(port)
        print '-----------------------------------------------------------'
        print 'IP: %s' %item['ip']
        print 'MAC: %s' %item['arp']
        print 'Open Ports: %s' %open_ports
        print 'DNS: %s' %item['dns']
        print 'TTL: %s' %item['ttl']
```

Figure 2.8: Printer

The `printer` function takes an array as a parameter. It is used almost at the end of the main method in Figure 2.13 and gives the human-readable representation of the infromations gathered throughout the enumeration.

```python
def ip_builder(end):
    return '192.168.69.%s' %str(end)
```

Figure 2.9: IP Builder

This simple function is used in main before calling the function that will then ping it. It returns the IP address that must be pinged and starts the whole process.

```python
def txt(pos):
    if pos == 'start':
        print '------- 4ct1v3 3num3r4t10n by Alessandro Buonerba -------'
    if pos == 'end':
        print '-------------------- GitHub: Dieman89 --------------------'
    if pos == 'summary':
        print '--------------------- Summary Report ---------------------'
```

Figure 2.10: Text Function

This function is used as a reference to some of the strings printed within the code.

```python
def summary(array, time_start):
    total_ips = ip_to - ip_from
    ok_counter = len(array)
    failed_counter = total_ips - len(array)
    time_elapsed = time.time() - time_start

    print 'Total IP Scanned: %s' %total_ips
    print 'IP Successfully scanned: %s' %ok_counter
    print 'IP that did not respond: %s' %failed_counter
    print 'Time elapsed: %.2f seconds' %time_elapsed
    print ''
    print 'Starting IP: 192.168.69.%s' %ip_from
    print 'Ending IP: 192.168.69.%s' %ip_to
```

Figure 2.11: Summary Function

The `summary` function takes and array a string that stores the start time of the program. The array is passed when the objects within it are fully populated and is used as a reference to calculate some of the metrics such as ip successfully and failed scanned ips. The start time, instead is passed from the main and re-used in the function where a new time method is called to calculate the time elapsed. At the end, all the data is printed to the user in a human-readable way.

```
def input_range():
    global ip_from
    global ip_to
    txt('start')
    print 'Scan from 192.168.69.???, enter last digits from 0 to 255'
    ip_from = int(input())
    print 'Scan till 192.168.69.???, enter last digits from 0 to 255'
    ip_to = int(input())
```

Figure 2.12: Input Range

The function above will be called as first thing at the start of the program in order to ask the user for the range of IPs to be scanned. It sets the global within the methods for readability and better understanding, as they will then be used throughout the code.

```
def main():
    input_range()
    time_start = time.time()
    for end in range(ip_from, ip_to):
        print 'Pinging the next IP address and waiting for a response...'
        ip = ip_builder(end)
        ttl = ping(ip, ping_flag, ttl_grep)
        if (str(ttl)):
            print '... %s is online :)!' %ip
            format_arp(ip, array, str(ttl))
        else:
            print '... %s did not respond :(!' %ip
        if operative_system == 'Windows':
            subprocess.Popen('cls', shell=True).communicate()
        else:
            print('\033c')
        txt('start')
        printer(array)
        txt('summary')
        summary(array, time_start)
        txt('end')
```

Figure 2.13: The Main

Finally the `main`. This has been referenced multiple times throughout the report of this lab and probably does not need to be explained further. Few things that are still not explained are the subprocess object with the Popen method used to clear the terminal on Windows, and the `print('\033c')` used to clear the terminal on Unix. On a note, this is where a time sleep would be implemented in order to bypass network bandwith overload as specified and asked in one of the tasks.

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

Figure 2.14: Name variable as main

As in almost any Python code, this sets the name variable as main and then call the main method.

```python
import socket
import os
import time
import platform
import subprocess
import re

operative_system = platform.system()
ping_flag = 'n' if operative_system == 'Window' else 'c'
ports = [20, 22, 25, 53, 80, 587, 631, 3306, 10000, 65000]
ttl_grep = 'grep -o ttl=[0-9][0-9]*'
mac_grep = 'grep -o ..:..:..:..:..:..'
array = []

def regex_chars(str):
    return re.sub('\W+', '', str)

def arp(ip, grep):
    return os.popen('sudo arping -c 1 %s | %s' %(ip, grep)).read()

def ping(ip, flag, grep):
    return os.popen('ping -%s 1 %s | %s' %(flag, ip, grep)).read()

def format_dns(obj):
    dns = os.popen('host -l %s' %(obj['ip'])).read()
    if 'not found' in str(dns):
        obj['dns'] = 'null'
    else:
        obj['dns'] = regex_chars(str(dns).split(' ')[4].rstrip())

def format_ports(obj):
    for port in ports:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ip = (obj['ip'], port)
        open_port = s.connect_ex(ip)
        if open_port == 0:
            obj['ports'].append(str(port) + ' ')
        s.close()

def format_arp(ip, array, ttl):
    print('------- Setting up the object for: %s -------' %ip)
    arped = arp(ip, mac_grep)
    ttl_num = ttl.split('=')
    obj = {
        'ip': ip,
        'ttl': ttl_num[1].rstrip(),
        'arp': str(arped).rstrip(),
        'ports': [],
        'dns': '' }
    array.append(obj)
    print('Populating the object with open ports...')
    format_ports(obj)
    print('Populating the object with DNS...')
    format_dns(obj)
    print('------- Object for: %s fully populated -------' %ip)

def printer(arr):
    for item in arr:
        open_ports = ''
        for port in item['ports']:
            open_ports += str(port)
        print('------------------------------------------------------------')
        print('IP: %s' %item['ip'])
        print('MAC: %s' %item['arp'])
        print('Open Ports: %s' %open_ports)
        print('DNS: %s' %item['dns'])
        print('TTL: %s' %item['ttl'])

def ip_builder(end):
    return '192.168.69.%s' %str(end)

def txt(pos):
    if pos == 'start':
        print('------- 4ct1v3 3num3r4t10n by Alessandro Buonerba -------')
    if pos == 'end':
        print('-------------------- GitHub: Dieman89 --------------------')
    if pos == 'summary':
        print('--------------------- Summary Report --------------------')

def summary(array, time_start):
    total_ips = ip_to - ip_from
    ok_counter = len(array)
    failed_counter = total_ips - len(array)
    time_elapsed = time.time() - time_start

    print('Total IP Scanned: %s' %total_ips)
    print('IP Successfully scanned: %s' %ok_counter)
    print('IP that did not respond: %s' %failed_counter)
    print('Time elapsed: %.2f seconds' %time_elapsed)
    print('')
    print('Starting IP: 192.168.69.%s' %ip_from)
    print('Ending IP: 192.168.69.%s' %ip_to)

def input_range():
    global ip_from
    global ip_to
    txt('start')
    print('Scan from 192.168.69.???, enter last digits from 0 to 255')
    ip_from = int(input())
    print('Scan till 192.168.69.???, enter last digits from 0 to 255')
    ip_to = int(input())

def main():
    input_range()
    time_start = time.time()
    for end in range(ip_from, ip_to):
        print('Pinging the next IP address and waiting for a response...')
        ip = ip_builder(end)
        ttl = ping(ip, ping_flag, ttl_grep)
        if (str(ttl)):
            print('... %s is online :)!' %ip)
            format_arp(ip, array, str(ttl))
        else:
            print('... %s did not respond :(!' %ip)
        if operative_system == 'Windows':
            subprocess.Popen('cls', shell=True).communicate()
        else:
            print('\033c')
        txt('start')
        printer(array)
        txt('summary')
        summary(array, time_start)
        txt('end')

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

Figure 2.15: Full Code

Above the full code for better readability.

2.4 CONCLUSION

This has been one of the most fun lab I have ever done at the University. I have learned more about active enumeration and basically created a nmap clone with very primitive tooling. The only downside is the isolation of the machine from internet, and the fact that is very very slow. It created a very slow and far from good developer experience but I understand how not much can be done to fix it. Research has been done on Python syntax as it is not my main language. Overall a very positive experience, and I am very happy with the final product.

3

LAB 3:

3.1 HELLO WORLD

This guy said that I'm awesome (Fu, n.d.)

BIBLIOGRAPHY

Cooper, Zach (2020). *What's the Difference between Active and Passive Reconnaissance?* URL: https://www.itpro.co.uk/penetration-testing/34465/whats-the-difference-between-active-and-passive-reconnaissance (visited on 10/07/2021).

Fu, Anthony (n.d.). *Slidev*. URL: https://sli.dev/ (visited on 10/22/2021).

IBM (2021). *IBM Docs*. URL: https://prod.ibmdocs-production-dal-6099123ce774e592a519d7c33db8265e-0000.us-south.containers.appdomain.cloud/docs/en/aix/7.1?topic=tcpp-tcpip-address-parameter-assignment-dynamic-host-configuration-protocol (visited on 10/07/2021).

Insam, Edward (2020). *Application Layer Protocol - an Overview*. URL: https://www.sciencedirect.com/topics/computer-science/application-layer-protocol (visited on 10/07/2021).

Scarpati, Jessica (2020). *What Is Simple Network Management Protocol (SNMP)? Definition from SearchNetworking*. URL: https://www.techtarget.com/searchnetworking/definition/SNMP (visited on 10/07/2021).

Sheldon, Robert and Jessica Scarpati (2020). *What Is the Server Message Block (SMB) Protocol? How Does It Work?* URL: https://www.techtarget.com/searchnetworking/definition/Server-Message-Block-Protocol (visited on 10/07/2021).