UNIVERSITÀ
DI TRENTO

Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Automated framework for IoT security testing

Supervisor

Bruno Crispo

Student

Matteo Mariotti

Academic year 2022/2023

Acknowledgements

I would like to thank my parents and my sister for supporting me during these years, and all my family for always being there for me.

I would also like to thank my friends Michele, Pietro and Elisabetta for sharing with me this experience and for all the time we spent together. I hope we will continue to support each other in the future, particularly for the rest of our years here at the university.

Last but not least, I would like to thank my supervisor, Prof. Bruno Crispo, for giving me the opportunity to work on this project and for his precious advice and support.

Contents

Abstract

In the last few years, the security of IoT devices has become a major concern in the field of Computer Science. In fact we are surrounded by always new devices, such as cameras, smart speakers, and domotic devices in general, that have become an integral part of our lives.
This means we are more and more exposed to attacks that can compromise our privacy and security.
The focus of this thesis is to automate the process of finding vulnerabilities in such devices, by the means of finding the right tools and scripting the various phases of the process.

This work is divided into two parts: a first more theoretical part, where we will analyze the proposed tools and compare them to the ones used in previous works. We will also analyze the scripts created to help the automation and explain their capabilities.
In a second, more practical part, we will test these tools on a real device, focusing on the actions that have been successfully automated, and hence require little to no user interaction.

Part I

Tools

1 Introduction

In this part of the dissertation we will discuss the improvements over previous work [24], both in terms of alternative tools with respect to the ones presented in that thesis, and in terms of additional areas where we can search for vulnerabilities.

In particular we will discuss how to gain access to the target network and list the various programs that already exist for this purpose. We will also discuss an extension to the previous work that allows to retrieve the memory image of the target device and analyze it offline, aiming to find useful information about what processes were running on said device and inspecting their memory.

Finally we will discuss what scripts were developed to automate the process and how to use them in practice, but leaving the actual testing to the second part of the thesis, and focusing only on their options and requirements.

2 Wi-Fi access

In the thesis on which[24] this work is based it was suggested to use the Fern wifi cracker[6] to gain access to the target newtwork. This tool is basically a GUI for the aircrack-ng suite[1] and exposes all the features of the command line tool. It can in fact perform a lot of other attacks, aside from the WPA/WPA2 cracking, such as:

- Automatic attacks on the access point
- MITM attacks
- Bruteforce attacks using protocols such as FTP, HTTP/HTTPS and TELNET
- Deauthentication, access point spoofing and replay attacks

Nevertheless, as the goal of this work is to automate the process, the command line tool may prove to be more useful, as it can be easily integrated in a script.
The Aircrack-ng suite is the most portable of the tools presented here, as it can be used from virtually every operating system broadly used today, such as Windows, MacOS, a lot of Linux distributions and even from BSD operating systems.
There is also a Docker container for running it without installing it on the main system. It can be installed using the right packaged version for the host OS or by compiling it from source[2].

Other tools that can be employed instead of aircrack-ng are:

- Reaver[14]: a tool that exploits a flaw in the WPS implementation of some routers using the WPS Pixie-Dust and other brute-force attacks
- Wifite[21]: a python script that is a frontend for a lot of tools, including aircrack-ng, reaver, cowpatty, pyrit, tshark, etc. It is therefore perfect to automate the process of vulnerability assessment of a wifi network
- Bully[3]: similar to Reaver

The suggested tool is Wifite, as it is the most complete and it is a frontend to all other tools presented here. It is also a command line program, so it can be easily used in a script and it's also very easy to use because of the almost complete automation of the process. Unfortunately it needs many dependencies to reach its full potential, but many of them are optional. It can be downloaded using git in the following way:

```
git clone https://github.com/derv82/wifite2.git
```

It can then be run opening the terminal in the cloned directory and running:

```
sudo ./Wifite.py
```

3 Memory analysis

3.1 Dump creation

In this chapter we will focus on memory analysis: a type of vulnerability assessment not part of the framework on which this thesis is based. The goal will be to retrieve the memory dump of a running device and analyze it to find out if it's vulnerable.

We will assume the target device runs a Linux based operating system, because in case of IoT devices this is the most common scenario. If the device has a proprietary OS the analysis will be more difficult, but the same steps will apply (dump creation, retrieval and analysis).
We also assume the attacker already has a working root shell on the device, as all operations of memory read and kernel module loading are only allowed in case of privileged access.

In order to do this we first have to extract the memory dump from the device and put it in a file. This can be done using the dd command, present on all Linux installations.
It can be used for example to copy the content of **/dev/mem** calling it with the following syntax:

```
    dd if=/dev/mem of=memdump bs=1M count=1
```

This command will copy the specified number of blocks (count) of the specified size (bs) to the specified output file.

Unfortunately we need a suitable device from which to extract the memory and **/dev/mem** may have been configured to allow only a small amount of memory to be read: this depends on the OS configuration so it's impossible to know in advance.
This limit may be of 1Mb or 1Gb, depending on the architecture and configuration, but it's only available since version 2.6.26 of the Linux kernel[5] and only if the kernel has been compiled with the **CONFIG_STRICT_DEVMEM** option.
On the x86 architecture it almost completely disallows memory access, allowing only some memory addresses, such as memory mapped PCI devices, to be read. The fact that the restriction is only available since a specific version may be an advantage for an attacker because it means that older versions of the kernel (usually used on IoT devices) may not have it.

A similar device to **/dev/mem** is **/dev/kmem**, which works the same way, but uses kernel virtual memory addresses instead of physical memory addresses.
Moreover it's only available if the kernel has been compiled with the **CONFIG_DEVKMEM** option, so it's not always available and not a good choice for our purposes.

Another device from which we can obtain a memory dump is **/dev/fmem**[4], which can be created with the use of a kernel module. It allows to read the whole physical memory without restrictions, but needs the installation of the kernel module and is not guaranteed to work with the specific kernel version or architecture. In fact it's not been updated for a while, but this may not be a problem, because IoT devices rarely ship with the latest kernel versions.
The last option that we'll analyze is the use of the **LiME** (Linux Memory Extractor)[9]. It's the most advanced option, because it offers a lot of convenient features that can help to retrieve the memory dump, with the downside (as before) of needing the installation of a kernel module.
It can be used with the following syntax:

```
    insmod ./lime.ko "path=<outfile | tcp:<port>> format=<raw|padded|lime>
        [digest=<digest>] [dio=<0|1>]"
```

The module is able to:

- Dump memory to a file or listen on a tcp port and send the memory dump to a remote host
- Use different formats for the memory dump
- Use different digest algorithms to verify the integrity of the memory dump

QEMU memory dump

In case the attacker succeeded in emulating the device image with QUEMU (for example using **FYRMADYNE**[7], as suggested in [24]), then it is possible to obtain a memory dump of the emulated device.
This can be done with the **pmemsave** or the **dump-guest-memory** quemu commands.

File retrieve

Once we have created the memory dump we need to retrieve it from the device.
If LiMe was not used than it's possible to use the commands **netcat**[13] or **scp**[15] to upload the file to the attacker's machine.
As we assume the target device to be a Linux machine, scp will probably be installed by default, while netcat will need to be installed.

3.2 Analysis

To analyze the dump file it's possible to use the **Volatility**[18] framework, which is a collection of python tools to analyze memory images and extract information from them. We suggest using the original version, but there is also another one, more up to date and written in Python 3 instead of Python 2, called **Volatility3**[19], still in development.

It can be used for example to see open files and running processes at the time of the dump, but has many capabilities, listed on the official documentation. Volatility supports different file formats (raw, LiME, etc.) and different operating systems (Windows, Linux, etc.), but for systems other than Windows you need to configure the right profile, or even to write your own (in case there is no suitable one).

For Linux there are already some profiles available, but it's very likely that an attacker will need to create his own ones, because of the huge variety of IoT devices. However the instructions can be found on the official documentation and on other good tutorials[10][20].

Once the profile is configured it's possible to use the framework to analyze the dump file, to retrieve all important information about the system. It's also possible to automate the process using the tool inside scripts, because it's a fully automated framework.

To analyze the file we can use the following command:

```
python2 vol.py -f <dump_file> <plugin_name> --profile=<profile_name>
```

The plugin name specifies which action to perform on the file, so we can search for different information changing this parameter. A list of all Linux plugins can be obtained with this command:

```
python2 vol.py --info | grep linux_
```

4 Scripts

Here we will analyze the capabilities of the scripts developed for the project.

4.1 Scan

First of all we will analyze the script for the scanning and testing of the hosts on the network. It's a Bash script that uses **nmap** to discover hosts on the network, search for open ports and run test script on those that are found.
If we run the script with the following option:

```
scan -h
```

we will get all the options it can accept and the output will be the following:

```
Usage: scan [OPTIONS]
    -u            perform udp scan
    -a            perform attack with default scripts
    -t            perform tcp scan
    -f            full scans
    -d            discovery mode (only -i and -o relevant)
    -v            scan for OS version
    -o FILENAME   redirect output to FILENAME
    -i IP         scan specified IP
    -m MODE       tcp scan mode
```

The **-d** option can be used to search for hosts on the network, and only the **-i** and **-o** options will be relevant, all the others will be discarded.
The **-i** option can be used to specify the IP address of the host to scan, or of the network in case we are using the discovery mode (in which case we have to specify the address using the CIDR notation).
The **-o** option will cause all the outputs of the commands to be written also to that file, and not only to standard output. In case we don't want to retain the output we can specify the **/dev/null** file. If no file is specified it will default to "**output**".

We can pass the **-v** option to perform an OS scan on the specified host, retrieving OS type and version.
The options **-t** or **-u**, which can be combined, will perform a TCP or UDP scan respectively. The **-f** options will cause a full scan to be performed, instead of only searching for the 1000 most common ports, and it will affect bot UDP and TCP scans. We can also specify the TCP scan mode to use with the **-m** option, in which case it's not necessary to pass **-t**, because it is subsumed.
If the **-a** option is specified, the script will run the default nmap scripts on the ports that are found open.

The script will also print every command executed and it's output, and it will print what ports are found open before starting to, eventually, test them.

4.2 New hosts discovery

This script uses the **nmap** tool as before, to discover the newly connected hosts on the network.
To obtain a help message from the script we can use the same syntax as before:

```
find_hosts -h
```

and we will get the following output:

```
Usage: find_hosts [OPTIONS]
-h              print this help message
-f FILE         get old IP addresses from FILE
-i IP           scan specified IP
```

We see that we can use the **-i** option (required) to specify which network to scan, passing an IP address in CIDR notation. The script will perform a first scan and print the hosts that are found. It will then wait for the user to press enter, after the new devices have been connected to the network, and it will perform another scan, printing the IP addresses of the new hosts.

The process of waiting for the user to press enter can be avoided by passing the **-f** option, which will cause the script to read the initial IP addresses from a file (one address per line) and proceed directly to the second scan.

4.3 Sniffing

This script relies on the **ettercap** tool to sniff the packets between the two specified hosts. It doesn't add much functionality, but it's useful to automate and standardize the process, also simplifying its usage.
If we run the script with the following option:

```
sniff -h
```

we will see the following output:

```
Usage: sniff [OPTIONS]
-h              print this help message
-1 IP           specify first IP address
-2 IP           specify second IP address
-o FILE         write output to FILE
-i IFACE        which interface to listen on
-m MODE         what mode to use for mitm (default is arp)
```

We can use the **-1** and **-2** options to specify the IP addresses of the two hosts we want to sniff (both are mandatory). The **-o** option can be used to specify the file to which we want to write the output, and if it's not specified it will default to "**output.pcap**".
The **-i** option can be used to specify the interface to listen on, and if it's not specified it will default to "**wlo1**".
Finally, the **-m** option can be used to specify the mode to use for the mitm attack, defaulting to **arp** if not specified.
This script, once started, will open the textual user interface of ettercap, and will wait for the user to press **q** to quit, saving all packets to the target output file.

Part II

Scripts testing

5 Device description

The device we are going to use in this second part is produced by the same manufacturer of the one used by Alessandro Bringhenti [24], but is a different model, in particular it is the Sricam SP017. It's a wireless IP camera, with night vision, motion detection and two-way audio. It also integrates a microSD card slot and a wireless access point, used to connect the phone for the first configuration.
As with the other model, at the bottom of the camera there is the camera ID, also encoded as a QR code, and the default password.
The image of the device can be found in **Figure 5.1**

We chose this device because of the cost (it can be bought on Amazon for 34€) and because it has a two-way audio, so it could be more interesting to see which services will be offered by the cam and test them.

To test the tools we will assume that the attacker is directly connected to the same network of the attacker, so we will not cover the tolls presented to crack Wi-Fi passwords.

Figure 5.1: Sricam SP017

6 Before device setup

To bring up the device it's sufficient to connect it to the power supply, then it will start emitting periodically a ”beep” sound to indicate that it's ready to be configured.
Then we need to connect our phone and our computer to the device's wifi network: the SSID is ”IPC_***” and the default password is ”12345678”.

After that we can start to scan the device using the script we wrote. First of all we need to get the address IP of the device, and we can do that using the command:

```
ip route
```

The output will be something like this:

```
default via 192.168.66.1 dev wlo1 proto dhcp src 192.168.66.100 metric 20600
...
```

So we can conclude that the IP address of the device is **192.168.66.100**.

Now we can start the script using the command:

```
sudo ./scan -futav -o $FILENAME -i $IP
```

where **$FILENAME** is the name of the file where we want to save the output and **$IP** is the IP address of the device. As previously said this command requires to be run using root privileges because of the use of the nmap command.

Regarding the OS version we got the following output:

```
No exact OS matches for host (If you know what OS is running on it, see
    https://nmap.org/submit/ ).
```

from which we can understand that nmap was not able to identify the OS running on the device.

We can also see that the device exposes various open ports:

```
137/udp open netbios-ns
138/udp open|filtered netbios-dgm
139/tcp open netbios-ssn
1716/tcp open xmsg
1716/udp open|filtered xmsg
41058/udp open|filtered unknown
445/tcp open microsoft-ds
445/tcp open netbios-ssn
51225/udp open|filtered unknown
5353/udp open zeroconf
6566/tcp open sane-port
6566/tcp open tcpwrapped
8828/tcp open unknown
```

We can see that the UDP ports number 137 and 138 and the 139 TCP port are open, which means that the device is exposing the **netbios** protocol, and the ports are used respectively to provide name lookup (137), the datagram service (138) and the session service (139) [12][16].

We can also see that the port 1716 is open for both TCP and UDP (although filtered), and exposes **xmsg** services, used to share information via XML documents.

Then we have port 445 (both UDP and TCP) which is associated with **SMB**: a protocol developed by Microsoft to share files between machines over network.
The port for the **zeroconf** service is used to dynamically configure the hosts on the network [23], and **sane-port** is a protocol used to share scanner devices with other hosts. The latter is also reported as a tcpwrapped port, which means that the port is protected by TCP Wrappers, an ACL system used to filter incoming packages using rules (such as allow only some hosts) in a similar way to firewalls.

Then we also have two ports that are unknown, because they are not standard, therefore we don't know what services are associated with them.

Then we have the output associated with the scripts run by nmap, which reports the following:

```
...
PORT   STATE SERVICE
139/tcp open netbios-ssn

Host script results:
|_nbstat: NetBIOS name: , NetBIOS user: <unknown>, NetBIOS MAC: <unknown> (unknown)
| smb2-security-mode:
|   3:1:1:
|_   Message signing enabled but not required
| smb2-time:
|   date: 2023-08-14T14:36:09
|_  start_date: N/A

...
PORT   STATE SERVICE
445/tcp open microsoft-ds

Host script results:
| smb2-time:
|   date: 2023-08-14T14:36:51
|_  start_date: N/A
| smb2-security-mode:
|   3:1:1:
|_   Message signing enabled but not required
|_clock-skew: -1s
|_nbstat: NetBIOS name: , NetBIOS user: <unknown>, NetBIOS MAC: <unknown> (unknown)

...
```

This output is generated by the **smb2-security-mode** script [17], which complains about the fact that the message signing is enabled but not required, and this could lead to security issues.

Then we have the output of the **nbstat** script [11], which unfortunately doesn't find any hostname and is not able to retrieve the netbios user or the MAC address.

7 After setup

7.1 Setup

To setup the camera it is sufficient to connect the phone to the camera wifi and add the device to the app. The app will automatically connect to the camera and we will be able to configure the camera to connect to the local wifi. In case the camera has been connected using the ethernet cable, it is sufficient to add it to the app using the QR code on the back of the camera.

After having configured the camera we can use the script created to list new hosts in a network to find the IP addresses of the camera and the phone.
It can be run using this command:

```
sudo ./find_hosts -i 192.168.200.223/24
```

where the IP address is the one of the local network (which can be found using the **ifconfig** command). An example of the output of the script is the following:

```
...
192.168.200.237
```

We can confirm this is the IP address of the camera by checking on the app. The address of the phone can be found in the same manner and in our case it was **192.168.200.226**.

7.2 Scanning

We can now repeat the scan of the camera to see which ports are open, now that the configuration has changed.
We can run the script using the same command as before command:

```
sudo ./scan -futav -o $FILENAME -i 192.168.200.237
```

Strangely enough, the script did not recognize the camera operating system and MAC address the first time it was run.
But the second time correctly gave the following output:

```
...
MAC Address: E0:51:D8:ED:73:85 (China Dragon Technology Limited)
Device type: general purpose
Running: Linux 2.6.X|3.X
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3
OS details: Linux 2.6.32 - 3.5
Uptime guess: 0.004 days (since Tue Aug 22 17:34:17 2023)
Network Distance: 1 hop
...
```

We can see that it runs a Linux kernel between 2.6.32 and 3.5, so it's not the latest version, and this may lead to some security issues.

We found that the open ports are the following:

```
1300/tcp open h323hostcallsc
19096/udp open|filtered unknown
```

```
3702/udp open|filtered ws-discovery
443/tcp open ssl/https
49167/udp open|filtered unknown
80/tcp open http
8000/udp open|filtered irdmi
843/tcp open unknown
```

The script also complained about two services being unrecognized, despite having returned data during the first scan.
We can see that the ports associated to the SMB protocol are not open anymore, but we have some new ports, such as the 80 and 443 over TCP, which are used by **HTTP** and **HTTPS**, probably for the administration interface of the camera.

Another interesting port is the number 1300, which is used by the **H.323** protocol, which is used to send and receive audio and video over the network [8].
Other open ports are the ones associated with **ws-discovery**, a protocol to search for web services on a network [22], and **irdmi**, a protocol developed by Intel that provides a remote desktop management interface for the device.
As before we also have two ports that are not recognized and are therefore flagged as unknown.

During the test phase of those ports, we got the following output:

```
...

PORT   STATE SERVICE
443/tcp open https
| ssl-cert: Subject:
    commonName=localhost/organizationName=Example.com/stateOrProvinceName=Washington/countryName=
| Issuer:
    commonName=localhost/organizationName=Example.com/stateOrProvinceName=Washington/countryName=
| Public Key type: rsa
| Public Key bits: 2048
| Signature Algorithm: sha256WithRSAEncryption
| Not valid before: 2015-10-29T00:57:15
| Not valid after: 2025-10-26T00:57:15
| MD5:  2b31:8848:da07:3366:8eba:ecf3:9c96:0be3
|_SHA-1: 48f0:92d1:bc58:b700:b956:2fb3:5819:efee:c4e8:8eab

...

PORT  STATE SERVICE
80/tcp open http
|_http-favicon: Unknown favicon MD5: 761B3CBCC87F3F8BE5FCF40C779E98B9
| http-methods:
|_  Supported Methods: GET HEAD POST
| http-title: index
|_Requested resource was http://192.168.200.237/index.html

...
```

The only useful information we can get from this output is what HTTP methods are allowed by the server, which are **GET**, **HEAD** and **POST**, and some information about the SSL certificate used by the HTTPS server.

7.3 Packet sniffing

We can now try to sniff the traffic between the camera and the router and between the phone and the router.
We can use the script created to sniff the traffic and save it to a file, and analyze later the pcap using Wireshark.
We can run the script using the following command:

```
    sudo ./sniff -1 192.168.200.226 -2 192.168.200.1 -i wlo1
```

to start the sniffing process for the camera. Unfortunately, the analysis of the pcap file did not give any useful information, so we were not able to get the source for the firmware updates of the camera and we could not analyze it further.

8 Conclusions

The goal of this project was to develop a set of scripts to automate the process of vulnerability assessment of IoT devices and it has been, for the part concerning the network, achieved. The scripts developed can be used to scan the network, either to to discover all hosts or only the ones that have been newly connected, and they can also be used to find open ports on those hosts testing them for known vulnerabilities.
Moreover we can use the packet sniffer script to capture the communication between two hosts and analyze it to find vulnerabilities, using standard tools such as Wireshark.
We also found some tools that can be used as alternatives to the ones we used, and we presented some ways to retrieve and analyze the memory of IoT devices to extract information from it.

Unfortunately we were not able to find a way to analyze the firmware or memory of the device, so we had to stop the analysis at the network level, but the main goals of the project have been achieved, so others can use the same tools to continue the analysis, with whe same device or with others.

Bibliography

[1] Aircrack-ng. https://www.aircrack-ng.org/.

[2] Aircrack-ng git repository. https://github.com/aircrack-ng/aircrack-ng.

[3] Bully. https://github.com/aanarchyy/bully.

[4] /dev/fmem github repository. https://github.com/NateBrune/fmem.

[5] /dev/mem manual page. https://man7.org/linux/man-pages/man4/mem.4.html.

[6] Fern wifi cracker. https://github.com/savio-code/fern-wifi-cracker.

[7] Fyrmadyne. https://github.com/firmadyne/firmadyne.

[8] H.323 protocol overview. https://www.cisco.com/c/en/us/td/docs/ios-xml/ios/voice/H-323/configuration/15-mt/voi-h323-overview.html.

[9] Lime memory extractor. https://github.com/504ensicslabs/lime.

[10] Linux volatility configuration. https://github.com/volatilityfoundation/volatility/wiki/Linux.

[11] Nbstat script. https://nmap.org/nsedoc/scripts/nbstat.html.

[12] Netbios wireshark description. https://wiki.wireshark.org/NetBIOS.

[13] Netcat manual page. https://linux.die.net/man/1/nc.

[14] Reaver. https://code.google.com/archive/p/reaver-wps/.

[15] Scp manual page. https://linux.die.net/man/1/scp.

[16] Smb netbios. https://learn.microsoft.com/it-it/troubleshoot/windows-server/networking/direct-hosting-of-smb-over-tcpip.

[17] Smb security script. https://nmap.org/nsedoc/scripts/smb-security-mode.html.

[18] Volatility. https://github.com/volatilityfoundation/volatility.

[19] Volatility 3 git repository. https://github.com/volatilityfoundation/volatility.

[20] Volatility tutorial. https://opensource.com/article/21/4/linux-memory-forensics.

[21] Wifite. https://github.com/derv82/wifite2.

[22] Ws-discovery specification. https://docs.oasis-open.org/ws-dd/discovery/1.1/os/wsdd-discovery-1.1-spec-os.html.

[23] The zeroconf protocol. https://www.ibm.com/docs/en/snips/4.6.0?topic=networking-what-is-zero-configuration.

[24] Bringhenti Alessandro. A framework for the security analysis of iot devices, 2022.

Appendix A Scripts

A.1 Network scanning script

```
#!/bin/bash

# Colors
RED='\033[0;31m'
GREEN='\033[0;32m'
NORMAL_COLOR='\033[0m'
OUTPUT_FILE="output"
IP=""
FULL=false
UDP=false
TCP=false
OS=false
MODE="S"
DISCOVERY=false
ATTACK=false

# Helper functions
error(){
    printf "${RED}$1${NORMAL_COLOR}"
    exit 2
}

print_header(){
    printf "${GREEN}--------------------\n"
    printf "${GREEN}$1${NORMAL_COLOR}\n"
}

run_command(){
    printf "\n\n" >> $OUTPUT_FILE
    printf "${GREEN}--------------------\n"
    printf "${GREEN}$1\n"
    printf "${GREEN}***Executing command: $2 ${NORMAL_COLOR}\n"
    $2 | tee -a $OUTPUT_FILE || error "Failed executing $1"
}

print_usage(){
    printf "Usage: scan [OPTIONS]\n"
    printf "\t-h\t\tprint this help message\n"
    printf "\t-u\t\tperform udp scan\n"
    printf "\t-a\t\tperform attack with default scripts\n"
    printf "\t-t\t\tperform tcp scan\n"
    printf "\t-f\t\tfull scans\n"
    printf "\t-d\t\tdiscovery mode (only -i and -o relevant)\n"
    printf "\t-v\t\tscan for OS version\n"
    printf "\t-o FILENAME\tredirect output to FILENAME\n"
    printf "\t-i IP\t\tscan specified IP\n"
    printf "\t-m MODE\t\ttcp scan mode\n"
}


# Perform TCP scan
tcp_scan(){
    if [ $FULL = true ]
    then
        # Full scan
        run_command "Full TCP scan" "nmap -p- -s${MODE} ${IP}"
    else
        # Default scan
        run_command "Default TCP scan" "nmap -s${MODE} ${IP}"
    fi
}

# Perform UDP scan
udp_scan(){
    if [ $FULL = true ]
    then
        # Perform UDP scan
        run_command "Full UDP scan" "nmap -sU -p- $IP --max-rtt-timeout 20ms --max-retries 0"
    else
        # Perform UDP scan
        run_command "Default UDP scan" "nmap -sU $IP --max-rtt-timeout 20ms --max-retries 0"
    fi
}

# Discover hosts in the network
discovery(){
    run_command "Scanning network" "nmap -sn $IP"
    exit 0
}

attack(){
    print_header "Starting to attack discovered ports"
    PORTS=$(cat $OUTPUT_FILE | grep open | cut -d " " -f 1,2,3,4 | grep tcp | sort | uniq | cut -d "/" -f 1)

    for p in $PORTS
    do
        run_command "Testing port $p" "nmap -sC -p$p $IP"
    done
}

while getopts 'avdhuftm:o:i:' flag; do
    case "${flag}" in
        u) UDP=true ;;
        t) TCP=true ;;
        f) FULL=true ;;
        o) OUTPUT_FILE="${OPTARG}" ;;
        i) IP="${OPTARG}" ;;
        v) OS=true ;;
        a) ATTACK=true ;;
        d) DISCOVERY=true ;;
        m) MODE="${OPTARG}"; TCP=true ;;
        h|*) print_usage && exit 1 ;;
    esac
done

```

```
[ -z $IP ] && error "You must specify an ip address"

echo "" > $OUTPUT_FILE

[ $DISCOVERY = true ] && discovery


[ $OS = true ] && run_command "First scan and os version" "nmap -O -sV $IP"

[ $TCP = true ] && tcp_scan

[ $UDP = true ] && udp_scan

print_header "Scanning results:\n"
cat $OUTPUT_FILE | grep open | awk '{ print $1 " " $2 " " $3 }' | sort | uniq

[ $ATTACK = true ] && attack
```

A.2 Script to find new hosts

```
#!/bin/bash

IP=""
RED='\033[0;31m'
NORMAL_COLOR='\033[0m'
GREEN='\033[0;32m'
OLD_HOSTS=""
NEW_HOSTS=""
FILE=""

print_usage(){
   printf "Usage: find_hosts [OPTIONS]\n"
   printf "\t-h\t\tprint this help message\n"
   printf "\t-f FILE\t\tget old IP addresses from FILE\n"
   printf "\t-i IP\t\tscan specified IP\n"
}

error(){
   printf "${RED}$1${NORMAL_COLOR}"
   exit 2
}

print_header(){
   printf "${GREEN}--------------------\n"
   printf "${GREEN}$1${NORMAL_COLOR}\n"
}

scan(){
   nmap -sn $IP | grep report | awk '{print $5}' | sort
}


while getopts 'hi:f:' flag; do
   case "${flag}" in
      i) IP="${OPTARG}" ;;
      f) FILE="${OPTARG}" ;;
      h|*) print_usage && exit 1 ;;
   esac
```

```
done

[ -z $IP ] && error "You must specify an ip address"

if [ -z $FILE ]
then
    print_header "First scan:"
    OLD_HOSTS=$(scan)
else
    print_header "Retrieving hosts from file:"
    OLD_HOSTS=$(cat $FILE | sort)
fi

echo "Hosts found:"
echo $OLD_HOSTS | tr ' ' '\n'

[ -z $FILE ] && read -p "Press enter to continue"

print_header "Second scan:"
NEW_HOSTS=$(scan)

echo "Hosts found:"
echo $NEW_HOSTS | tr ' ' '\n'

print_header "New hosts:"
echo $NEW_HOSTS | tr ' ' '\n' | grep -v -x -f <(echo $OLD_HOSTS | tr ' ' '\n') | sort
```

A.3 Packet sniffing script

```
#!/bin/bash

FIRST_IP=""
SECOND_IP=""
FILE="output.pcap"
IFACE="wlo1"
MODE="arp"

print_usage(){
    printf "Usage: sniff [OPTIONS]\n"
    printf "\t-h\t\tprint this help message\n"
    printf "\t-1 IP\t\tspecify first IP address\n"
    printf "\t-2 IP\t\tspecify second IP address\n"
    printf "\t-o FILE\t\twrite output to FILE\n"
    printf "\t-i IFACE\twhich interface to listen on\n"
    printf "\t-m MODE\t\twhat mode to use for mitm (default is arp)\n"
}

error(){
    printf "${RED}$1${NORMAL_COLOR}"
    exit 2
}

while getopts 'h1:2:o:i:m:' flag; do
    case "${flag}" in
        1) FIRST_IP="${OPTARG}" ;;
        2) SECOND_IP="${OPTARG}" ;;
        o) FILE="${OPTARG}" ;;
        i) IFACE="${OPTARG}" ;;
        m) MODE="${OPTARG}" ;;
        h|*) print_usage && exit 1 ;;
    esac
done

[ -z $FIRST_IP ] && error "You must specify both ip addresses"
[ -z $SECOND_IP ] && error "You must specify both ip addresses"

ettercap -T -i $IFACE -w $FILE -M $MODE /$FIRST_IP/// /$SECOND_IP/// || error "Failed
    executing ettercap"
```