TU WIEN Informatics

Large-Scale Fuzzing of Embedded Web Interfaces

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Marc Sarri
Registration Number 12044794

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dipl.-Ing. Mag. Dr. techn. Edgar Weippl, TU Wien
Assistance: Phd. Christian Kudera, TU Wien
Phd. Georg Merzdovnik, TU Wien
Dr. Jose Luis Muñoz, UPC

Vienna, 15th June, 2021

Marc Sarri

Edgar Weippl

Acknowledgements

First of all to my supervisors Christian and Georg, who have helped me in every possible problem I have encountered. To my roommate who has been there all of the journey helping with motivation and late night beers. To my girlfriend for the support and help in writing.

And last but not least to the city of Wien and the people in it.

Abstract

Every day IoT is being used more and more, they connect from our fridge to our security cameras. Recent studies have showed that they are a huge security hole, especially the web interfaces. Web interfaces are really difficult to secure, as there still aren't any specific guidelines on how to completely secure them.

In this thesis we investigate the efficiency of an unexplored method when trying to emulate these web interfaces. Instead of emulating the whole operating system we try to extract the necessary files and serve the web page using a Docker container.

We present the hurdles that we have had to jump and the ones we have not been able to. And finally we present why this method is not recommended when trying to create a large-scale fuzzing framework.

Resum

Cada dia es fa servir més i més l'Internet de les Coses (IoT), connecten des de la nostra nevera fins a les càmeres de seguretat. Estudis recents han demostrat que són un gran forat de seguretat, especialment les interfícies web. Aquestes interfícies són realment difícils de protegir, ja que encara no hi ha pautes específiques sobre com protegir-les del tot.

En aquesta tesi investiguem l'eficàcia d'un mètode inexplorat per intentar emular aquestes interfícies web. En lloc d'emular tot el sistema operatiu, intentem extreure els arxius necessaris i servir la pàgina web fent servir un contenidor Docker.

Presentem els obstacles que hem hagut de saltar i els que no hem aconseguit. I finalment presentem per què no es recomana aquest mètode quan s'intenta crear una eina per al fuzzing a gran escala.

Resumen

Cada día se usa más y más el Internet de las Cosas (IoT), conectan desde nuestro frigorífico a nuestras cámaras de seguridad. Estudios recientes han demostrado que son un gran agujero de seguridad, especialmente las interfaces web. Éstas interfaces web son realmente difíciles de proteger, ya que todavía no existen pautas específicas sobre cómo protegerlas por completo.

En esta tesis investigamos la eficacia de un método inexplorado para intentar emular estas interfaces web. En lugar de emular todo el sistema operativo, intentamos extraer los archivos necesarios y servir la página web usando un contenedor Docker.

Presentamos los obstáculos que hemos tenido que saltar y los que no hemos podido. Y finalmente presentamos por qué no se recomienda este método cuando se intenta crear una herramienta para el fuzzing a gran escala.

Contents

CHAPTER 1

Introduction

The IoT demand has been in a continued growth since its “beginning”. This has lead the industry to make devices without proper consideration of their security. In 2017 it was estimated that 127 new IoT devices were connected to the internet every second [18]. As a result of the widespread use of insecure IoT, malicious attackers have an easy way into the internal network. In 2017[22] hackers managed to extract 10GB of data from a North American casino. They managed to breach the casinos' network by first hacking into an internet-connected fish tank.

There are a lot of possible security holes in an IoT machine (or machines in general), but we will classify them in 3 main ways. Firstly there are open services that shouldn't be used, like for example, forgetting to stop telnet. Afterward, there are open services that are intended to be used but have vulnerabilities in them (be it because they are 0-days or because haven't been updated). Last but not least there are the web servers, this opens up a plethora of security risks as coding a secure website is extremely difficult.

In this thesis we investigate the security of the websites in IoT devices. In order to create a framework that automatically audits the web pages of IoT devices, we need to emulate them. As explained in chapter 3 we opt for extracting the web files and using Docker to serve them.

In this thesis, we present the first steps for developing a firmware that fuzzes embedded web interfaces and we give reasons why the chosen approach isn't the most optimal one. If in future work this method is perfected, it could be more efficient than other methods used before, as it does not operate the whole Operating System.

In summary, the main contributions are:

- Develop scripts which can aid in the automatic emulation of the firmwares.

- Present the starting steps for emulating 5 firmware's web interfaces from 5 different vendors.
- Conclude that using Docker to serve only the web app is not efficient.

CHAPTER 2

Workplan

2.1 Incidences

When working on Work Plan 3 (Find Common Virtualization Errors and Solve them), which was supposed to be finished on the 18 of May, the problems found weren't able to be solved, so we got stuck completely trying to solve it.

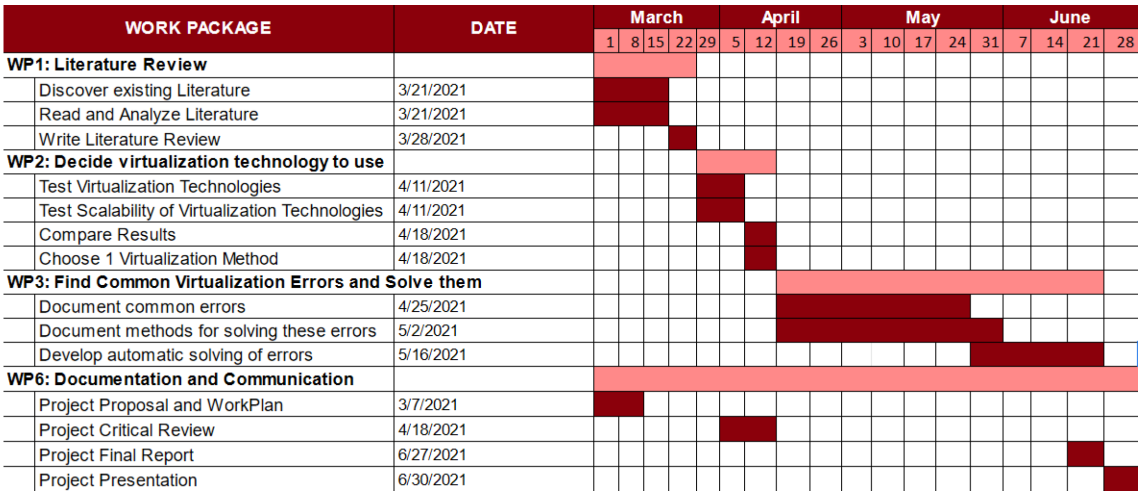
2.2 Updated Work packages

Compare Web Vulnerability Scanners	
Major constituent: SW	WP ref: WP 1
Short description: In order to discover the current solutions to the problem and find which ones are applicable to the project, there is a need for a state of the art Literature Review.	Start event: 01-03-2021 End event: 28-03-2021
Tasks: - Discover existing Literature - Read and Analyze Literature - Write Literature Review	

Decide virtualization technology to use	
Major constituent: SW	WP ref: WP 2
Short description: There are numerous ways to virtualize an embedded web interface, and in the WP 1 we find out all of the existing options and compare them with our study case. In this WP we test the different virtualization technologies and decide which one we will finally use.	Start event: 29-03-2021 End event: 18-04-2021
Tasks: - Test Virtualization Technologies - Test Scalability of Virtualization Technologies - Compare Results - Choose 1 Virtualization Method	

Find Common Virtualization Errors and Solve them	
Major constituent: SW	WP ref: WP 3
Short description: When virtualizing embedded devices, there can be loads of errors resulting from not having the correct peripherals. The aim of sWP is to find the most common of these errors and find a way to solve them automatically without any manual intervention	Start event: 19-04-2021 End event: 27-06-2021
Tasks: - Document common errors - Document methods for solving these errors - Develop automatic solving of errors	

2.3 Updated Time Plan



WORK PACKAGE	DATE	March					April				May					June			
		1	8	15	22	29	5	12	19	26	3	10	17	24	31	7	14	21	28
WP1: Literature Review		▨	▨	▨	▨														
Discover existing Literature	3/21/2021	🟥	🟥	🟥															
Read and Analyze Literature	3/21/2021	🟥	🟥	🟥															
Write Literature Review	3/28/2021				🟥														
WP2: Decide virtualization technology to use						▨	▨	▨											
Test Virtualization Technologies	4/11/2021					🟥	🟥												
Test Scalability of Virtualization Technologies	4/11/2021					🟥	🟥												
Compare Results	4/18/2021							🟥											
Choose 1 Virtualization Method	4/18/2021							🟥											
WP3: Find Common Virtualization Errors and Solve them									▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	
Document common errors	4/25/2021								🟥	🟥	🟥	🟥	🟥	🟥					
Document methods for solving these errors	5/2/2021								🟥	🟥	🟥	🟥	🟥	🟥	🟥				
Develop automatic solving of errors	5/16/2021														🟥	🟥	🟥	🟥	
WP6: Documentation and Communication		▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
Project Proposal and WorkPlan	3/7/2021	🟥	🟥																
Project Critical Review	4/18/2021						🟥	🟥											
Project Final Report	6/27/2021																	🟥	
Project Presentation	6/30/2021																		🟥

CHAPTER 3

State of the Art

There are many challenges in emulating web interfaces in IoT firmwares and analyzing them, furthermore, there are a lot of different techniques and every one with a different scope and difficulties.

The main steps we have to take are:

1. Extracting the firmware
2. Analyse the extracted firmware to find relevant information
3. Emulate the web interface
4. Find and solve errors in the emulation
5. Execute vulnerability analysis

In steps 1 and 2 we extract the binaries from the firmware and find relevant files that will next be used in the emulation phase, like finding the root partition and finding the web server executable.

In order to dynamically analyze a web application, it's necessary to have a functioning web interface. There are various ways to launch the web interface, however, none are perfect. The different approaches are described in Figure 3.1.

The best and most accurate solution would be to have the device physically, but as we are searching for a large-scale and automatic approach this is not feasible.

Our next option is developing a "perfect" emulator, but the difficulty of programming such an emulator for every hardware defeats the purpose of a large-scale framework.

System-level emulation has been extensively researched and tested, and there are different solutions for the various problems that arise.

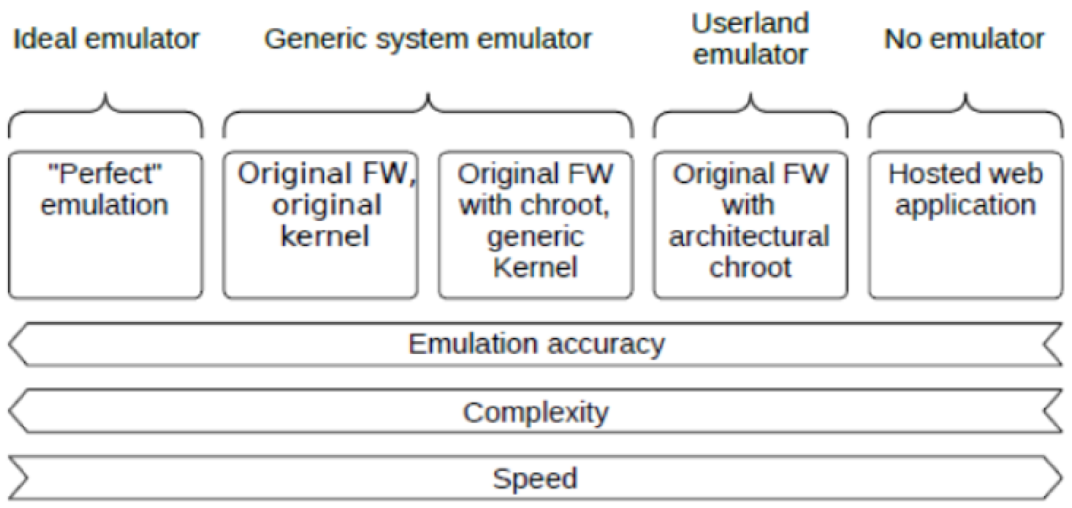


Figure 3.1: Ways to emulate embedded web interfaces. [10]

Costin et al [10] develop a framework in which they pre-compile Debian for `ARM`, `MIPS`, and `MIPSel` under `QEMU`. They pre-compile the kernel as only 5% of their firmware samples contained the kernel. Out of the 1925 firmware samples they started with they were only able to emulate and `CHROOT` into 488 (25,35%) of them, and out of these only 246 (12,77%) of them were able to execute the web server. Although they claim that they could emulate up to 97% of the firmware samples if they pre-compile Debian for all architectures (`ARM`, `MIPS`, `MIPSel`, `Axis CRIS`, `bFLT`, `PowerPC`, `Intel 80386` and more).

Chen et al develop Firmadyne [11], a framework that works similarly to the one developed by Costin et al. They implemented 60 known exploits from the Metasploit exploit framework and discovered 14 previously unknown vulnerabilities. Out of 23,035 firmware images only 9,486 (41%) were successfully extracted.

In order to solve Firmadyne's emulation issues, FirmAE [15] coins an innovative technique called "Arbitrated Emulation". They apply heuristics to solve various emulation problems. Only by adding the arbitrated emulation technique, they have 79% of success in emulating the web interface.

Stepping out of the system-level emulation we've got FirmAFL [25], in which they implement a novel method where they combine system-level and user-level emulation.

At first, the IoT firmware boots up in the system-mode emulator and the user-level programs are launched properly inside the emulator. After the program to be fuzzed has reached a predetermined point, the process execution is then migrated to the user-mode emulation in order to gain high execution speed. Only on rare occasions, the execution is migrated back to the system-mode execution to ensure the correctness of execution.

This approach is meant to fuzz various applications, without specifying the need for a web interface, so the fuzzing techniques are different from our aim.

As seen there are no current solutions in which the user-level emulation is fully used. Costin et al [10] suggest this is not a viable solution when trying to emulate web interfaces because 57% of their samples use binary CGI's or were in some way bound to the hardware platform.

But as no one has used this method we want to find its effectiveness and see if, where other solutions fail, this one succeeds. This way future solutions could combine different techniques in order to have a higher success rate.

Having decided on using user-level emulation we have to use a different base emulator as all of the previous solutions use QEMU[7]. We choose to use Docker[14], as it's got a large user-base and support available.

The post-emulation phase is considered one of the most important, as a lot of successful emulations will probably lack all of the functionality. It will be necessary to find the source of these problems and fix them.

In various previous studies, these checks are also performed, but the problems they face are usually with memory issues and hardware-related issues. As we will be using user-level emulation we won't have these problems so their work isn't relevant to this scenario.

After successfully emulating the web interface comes the vulnerability analysis. The most common technique used to find vulnerabilities is fuzzing (first developed by Takanen et al. [23]), it consists of automatically injecting semi-random data into a program/stack and detect bugs. This automatic approach makes it very useful because it's not limited to a specific vulnerability set, so you can find 0-Day and 1-Day vulnerabilities.

Firmadyne doesn't use Fuzzing in order to find vulnerabilities, they implemented 60 exploits from the Metasploit Framework. When implementing known exploits, you limit the results to those known vulnerabilities and won't find new vulnerabilities.

In Costin et al. they implement open-source web penetration tools available such as Arachni, Zed Attack Proxy (ZAP), and w3af. These 3 penetration tools have fuzzing capabilities but it isn't the main way they find vulnerabilities.

- Arachni[16] has been discontinued since 2020 so we can discard it.
- ZAP[17] is a man-in-the-middle proxy which has capabilities of crawling and fuzzing.
- w3af[21] uses plugins to find already discovered vulnerabilities.

Wang et al. [24] develop a blackbox fuzzer (WMIFuzzer) for the purpose of fuzzing COTS IoT Web Management Interfaces. They use Chrome, Selenium, mitmproxy, and fuzzdb in order to test the Web Interfaces.

- Selenium[13] is a playback tool that imitates human interactions with a web page.
- mitmproxy[2] is like ZAP in which it can modify HTTP & HTTPS requests and responses
- FuzzDB[1] claims to be the first and most comprehensive open dictionary of fault injection patterns, predictable resource locations, and regex for matching server responses.

Searching through open source projects there is a large community of people developing fuzzers for everything, from kernel fuzzing, to application fuzzing, to fuzzing cryptography protocols.

For web fuzzing there are some interesting project which could be useful:

- WFuzz[4], a framework to automate web applications security assessments.
- SSRFmap[3], framework for finding and exploiting Server Side Request Forgery vulnerabilities.
- Boofuzz[19], although not a web fuzzer, it's a network protocol fuzzer, which may come in handy.
- Honggfuzz[12], is a general purpose fuzzer, but has various CVEs from web interfaces so it could end up being a tool to use in conjunction with fuzzdb.

CHAPTER 4

Background

Before starting, we are going to explain some of the main concepts necessary.

4.1 Internet of Things

The Internet of Things (IoT) is the network of daily objects connected to the Internet. The IoT concept was first proposed by Kevin Ashton in 1999 [6]. He talked about all of the objects (not humans) that create information for the internet. Today this idea has transformed the way we live and interact with technology, now we have a vast amount of computer generated information at the tip of our fingers. It's expected that by 2025 the amount of data generated by IoT devices will reach 73.1 ZB[1] [9].

4.2 Firmware

A firmware is any software that is embedded in a piece of hardware. They provide low-level control for a device's specific hardware. In complex hardware it may contain only elementary basic functions of a device and may only provide services to higher-level software. For less complex devices, firmware act as the device's complete operating system, performing all control, monitoring and data manipulation functions.

In this thesis we only focus on the firmwares used by IoT devices, specifically we only use firmwares that contain a whole Operating System. These firmwares are normally obtained from the vendor's website downloading them as an update.

[1] 1 ZB = 2^{70} bytes

4.3 NVRAM

Non-Volatile Random-Access Memory (NVRAM) is a type of memory with the characteristic that can retain data even when no power is available.

This type of memory is used on practically all of the embedded systems in order to retain information when powered down. One example could be in a router, it saves the configuration changes that are manually made. When used as Read-Only-Memory it can be used for storing the system firmware in an embedded device.

4.4 Emulation

Emulation is the ability of one computer system (hardware or software) to behave like another computer system. An emulator enables the host system to run software built for the guest system. So if your computer is running Windows 10, you could emulate any other OS, like past versions of Windows, Linux, etc, and run software that only runs on that OS.

There has been a lot of development into emulators. VirtualBox[8] is a powerful x86 and AMD64/Intel64 virtualization product. QEMU[7] performs hardware virtualization using dynamic binary translation and provides a set of different hardware and device models to emulate. It can run virtual machines at near-native speed and it can emulate user-level processes, allowing applications compiled for one architecture to run on another without running the whole operating system

4.5 OS-Level Virtualization

OS-Level Virtualization differs from full virtualization in that it doesn't have to simulate the whole underlying hardware and kernel. The kernel is shared and then there is the possibility to put as many guest operating systems as the hardware can manage.

Each container (a virtualized system) is seen as a real computer from the point of view of the software running in it, and every container running on the same system can not see each other, unless a network is created. This provides security and control of the accessed resources.

Docker[14] is the implementations of OS-Level Virtualization that is used in this thesis.

4.6 YARA

YARA is an ancronym for: YARA: Another Recursive Ancronym, or Yet Another Ridiculous Acronym.[20] YARA was developed by Victor Alvarez from VirusTotal[5]. The main aim of the tool is helping malware researchers identify and classify malware samples.

It works by defining rules, in which you can specify strings, binary patterns, wild cards, regular expressions and conditions. After you have a description of whatever you want to describe (commonly malware) and you can include it in a python script to check the conditions.

CHAPTER 5

Implementation

This section covers the methodology developed before starting to emulate the firmwares, after we identify some possible problems we which we finally encountered and propose some solutions to them. We then present the steps taken to try and emulate the firmwares and how it was not successful.

5.1 Methodology

First of all we have to develop a methodology for emulating the firmwares.

The firmware samples used are downloaded from the FirmAE dataset. There are a total of 1127 firmware samples out of 8 different vendors.

First of all we want to manually emulate at least one firmware sample in order to have the process needed for the later automation.

When trying to emulate the firmwares we will follow these steps:

1. Extract filesystem using `binwalk`[1].
2. Search through the boot parameters in order to find the init file executed at boot.
    a) If we find it continue to 3.
    b) If there are no boot parameters try and manually find it and continue to 3.
3. Analyse the init file in order to find where the web server is located and what parameters are used.

[1]Binwalk is a tool for searching a given binary image for embedded files and executable code.

Table 5.1: Problems a priori

First Problem	Solution
When executing the web server, it will source the NVRAM for configuration parameters, but as we are emulating in a Docker container we don't have access to it	Security researcher Zachary Cutlip has developed a library that intercepts the calls to `nvram_get()` and you can manually put the return values using `LD_PRELOAD`.
Second Problem	**Solutions**
When doing the automatic analysis of the firmware how are the files needed for emulation detected?	We can code a parser that reads the init file and finds the necessary files for emulation. As later seen, this option is discarded for the difficulty of decompiling the init file and parsing it.
	Using YARA rules[5] it's possible to find common files and folders used in firmwares and use them for emulation.

4. Write Dockerfile which includes all of the necessary files for executing correctly the web server.
5. Test Docker container.
6. Debug.

After emulating at least 5 devices from different vendors and documenting the process we can develop some python scripts which will automate the extraction, analysis and emulation of the firmware.

5.2 Problems a priori

After the methodology was developed we identified 2 possible problems that we might encounter and found possible solutions to them. We can see them at Table 5.1

5.3 Scripts

During the manual analysis of the firmwares we have developed a some simple scripts that would aid us during the automatic part of the analysis, although as explained, in the end the emulation of only the web app is not a valid solution.

The code is divided in two classes, Firmware and Crawler.

In the firmware class there is all of the code related to the analysis of the firmware root folder.

And in the crawler class you give it a folder, and it tries to extract and find the firmware root folders.

The full code of these two classes can be found in the Annex [8]

5.3.1 Firmware class

The tree main objectives of the Firmware class are finding the achitecture of the firmware, the init file and the web server files.

Find architecture

Right now the only capability it has is finding the architecture of the firmware. It achieves this by using the `file` command. The `file` command give us information about the file passed as argument, if this file is an executable file it also gives us the architecture for which it was compiled. A sample output would be:

```
rc: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked, interpreter /lib/ld-uClibc.so.0, stripped
```

In this case we've it's compiled for ARM 32 bits LSB (Least Significant Byte)

In order to extract this we first check if the second word is `ELF`, and then we extract the 6th word, the 3rd and the 4th, in that order.

Find init

The reason for finding the init file would be to parse it later and find the server files needed. But as later explained in Section 5 the parsing of binary files is extremely complicated, and so we discarded coding this capability.

In case this investigation is continued here are 2 possible methods to achieve it.

First of all would be reading the boot parameters. When extracting the firmware a data file appears in which if there are boot parameters there is the init file.

- If an init file is found it shows up like this:

  ```
  root=/dev/mtdblock2 console=ttyS0,115200 init=init
  earlyprintk debug
  ```

- If no init file is found:

  ```
  No init found.  Try passing init= option to kernel.
  ```

If the init file is found we can simply return the value given.

In case there are no boot parameters, we should have a list of common init files and search the the filesystem for the most common folders. This approach would not have a 100% effectivity rate as some firmwares have custom init scripts/binaries.

Find server files

As seen in section 5, out of 5 firmwares, none of them shared exactly the same configuration when executing the web servers.

The best approach to find the necessary files to emulate the web app would be having a huge data-set in which we had information about where are they usually located. Having to manually analyze this many firmwares is out of the scope of this thesis and is left as future work.

5.3.2 Crawler class

The Crawler class has two main objectives:

- Extract any binary
- Find the root folder of the firmware

In order to accomplish this we decide the hierarchy of the folders, at the root we will have folders named after the vendor, then inside every file must be a packed firmware file with the name of the firmware/update.

```
asus

\ FW_BLUECAVE_300438432546.zip

Belkin

\ F9K1122v1_Belkin_1.00.33_upg.bin

\ F7D4301-8301_WW_1.00.30.bin
```

Figure 5.1 explains how the crawler navigates the files and extracts them. The criteria for extracting a file is looking at the extension, if it is a `.zip`, `.bin`, `.img`, `.trx` or `.rar`.

Recursivity is used because some firmwares have packed images inside of zip files.

Every time the Crawler goes into a folder it searches for the following folders `/bin`, `/lib`, `/sbin`, `/usr` or `/tmp`. If it finds at least 4 we consider it to be the root filesystem of the firmware and we create a Firmware object with the root path, the name of the firmware and the vendor.

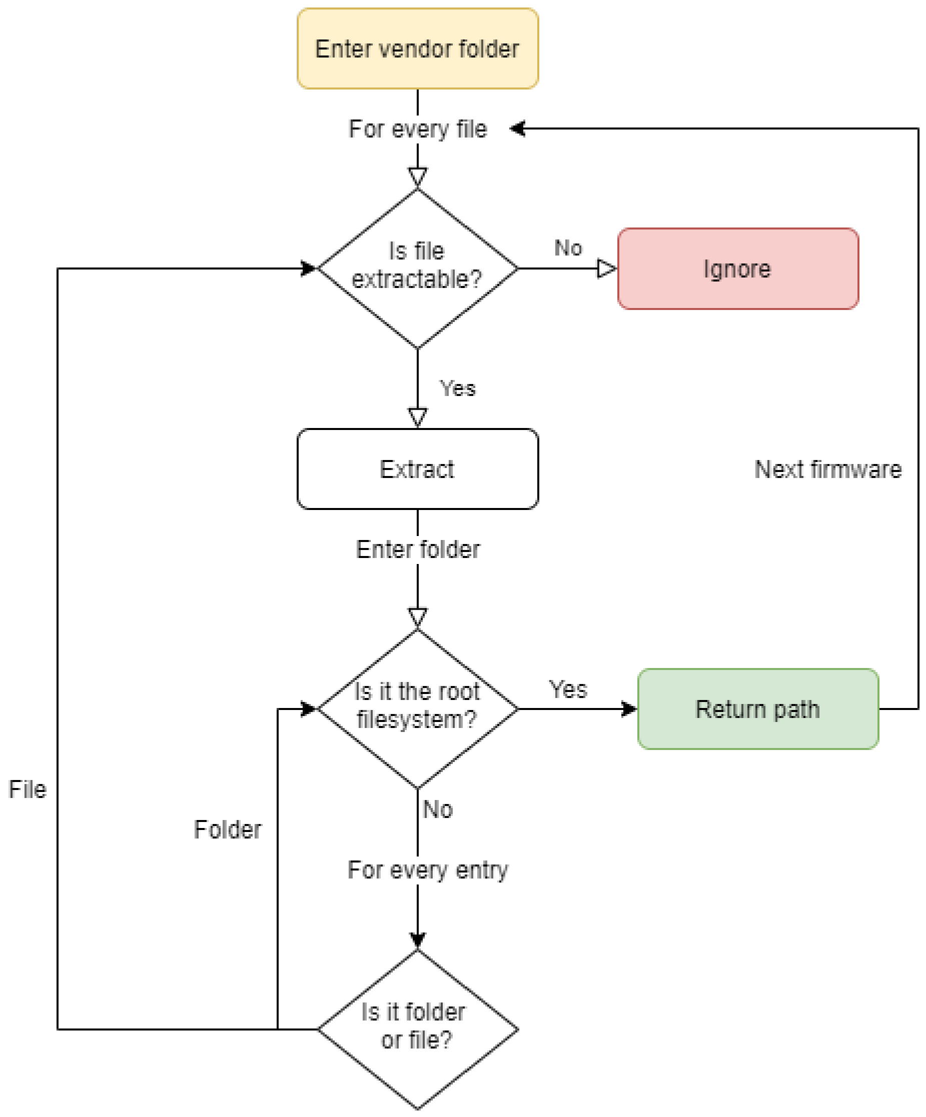


Figure 5.1: Workflow of Crawler.

5.4 Emulation

Applying the methodology developed previously we begin trying to emulate 5 devices, here is the process

5.4.1 Tenda AC15

Analysis of the firmware

The Tenda ac15 is a dual-band Gigabit WiFi router. The reason for choosing this device first was made because GitHub user Saumil Shah has a very detailed explanation of how to emulate the full operating system using QEMU. This walkthrough isn't completely useful for our purpose but it gives us a head start in understanding the filesystem and boot parameters.

Following the methodology already described we start by unpacking the firware using binwalk. When unpacked we obtain the squashfs-root folder, and a data file 5C containing the firware headers. Using the strings command and piping it to a .txt file we can search for the boot parameters and we find the following line:

```
root=/dev/mtdblock2 console=ttyS0,115200 init=init earlyprintk debug
```

Here we can clearly see that the first file read when booting is the file located at the root folder called `init`. This file is a symlink that points to `bin/busybox`.[2]

We tried to decompile the Busybox binary using Ghidra but couldn't find how the http daemon was started.

At this point we relied on the blog post mentioned above and found that Busybox looks at the file `/etc/inittab`. We confirm that by using `grep -r` which also searches for strings inside a binary.

As later seen this is a common file found in various vendors.

The `/etc/inittab` file contains various files that have to be called when doing certain things, for example when initialising the system, or when shutting down.

In this case the line we are interested is this one:

```
sysinit\:/etc_ro/init.d/rcS
```

This means that when initialising the system the file `/etc_ro/init.d/rcS` has to be called. This bash script creates various files and mounts some filesystems, but what is most important for us is:

```
cp -rf /etc_ro/* /etc/
cp -rf /webroot_ro/* /webroot/
sh /etc/nginx/conf/nginx_init.sh
```

We now know that some files from etc will probably be needed, that we have the static files of the web page at webroot and that there is a script that initialises nginx.

Looking at the `nginx_init.sh` script we find that it starts nginx and starts a fastCGI server in port 8188 that points to the binary `/usr/bin/app_data_center`

Checking the `nginx.conf` file located under the `etc_ro/nginx/conf` folder we see that nginx only listens to the port 8180 and in no case looks at the webroot folder. Looking

[2]Busybox is a software suite that provides several Unix utilities in a single executable file. The authors call it "The Swiss Army Knife of Embedded Linux". Busybox can read the initial symlink reading the argv[0] parameter.

for references to webroot we find two binaries that reference the folder, `bin/dhttpd` and `bin/httpd`.

The process by which the servers communicate is explained in Figure 5.2

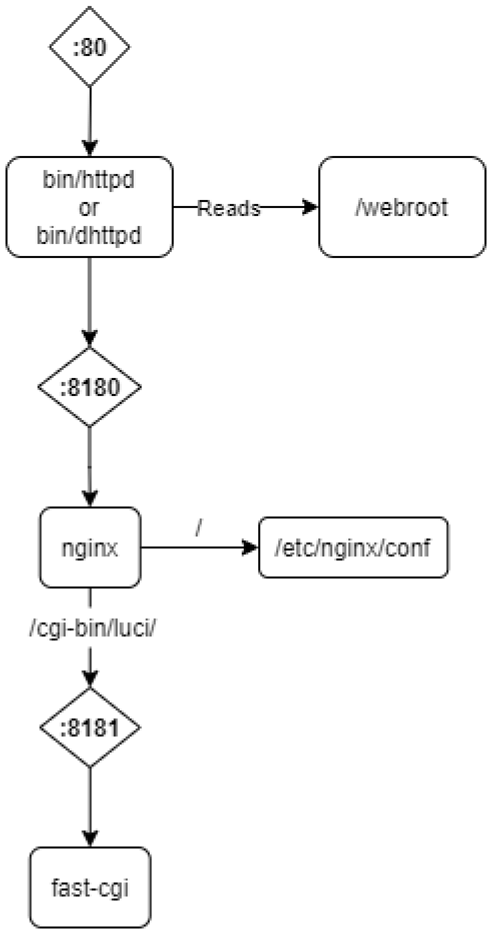


Figure 5.2: Diagram of web servers.

There is a certain degree of uncertainty in this diagram as we can't read the contents of the binaries, we can only guess.

First emulation

For starting we try to only emulate the static web files.

The most simple way to achieve this is by using a docker image with NGINX preinstalled, copy all of the contents of `webroot_ro` to `/usr/share/nginx/html` (which is the default directory for serving static content).

This method gives us the expected results, we are able to enter the configuration page, and we can interact with the elements, but we can't change any configuration.

Second emulation

The first step would be creating a Dockerfile that includes all of the necessary files and folders, and executes them. Doing this we encounter that none of the binaries execute, as they are compiled for ARM32 LSB.

Third emulation

On our second emulation we modify the Dockerfile so we can install QEMU usermode. QEMU User space emulator allows us to execute binaries with the desired architecture.

5.4.2 Asus Blue Cave

Analysis of the firmware

The Asus Blue Cave is a dual-band WiFi router specifically designed for smart homes.

Following the methodology, the first step after unpacking is searching the boot parameters for an init file. There are no boot parameters available, this means that probably the firmware downloaded is only an update that doesn't include everything.

Now we have to search manually for the init files. Using grep we find various potential files, after double checking we confirm that `/sbin/init` is the one. In this case the init file points to `/sbin/rc`.

As seen in the following firmwares, this `rc` file is commonly used, it seems to be a replacement for busybox.

Trying to decompile the `rc` binary again is an enormous task, so we opt for manually finding the next files.

Using `grep` searching for `rc.d` scripts we find `/rom/opt/lantiq/etc/rc.d/flow-aware-qos.sh` which initialises the IP tables for the router, but it doesn't start any web page. We can find a reference to `/etc/rc.d` but the `/etc/` is a broken symlink, so it's probably initialised by the `rc` binary.

UsingAsus Blue Cave `grep` searching for `httpd` we find various interesting binaries: `/usr/sbin/httpd`, `/usr/sbin/lighttpd`, `/usr/sbin/lighttpd-arpping` and `/usr/sbin/lighttpd-monitor`

There is also the /www folder which contains `.asp`, `.htm`, `.dict`, `.js`, `.css` and `json` files. The `.asp` files are served by an ASP.NET server.

First emulation

First we are going to try and serve directly the files without using any of the given binaries. To achieve this we use `mono-fastcgi-server`.

After writing the Dockerfile and executing it we get a `HTTP 500 Error`, and every file we try to access is forbidden. Giving full permissions to the files didn't solve the error.

Second Emulation

Changing the focus we try to execute the binaries provided.

First we check for what architecture the binary was compiled using the `file` command. It's compiled for MIPS 32-bit Most significant byte, so we will use the `qemu-mips` command.

In the Docker container we copy the contents of `/www`, the `lighttpd` binaries and the binaries located under `/lib` as they are necessary for the execution.

Executing the main `lighttpd` binary gives us an error that the cache located at `/etc/ld.so.cache` is corrupted.

As already mentioned the folder `/etc` is a broken symlink so we can't try and replicate the cache.

Third emulation

Although seemingly the last error was not related to the nvram we try to emulate it any ways.

In order to use `nvram_faker` it's necessary to preload the library when executing the binary, this is done by putting `LD_PRELOAD` before the execution of the command and using the `nvram_faker` compiled for the specific architecture.

Doing this gave us the same error.

5.4.3 Linksys WAG320N

Analysis of the firmware

The Linksys WAG320N is a dual-band WiFi router.

Following the methodology we don't find any boot parameters after unpacking.

Manually searching for the init file we find `/sbin/init` which is a symlink to `/bin/busybox`. As already explained before decompiling the busybox binary has a high complexity.

We don't find any `inittab` file but there is are 3 `rcS` files. 2 of them are identical, and they start the `httpd` service through the `/usr/sbin/rc` binary. It also copies `/usr/sbin/mini_httpd` into the `/tmp` directory. It also executes `/usr/sbin/mini_httpd`.

The third `rcS` script is not relevant.

There are 2 folders containing web files, `www.ar` and `www.eng`. They don't share the same content, although searching for any reference to them, only `www.eng` is mentioned, so we can assume that `www.ar` is not used.

There are 3 CGI files that are symlinks, we can simply copy them over to the folder.

The steps we have to take to serve the web page are the following:

1. Execute `rc_server`.
    - It also starts the `mini_httpd` as it appears in the `rc_server` binary.
2. Start `httpd` through `rc`.
3. Add the `/lib` directory so the binaries can work.

Emulation

Firstly we have to copy the necessary files.

But when trying to execute the `rc_server` gives us the same error as the last firmware in which it says that the cache is corrupted. And using `nvram_faker` does not solve the problem.

5.4.4 Belkin N600

Analysis of the firmware

The Belkin N600 is a dual-band WiFi range extender.

After unpacking the firmware we don't find any boot parameters, so we manually search for the init file.

We find various interesting files: `/bin/init.sh`, `/etc/init.d/rcS`, `/etc/inittab`, which contains: `::sysinit:/etc/init.d/rcS` and `/sbin/init -> /bin/busybox`

And searching for `httpd` we find the `/bin/webs` binary. This binary is a BOA web server.

BOA is a web server mainly used in embedded systems.

The binary `/bin/webs` is started at `/bin/init.sh`.

The web files are located at `/web1` and the files are `.asp`, which means that the web server is ASP.NET.

Taking a closer look at the web files we see that instead of using `nvram_get` to get information from the NVRAM it uses `getVar` which looks into `/bin/webs`.

In order to emulate the web page we have to execute `/bin/webs` making sure that the configuration file `/etc/boa/boa.conf`, the `/web1` and the `/lib` directories are in place.

First emulation

First we copy the necessary files inside the container. We have to use the command `qemu-mips` to execute the `/bin/webs` binary. Doing so gives us the next error: `Could not chdir to "/etc/boa": aborting`

Seccond emulation

First we create the `/etc/boa` directory.

Executing the web server it starts executing but it suddenly crashes with the following error:

```
caught SIGSEGV, dumping core in /tmp

qemu: uncaught target signal 6 (Aborted) - core dumped
```

5.4.5 D-Link DIR-600 B1

Analysis of the firmware

The D-Link DIR-600 B1 is a WiFi Router.

After unpacking the firmware we don't find any boot parameters, so we manually search for the init file.

Using `grep` we don't find any reference to an init file, although there is a `rcS` script that executes all of the scripts in the same folder, which are none. Looking any reference to `/etc/init.d/rcS` there is one in `/bin/busybox`, so probably when initialising the router it copies the necessary files to the folder.

Searching for any `httpd` binary there is none, but there is a script at `/etc/scripts/config.sh` that executes `/etc/templates/webs.sh` start.

Looking inside of this script we find:

```
case "$1" in
...
start|restart)
        [ -f /var/run/webs_stop.sh ] &&
            sh /var/run/webs_stop.sh > /dev/console
        rgdb -A $TROOT/httpd/webs_run.php -V
            generate_start=1 > /var/run/webs_start.sh
        rgdb -A $TROOT/httpd/webs_run.php -V
            generate_start=0 > /var/run/webs_stop.sh
        sh /var/run/webs_start.sh > /dev/console
        ;;
...
```

So the startup script is created by this `rgdb` command, which is a symlink to `/usr/sbin/xmldb`.

Looking at some of the scripts we can see that it doesn't use the `nvram_get` in order to get stored configuration, instead it uses `devdata get`. the `devdata` binary is a symlink to `/usr/sbin/rgbin`.

The web files are located under `/www` and it consists mainly of `.php` files.

Emulation

Copying the necessary binaries and executing them with QEMU gave us the same error as the last firmware.

```
qemu: uncaught target signal 6 (Aborted) - core dumped
```

CHAPTER 6

Budget

During the thesis the tools used have been a Lenovo laptop, which could have been any computer with no limitation in the needed specifications, and all of the software used is free (as in free beer) and open-source like Docker and Python.

Then the main costs of the theses then comes from the hours spent on it by the student and the supervisors. Taking into consideration the wages in Austria, 25 hours/week for the student and 2 hours/week for the supervisors combined we get the calculations from table 6.1

Position	Amount	Wage/hour	Dedication
Junior engineer	15,00 €/h	25 h/week	7.500 €/h
Senior engineer	30,00 €/h	3 h/week	1.800 €/h
TOTAL: 9.300 €/h			

Table 6.1: Calculations for the wages

CHAPTER 7

Discussion

When starting this thesis we wanted to emulate the web page of IoT devices. We found that there are already a lot of different implementations in order to achieve this, but none of them had an extremely high effectivity rate. So we decided to take a different route, instead of emulating the whole operating system, we extract the web files used to serve the web page and serve it in a Docker container, which consumes less resources, and so should be more efficient. As seen in the Emulation section [5.4] we didn't even manage to emulate them manually.

After writing the State of the Art [3] we had an idea of how we thought the web pages were served. We thought that the `init` script would be a bash script which executed a web server like `NGINX` or `lighttpd`. Then the web server would search for static files and serve them, or if dynamic files were needed there would be a simple CGI server which would be easy to emulate.

Instead what we found were binary files very difficult to decompile, which in turn executed a bunch of services related to the web server. And every firmware did this in it's own way so the process of identifying the necessary files has to be repeated for each one. This complicates enormously the task of automatically identifying and serving the web server. Furthermore, as we would be using a large number of binaries and libraries, the whole purpose of only emulating the web server in order to lower the resources used to emulate would be defeated.

CHAPTER 8

Conclusion

In this thesis we demonstrate that trying to extract a web from a firmware and serving it using a container is not the most effective approach of achieving the desired results. In order to try to achieve this we developed some scripts which automatically extract the firmware.

In order to address the main hurdles that appear when trying to emulate, there should be an enormous work done that may or may not work.

We conclude that right now the most optimal solution for a large scale fuzzing framework is emulating the whole Operating System.

As future work we leave:

- Analysing a large number of firmwares in order to have statistics on the most used init files, web servers and binaries.
- Create YARA rules that define the most common init files, web servers and binaries.
- Create a script that parses binary init files.
- Fix errors when executing binaries with QEMU.

List of Figures

List of Tables

Bibliography

[1] fuzzdb: Dictionary of attack patterns and primitives for black-box application fault injection and resource discovery. `https://github.com/fuzzdb-project/fuzzdb`.

[2] mitmproxy - an interactive https proxy. `https://mitmproxy.org/`.

[3] Ssrfmap: Automatic ssrf fuzzer and exploitation tool. `https://github.com/swisskyrepo/SSRFmap`.

[4] wfuzz: Web application fuzzer. `https://github.com/xmendez/wfuzz`.

[5] Victor Alvarez. Yara - the pattern matching swiss knife for malware researchers. `https://virustotal.github.io/yara/`.

[6] Kevin Ashton. That 'internet of things' thing. `https://www.rfidjournal.com/that-internet-of-things-thing`, 2009.

[7] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[8] Virtual Box. Oracle virtualbox, virtualization software. `https://www.virtualbox.org/`, 2007.

[9] International Data Corporation. Iot growth demands rethink of long-term storage strategies, says idc. `https://www.idc.com/getdoc.jsp?containerId=prAP46737220`, 2020.

[10] Andrei Costin. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. 2016.

[11] Daming D. Chen. Towards automated dynamic analysis for linux-based embedded firmware. 2016.

[12] Google. honggfuzz: Security oriented software fuzzer. `https://github.com/google/honggfuzz`.

[13] Jason Huggins. Seleniumhq browser automation. https://www.selenium.dev/.

[14] Solomon Hykes. Docker project site. https://www.docker.com/, 2013.

[15] Mingeun Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. 2020.

[16] Tasos Laskos. Github - arachni/arachni: Web application security scanner framework. https://github.com/Arachni/arachni.

[17] OWASP. Owasp zed attack proxy. https://www.zaproxy.org/.

[18] Mark Patel. What's new with the internet of things? https://www.mckinsey.com/industries/semiconductors/our-insights/whats-new-with-the-internet-of-things, 2017.

[19] Joshua Pereyda. boofuzz: Network protocol fuzzing for humans. https://github.com/jtpereyda/boofuzz.

[20] Victor Alvarez [@plusvic]. "yara is an ancronym for: Yara: Another recursive ancronym, or yet another ridiculous acronym. pick your choice.". https://twitter.com/plusvic/status/778983467627479040.

[21] Andres Riancho. w3af - open source web application security scanner. http://w3af.org/.

[22] Alex Schiffer. How a fish tank helped hack a casino - the washington post. https://www.washingtonpost.com/news/innovations/wp/2017/07/21/how-a-fish-tank-helped-hack-a-casino/?noredirect=on, 2017.

[23] Ari Takanen. Fuzzing for software security testing and quality assurance. 2008.

[24] Dong Wang. Discovering vulnerabilities in cots iot devices through blackbox fuzzing web management interface. 2019.

[25] Yaowen Zheng. Firm-afl: High-throughput greybox fuzzing of iot firmware via augmented process emulation. 2019.

Annex

crawler.py

```
case "$1" in
# !/usr/bin/python3

import os
import firmware

class Crawler():
 def __init__(self, path):
  self.path = path
  self.firmwares = []
  self.extensions_firm = {"bin"} # DEBUG VARIABLE
  self.extensions_extra = {"7z"} # DEBUG VARIABLE

 # Crawls the directory
 # It assumes that the first directories are the name of the vendor
 def crawl(self):
  os.chdir(self.path)

  # Enter every vendor's folder
  for v in os.scandir():
   print(v.name)
   if(v.is_dir()):
    vendor = v.name.split("_")[0]
    for f in os.scandir(v.path):
     # We assume the name of the firmware is the name of the file,
     # be it a file or folder
     name = f.name
     root = None
     print(vendor + ": " + name)
     if(f.is_dir()):
      print("Is directory")
     root = self.find_root(f.path, 0)
    elif(f.is_file()):
     print("Is file")
     extract = self.extract(f.path)
     if(extract): root = self.find_root(extract, 0)

    if(root):
     # print(root)
     firm = firmware.Firmware(root, name, vendor)
     self.firmwares.append(firm)
    print("------------------------------------------")
 print(len(self.firmwares))
 print("END")

# Recursive function that crawls inside the
# directory of any firmware to find the root folder
def find_root(self, path, count):
 if(count > 3): # Check if we have executed 3 times the find root
  print("Count is 3!!!!!")
  return None
 print("Looking into folder: " + path)
 # These folders have to be there in order for it to be a filesystem
 check_folders = {"bin", "lib", "sbin", "usr", "tmp"}
 folders = os.listdir(path)
 # Exclude files
 tmp = []
 for f in folders:
  if(os.path.isdir(os.path.join(path, f))): tmp.append(f)
 folders = tmp

 # Check if more than 3 folders coincide
 if(len(check_folders.intersection(folders)) > 3):
  print("Found Root: " + path)
  return path
 else:
  for f in os.scandir(path):
   # If we find a folder call this function to do the same
   r = None
   if(f.is_dir()):
    print("Is Directory " + f.name)
    r = self.find_root(f.path, count + 1)
   # If we find a file send it to self.extract() in
   # order to decide if it has to be extracted
```

```
    elif(f.is_file()):
     print("Is file: " + f.name + " - Sending to extract")
     extract = self.extract(f.path)
     if(extract): r = self.find_root(extract, count + 1)
    if(r != None): return r

  return None


# Returns None if already extracted or isn't a packed firmware
# Returns path of extracted firmware
def extract(self, path):
 print("Extracting: " + path)
 extensions = ["zip", "ZIP", "bin", "img", "trx", "rar"]
 # Check if extension is one of the supported ones
 if(not (os.path.basename(path).split(".")[-1] in extensions)):
  return None
 # Check if it was already extracted by binwalk looking
 # for a directory named _{original name}.extracted
 if("_"+os.path.basename(path)+".extracted" in
 os.listdir(os.path.dirname(path))):
  print("Already extracted")
  return None
 # Check if the file found is a .zip or .rar and
 # if the parent directory is an already extracted firmware,
 # then don't extract anything
 if((os.path.basename(path).split(".")[-1] == "zip" or
 os.path.basename(path).split(".")[-1] == "rar")
 and os.path.dirname(path).split(".")[-1] == "extracted"):
  print("Already in an extracted folder, FUCK UUUUUUU")
  return None
 # Extraction of the firmware
 command = "binwalk -e " + path.replace("(","\(").replace(")","\)")
 print(command)
 os.system(command)
 filename = os.path.join(os.path.dirname(path),
 "_"+os.path.basename(path)+".extracted")
 # Check if the folder was created
 if("_"+os.path.basename(path)+".extracted" in
 os.listdir(os.path.dirname(path))):
  print("Returning this filename: " + filename)
  return filename
 elif("_"+os.path.basename(path)+"-0.extracted" in
  os.listdir(os.path.dirname(path))):
   return os.path.join(os.path.dirname(path),
   "_"+os.path.basename(path)+"-0.extracted")
  else:
   print("Did not return: " + filename)
   print("These are the files in folder: ")
   print(os.listdir(os.path.dirname(path)))

  return None
```

firmware.py

```
# !/usr/bin/python3

import os
import firmware

class Crawler():
 def __init__(self, path):
  self.path = path
  self.firmwares = []
  self.extensions_firm = {"bin"} # DEBUG VARIABLE
  self.extensions_extra = {"7z"} # DEBUG VARIABLE

 # Crawls the directory
 # It assumes that the first directories are the name of the vendor
 def crawl(self):
  os.chdir(self.path)

  # Enter every vendor's folder
  for v in os.scandir():
   print(v.name)
   if(v.is_dir()):
    vendor = v.name.split("_")[0]
    for f in os.scandir(v.path):
     # We assume the name of the firmware is the name
     # of the file, be it a file or folder
     name = f.name
     root = None
     print(vendor + ": " + name)
     if(f.is_dir()):
      print("Is directory")
      root = self.find_root(f.path, 0)
    elif(f.is_file()):
     print("Is file")
     extract = self.extract(f.path)
     if(extract): root = self.find_root(extract, 0)

     # if("." in f.name): self.extensions_firm.add(f.name.split(".")[-1])

    if(root):
     # print(root)
     firm = firmware.Firmware(root, name, vendor)
     self.firmwares.append(firm)
    print("-------------------------------------------")
 print(len(self.firmwares))
 print("END")

# Recursive function that crawls inside the
# directory of any firmware to find the root folder
def find_root(self, path, count):
 if(count > 3): # Check if we have executed 3 times the find root
  print("Count is 3!!!!!")
  return None
 print("Looking into folder: " + path)
 # These folders have to be there in order for it to be a filesystem
 check_folders = {"bin", "lib", "sbin", "usr", "tmp"}
 folders = os.listdir(path)
 # Exclude files
 tmp = []
 for f in folders:
  if(os.path.isdir(os.path.join(path, f))): tmp.append(f)
 folders = tmp

 # Check if more than 3 folders coincide
 if(len(check_folders.intersection(folders)) > 3):
  print("Found Root: " + path)
  return path
 else:
  for f in os.scandir(path):
   # If we find a folder call this function to do the same
   r = None
   if(f.is_dir()):
    print("Is Directory " + f.name)
    r = self.find_root(f.path, count + 1)
   # If we find a file send it to self.extract()
```

```
        # in order to decide if it has to be extracted
        elif(f.is_file()):
         print("Is file: " + f.name + " - Sending to extract")
         extract = self.extract(f.path)
         if(extract): r = self.find_root(extract, count + 1)
        if(r != None): return r

      return None


    # Returns None if already extracted or isn't a packed firmware
    # Returns path of extracted firmware
    def extract(self, path):
     print("Extracting: " + path)
     extensions = ["zip", "ZIP", "bin", "img", "trx", "rar"]
     # Check if extension is one of the supported ones
     if(not (os.path.basename(path).split(".")[-1] in extensions)):
      return None
     # Check if it was already extracted by binwalk looking
     # for a directory named _{original name}.extracted
     if("_"+os.path.basename(path)+".extracted" in
     os.listdir(os.path.dirname(path))):
      print("Already extracted")
      return None
     # Check if the file found is a .zip or .rar and if
     # the parent directory is an already extracted firmware,
     # then don't extract anything
     if((os.path.basename(path).split(".")[-1] == "zip" or
     os.path.basename(path).split(".")[-1] == "rar")
     and os.path.dirname(path).split(".")[-1] == "extracted"):
      print("Already in an extracted folder, FUCK UUUUUUU")
      return None
     # Extraction of the firmware
     command = "binwalk -e " + path.replace("(","\(").replace(")","\)")
     print(command)
     os.system(command)
     filename = os.path.join(os.path.dirname(path),
     "_"+os.path.basename(path)+".extracted")
     # Check if the folder was created
     if("_"+os.path.basename(path)+".extracted" in
     os.listdir(os.path.dirname(path))):
      print("Returning this filename: " + filename)
      return filename
```

```
elif("_"+os.path.basename(path)+"-0.extracted" in
os.listdir(os.path.dirname(path))):
 return os.path.join(os.path.dirname(path),
 "_"+os.path.basename(path)+"-0.extracted")
else:
 print("Did not return: " + filename)
 print("These are the files in folder: ")
 print(os.listdir(os.path.dirname(path)))

return None
```