Department of Information Engineering and Computer Science

Bachelor's Degree in
Computer Science

Final Dissertation

Automated framework for IoT security testing

Supervisor

Crispo Bruno

Student

Mariotti Matteo

Academic year 2022/2023

Acknowledgements

...thanks to... hello

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 attack 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 will divided into two parts: a first more theoretical part, where we will analyze the proposed tools and compare them to the 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 works[22], 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 the following topics:

- Gaining Wi-Fi access to the target network and exploring its weak points
- Attacks to the target network from the inside
- Attacks to the exported services of the target device, in particular web services
- An extension to the previous work, which allows to retrieve the memory image of the target device and analyze it offline

2 Wi-Fi access

In the thesis on which[22] this work is based it was suggested to use the Fern wifi cracker[6] to gain access to the target newtwork. Tis 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[13]: a tool that exploits a flaw in the WPS implementation of some routers using the WPS Pixie-Dust and other brute-force attacks
- Wifite[20]: 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 is 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 well 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 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, retrieve 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 it 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 textbf/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)[8]. 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 [22]), 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**[12] or **scp**[14] 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, whereas netcat will need to be installed.

3.2 Analysis

To analyze the dump file it's possible to use the **Volatility**[17] 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**[18], 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[9][19].

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.

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 syntax:

```
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 script can be passed the **-t** or **-u** options, also in combination, to 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. 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.

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 [22], 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 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) [11][15].

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 [21], and **sane-port** is a protocol used to share scanner devices over network. 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 [16], 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 [10], which unfortunately doesn't find any hostname and is not able to retrieve the netbios user or the MAC address.

7 After setup

Bibliography

[1] Aircrack-ng. https://www.aircrack-ng.org/. accessed 15/06/2023.

[2] Aircrack-ng git repository. https://github.com/aircrack-ng/aircrack-ng. accessed 15/06/2023.

[3] Bully. https://github.com/aanarchyy/bully. accessed 15/06/2023.

[4] /dev/fmem github repository. https://github.com/NateBrune/fmem. accessed 15/06/2023.

[5] /dev/mem manual page. https://man7.org/linux/man-pages/man4/mem.4.html. accessed 15/06/2023.

[6] Fern wifi cracker. https://github.com/savio-code/fern-wifi-cracker. accessed 15/06/2023.

[7] Fyrmadyne. https://github.com/firmadyne/firmadyne. accessed 15/06/2023.

[8] Lime memory extractor. https://github.com/504ensicslabs/lime. accessed 15/06/2023.

[9] Linux volatility configuration. https://github.com/volatilityfoundation/volatility/wiki/Linux. accessed 15/06/2023.

[10] Nbstat script. https://nmap.org/nsedoc/scripts/nbstat.html. accessed 15/06/2023.

[11] Netbios wireshark description. https://wiki.wireshark.org/NetBIOS. accessed 15/06/2023.

[12] Netcat manual page. https://linux.die.net/man/1/nc. accessed 15/06/2023.

[13] Reaver. https://code.google.com/archive/p/reaver-wps/. accessed 15/06/2023.

[14] Scp manual page. https://linux.die.net/man/1/scp. accessed 15/06/2023.

[15] Smb netbios. https://learn.microsoft.com/it-it/troubleshoot/windows-server/networking/direct-hosting-of-smb-over-tcpip. accessed 15/06/2023.

[16] Smb security script. https://nmap.org/nsedoc/scripts/smb-security-mode.html. accessed 15/06/2023.

[17] Volatility. https://github.com/volatilityfoundation/volatility. accessed 15/06/2023.

[18] Volatility 3 git repository. https://github.com/volatilityfoundation/volatility. accessed 15/06/2023.

[19] Volatility tutorial. https://opensource.com/article/21/4/linux-memory-forensics. accessed 15/06/2023.

[20] Wifite. https://github.com/derv82/wifite2. accessed 15/06/2023.

[21] The zeroconf protocol. https://www.ibm.com/docs/en/snips/4.6.0?topic=networking-what-is-zero-configuration. accessed 15/06/2023.

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

Appendix A Scripts

A.1 Network scanning script

```
1  #!/bin/bash
2
3  # Colors
4  RED='\033[0;31m'
5  GREEN='\033[0;32m'
6  NORMAL_COLOR='\033[0m'
7  OUTPUT_FILE="output"
8  IP=""
9  FULL=false
10 UDP=false
11 TCP=false
12 OS=false
13 MODE=""
14 DISCOVERY=false
15 ATTACK=false
16
17 # Helper functions
18 error(){
19     printf "${RED}$1"
20     exit 2
21 }
22
23 print_header(){
24     printf "${GREEN}--------------------\n"
25     printf "${GREEN}$1${NORMAL_COLOR}\n"
26 }
27
28 run_command(){
29     printf "\n\n" >> $OUTPUT_FILE
30     printf "${GREEN}--------------------\n"
31     printf "${GREEN}$1\n"
32     printf "${GREEN}***Executing command: $2 ${NORMAL_COLOR}\n"
33     $2 | tee -a $OUTPUT_FILE || error "Failed executing $1"
34 }
35
36 print_usage(){
37     printf "Usage: scan [OPTIONS]\n"
38     printf "\t-u\t\tperform udp scan\n"
39     printf "\t-a\t\tperform attack with default scripts\n"
40     printf "\t-t\t\tperform tcp scan\n"
41     printf "\t-f\t\tfull scans\n"
42     printf "\t-d\t\tdiscovery mode (only -i and -o relevant)\n"
43     printf "\t-v\t\tscan for OS version\n"
44     printf "\t-o FILENAME\tredirect output to FILENAME\n"
45     printf "\t-i IP\t\tscan specified IP\n"
46     printf "\t-m MODE\t\ttcp scan mode\n"
```

```
47 }
48
49
50 # Perform TCP scan
51 tcp_scan(){
52     if [ $FULL = true ]
53     then
54         # Full scan
55         run_command "Full TCP scan" "nmap -p- -s${MODE} ${IP}"
56     else
57         # Default scan
58         run_command "Default TCP scan" "nmap -s${MODE} ${IP}"
59     fi
60 }
61
62 # Perform UDP scan
63 udp_scan(){
64     if [ $FULL = true ]
65     then
66         # Perform UDP scan
67         run_command "Full UDP scan" "nmap -sU -p- $IP --max-rtt-timeout 20ms --max-retries
               0"
68     else
69         # Perform UDP scan
70         run_command "Default UDP scan" "nmap -sU $IP --max-rtt-timeout 20ms --max-retries 0"
71     fi
72 }
73
74 # Discover hosts in the network
75 discovery(){
76     run_command "Scanning network" "nmap -sn $IP"
77     exit 0
78 }
79
80 attack(){
81     print_header "Starting to attack discovered ports"
82     PORTS=$(cat $OUTPUT_FILE | grep open | awk '{ print $1 " " $2 " " $3 }' | grep tcp |
           sort | uniq | uniq | cut -d "/" -f 1)
83
84     for p in $PORTS
85     do
86         run_command "Testing port $p" "nmap -sC -p$p $IP"
87     done
88 }
89
90 while getopts 'avdhuftm:o:i:' flag; do
91     case "${flag}" in
92         u) UDP=true ;;
93         t) TCP=true ;;
94         f) FULL=true ;;
95         o) OUTPUT_FILE="${OPTARG}" ;;
96         i) IP="${OPTARG}" ;;
97         v) OS=true ;;
98         a) ATTACK=true ;;
99         d) DISCOVERY=true ;;
100        m) MODE="${OPTARG}"; TCP=true ;;
101        h|*) print_usage && exit 1 ;;
102    esac
103 done
104
105 [ -z $IP ] && error "You must specify an ip address"
```

```
106
107 echo "" > $OUTPUT_FILE
108
109 [ $DISCOVERY = true ] && discovery
110
111
112 [ $OS = true ] && run_command "First scan and os version" "nmap -O -sV $IP"
113
114 [ $TCP = true ] && tcp_scan
115
116 [ $UDP = true ] && udp_scan
117
118 print_header "Scanning results:\n"
119 cat $OUTPUT_FILE | grep open | awk '{ print $1 " " $2 " " $3 }' | sort | uniq
120
121 [ $ATTACK = true ] && attack
```