UNIVERSITÉ LIBRE DE BRUXELLES
Faculté des Sciences
Département d'Informatique

Emulation-based dynamic analysis plugins for the FACT framework

Hanne Peeters

Promotor :
Prof. Wim Mees

Master Thesis
in Cybersecurity

Academic year 2019 - 2020

Preface

First, I would like to thank my promotor for the guidance and feedback. I would also like to thank the jury for reading the text. Finally, my sincere gratitude goes also to my family, to my friends and especially to Ward for their support.

Hanne Peeters

Contents

Abstract

The amount of embedded devices increases each year. Consequently, there is the need for large-scale analysis of their firmware images. To support this the FACT framework was developed, that currently consist mainly out of plugins for static analysis. Therefore, we will develop two dynamic analysis plugins for the FACT framework in this thesis. Here, scalability, both in terms of execution time and in terms of analysis of firmware images at the same time, is an important factor.

The first plugin, of which the functionality is divided over two actual plugins `CVE exploits` and `CVE firmadyne`, performs dynamic CVE verification. This contributes to reducing the amount of manual analysis needed to confirm a possible vulnerability for a specific firmware image. For this plugin, we observed that the number of Metasploit exploits available for the identified CVEs for a firmware image is limited. Next, we saw that it is difficult to execute the exploits in an automated way as required settings are often not set, which means that the exploit can not be tested. Further, the plugin does not work for privilege escalation exploits, which is a limitation. Finally, we found that the support currently provided by FACT framework is not sufficient to implement the interaction between the two plugins in an scalable way.

The second plugin `fuzz tracing` enables the user to compare the code coverage of different fuzzer configurations for the embedded web interface of a firmware image. As full program coverage is often the goal of fuzzing, this plugin enables the user to determine which fuzzer configuration has the highest code coverage or results in the most unique code coverage. Therefore, the value of this plugin is that the ratio between the number of executed fuzzers and the number of detected vulnerabilities can be optimised. The scalability of this plugin, in terms of needed execution time, depends on two factors: the firmware image under test and the number of fuzzer configurations executed. We observed that the execution time varies greatly between firmware images and even within vendors. This plugin scales well for the amount of fuzzer configurations that can be tested while the execution time remains reasonable. A limitation of the plugin is the lack of context for the obtained code coverage per fuzzer configuration.

The two plugins make use of Firmadyne to obtain software-based full system emulation. This limits the firmware images that can be tested to network-connected Linux-based embedded firmware images for which the root file system is present. The supported CPU architectures are little-endian ARM, little-endian MIPS and big-endian MIPS.

List of Figures

List of Tables

List of Abbreviations

Abbreviations

CPE	Common Platform Enumeration
CVE	Common Vulnerabilities and Exposures
FACT	Firmware Analysis and Comparison Tool
NTP	Network Time Protocol
NVRAM	Non-volatile random-access memory
SMB	Server Message Block
TDS	Tabular Data Stream
QEMU	Quick EMUlator

Chapter 1

Introduction

1.1 Embedded systems

Nowadays, embedded systems are ubiquitous. Their applications range from washing machines, medical implants and routers to car elements and traffic lights [9, 12]. Further, more and more embedded devices are connected to the internet, thereby forming the Internet of Things, which is estimated to consist of in the order of 20 billion devices by 2020 [20].

The large number of devices connected to the internet combined with the presence of vulnerabilities, makes attacks possible such as the Mirai botnet in 2016. The Mirai botnet was used by attackers to launch distributed denial-of-service (DDoS) attacks at various high-profile targets. Retrospective research pointed out that security cameras, Digital Video Recorders (DVRs) and consumer routers accounted for the majority of devices present in the Mirai botnet [2]. This shows the importance of developing secure embedded devices and finding their vulnerabilities early on so that they can be patched.

1.2 Problem statement

In this section we start with a discussion of the concept of *firmware* in the context of embedded systems. Firmware is defined by Costin *et al.* as being 'the software that is embedded in a hardware device' [9]. It resides on non-volatile memory and can be broken down into three principal components: the kernel, the bootloader(s) and the file system(s) [21, 24]. Together, the kernel and the file system(s) make up the entire operating system of an embedded device [36]. Firmware images can also be monolithic, in which case there is no clear distinction between the components [8].

The kernel can be viewed as being the core of an operating system. It mainly provides services such as: controlling and managing the execution order of processes, memory usage and communication with peripheral devices and networks [24].

The bootloader takes care of the initialisation of the various hardware components (e.g. RAM, flash storage, I/O) of the embedded system. It also ensures that the kernel is loaded into memory for execution. This process can be carried out in one,

two or three stages, thereby using a corresponding number of bootloaders. Each of these bootloaders has its specific functionality and loads the bootloader of the next stage [36].

The file system of an embedded device usually includes a root file system with possibly other file systems mounted on it [8].

Firmware images are often customised for a specific vendor, device, chipset or geographic retail location [8, 11, 12]. Vendors can distribute their firmware either as the full firmware image (containing the kernel, one or multiple bootloaders and one or multiple file systems) or as an update (missing the kernel and/or the root file system) [9, 10, 36]. Further, firmware can be acquired under one of two forms: as source code (before compilation) or in a binary format. The latter form is more often available e.g. due to proprietary firmware [22, 34].

As the example attack given in the previous section points out, it is important to be able to identify weaknesses and vulnerabilities in the firmware of embedded devices. Next, we give a general overview of the principal methods used to identify vulnerabilities in already released firmware images.

For security-focused analysis of firmware images, four kinds of methods are used: static analysis, dynamic analysis, symbolic analysis and fuzzing [22, 25]. Static analysis methods attempt to find vulnerabilities by inspecting the code, whereas dynamic analysis, symbolic analysis and fuzzing execute the firmware on the real embedded device or on an emulator [30, 37]. The main difference between dynamic and symbolic analysis, lies in the support of symbolic program memory by the latter [22]. Fuzzing attempts to discover unknown vulnerabilities in software by executing the target program with random input data and monitoring the program behaviour for anomalies [26, 30, 39].

The main challenge in the context of dynamic analysis and fuzzing on firmware of embedded devices, is designing an emulator that makes running the firmware and therefore dynamic analysis possible. This is a challenge because of the great variety of hardware environments that are used in embedded systems [38]. Usually, emulators that are designed for embedded devices need access to the actual hardware, which means that it does not scale very well [38]. The solution to this is software-based full system emulation, which was applied in earlier research for Linux-based embedded firmware [8, 10, 35]. The choice of the researchers to focus the emulation effort on embedded Linux is mainly motivated by its observed dominating occurrence in the obtained datasets in both researches, which was also pointed out by a previous large-scale analysis of embedded devices [8, 9, 10].

As mentioned earlier, the increase of the number of embedded devices employed nowadays requires an evolution towards automated large-scale analysis methods for firmware images. The Firmware Analysis and Comparison Tool (FACT)[1] is an extendable open-source framework that facilitates carrying out firmware analysis, by storing uploaded firmware images and their analysis results in a local database for later usage. The different analysis tools, currently principally static analysis methods, are provided as plugins, which makes it straightforward to add additional tools. An

[1] https://fkie-cad.github.io/FACT_core/

example of static analysis performed with an analysis plugin is the identification of the different software components present in the firmware. Currently, only one analysis plugin provides dynamic analysis for a firmware image within the FACT framework.

1.3 Research questions

In this master thesis, we will focus on providing dynamic analysis plugins for the FACT framework. We will provide plugins for the following functionality:

- using directed dynamic analysis on a firmware image to verify earlier found CVEs by static analysis
- comparing code coverage obtained by different fuzzer configurations for a firmware image

1.4 Thesis structure

The remainder of this thesis is structured as follows. In chapter 2, we provide a discussion of the existing FACT framework and the tools that we will use to perform emulation and to carry out dynamic analysis. Next, in chapter 3, we treat the data collection of firmware images that will be used to test the plugins on. The two developed plugins are discussed in chapter 4 and chapter 5. The discussion is provided in chapter 6 and chapter 7 contains the conclusion.

Chapter 2

FACT framework and existing tools

In this chapter, we will discuss the context for both plugins. First, in section 2.1 the FACT framework, for which the plugins are developed will be described in detail. Furthermore, we treat two open-source tools that will be used by both plugins. Firmadyne, which will be used to emulate the firmware, is discussed in section 2.2 and Metasploit, which will be used to carry out the dynamic analysis, is discussed in section 2.3.

2.1 FACT framework

The Firmware Analysis and Comparison Tool (FACT)[1] is an open-source extendable analysis framework for binary firmware images developed by Fraunhofer FKIE[2]. Its goal is to automate a large part of the process of firmware analysis, which is focused on the security of the firmware images. Consequently, this makes large-scale analysis of firmware images feasible and manageable. The framework is both accessible through a web user interface as with a REST API. It integrates many unpacking tools and various plugins for static analysis, which allows the user to adapt the performed analysis to the firmware image in question. At the moment, the FACT framework does provide one dynamic analysis plugin, `firmadyne`, which is discussed in detail in the next section. There are three categories of plugins within the FACT framework: unpacker plugins, analysis plugins and compare plugins [17]. The compare plugins make it possible to compare two firmware images, for example a vulnerable firmware image and its update. Furthermore, all uploaded firmware images are kept in a local database, which can be queried. The framework provides also statistics about earlier analysed firmwares [13].

The following procedure is used to perform an analysis for a firmware image within the FACT framework. Via the web user interface, the user can upload the firmware image and its corresponding metadata. On this page, the wanted analyses

[1] https://github.com/fkie-cad/FACT_core
[2] https://www.fkie.fraunhofer.de/

can be checked off. This is displayed in Figure 2.1. Next, the specified analyses for the firmware image are carried out and the page that contains the analysis results for the firmware image is updated. This page is shown in Figure 2.2. Here, the obtained analysis results can be seen in general for the whole firmware image by clicking on the wanted results in the 'Analysis Results' column and per file by expanding the file tree and clicking on the wanted file.

Plugins for the FACT framework are written in Python. All plugins within the FACT framework are open-source and users can write their own plugins. The development guide mentions that an analysis plugin should follow the code structure shown in Figure 2.3[3] [16]. Here, we see that only the `code` folder, which contains the actual plugin, is required. The other folders are optional. In the `install.sh` file the necessary installations for the plugin can be provided. These will be carried out during the execution of the FACT installation script. The `internal` folder is meant to contain additional code or sources for the plugin. In the `test` folder tests can be provided to be executed with pytest. The `view` folder contains the customised view for the web user interface that can be provided [16].

New plugins are automatically detected if they are provided in the right place in the structure of the FACT framework. Next, by default an analysis is performed on each file that the firmware image contains, but there is also the option to perform it only on the outer container. A plugin `analysis A` can be dependent on another plugin `analysis B`, which means that for each file the plugin `analysis B` is executed first. Through this dependency, plugin `analysis A` can use the results of the analysis carried out by plugin `analysis B`, which are by default passed per file. Here, it is not possible for plugin `analysis B` to access the overall result of plugin `analysis A` [16].

The FACT framework already provides a plugin for software-based full system emulation of firmware images, which is discussed next.

2.2 Firmadyne

The `firmadyne` plugin[4] integrates Firmadyne within the FACT framework. Firmadyne[5] was developed by Chen *et al.* in 2016 and provides software-based full system emulation for network-connected Linux-based embedded firmware images of which the root file system is present. In order to achieve this kind of emulation, the researchers replaced the existing kernel with one modified by them (and compatible with the original CPU architecture) to match the emulation environment. Further, they also provided software emulation of the NVRAM hardware peripheral, as half of their dataset accessed it. Afterwards, the modified kernel together with the extracted, original file systems are booted by using QEMU full system emulator. Replacing the original kernel means that only vulnerabilities affecting the file system of the firmware can be identified in this way [8, 37]. When the emulation is successful,

[3] `__init__.py` files are omitted for better readability
[4] https://github.com/fkie-cad/FACT_firmadyne_analysis_plugin
[5] https://github.com/firmadyne/firmadyne

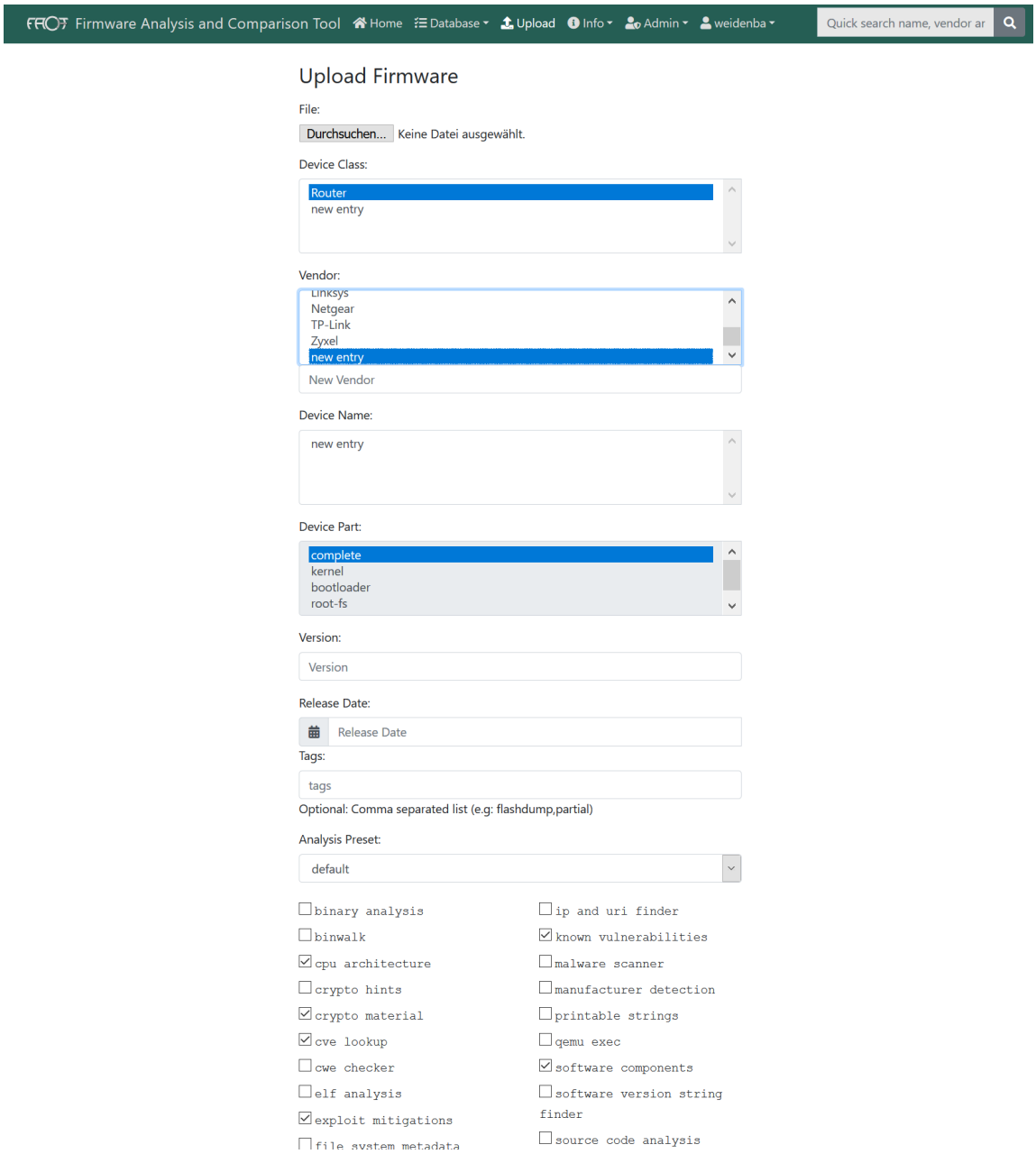

FIGURE 2.1: FACT upload page [18].

Firmadyne provides the possibility to carry out three basic automated analyses on the firmware image: an analysis for publicly available web pages, an analysis that dumps all unauthenticated SNMP information and an analysis that tries to exploit possible vulnerabilities. For the last one, the researchers made a default selection of 60 Metasploit exploits well suited for embedded devices and provided 14 self-written

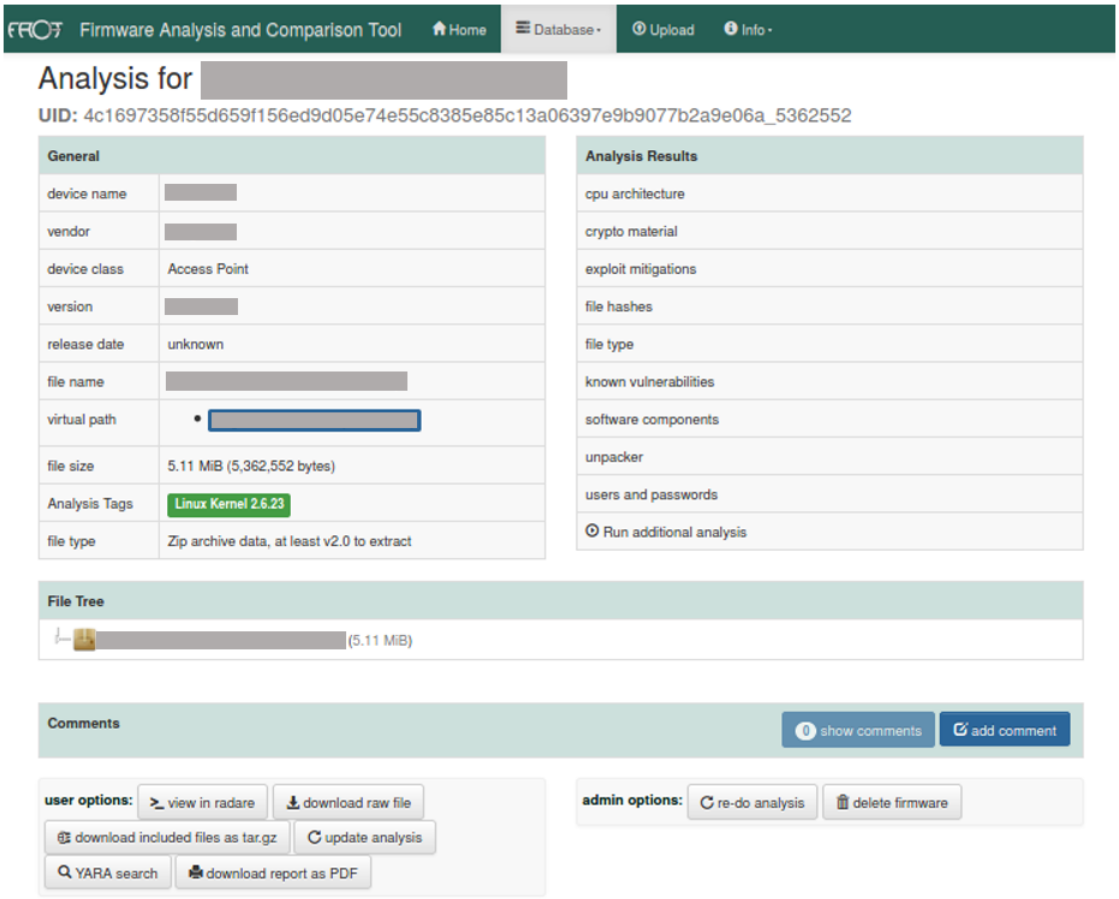

Figure 2.2: FACT analysis results page for firmware image.

```
analysis plugin
├── install.sh [OPTIONAL]
├── code
│   └── PLUGIN_NAME.py
├── internal [OPTIONAL]
│   └── ADDITIONAL_SOURCES_OR_CODE
├── test [OPTIONAL]
│   ├── test_PLUGIN_NAME.py
│   └── data
│       └── SOME_DATA_FILES_TO_TEST
└── view
    └── PLUGIN_NAME.html [OPTIONAL]
```

Figure 2.3: Code structure for an analysis plugin [16].

exploits for vulnerabilities they discovered. These exploits are executed within the Metasploit Framework [8].

The code structure for the `firmadyne` plugin is shown in Figure 2.4. This is a recursive analysis plugin and works as follows. As it executes its analysis in a recursive way, each file is processed by the plugin. To be able to emulate a firmware image with Firmadyne, the root file system needs to be present. Therefore, the plugin code waits for a file of which the content represents a file system to execute the code of Firmadyne. Once the analysis is started for the file system, the code located in the `internal` folder is executed, which automates the different steps that need to be performed for Firmadyne.

```
firmadyne
├── install.sh
├── bin
│   ├── Firmadyne
│   └── Metasploit Framework
├── code
│   └── firmadyne.py
├── internal
│   ├── steps
│   │   ├── prepare.py
│   │   ├── emulation.py
│   │   └── analysis.py
│   └── firmadyne_wrapper.py
├── test
│   ├── test_firmadyne.py
│   └── data
└── view
    └── firmadyne.html
```

FIGURE 2.4: Code structure for the `firmadyne` plugin.

2.3 Metasploit

Metasploit is an open-source Ruby-based framework used for penetration testing. At a high level, the following components can be distinguished for the Metasploit Framework: auxiliaries, payloads, exploits, encoders and post-exploitation activities [32]. In the context of this thesis, we are mostly interested in the exploits (for the plugin of chapter 4) and the auxiliaries (for plugin of chapter 5) components. The structure of the exploits component is displayed in Figure 2.5. Here, we see that the actual exploits are categorised in different categories corresponding to the target they are applicable for [32].

Aix	Android	Apple_ios
Bsdi	Dialup	Firefox
Freebsd	Hpux	Irix
Linux	Mainframe	Multi
Netware	Osx	Solaris
Unix	Windows	

Figure 2.5: Categories of exploits in the Metasploit Framework [32].

Next, we have the auxiliaries component, of which the structure is shown in Figure 2.6. For this component, we are in particular interested in the provided fuzzers. Currently, Metasploit provides a number of protocol-based fuzzers for the following protocols: DNS, FTP, HTTP, NTP, SMB, SMTP, SSH and TDS [1].

Admin	Analyze	Bnat
Client	Crawler	Docx
Dos	Fileformat	Fuzzers
Gather	Parser	Pdf
Scanner	Server	Sniffer
Spoof	Sqli	Voip
Vsploit		

Figure 2.6: Categories of auxiliaries in the Metasploit Framework [32].

Chapter 3

Collection of firmware images

In this chapter, we discuss the collection of firmware images that will be used to evaluate the developed plugins in the next chapters (chapter 4 and chapter 5) on. For this purpose, it should be possible to successfully emulate the firmware image with Firmadyne, as this will be the emulation tool used in both plugins.

This chapter is structured as follows. First, in section 3.1, the collection of the firmware images is discussed. Next, in section 3.2, we carry out the different steps in the Firmadyne emulation process to determine which firmware images can be used to evaluate the plugins. In section 3.3, we will see some characteristics of the resulting dataset. Finally, section 3.4 contains the conclusion for this chapter.

3.1 Sources

Due to the lack of publicly available databases for firmware images, it was necessary to perform our own collection of firmware images. For this task, we used the scrapers distributed together with Firmadyne [8]. In order to test and evaluate Firmadyne, Chen *et al.* collected 23.035 firmware images developed by 42 vendors. This collection was done either manually or in an automated way by using their scrapers. Of these 42 vendors, only 14 vendors had at least one of their firmware images reach the 'network reachable' phase. This means that their embedded web interface became reachable over the network and that the emulation succeeded. Of the 23.035 collected firmware images, 1971 did emulate successfully [8].

As we will use the Firmadyne tool for the emulation of the firmware images in both plugins, we therefore focused on collecting firmware images of those 14 vendors with at least one successful emulation with Firmadyne, being: Belkin, CenturyLink, D-Link, Huawei, Linksys, Netgear, On Networks, OpenWrt, Tenda, Tenvis, Tomato by Shibby, TP-Link, TRENDnet and ZyXEL. For 12 of these vendors, we were able to download at least one firmware image. We used Firmadyne's scrapers wherever possible, but as some of the websites changed, not all of them worked. Further collection was performed manually. Here, our aim was to collect firmware images from as many different hardware products as possible, to maximise the number of firmware images found that could possibly be emulated with Firmadyne. For On

Networks, we could not find a download site that contained firmware images for the products provided by them. Tenvis on the other hand deleted the links for firmware downloads on their site. Furthermore, we encountered vendors that required an account for downloading some of their firmware images. This was for example the case with Huawei, which explains why we could download only two firmware images for this vendor. A side note is that OpenWrt and Tomato by Shibby do provide firmware for devices of other hardware vendors [8]. Their third-party firmware images are Linux-based and thus in theory suitable for emulation with Firmadyne [6, 31].

3.2 Firmware images suitable for emulation with Firmadyne

In total, we collected 2699 firmware images, of which the distribution per vendor is displayed in Table 3.1.

Table 3.1: The stages reached for each of the collected firmware images per vendor.

	stage reached			
vendor	downloaded	Linux file system	supported CPU architecture	emulated
Belkin	65	0	0	0
CenturyLink	21	16	16	0
D-Link	1051	212	184	45
Huawei	2	0	0	0
Linksys	111	58	58	7
Netgear	47	25	24	1
OpenWrt	50	41	38	1
Tenda	294	130	129	0
Tomato by Shibby	674	672	671	17
TP-Link	22	17	17	2
TRENDnet	52	24	24	5
ZyXEL	310	113	113	9
Total	2699	1308	1274	87

Next, we used the Firmadyne extractor to determine if the firmware images contain a Linux-based file system. This was necessary because other file systems are not supported by Firmadyne. The extractor script of Firmadyne, attempts to extract the firmware image and outputs the found file system(s). The different file systems and their frequency, found for both firmware images of which the extraction failed and for firmware images of which the extraction was successful, are displayed in Table 3.2. Here, we see that most firmware images in our dataset contain a Squashfs

root file system. For the firmware images for which no root file system could be detected, the firmware image could either not be unpacked by Firmadyne or it is not a complete firmware image (e.g. contains only release notes, is a partial firmware update) [10]. This leaves us with 1308 candidates for successful emulation.

TABLE 3.2: Detected root file system for both not extracted and successfully extracted firmware images with Firmadyne.

file system	not extracted	extracted
Cramfs	20	2
EXT	0	3
JFFS2	14	78
Linux	0	102
Minix	5	0
MPFS	6	0
PFS	3	0
romfs	2	0
Squashfs	44	1115
TROC	8	1
UBIFS	2	4
YAFFS	15	3
no file system	1272	-
Total	1391	1308

After that, the Firmadyne tool was used to determine the CPU architecture that the firmware was designed to run on. As emulation by Firmadyne supports only little-endian ARM, little-endian MIPS and big-endian MIPS, we omitted the firmware images with other CPU architectures in Table 3.1. For 10 firmware images, we could not determine the CPU architecture due to an error and therefore these are also omitted from the table. A full overview of all found CPU architectures for the firmware images with a Linux file system per vendor can be found in Table 3.3. We see that in our dataset the majority of the firmware images with a Linux-based file system is made to be run on one of the supported CPU architectures by Firmadyne.

The next step is to check if the firmware image can be actually emulated with Firmadyne. We count a firmware image as successfully emulated, if the embedded web interface of a firmware image becomes reachable over the network. During this stage, a '60-second' learning phase of Firmadyne is executed in order to infer the expected network interfaces [8]. When a network interface was found, we continued with the actual emulation of the firmware image. For this final stage, we used a time limit of 300 seconds to determine if the embedded web interface is reachable on the host or not. Thus, there are three possible outcomes: a network interface could not be inferred, the network interface was not reachable on the host within 300 seconds or the emulation was successful. Finally, the firmware images that emulated

Table 3.3: Found CPU architectures for firmware images containing a Linux file system per vendor.

vendor	CPU architecture				
	little-endian ARM	little-endian MIPS	big-endian MIPS	big-endian MIPS 64-bit	big-endian PowerPC
CenturyLink	1	1	14	0	0
D-Link	89	6	89	16	5
Linksys	20	32	6	0	0
Netgear	15	1	8	0	1
OpenWrt	3	13	22	0	2
Tenda	27	72	30	0	0
Tomato	117	554	0	0	0
TP-Link	6	2	9	0	0
TRENDnet	2	4	18	0	0
ZyXEL	25	10	78	0	0
Total	305	695	274	16	8

successfully are the ones that will be used for testing the plugins. In our case, we end up with 87 firmware images. The exact division of those firmware images per vendor, is displayed in the last column of Table 3.1. For 1070 firmware images, there could no network interface be inferred. For 117 firmware images, the embedded web interface became not reachable on the host within the time limit of 300 seconds.

3.3 Resulting firmware images

In this section, we will look at characteristics of the resulting dataset that may help to interpret the results in the next chapters in a better way. The different categories of embedded devices for the resulting firmware images can be found in Table 3.4. We can see that our dataset mainly consists out of access points and routers. When we map the firmware images to their hardware products (not differentiating between different hardware versions), we have 40 unique products left.

If we look at the release year of the firmware images, we see that it ranges from 2007 until 2019. This means that Firmadyne is able to emulate firmware released after its active development ended in 2016, which is indicated on their Github page [7].

3.4 Conclusion

In this chapter, we saw that, despite our effort to target the vendors that had firmware images that successfully emulated with Firmadyne [8], only a fraction of

TABLE 3.4: Category of devices associated with firmware images and its frequency.

device category	frequency
Access Point	46
Router	30
Gateway	6
Gigabit Passive Optical Networking	2
Range Extender	1
Mesh Extender	1
Wireless Ethernet Adapter	1
Total	87

the collected firmware images could be emulated successfully. As both developed plugins will be using Firmadyne for the emulation of a specific firmware image, this is a first important limitation on the usability of the plugins. Further, we found that the obtained dataset consists mainly out of access points and routers.

Chapter 4

Plugin I. Dynamic CVE verification

In this chapter, we describe the first plugins that we developed for the FACT framework, that aim to automate the verification of earlier found Common Vulnerabilities and Exposures (CVEs). For this purpose, we use the already existing `CVE lookup` plugin of the FACT framework to identify all the present CVEs for a firmware image. The rest of the functionality is divided over two plugins: `CVE exploits` and `CVE firmadyne`. The first one performs the preparation for the dynamic CVE verification (i.e. searches for exploits for the identified CVEs). The second one, which is an adaptation of the `firmadyne` plugin, uses Firmadyne for software-based full system emulation. This enables the plugin to carry out the dynamic verification of the CVEs, for which an exploit was found, on the firmware image. The exploits used in the plugins, are those provided by the Metasploit Framework. The goal of the two plugins together is to reduce the need for manual analysis in order to confirm the existence of a CVE for the firmware image.

The chapter is structured as follows. First, in section 4.1 we explain what CVEs are and we describe the problem that our plugins try to solve. Next, in section 4.2 we discuss the `CVE lookup` plugin of the FACT framework that performs static analysis on a given firmware image to identify CVEs for it. Further, in section 4.3 we treat the plugins that we provided for the FACT framework: the `CVE exploits` plugin and the `CVE firmadyne` plugin. The evaluation of both these plugins in terms of correctness and usability is carried out in section 4.4. Finally, section 4.5 contains the conclusion for this chapter.

4.1 Problem statement

To check for known vulnerabilities in software, hardware devices or operating systems, the Common Vulnerabilities and Exposures[1] (CVE) and Common Platform Enumeration[2] (CPE) standards are often used [33].

[1] https://cve.mitre.org/
[2] https://nvd.nist.gov/products/cpe

As stated on its website, CVE is a list filled with entries of public disclosed cybersecurity vulnerabilities and exposures. Each entry contains a standardized common identification number (referred to as CVE ID number), a description for the vulnerability or exposure and at least one public reference (e.g. vulnerability reports) [28]. These CVE ID numbers are of the form of `CVE-<year>-<sequence number>` [29]. In the rest of this thesis, we will refer to these CVE entries as CVEs.

Further, to make it possible to identify software, hardware devices and operating systems affected by a CVE, each CVE entry is associated with a list of products, described according to the CPE standard. This way, it is possible to find vulnerabilities for specific products by searching for their CPE identifier [33].

In general, static analysis yields often false positives, which need to be identified through manual analysis [8, 10]. In this context and with the growing number of embedded devices kept in mind, it is therefore interesting to apply dynamic analysis directed towards the found vulnerabilities by the used static analysis tool. This method was suggested and applied by Costin *et al.* in 2016 in their large-scale analysis of embedded web interfaces of embedded devices [10]. Methods like this are interesting because they provide a way to focus manual analysis on confirmed vulnerabilities.

In this chapter, we will therefore develop a plugin that can confirm possible CVEs through dynamic analysis. The two main criteria for the plugin are the following. First, the plugin has to work within the FACT framework. Second, it has to determine and output which CVEs were confirmed by dynamic analysis for a given firmware image.

4.2 CVE identification by static analysis

The FACT framework does already contain an analysis plugin that matches a given firmware image with the present CVEs in that firmware image. This plugin, `CVE lookup`, looks for CVEs by using the results of another FACT plugin, `software components`, which identifies the different software components (e.g. kernel version) that are present in the firmware image. After this, each identified software component is looked up in the CPE database[3] to obtain their unique CPE ID. Next, these CPE IDs are used to find the corresponding CVEs.

This analysis is performed in a recursive way, which means that as soon as the `software components` plugin has processed a file, the CVEs for the file can be searched. Therefore, it is possible to consult the results for this plugin per file, although a summary overview of all found CVEs is also provided. The results of this plugin contain a list with all the found CVEs (i.e. to generate the summary overview), which we will be using in the `CVE exploits` plugin. We used version 0.0.3 of the `CVE lookup` plugin in the development of the other plugins.

[3] https://nvd.nist.gov/products/cpe

4.3 CVE verification by dynamic analysis

After we have obtained the CVEs for a firmware image, the next steps, to be able to confirm a possible CVE for a firmware image, are the following. First, we need to find the exploits available for each identified CVE. This will be implemented by the `CVE exploits` plugin and is treated in subsection 4.3.1. After that, we need to emulate the firmware image, perform the dynamic analysis by executing the found exploit and interpret the results. This will be implemented by the `CVE firmadyne` plugin and is discussed in subsection 4.3.2.

A visualisation of the interaction of the different plugins and their dependencies is provided in Figure 4.1.

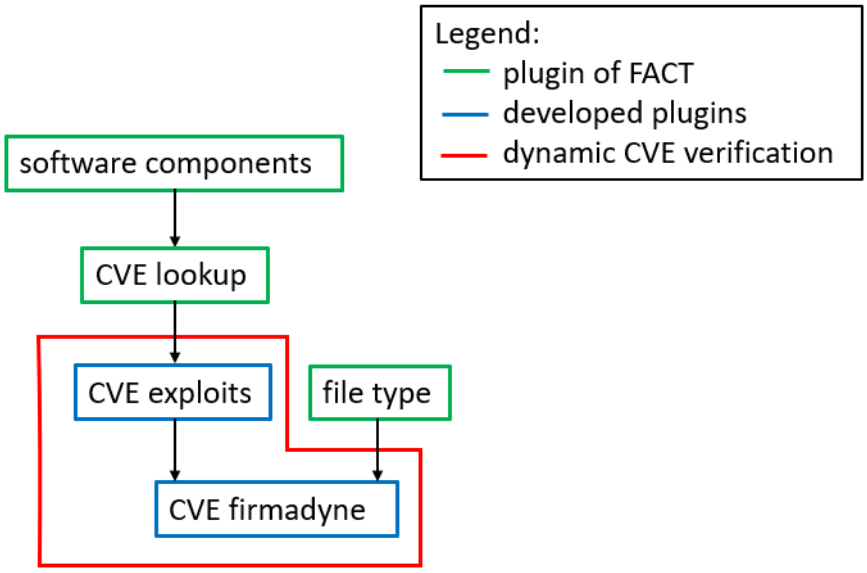

FIGURE 4.1: Visualisation of the present dependencies for the dynamic CVE verification functionality.

4.3.1 `CVE exploits` plugin

First, we need a framework that contains exploits for the identified CVEs and supports their execution. For these tasks, we chose to use Metasploit. This choice was made for the following reasons. First of all, it can be easily used in combination with Firmadyne as this was done in the original research [8]. It is thus already implemented in the FACT framework together with Firmadyne. Further, it is open-source as we mentioned earlier. Lastly, their Vulnerability and Exploit database[4] allows to search on CVE ID number to find a corresponding Metasploit exploit. A limitation of the choice for Metasploit is that exploits will not be found for each

[4] https://www.rapid7.com/db/

identified CVE as the exploit database currently contains only 2005 exploits[5] while the 100 000th CVE entry was published in 2018 [3].

Our implemented plugin, which we call `CVE exploits`, will be an analysis plugin and can therefore use the results of other plugins to perform its analysis [15]. To be able to use the found CVEs by the `CVE lookup` plugin, we need to make our plugin dependent on the `CVE lookup` plugin. This means that for each file of the firmware image, the `CVE lookup` analysis is performed before carrying out our `CVE exploits` analysis. As the `CVE lookup` plugin works in a recursive way, our plugin has to perform its analysis in the same recursive way because of the established dependency. This dependency is represented by the arrow between the two plugins in Figure 4.1.

The plugin uses the following scheme to find the available Metasploit exploits for the identified CVEs. For each CVE, it determines if there is a Metasploit exploit available by looking it up in the Vulnerability and Exploit Database. If there is one available, the path in the Metasploit Framework for the exploit is extracted. The results of this plugin contain a mapping of CVEs to their corresponding Metasploit exploits (if available). The summary contains all the found Metasploit exploits. Further, the plugin provides the exploits file for the found exploits to be used in the `CVE firmadyne` plugin. However, to be able to generate this file in the right way, multithreading needs to be disabled for the plugin. This could increase the needed execution time. We made the choice to generate the exploits file by the `CVE exploits` plugin. This because the recursive way in which all of the plugins work, does not allow the `CVE firmadyne` plugin to retrieve all the found exploits before starting the emulation of the firmware image.

We implemented this functionality as a separate plugin as it allows the user to get an overview of all available Metasploit exploits for the firmware image before any emulation is executed. This can be useful to avoid a dynamic CVE verification in the case that no Metasploit exploits are available. Further, it can give an insight for which CVEs the user needs to look for exploits implemented by other frameworks or if the user needs to write its own exploits.

4.3.2 `CVE firmadyne` plugin

Next, we need to be able to emulate the firmware image and execute the exploits corresponding to the found CVEs. Therefore, this plugin, which is called `CVE firmadyne` (an adaptation of the `firmadyne` plugin[6] provided by the FACT framework), is going to be dependent on the previous discussed `CVE exploits` plugin as shown in Figure 4.1. This way, the exploits file used by Firmadyne to execute Metasploit exploits is adapted to contain only the found Metasploit exploits (i.e. instead of the default selection of Metasploit exploits made by the researchers of Firmadyne [8]).

The main challenge in this context is the file-based processing mechanism, which the FACT framework uses for recursive analysis plugins. If the first plugin in a series of dependencies performs its analysis in a recursive way, the other plugins need to do it as well in a recursive manner. As the file type, for which the `CVE Firmadyne`

[5] Metasploit `v5.0.87-dev-`

[6] https://github.com/fkie-cad/FACT_firmadyne_analysis_plugin

plugin looks to start the analysis (i.e. file system)[7], is skipped by both the `CVE lookup` and `CVE exploits` analyses, this file will be one of the first files that is ready to be processed by `CVE firmadyne`. This means that Firmadyne will start the emulation before all the CVEs and exploits are looked up for a firmware image and thus with none or an incomplete collection of exploits to be tested. Currently, the FACT framework has no easy way to detect if an analysis is terminated for all files of a given firmware image. Therefore, we tried to implement the suggestion that was made on their Github: to check if the analysis was done for all included files of the firmware image [19]. We used the results of the `unpacker` tool to determine the files present in the firmware image and then checked for all these files if the `CVE exploits` analysis was already carried out. However, this does not solve the problem as the FACT scheduler does not wait for the `unpacker` to finish before other scheduled analyses are carried out. Neither is there another way to detect if the `unpacker` has finished. This is a problem as the number of files that a firmware image contains is not known beforehand. Therefore, we queried two times for the results of the `unpacker` by using the REST API with a time interval between the two queries. The chosen time interval should be large enough to allow the `unpacker` to process additional files. In this case, if the results of the two queries are equal, the `unpacker` has finished. We used the REST API for this task because it returns the analysis results for the whole firmware image (when queried with the right identifier) and can display for a chosen file the analyses that were already carried out.

When the process of searching Metasploit exploits for the found CVEs is finished, the emulation of the firmware image can be carried out. As already mentioned before, instead of using the default exploits file, we will use a customised one that only contains the found exploits. This exploits file was created by the `CVE exploits` plugin. After this, the Metasploit logs are interpreted. For executing the exploits with Metasploit, we have the limitation that it may be the case that a required setting for an exploit is not set (i.e. a setting that has no default value such as LHOST). Currently, we only set the RHOST and RHOSTS settings and therefore it can be the case that the exploit can not be executed due to a missing setting. Further, we provided a configuration file for Metasploit to which configurations (i.e. the name of the setting and the value for the setting) can be added to improve the number of exploits that can be executed.

The results of this plugin consist of the successfulness of the different steps of the emulation process, the same way as is done for the `firmadyne` plugin. Next, the results for the Metasploit exploits are displayed, which consist out of the CVE, the tested exploit, if the exploit was executed successfully on the firmware image, the parameters which were missing and the corresponding Metasploit log file. It is useful to include this log in the results to be able to know if the exploit could not be executed due to a missing setting. A screenshot of the results of the plugin can be found in Figure A.1 in Appendix A.

[7] functionality originating from `firmadyne` plugin

4.4 Evaluation

In this section, we will evaluate both the correctness and the usability of the `CVE exploits` and `CVE firmadyne` plugins separately and the correctness of their interaction.

4.4.1 `CVE exploits` plugin

An overview of the code structure for the `CVE exploits` plugin is given in Figure 4.2. The `internal` folder contains the template for the exploits file.

```
cve exploits
├── code
│   └── cve_exploits.py
├── internal
│   └── template_runExploits.py
├── test
│   └── test_cve_exploits.py
└── view
    └── cve_exploits.html
```

Figure 4.2: Code structure for the `CVE exploits` plugin.

Correctness

First, we look at the correctness of the `CVE exploits` plugin. To evaluate this, we implemented a number of test cases, which are displayed in Appendix C in subsection C.1.1. With these test cases we obtained a test coverage of 99% for the code of our plugin.

However, when testing the plugin in the FACT framework, there occurs a timeout error for the files that do contain exploits. This means that there are no results registered for this file and the results for this file can not be queried. We obtain only results for the files with CVEs that do not match any Metasploit exploit. We were not able to solve this. This hinders the usability of the `CVE exploits` plugin itself, but as the exploits file is generated correctly the overall dynamic CVE verification functionality is not affected.

Usability

To assess the usability of the plugin, we look at its performance in terms of time-consumption, the range of firmware images on which it can be used and the usability of its results when tested on the collected firmware images in chapter 3.

We tested the execution time of the plugin with and without multithreading for the same firmware image to get an idea of the impact of disabling this option on the execution time. With the multithreading option of the FACT framework enabled for

this plugin, the execution time is 59 minutes. Without the multithreading option of the FACT framework, the execution time is 60 minutes. This is a negligible increase in execution time, which could be due to the fact that the `CVE lookup` plugin is still executed with multithreading and that the execution time per file of the `CVE exploits` plugin makes no significant contribution to the total execution time. The needed execution time for this plugin depends on the size of the firmware image that is analysed. The execution time was tested with a firmware image of size 8.5 MB.

As this part of the functionality is rather a preparation for the verification phase, which takes place in the `CVE firmadyne` plugin, the plugin is not limited to firmware images that can be emulated with Firmadyne and can be used on all firmware images that can be unpacked by the FACT framework.

Next, we tested the plugin on the collected firmware images that emulated successfully with Firmadyne. First, we discuss the amount of CVEs identified by the `CVE lookup` plugin. The lowest amount of CVEs found for a firmware image was 12. The highest amount of CVEs found for a firmware image was 702. The median of the number of found CVEs lies on 405. For these results obtained by the `CVE lookup` plugin, we need to note that they display stability issues in identifying the right CVEs due to inconsistent use of keywords used to look up the exact software component (e.g. linux kernel and kernel).

For these CVEs, we found the following amount of Metasploit exploits. For 33 firmware images there were no Metasploit exploits available for the identified CVEs. For 28 firmware images there is one Metasploit exploit found and for 24 firmware images we found two Metasploit exploits. Further, one firmware image had three Metasploit exploits and one firmware image had four Metasploit exploits. For the whole dataset, we obtained six unique Metasploit exploits.

Here, we see that only a fraction of the identified CVEs has an available Metasploit exploit. This limits the usability of the `CVE firmadyne` plugin and of the whole dynamic CVE verification process.

4.4.2 `CVE firmadyne` plugin

An overview of the code structure for the `CVE firmadyne` plugin can be found in Figure 4.3. Here, the code structure of the `firmadyne` plugin was maintained. The implementation of the functionality required changes in the `cve_firmadyne.py` code and in the `analysis.py` code.

Correctness

To evaluate the correctness of the `CVE firmadyne` plugin, we implemented a number of test cases to be executed with pytest. The correctness of the waiting mechanism, which is implemented in `cve_firmadyne.py` is discussed in subsection 4.4.3. Once the right exploits file is available, the rest of the plugin (i.e. the part implement in the `internal` folder) can be tested via the implemented tests. For the `cve_firmadyne.py`

[7] version numbers are omitted for readability

```
cve_firmadyne
├── install.sh
├── bin
│   ├── Firmadyne
│   └── Metasploit Framework
├── code
│   └── cve_firmadyne.py
├── internal
│   ├── metasploit_configuration.csv
│   ├── steps
│   │   ├── prepare.py
│   │   ├── emulation.py
│   │   └── analysis.py
│   └── firmadyne_wrapper.py
├── test
│   └── test_cve_firmadyne.py
└── view
    └── cve_firmadyne.html
```

Figure 4.3: Code structure for the `CVE firmadyne` plugin.

code, we obtained a test coverage of 43%. For the `analysis.py` code, we have currently a test coverage of 96%.

Usability

The execution time of this plugin (around 150 seconds) represents only a fraction of the whole execution time for the dynamic CVE verification process.

The `CVE firmadyne` plugin can only perform its analysis on firmware images that can be emulated with Firmadyne and therefore the application range of the whole dynamic CVE verification process is limited in terms of support for firmware images.

For 54 firmware images we found at least one Metasploit exploit. Of these, none were executed successfully and thus effective in exploiting the firmware image. For each of the six unique exploits, there was the same required setting missing (i.e. SESSION). All these exploits are privilege escalation exploits, for which another exploit needs to be executed first and the number of the opened session by this other exploit needs to be set as SESSION. The execution of an exploit to enable the execution of another exploit is not supported by our plugin.

4.4.3 Interaction between `CVE exploits` plugin and `CVE firmadyne` plugin

Here, we discuss the correctness of the interaction between the two plugins.

Correctness

For part of the plugin that implements the waiting mechanism, it is not possible to test it via test cases per plugin due to the fact that it needs the FACT database and the `CVE exploits` plugin to be running. Therefore, this interaction was tested manually. Although the waiting mechanism works, there is a more structural solution in the FACT framework needed that enables a recursive plugin to wait on all the results of another recursive plugin before carrying out its analysis. Further, it should be made possible to retrieve the ID of a firmware image in a more easy way starting from a file of it such that all analysis results for all files of the firmware image can be found by the plugin. This would eliminate the need for the `CVE exploits` plugin to generate the exploits file. This would make the processing of multiple firmware images at the same time possible.

4.5 Conclusion

In this chapter, we developed two plugins (`CVE exploits` and `CVE firmadyne`) that together dynamically verify possible CVEs for firmware images in the context of the FACT framework. This plugin helps in reducing the amount of manual analysis needed to confirm a CVE for a firmware image. The two criteria for this plugin are satisfied since it works in the FACT framework and the results report if a possible CVE is confirmed or not.

The advantage of this plugin is that a large part of the analysis is automated. The first limitation for this plugin is the time needed for the analysis as the execution of the `CVE firmadyne` plugin waits for the `CVE lookup` and `CVE exploits` plugins to terminate. Furthermore, the scalability of the plugin is limited due to the fact that the `CVE exploits` can not be executed for multiple firmware images at the same time. Disabling the multithreading option for the `CVE exploits` plugin in the FACT framework did have a negligible impact on the execution time as the `CVE lookup` plugin is still executed with multithreading.

Next, we have some limitations of the plugin that are related to the usability. First, there is a limited number of Metasploit exploits available for possible CVEs. Next, we observed that all executed exploits for the earlier collected firmware images missed the same required setting and could therefore not be tested. This setting could not be set in the configuration file for Metasploit because another exploit needs to be executed successfully to set it. Furthermore, there is the reduction in firmware images that can be tested due to the use of Firmadyne as emulator.

Finally, we needed to be creative in the implementation of the interaction between the two plugins as this functionality is currently not supported by the FACT framework. However, for this a structural solution should be provided.

Chapter 5

Plugin II. Comparing fuzzer-based code coverage

This chapter treats the second developed analysis plugin for the FACT framework. The aim of this plugin is to make it possible to compare the code coverage obtained by the use of different fuzzer configurations for embedded web interfaces of firmware images. These execution traces are gathered by emulating the firmware images with Firmadyne while replacing the original kernels with QEMU kernels for which tracing capabilities are enabled. The file system of the firmware images, which contains the embedded web interface if there is one available, remains untouched for this execution tracing.

The structure of this chapter is as follows. An introduction to fuzzing and the problem statement for the plugin is given in section 5.1. Next, section 5.2 discusses obtaining code coverage of an emulated firmware image. In section 5.3 the fuzzing of embedded web interfaces is treated. Further, in section 5.4 the analysis of the obtained execution traces is dealt with. The results of the plugin for one firmware image are discussed in section 5.5. The evaluation of the plugin in terms of correctness and usability is treated in section 5.6. Finally, section 5.7 contains the conclusion for this chapter.

5.1 Problem statement

Muench *et al.* provide the following definition for fuzzing:

> *"Fuzz-testing or fuzzing describes the process of automatically generating and sending malformed input to the software under test, while monitoring its behavior for anomalies [30]."*

In general, it is used to identify previously unknown vulnerabilities in the software under test and it does so in a effective, fast and scalable way [26, 30]. The goal with fuzzing is often to reach full program coverage, as this results in a higher probability of finding vulnerabilities [25, 27].

Fuzzing of embedded devices brings additional challenges with it compared to regular fuzzing. According to Muench *et al.* one of those challenges is instrumentation, which is more difficult to obtain in the case of embedded devices, because source code is rarely available and rewriting of binary code is time-intensive [30].

Our plugin does not solve the need for using instrumentation (or another solution) to detect faults resulting by fuzzing a firmware image. However, with the goal of full program coverage in the back of our minds, it can be valuable information to know which fuzzer configuration results in the highest code coverage or in the most unique code coverage (i.e. code not reached by other fuzzer configurations). With the term *fuzzer configuration*, we mean the entirety of the fuzzer module and the chosen values for its settings.

We will thus develop a plugin which makes it possible to map a fuzzer configuration to the corresponding obtained code coverage for a given firmware image. Further, will we limit us here to measuring reached code by fuzzing the embedded web interface of firmware images as it can be easily determined when the embedded web interface is ready for fuzzing. Another argument for this choice was made by Costin *et al.*, who pointed out the considerable role of embedded web interfaces as attack surface for embedded devices [10].

We have two main criteria for this plugin. First, it has to work within the FACT framework. Second, it has to determine and to output the code coverage obtained by executing different fuzzing configurations for a given firmware image. Thus, it needs to emulate the firmware image in order to be able to perform fuzzing on its embedded web interface. Next, it will be necessary to log execution traces to determine the corresponding code coverage for each fuzzer configuration.

We will tackle this problem step-by-step. First, we need to achieve emulation of the firmware image. As discussed before in the other chapters of this thesis, we opt to use Firmadyne for this task as it is already available as plugin in the FACT framework and as there is no hardware associated to the firmware required, which makes it scalable. Next, we need to adapt the execution environment for execution tracing in order to obtain code coverage of the firmware image. This is discussed in section 5.2. After that, we explore more in depth the fuzzing that needs to be performed on the embedded web interfaces, which is described in section 5.3. Finally, we have the analysis of the obtained trace files to obtain the code coverage achieved by each fuzzing configuration. This is treated in section section 5.4.

5.2 Code coverage of emulated firmware

As mentioned in the previous section, we will be using Firmadyne to achieve software-based full system emulation [8]. This means that we do not need the hardware associated with the firmware image to be able to run it. Therefore, our plugin is more scalable. Furthermore, Muench *et al.* stated in their research about fuzzing embedded devices, that the best fuzzing results in terms of efficiency are obtained when using full emulation [30]. Finally, the fact that Firmadyne replaces the kernel of the firmware image poses no problem for our plugin, as the web server that provides

the embedded web interface for a firmware image are located in the root file system [10, 11].

To obtain the code coverage for a firmware image, we need to be able to log execution traces. As stated by Felbinger *et al.* there exist three different methods to obtain code coverage: instrument the source code, instrument the object code or obtain the execution traces from the execution platform [14]. Since the first two methods are not possible in our situation as we only have already compiled firmware at our disposal, we opted to obtain the execution traces from the execution platform, which is Firmadyne. Another advantage of this approach is that the extraction of execution traces happens in a non-intrusive way as the firmware image itself is not instrumented [14]. To carry out the full system emulation, Firmadyne uses the underlying QEMU full system emulator. It extracts the file system of the firmware image and boots it together with a pre-built kernel in the QEMU full system emulator [8].

5.2.1 QEMU full system emulation

QEMU emulates a target CPU on a (possibly) different host CPU and therefore makes it possible to run a complete operating system in a virtual machine [4]. The target CPU instructions are translated into host CPU instructions by the dynamic translator of the CPU at runtime and it provides a simulated program counter [4]. This means that we can trace the code coverage by using this simulated program counter. To achieve this, we need to replace the pre-built kernel provided and used by Firmadyne by a QEMU kernel that has this kind of tracing enabled.

In the context of the COUVERTURE project, the researchers extended a version of QEMU for logging execution traces [5]. Because this extended version of QEMU is not publicly available anymore, we opted to use the tracing feature provided by newer versions of QEMU kernels as was done and documented by Jones in his blog [23]. By enabling this tracing feature, the simulated program counters by QEMU are logged for specified trace-events. As Firmadyne currently only supports the CPU architectures little-endian ARM, little-endian MIPS and big-endian MIPS [8], we only made pre-built tracing-enabled QEMU kernels available for these CPU architectures. An advantage of pre-building QEMU kernels in this way with tracing enabled, is the possibility to easily extend the provided kernels if more CPU architectures would be supported by Firmadyne. The second adaptation, that we need to make is to provide the trace file (which contains the specified trace-events) to the QEMU run command.

5.3 Fuzzing embedded web interfaces

In this section, we describe the implementation of fuzzing of embedded web interfaces. Administration of embedded devices happens often through web interfaces, for which the web server and web application files are present in the file system of the firmware image [10, 11].

As fuzzing framework, we opted to use the Metasploit Framework. The main reasons for this choice are that it is already available in the FACT framework, it is open-source, it works well with Firmadyne and it has fuzzer modules available, of which the configuration can easily be adapted [32].

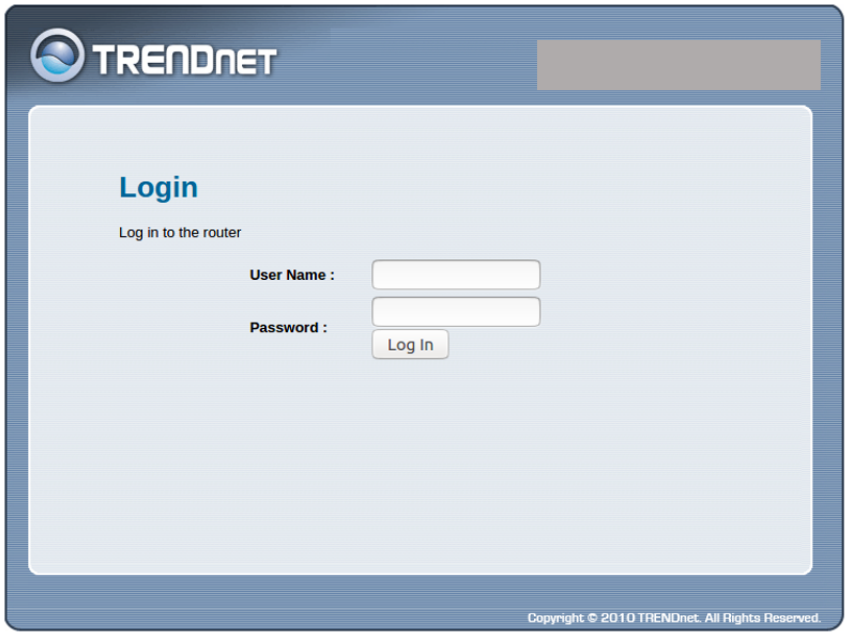

Figure 5.1: An example of an embedded web interface of a collected firmware image.

A typical example of an embedded web interface (originating from one of the earlier collected firmware images) can be found in Figure 5.1. For the firmware images collected in chapter 3, we found that all the firmware images, that could be emulated, have similar embedded web interfaces that are username and password protected or only password protected. This is useful information for choosing the right fuzzer modules.

Metasploit has currently three available protocol-based fuzzers for the HTTP protocol: `http_form_field`, `http_get_uri_long` and `http_get_uri_strings`. The exact functionality of these fuzzers can be found in the description in their scripts. The `HTTP Form Field Fuzzer` collects the available forms and subsequently performs POST actions on those forms in order to fuzz them. The `HTTP GET Request URI Fuzzer (Incrementing Lengths)` sends HTTP GET requests with incrementing URL lengths. The `HTTP GET Request URI Fuzzer (Fuzzer Strings)` sends a series of HTTP GET requests with malicious URIs. This means that we have one fuzzer for the available username and password fields and two fuzzers for trying if some other URL is also available for the firmware image.

Those three fuzzers will be used in the plugin in the context of this chapter. A current limitation of the FACT framework, is the lack of possibility to provide a

plugin with custom input at the start of the analysis, e.g. to indicate the wanted fuzzers or some specific fuzzer configuration. Therefore, we provide a default fuzzing file, which can be extended and adapted, but not yet in a user-friendly way.

Each fuzzer has different settings available that together form the configuration for the fuzzer. In our implementation, multiple values for one setting can be specified whereafter all possible configurations for the fuzzer are determined. After the embedded web interface becomes reachable, each fuzzer configuration is executed for a beforehand specified period. During this, the start and end time of each phase are registered (e.g. when the embedded web interface became available, when an execution phase for a fuzzer configuration is started).

5.4 Analysis of obtained execution traces

After the fuzzing, we need to analyse the code coverage obtained by the execution tracing. First, the trace file needs to be converted to a readable trace file so it can be further processed. For this, we used the script for this purpose provided with QEMU, `simpletrace.py`. It is however important to note that this conversion is slow and the main bottleneck of this plugin.

Next, the relative timestamps are converted to absolute timestamps and the timing of the trace file is corrected. This correction is needed because there is a difference between the total execution time measured by the trace file and the total execution time measured by our own the timing registering. This is probably due to the fact that Firmadyne first sets up the network interface on the host before it starts to emulate the firmware image.

After that, the registered program counters can be assigned to their corresponding execution phase. Here, there are in general two possibilities; either the program counter belongs to the start-up phase of the firmware image (thus before the web interface becomes reachable) or it belongs to a fuzzing execution phase. Here, we made the assumption that the amount of program counters that are not the result of the execution of the fuzzer configuration (e.g. background processes, influence of earlier executed fuzzer configuration) is negligible or consistent during the whole execution (e.g. present during all execution phases).

Now, it is possible to determine how many program counters are reached by each fuzzer. For this plugin, we determined three numbers for the code coverage per fuzzer configuration. It is important to note that those numbers have no meaning on itself, as the total number of program counters to be reached (to obtain full coverage) is unknown. The importance of these numbers lies in the possibility to compare them to the numbers obtained by the other fuzzer configurations during the same execution.

First, we determined how many program counters each fuzzer configuration reached in total. Second, we determined how many unique program counter were reached by each fuzzer configuration. Comparing the first and the second number gives an idea of the amount of times that the same program counter was executed. Third, we calculated how many unique program counters were reached by each fuzzer

configuration that were not reached by another fuzzer configuration. This gives an idea about the amount of program counters that were reached only by this fuzzer configuration. A limitation of tracing program counters is the lack of context. Jones showed in his blog that some context could be added by mapping program counters to their corresponding kernel symbols (and removing them from the final result as the kernels are not part of the original firmware image), but this very time-intensive process is not desirable for a plugin [23].

5.5 Results

Next, we discuss the results of the plugin for one firmware image, as it would not be meaningful to compare them between firmware images as was discussed in the previous section. The plugin returns (besides the successfulness of the different steps in the Firmadyne emulation process) four results: the settings of the executed fuzzer configurations, the total number of reached program counters by each fuzzer configuration, the unique number of reached program counters by each fuzzer configuration and the global unique number of reached program counters only by that fuzzer configuration.

We will use all three available Metasploit fuzzers for the HTTP protocol. For the `HTTP Form Field Fuzzer`, we will define two possible values for the handle cookies option (true and false). All other settings are set on their default values (if there is one). In total, we have thus four different fuzzer configurations that will be executed. The exact used values per fuzzer configuration can be found in Appendix B in section B.1. As execution time for each fuzzer configuration, we took 20 seconds. This experiment was performed with the `Netgear WNAP320 firmware image (version 2.0.3)`, which is provided as example on the Firmadyne Github page[1].

A screenshot of the result in our developed plugin can be found in section A.2 in Appendix A.

5.6 Evaluation

In this section we will evaluate the developed plugin in terms of correctness and usability.

5.6.1 Fuzz tracing plugin

The high-level code structure for the plugin is displayed in Figure 5.2. It follows the same code structure as the `firmadyne` plugin[2] with the addition of some scripts and of two folders in the `internal` folder (`kernels` and `trace`) and a `results` folder. The `runFuzzers.py` script is an adaptation of the `runExploits.py` file present in Firmadyne and performs the execution of the Metasploit fuzzers. The wanted fuzzer

[1] https://github.com/firmadyne/firmadyne
[2] https://github.com/fkie-cad/FACT_firmadyne_analysis_plugin

configurations to be tested can be adapted in this file. The `kernels` folder contains the QEMU kernels for which tracing is enabled. The `trace` folder contains the code used to convert the trace file to a readable trace file usable for further processing. This code originates from QEMU. The `results` folder contains all the files generated during processing (e.g. program counters per phase).

```
fuzz tracing
├── install.sh
├── bin
│   ├── Firmadyne
│   └── Metasploit Framework
├── code
│   └── fuzz_tracing.py
├── internal
│   ├── steps
│   │   ├── prepare.py
│   │   ├── emulation.py
│   │   └── processing.py
│   ├── runFuzzers.py
│   ├── kernels
│   │   ├── qemu-system-arm
│   │   ├── qemu-system-mips
│   │   └── qemu-system-mipsel
│   ├── trace
│   │   ├── simpletrace.py
│   │   ├── tracefile
│   │   └── trace-events-all
│   └── firmadyne_wrapper.py
├── results
├── test
│   ├── test_fuzz_tracing.py
│   └── data
└── view
    └── fuzz_tracing.html
```

FIGURE 5.2: Code structure for the `fuzz tracing` plugin.

Correctness

As this plugin is an adaptation of the `firmadyne` plugin, we limited the additional implemented test cases to the code that we changed. This means that we discuss the code coverage obtained by our test cases for `prepare.py`, `emulation.py` and `processing.py`. For the preparation phase, we obtained a code coverage of 88%. For the emulation phase, we got 94%. For the processing phase, the code coverage comes to 97%.

Usability

In this section, we will investigate the usability of the developed plugin.

First, it would be interesting to test if the plugin produces a stable result under the same execution conditions. It is important to have a stable result for it to be usable. This experiment was also performed with the the `Netgear WNAP320 firmware image (version 2.0.3)`. We used the same fuzzer configurations as in the experiment in the previous section, of which the exact used values for each setting can be found in Appendix B and choose the execution time per fuzzer configuration to be 20 seconds. With these settings, the plugin was executed 10 times. The results for this experiment are displayed in Figure 5.3. In this figure, fuzzer configuration

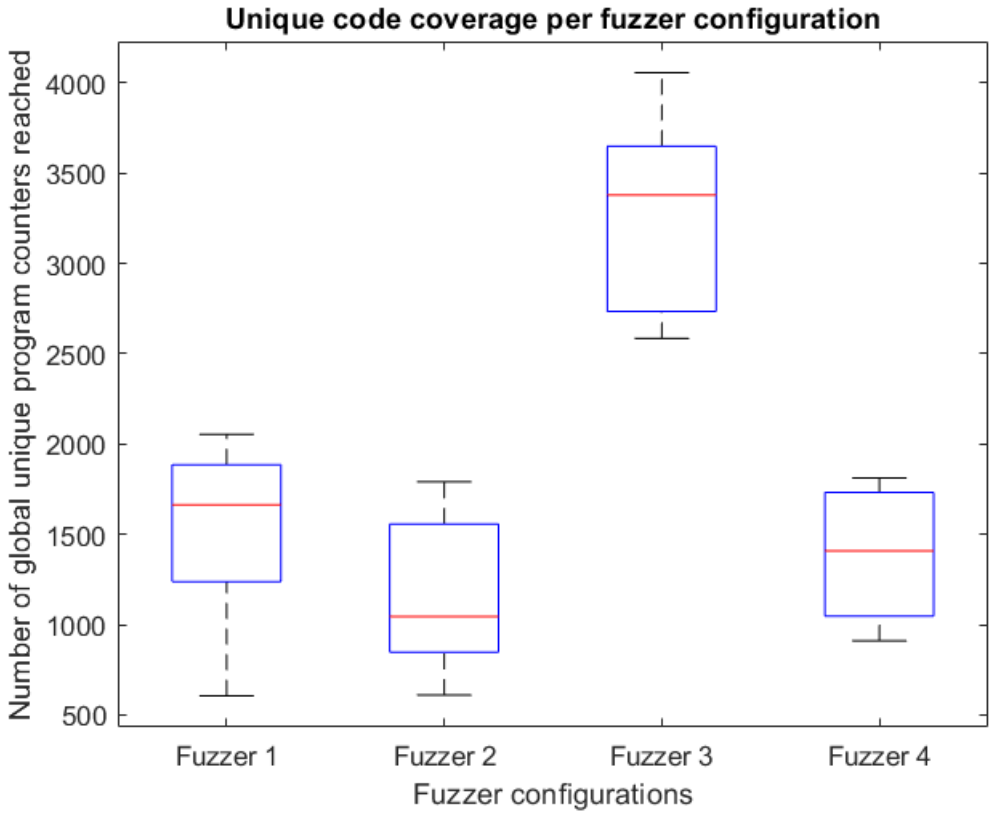

Figure 5.3: Unique code coverage reached per fuzzer configuration during ten executions.

1 corresponds to the `HTTP Form Field Fuzzer` with the handle cookies setting set on true. Fuzzer configuration 2 corresponds to `HTTP Form Field Fuzzer` with the handle cookies setting set on false. Fuzzer configuration 3 corresponds to `HTTP GET Request URI Fuzzer (Fuzzer Strings)`. Fuzzer configuration 4 corresponds to the `HTTP GET Request URI Fuzzer (Incrementing Lengths)`. From this figure,

we can derive that fuzzer configuration 3 reaches more code than the other fuzzer configurations and is stable in doing so. For the other fuzzer configurations it is not clear from this experiment if one performs better than another.

When looking at the total unique number of program counters reached by each experiment, we see that the minimal amount is 6943 program counters and the maximal amount is 7833 program counters. The median lies on 7428 program counters. For this, we see three possible factors. First, the difference could occur due to the fact that the fuzzer configurations were not always executed in the same order. However, these differences are also present among experiments that used the same execution order for the fuzzer configurations. Second, noise of background processes could attribute to the distribution of the obtained values. However, as the program counters reached by any other fuzzer configuration are filtered out, this would eliminate the background processes present in at least two fuzzing phases. Therefore, this noise would be negligible. Third, it could be the case that a fuzzer configuration reaches deeper paths (and covers more program counters) due to differences in the random generated data.

Next, we conducted an experiment to see how the amount of executed fuzzer configurations influences the total execution time of the plugin, as this execution time is an important property of a plugin. We used the exact same configuration for the plugin as in the previous experiment. The execution time for each amount of fuzzer configurations was measured 5 times. Next, the mean of the execution time for each amount of executed fuzzer configurations was calculated. These results are plotted in Figure 5.4. In this figure, we see that within 6000 seconds, 98 fuzzer configurations could be tested and compared in terms of code coverage, which means that the plugin scales well for multiple fuzzer configurations. As mentioned earlier, the bottleneck of the plugin is the conversion of the trace file to a readable trace file by the script provided by QEMU. Adding an additional fuzzer configuration means that the execution of the firmware image is traced for an additional 20 seconds in this case and that the total execution time increases. However, the size of the trace file did not increase significantly. Therefore there is another bottleneck present in this case, which is the division of program counters per phase.

For our next experiment, we will be using the earlier collected firmware images that could be emulated with Firmadyne in chapter 3. We will be comparing the execution times of these firmware images for exact the same fuzzer configurations and carried out in the same execution environment. The results of this experiment are displayed in Figure 5.5. Each item represents a firmware image and its colour and shape indicate its vendor. The lowest observed execution time was 676 seconds. This firmware image originated from a Wireless Range Extender of ZyXEL. The highest observed execution time was 3329 seconds, belonging to a Wireless Access Point of TRENDnet. The median of all execution times lies on 1634 seconds. For the remaining 12 firmware images no results could be obtained due to a too big trace file (i.e. 4 GB and more) which results in a memory error while converting it to a readable trace file. The generation of so many program counters could be attributed to specific missing hardware for the firmware image [10].

In Figure 5.5, we observe even within the same vendor variety in the obtained

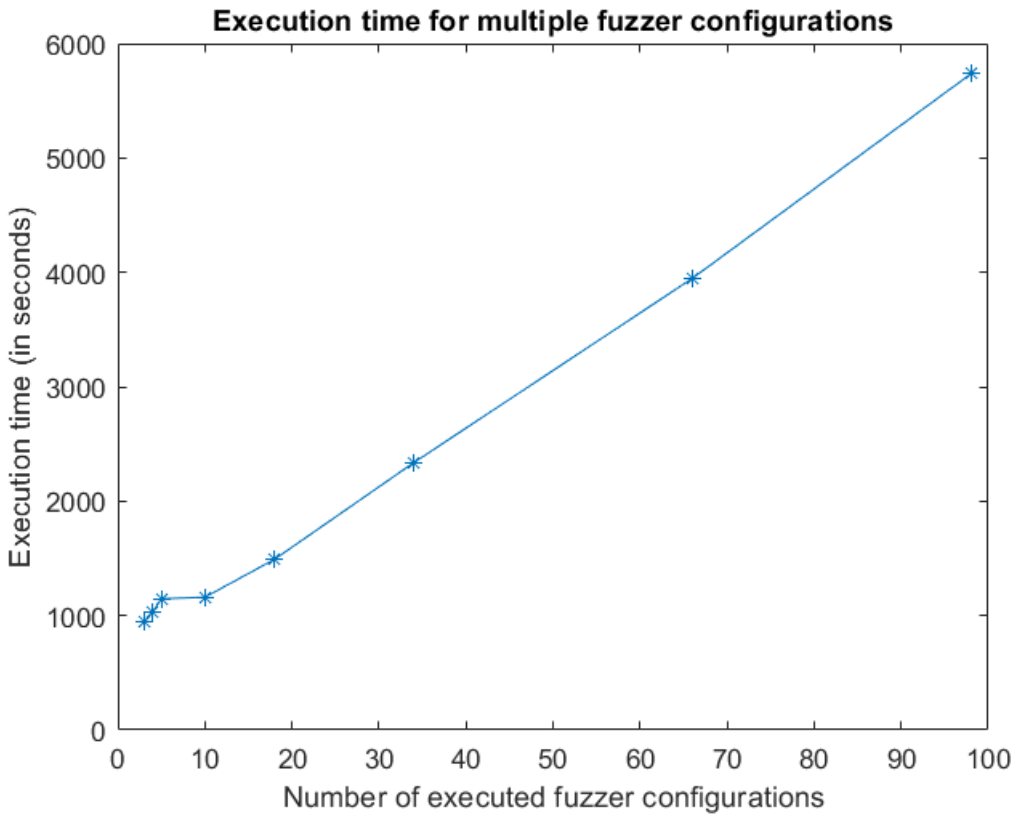

Figure 5.4: The execution time of the plugin per number of executed fuzzer configurations.

execution times. However, all those firmware images have similar characteristics due to the fact that emulation with Firmadyne is necessary in order to be successfully run with this plugin.

5.7 Conclusion

In this chapter, we developed a plugin for comparing code coverage obtained by executing different fuzzer configurations on an embedded web interface of an emulated firmware image in the context of the FACT framework. This plugin helps in choosing the most promising fuzzer configurations in terms of most code coverage and most unique code coverage. The two criteria for this plugin where met, as it works in the FACT framework and as it gives an idea of the code coverage obtained for the different fuzzer configurations.

The advantages of this plugin are its scalability in terms of the number of fuzzer configurations that can be tested while the execution time remains reasonable and

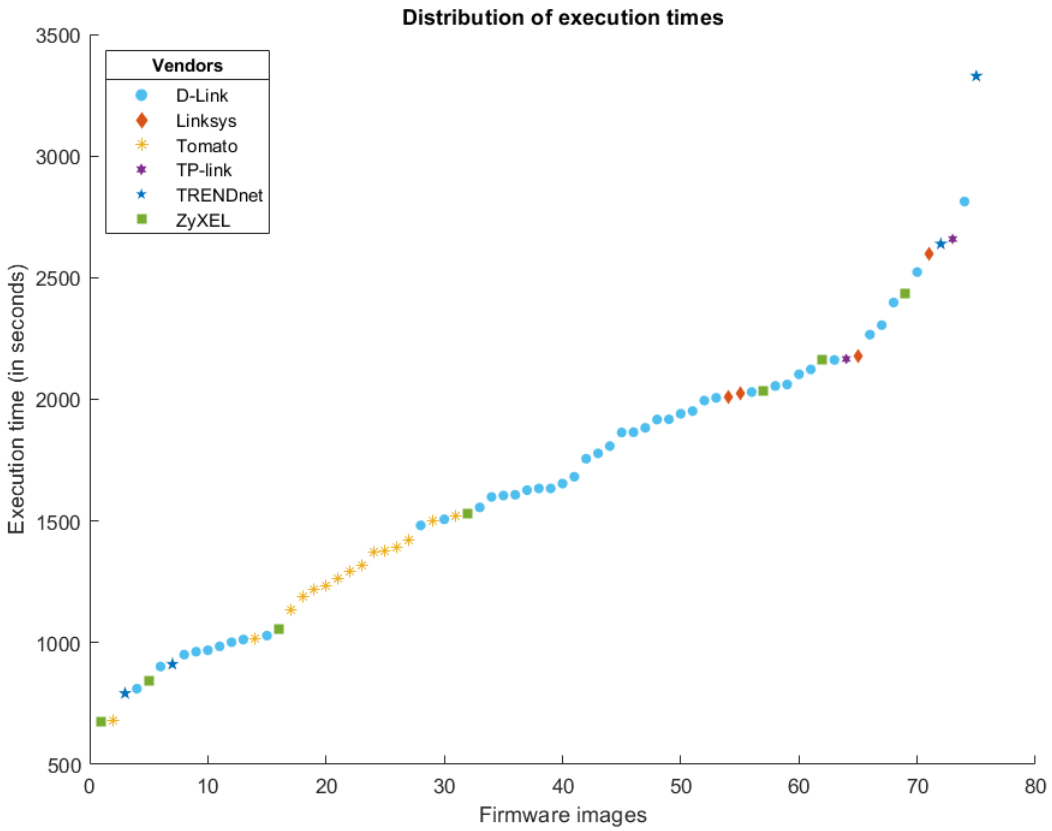

FIGURE 5.5: Distribution of the execution times of the collected firmware images with the same settings for fuzzing.

its relative stability over multiple executions. The first limitation of this plugin is the restricted amount of firmware images that are well suited for this plugin due to the use of Firmadyne for emulation, which is further restricted by firmware images that could not be analysed due to the generation of a too big trace file. The second limitation of the plugin consist of the lack of context for the code coverage results.

Chapter 6

Discussion

We will start this chapter with a discussion of previous research related to the implemented functionality of both plugins in section 6.1. Afterwards, we will discuss the overall usability of the plugins in section 6.2 and their results in section 6.3.

6.1 Previous related research

In this section, we will discuss previous research related to the dynamic analysis implemented in both plugins.

6.1.1 Dynamic CVE verification

Costin *et al.* used in 2016 the method of looking for vulnerabilities for firmware images with static analysis and afterwards applying dynamic analysis to be able to confirm the vulnerability for a firmware image in their large-scale analysis of web interfaces of embedded devices. The framework they developed for this purpose is fully automated and they also used the approach of emulating the firmware image with a generic kernel and the original root file system to be able to perform the dynamic analysis. However, they did not limit the dynamic analysis to the possible vulnerabilities found by their static analysis phase, but rather guided the dynamic analysis to certainly test the possible high impact vulnerabilities found. Another difference with our approach, is that the researchers focussed on the PHP source code present in firmware images and did identify previously unknown vulnerabilities [10]. In our case, the found vulnerabilities were already publicly known for the different software components used in the firmware images and the dynamic analysis was used to confirm if the firmware image in question could be effectively exploited by those vulnerabilities.

6.1.2 Tracing code coverage of fuzzers for emulated embedded devices

In 2010, Bordin *et al.* adapted QEMU, in the context of the COUVERTURE project, to support the generation of execution traces. For this purpose, they provided two

kinds of execution traces: synthetic bounded-size traces and full historical traces, which both contain information about conditional branches [30]. We were not able to use this adapted version[1] of QEMU as it was not publicly available any more. We did find a version[2] provided by Adacore but we were not able to build this successfully. Therefore, we used the trace functionality provided by QEMU. The method we used to obtain code coverage [23], yields only the simulated program counters and provides no information about conditional branches encountered.

In 2018, Muench *et al.* performed a research to the challenges in fuzzing embedded devices and focused more in particular on memory corruptions. One of their findings is that with full system emulation, a better fuzzing efficiency can be achieved than with the actual physical devices [30]. Here, we can clearly see the added value that our plugin offers, namely to optimise the ratio between the fuzzer configurations that are executed and the number of detected vulnerabilities.

In 2019, Zheng *et al.* developed FIRM-AFL, a fuzzing solution for firmware that achieves both transparency (i.e. no modifications to the program are made) and efficiency (i.e. the fuzzing throughput approaches the one obtained under user-mode emulation). They showed that full system emulation contains three major bottlenecks that significantly increase the execution time for the fuzzers: memory address translation, dynamic code translation and syscall. Therefore, to improve this efficiency, they use augmented process emulation. This approach combines AFL and Firmadyne to obtain the advantages of both user-mode emulation and full system emulation [39].

Srivastava *et al.* developed FirmFuzz in 2019. This provides an analysis framework that combines static analysis with fuzzing to find deep vulnerabilities in firmware images and focusses on custom vendor-developed applications. The goal is to achieve a deeper analysis of the firmware images, which resulted in a less scalable framework. Furthermore, FirmFuzz attempts to increase the code coverage of the used fuzzer by using static analysis to extract the username-password pair for the web interface. If that fails, it tries to brute-force this authentication [35]. As the functionality of finding usernames and passwords for a firmware image is already provided in the FACT framework in the `users and passwords` plugin, the use of its results would be an interesting extension for the `fuzz tracing` plugin.

6.2 Usability of plugins

As both plugins are dynamic analysis plugins and both use Firmadyne to emulate the firmware images, they have both the limitation that only Linux-based embedded firmware images are supported. As we saw in chapter 3, only a fraction of the collected firmware images could be emulated with Firmadyne and can thus be tested with the developed plugins. Chen *et al.*, who developed Firmadyne, encountered the same kind of reduction when testing Firmadyne on all the collected firmware images [8]. However, the problem is not limited to the emulation provided by Firmadyne.

[1] http://forge.open-do.org/projects/couverture-qemu
[2] https://github.com/AdaCore/qemu

Srivastava *et al.* experienced the same reduction of usable firmware images while testing their tool FirmFuzz, for which the web interface of the firmware image needed to be accessible in order to carry out the fuzzing [35]. Costin *et al.* saw a similar reduction in the number of Linux-based embedded firmware images to those that were usable to test their large-scale dynamic analysis framework on (i.e. those for which the embedded web interface could be started) [10]. In all these cases, the researchers focussed on supporting Linux-based embedded firmware images because of its prevalence and because these firmware images are well structured into bootloaders, kernels and file systems, which makes the unpacking and emulation process easier [10].

Furthermore, we found that the encountered limitations of the FACT framework impact the usability of the plugins. For the dynamic CVE verification functionality, we have the problem that the `CVE firmadyne` plugin can not access all the results for all the files of the firmware image as it is not possible to derive the ID associated to the firmware image from a file of that firmware image. This was solved by having the `CVE exploits` plugin generate the exploits file. To achieve this, multithreading needed to be disabled, but this has a negligible impact on the execution time. For both plugins, the usability can be improved if there would be the possibility in the FACT framework to provide some input for a plugin. For the dynamic CVE verification plugin, this would consist of the indication of additional name-value pairs that are set for the configuration of Metasploit. In the context of the comparison for the code coverage obtained by different fuzzer configurations, this would consist of indicating how many fuzzer configurations should be tested and/or which the wanted values are for the settings of a fuzzer configuration. For both plugins, these settings are currently provided in a file in the `internal` folder.

Next, as the focus of the FACT lies on providing scalable analysis for firmware images, it is important to assess the plugins in terms of scalability. As mentioned earlier, their application field in terms of supported firmware images is limited due to the used emulation tool. Furthermore, the scalability of the first plugin is limited by its current implementation through which it is not possible to execute the analysis for multiple firmware images at the same time. For the `fuzz tracing` plugin, this restriction is not present and multiple firmware images can be analysed at the same time (with the same settings for the analysis). The scalability of this plugin depends on its execution time, which is mainly influenced by two factors: the firmware image to be tested and the number of fuzzer configurations to be executed.

6.3 Usability of results

In this section, we discuss the usability of the results produced by both plugins (i.e. if they produce valuable information).

For the dynamic CVE verification plugin, we observed that the automatic verification of CVEs by our plugin is limited due to the following two factors. First, there is only a fraction of exploits available for all CVEs in the Metasploit Framework. Second, there is the limitation caused by the required settings for which the value

needs to be set in advance. This is not possible for e.g. SESSION, a setting that is required for privilege escalation exploits, which requires the successful execution of another exploit on the firmware image. However, other settings could be provided beforehand by the user e.g. LHOST, hence we provided the possibility to specify the Metasploit configuration. When an exploit can be executed successfully for a firmware image or not (i.e. the values for all required settings are present), this provides useful information.

For the `fuzz tracing` plugin, the usability of the results is limited due to the lack of context for the logged program counters. Therefore, it can give no insight on which parts of the firmware image are reached, this could be an added value. However, the results obtained are relatively stable. With the current mechanism for obtaining execution traces in place, bug detection capabilities could be added for this plugin under the form of reboot detection. The program counters generated and logged during the operational phase (i.e. when the firmware image boots) could be compared with the program counters logged during one of the fuzzing phases to detect the boot pattern. A threshold could be set for the number of program counters of the boot pattern that should return in the same order during a fuzzing phase.

Chapter 7

Conclusion

In this chapter the main conclusions are discussed. An overview of the thesis and the obtained results is given in section 7.1. In section 7.2 our own contribution is discussed. Next, section 7.3 lists the main challenges that we experienced in the context of this thesis. Possible extensions for both plugins are treated in section 7.4.

7.1 Overview

In this thesis, we developed two emulation-based dynamic analysis plugins for the FACT framework. The main requirements for these plugins are that they function in the FACT framework and that they effectively implement the intended dynamic analysis. Furthermore, as the main goal of the FACT framework is to provide large-scale analysis of firmware images, scalability is an important factor.

The first plugin, which performs dynamic CVE verification, is divided in two actual plugins `CVE exploits` and `CVE firmadyne`. This plugin aims to reduce the amount of manual analysis needed to confirm a possible vulnerability for a specific firmware image. The results of this plugin are limited by two factors. First, there is only for a fraction of the identified CVEs an exploit available in the Metasploit Framework. Second, it is difficult to execute the exploits in an automated way as required settings are sometimes not set and the exploit can not be executed in that situation. To mitigate this issue, we provided a configuration file for Metasploit in which the value for settings can be specified. However, even with this adaption it is still not possible to support the testing of privilege escalation exploits on a firmware image with Metasploit. For the collected firmware images, we found only privilege escalation exploits and we were therefore unable to successfully execute an exploit. Finally, we found that the support provided by FACT framework is not sufficient to implement the interaction between the two plugins in a scalable way.

The second plugin `fuzz tracing` enables the user to compare the code coverage of different fuzzer configurations for the embedded web interface of a firmware image. As full program coverage is often the goal of fuzzing, this plugin enables the user to know which fuzzer configuration obtains the highest code coverage or results in the most unique code coverage. Therefore, the plugin can be used to optimise the ratio

between the number of executed fuzzers and the number of detected vulnerabilities. The scalability of this plugin in terms of needed execution time depends on two factors: the firmware image under test and the number of fuzzer configurations to be executed. The results of the plugin are relatively stable (i.e. in terms of which fuzzer configuration performs the best), but here we have a limitation by the lack of context of which parts of the firmware image are reached.

Both plugins could be improved in terms of usability if there would be a possibility to provide input at the start of the analysis for a plugin in the FACT framework.

The two plugins use Firmadyne to obtain software-based full system emulation and this limits the firmware images that can be tested to network-connected Linux-based embedded firmware images for which the root file system is present and that have little-endian ARM, little-endian MIPS or big-endian MIPS as CPU architecture.

7.2 Contribution

For this thesis, two dynamic analysis plugins were developed for the FACT framework. For the dynamic CVE verification plugin, the functionality is divided over two actual plugins: `CVE exploits` and `CVE firmadyne`. The first one was developed from scratch with the use of the provided template by the FACT framework. For the second one, we started with the `firmadyne` plugin[1] and made the necessary changes to achieve the wanted functionality. For the `fuzz tracing` plugin, we also started from `firmadyne` plugin and adapted the code from there to obtain the wanted functionality. For the part of this plugin that logs the reached program counters by building and using a tracing-enabled QEMU kernel, we followed the method provided by Jones [23].

7.3 Challenges

In the realisation of this thesis, we encountered several challenges. First, there was the lack of extensive publicly available documentation for the FACT framework. The explanation for the development of a plugin is concise and not all the options are covered. Second, we had the implementation of the interaction between the two developed plugins for the dynamic CVE verification process, which turned out to be quite time-intensive. Furthermore, while building the QEMU kernels with tracing-enabled, we encountered some system-specific errors. Finally, we encountered the challenge of obtaining a collection of firmware images, which was also quite time-intensive.

7.4 Possible extensions

For the dynamic CVE verification functionality, it would be interesting to examine if the combined use of multiple frameworks, that provide exploits, could lead to a

[1] https://github.com/fkie-cad/FACT_firmadyne_analysis_plugin

higher coverage of the CVEs.

For the `fuzz tracing` plugin, it would be interesting to test the influence of adding username-password authentication for the plugin on the results produced by the fuzzer configurations.

Appendices

Appendix A

Screenshots of web user interface

This appendix contains the screenshots made of the web user interface for the two developed plugins.

A.1 Plugin I. Dynamic CVE verification

Results Metasploit exploits: cve firmadyne

CVE	Result	Exploit	Missing parameters	Log file exploit
CVE-2014-0038	unsuccessful	exploits/linux/local /recvmmsg_priv_esc	SESSION	[*] Spooling to file [redacted]cve_firmadyne/internal/../bin/firmadyne/exploits /exploit.CVE-2014-0038.log... resource (script.rc)> use exploits/linux/local/recvmmsg_priv_esc resource (script.rc)> exploit -z [1m[31m[-] Exploit failed: The following options failed to validate: SESSION. [*] Exploit completed, but no session was created. resource (script.rc)> spool off
CVE-2016-8655	unsuccessful	exploits/linux/local /af_packet_chocobo_root_priv_esc	SESSION	[*] Spooling to file [redacted]cve_firmadyne/internal/../bin/firmadyne/exploits /exploit.CVE-2016-8655.log... resource (script.rc)> use exploits/linux/local/af_packet_chocobo_root_priv_esc resource (script.rc)> exploit -z [1m[31m[-] Exploit failed: The following options failed to validate: SESSION. [*] Exploit completed, but no session was created. resource (script.rc)> spool off

FIGURE A.1: Unique code coverage reached per fuzzer configuration during ten executions.

A.2 Plugin II. Comparing fuzzer-based code coverage

Results Fuzzing: fuzz tracing

Fuzzers	Configurations	Number of total reached program counters	Number of unique reached program counters	Number of unique reached program counters only by this fuzzer
http_get_uri_strings	['RPORT=80', 'URIBASE=/']	72445	1476	873
http_form_field	['ENDSIZE=40000', 'RPORT=80', 'FUZZHEADERS=false', 'HANDLECOOKIES=true', 'STEPSIZE=1000', 'DELAY=0', 'TIMEOUT=15', 'STOPAFTER=2', 'STARTSIZE=1000', 'TYPES=text,password,inputtextbox', 'CODE=200,301,302,303', 'CYCLIC=false']	28342	848	235
http_form_field	['ENDSIZE=40000', 'RPORT=80', 'FUZZHEADERS=false', 'HANDLECOOKIES=false', 'STEPSIZE=1000', 'DELAY=0', 'TIMEOUT=15', 'STOPAFTER=2', 'STARTSIZE=1000', 'TYPES=text,password,inputtextbox', 'CODE=200,301,302,303', 'CYCLIC=false']	42549	1042	426
http_get_uri_long	['RPORT=80', 'URIBASE=/', 'MAXLENGTH=16384']	24651	325	49

Figure A.2: Web user interface of FACT showing the results of the fuzz tracing analysis for a firmware image.

Appendix B

Configuration for experiments

This appendix contains configurations used in experiments with the plugins.

B.1 Fuzzer configurations

This section contains the fuzzer configurations for the Metasploit fuzzer modules used for the experiments in chapter 5.

B.1.1 HTTP Form Field Fuzzer

- CODE : [‘200,301,302,303’]
- CYCLIC : [‘false’]
- DELAY : [0]
- ENDSIZE : [40000]
- FUZZHEADERS : [‘false’]
- HANDLECOOKIES : [‘true’, ‘false’]
- RPORT : [80]
- STARTSIZE : [1000]
- STEPSIZE : [1000]
- TIMEOUT : [15]
- TYPES : [‘text,password,inputtextbox’]
- STOPAFTER : [2]

B.1.2 HTTP GET Request URI Fuzzer (Incrementing Lengths)

- MAXLENGTH : [16384]
- RPORT : [80]
- URIBASE : [‘/’]

B.1.3 HTTP GET Request URI Fuzzer (Fuzzer Strings)

- RPORT : [80]
- URIBASE : [‘/’]

Appendix C

Implemented test cases for plugin

This appendix contains the test cases for the `CVE exploits` plugin. The test cases are provided with each plugin. All tests can be executed with pytest.

C.1 Plugin I. Dynamic CVE verification

C.1.1 `CVE exploits` plugin

These test cases are provided in the `test_cve_exploits.py` file and can be executed with pytest.

```
import os
from objects.file import FileObject
from common_helper_files import get_dir_of_file

from test.unit.analysis.analysis_plugin_test_class import AnalysisPluginTest
from ..code.cve_exploits import
AnalysisPlugin,
collect_exploits,
search_metasploit_exploit_for_cve,
create_metasploit_files,
check_if_exploit_for_cve_is_present, add_exploit_to_run_exploits_file,
INTERNAL_DIRECTORY_PATH

TEST_FILE_PATH = os.path.join(get_dir_of_file(__file__), 'data')

CVE_EXPLOITS = {
'CVE-2015-8660': ['exploits/linux/local/overlayfs_priv_esc'],
'CVE-2010-3310': ['N/A'],
'CVE-2013-6376': ['N/A'],
'CVE-2016-2117': ['N/A'],
```

```
'CVE-2012-3430': ['N/A'],
'CVE-2016-5412': ['N/A']
}

CVE_EXPLOITS_2 = {
'CVE-2017-7308': ['exploits/linux/local/af_packet_packet_set_ring_priv_esc'],
'CVE-2010-3310': ['N/A'],
'CVE-2013-6376': ['N/A'],
'CVE-2016-2117': ['N/A'],
'CVE-2012-3430': ['N/A'],
'CVE-2016-5412': ['N/A']
}

CVES = ['CVE-2015-8660', 'CVE-2019-5747', 'CVE-2010-4250']

CVE_LOOKUP_ANALYSIS_RESULT = {
'summary':
['CVE-2015-1328', 'CVE-2019-19052', 'CVE-2010-2492',
 'CVE-2010-3067', 'CVE-2011-2525', 'CVE-2019-16995',
 'CVE-2012-0055', 'CVE-2015-4003', 'CVE-2019-19065',
 'CVE-2012-2745', 'CVE-2019-15030', 'CVE-2019-15807',
 'CVE-2018-10840', 'CVE-2019-20096', 'CVE-2011-2182',
 'CVE-2016-4997', 'CVE-2013-2897']
}

CVE_EXPLOITS_ANALYSIS_RESULT = {
'summary': ['exploits/linux/local/overlayfs_priv_esc'],
'tags': {'root_uid':
 'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855_0',
 'cve_exploits': {'color': 'info', 'propagate': True,
 'value': 'metasploit exploit'}},
 'exploits': {
     'CVE-2019-15030': ['N/A'], 'CVE-2019-16995': ['N/A'],
     'CVE-2016-4997': ['N/A'], 'CVE-2019-19052': ['N/A'],
     'CVE-2010-2492': ['N/A'], 'CVE-2019-15807': ['N/A'],
     'CVE-2012-0055': ['N/A'], 'CVE-2018-10840': ['N/A'],
     'CVE-2019-20096': ['N/A'], 'CVE-2011-2182': ['N/A'],
     'CVE-2015-4003': ['N/A'], 'CVE-2010-3067': ['N/A'],
     'CVE-2019-19065': ['N/A'], 'CVE-2012-2745': ['N/A'],
     'CVE-2011-2525': ['N/A'], 'CVE-2013-2897': ['N/A'],
    'CVE-2015-1328': ['exploits/linux/local/overlayfs_priv_esc']
     }
 }
```

```
CVE_LOOKUP_ANALYSIS_RESULT_2 = {'summary': ['CVE-2016-8655',
'CVE-2014-0038', 'CVE-2017-7308']}

CVE_EXPLOITS_ANALYSIS_RESULT_2 = {
'summary': [
    'exploits/linux/local/af_packet_chocobo_root_priv_esc',
    'exploits/linux/local/recvmmsg_priv_esc',
    'exploits/linux/local/af_packet_packet_set_ring_priv_esc'
],
'exploits': {
    'CVE-2017-7308': ['exploits/linux/local/af_packet_packet_set_ring_priv_esc'],
    'CVE-2014-0038': ['exploits/linux/local/recvmmsg_priv_esc'],
    'CVE-2016-8655': ['exploits/linux/local/af_packet_chocobo_root_priv_esc']
},
'tags': {
    'cve_exploits': {
        'color': 'info',
        'value': 'metasploit exploit',
        'propagate': True
        },
'root_uid': 'e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855_0'}
}

OUTPUT =
{'output': '\n7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov :
2016-05-21\np7zip Version 16.02 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,
64 bits,4 CPUs x64)\n\n
Scanning the drive for archives:\n1 file, 5362552 bytes (5237 KiB)\n\n
Extracting archive:
 /tmp/extractor/input/4c1697358f55d659f156ed9d05e74e55c8385e85c13a06397e9b9077b2a9e06a
 _5362552\n--\nPath = /tmp/extractor/input/4c1697358f55d659f156ed9d05e74e55c
 8385e85c13a06397e9b9077b2a9e06a_5362552\nType = zip\nPhysical Size = 5362552\n\n
 Everything is Ok\n\nFiles: 2\nSize:       5429867\nCompressed: 5362552\n',
 'number_of_unpacked_directories': 0,
 'entropy': 0.8959860330786459, 'size_unpacked': 5429867,
 'plugin_version': '0.7', 'size_packed': 5361056, 'summary': ['no data lost'],
 'plugin_used': '7z', 'analysis_date': 1589131192.4340925, 'number_of_unpacked_files': 2,
 'file_system_flag': False}


class TestAnalysisPluginsCVEexploits(AnalysisPluginTest):

    PLUGIN_NAME = 'cve_exploits'

    def setUp(self):
```

```
        super().setUp()
        config = self.init_basic_config()
        self.analysis_plugin = AnalysisPlugin(self, config=config)

    def test_process_object(self):
        '''
        Perform analysis for dummy file object with list of found CVEs.
        '''

        # setup
        test_file = FileObject(file_path='{}/file.txt'.format(TEST_FILE_PATH))
        test_file.processed_analysis['cve_lookup'] = CVE_LOOKUP_ANALYSIS_RESULT

        # perform analysis
        processed_file = self.analysis_plugin.process_object(test_file)
        result = processed_file.processed_analysis[self.PLUGIN_NAME]

        # check result
        assert result == CVE_EXPLOITS_ANALYSIS_RESULT

    def test_process_object_no_exploits(self):
        '''
        Perform analysis for dummy file object with list of found CVEs.
        '''

        # setup
        test_file = FileObject(file_path='{}/file.txt'.format(TEST_FILE_PATH))
        test_file.processed_analysis['cve_lookup'] = {'summary': ['CVE-2010-3310']}

        # perform analysis
        processed_file = self.analysis_plugin.process_object(test_file)
        result = processed_file.processed_analysis[self.PLUGIN_NAME]

        # check result
        assert result == {'summary': [], 'exploits': {'CVE-2010-3310': ['N/A']}}

    def test_process_object_all_exploits(self):
        '''
        Perform analysis for two dummy file object with lists of found CVEs.
        '''

        # setup for first analysis
        test_file = FileObject(file_path='{}/file.txt'.format(TEST_FILE_PATH))
        test_file.processed_analysis['cve_lookup'] = CVE_LOOKUP_ANALYSIS_RESULT_2
```

```
        # perform first analysis
        processed_file = self.analysis_plugin.process_object(test_file)
        result = processed_file.processed_analysis[self.PLUGIN_NAME]

        # check result for first analysis
        assert result == CVE_EXPLOITS_ANALYSIS_RESULT_2

        # setup for second analysis
        test_file_2 = FileObject(file_path='{}/file_2.txt'.format(TEST_FILE_PATH))
        test_file_2.processed_analysis['cve_lookup'] = CVE_LOOKUP_ANALYSIS_RESULT

        # perform second analysis
        processed_file_2 = self.analysis_plugin.process_object(test_file_2)
        result = processed_file_2.processed_analysis[self.PLUGIN_NAME]

        # check results for second analysis
        assert result == CVE_EXPLOITS_ANALYSIS_RESULT

        # check that right exploits file is generated with exploits of both analyses
        exploits_file = '{}/runExploits.py'.format(INTERNAL_DIRECTORY_PATH)
        verified_exploits_file = '{}/runExploits_2.py'.format(TEST_FILE_PATH)

        test_file = open(exploits_file, 'r')
        verified_file = open(verified_exploits_file, 'r')
        test_lines = test_file.readlines()
        verified_lines = verified_file.readlines()
        for line in test_lines:
            assert line in verified_lines


def test_collect_exploits():
    '''
    Check if the right exploits are found.
    '''

    # function under test
    result, exploits = collect_exploits(CVE_LOOKUP_ANALYSIS_RESULT_2['summary'])
    assert result == {
    'CVE-2017-7308': ['exploits/linux/local/af_packet_packet_set_ring_priv_esc'],
     'CVE-2014-0038': ['exploits/linux/local/recvmmsg_priv_esc'],
      'CVE-2016-8655': ['exploits/linux/local/af_packet_chocobo_root_priv_esc']
      }
    assert exploits == ['exploits/linux/local/af_packet_chocobo_root_priv_esc',
     'exploits/linux/local/recvmmsg_priv_esc',
     'exploits/linux/local/af_packet_packet_set_ring_priv_esc']
```

```
def test_search_metasploit_exploit_for_cve():
    '''
    Check if the right exploit paths are extracted.
    '''

    # function under test
    result = {}
    for cve in CVES:
        exploits = search_metasploit_exploit_for_cve(cve)
        result[cve] = exploits

    # CVE has one metasploit exploit
    assert result['CVE-2015-8660'] == ['exploits/linux/local/overlayfs_priv_esc']

    # CVE has no metasploit exploit but multiple vulnerabilities
    assert result['CVE-2019-5747'] == []

    # CVE has no metasploit exploit but multiple vulnerabilities
    assert result['CVE-2010-4250'] == []


def test_create_metasploit_files():
    '''
    Check if right exploits file is created.
    '''

    # setup file paths
    exploits_file = '{}/runExploits.py'.format(INTERNAL_DIRECTORY_PATH)
    verified_exploits_file = '{}/runExploits_3.py'.format(TEST_FILE_PATH)

    # cleanup
    if os.path.isfile(exploits_file):
        os.remove(exploits_file)

    # function under test
    for cve in CVE_EXPLOITS:
        exploits = CVE_EXPLOITS[cve]
        if exploits != ['N/A']:
            for exploit in exploits:
                create_metasploit_files(cve, exploit)

    # verify if exploit file is created
    assert os.path.isfile(exploits_file)
```

```
    # verify the file contents
    test_file = open(exploits_file, 'r')
    verified_file = open(verified_exploits_file, 'r')
    test_lines = test_file.readlines()
    verified_lines = verified_file.readlines()
    assert test_lines == verified_lines

    # cleanup
    if os.path.isfile(exploits_file):
        os.remove(exploits_file)


def test_check_if_exploit_for_cve_is_present():
    '''
    Check if the detection of the exploits that are already present in the exploits file i
    '''

    # setup file paths
    exploits_file = '{}/runExploits.py'.format(INTERNAL_DIRECTORY_PATH)

    # cleanup
    if os.path.isfile(exploits_file):
        os.remove(exploits_file)

    # prepare
    for cve in CVE_EXPLOITS:
        exploits = CVE_EXPLOITS[cve]
        if exploits != ['N/A']:
            for exploit in exploits:
                create_metasploit_files(cve, exploit)

    # function under test
    cve = 'CVE-2015-8660'
    exploit = 'exploits/linux/local/overlayfs_priv_esc'
    true_result = check_if_exploit_for_cve_is_present(cve, exploit)
    assert true_result

    cve = 'CVE-2016-8660'
    exploit = 'exploits/linux/local/overlayfs_priv_esc'
    false_result = check_if_exploit_for_cve_is_present(cve, exploit)
    assert not false_result

    # cleanup
    if os.path.isfile(exploits_file):
```

```
        os.remove(exploits_file)


def test_add_exploit_to_run_exploits_file():
    '''
    Check if exploits are added to the exploit file.
    '''

    # setup file paths
    exploits_file = '{}/runExploits.py'.format(INTERNAL_DIRECTORY_PATH)
    verified_exploits_file = '{}/runExploits.py'.format(TEST_FILE_PATH)

    # cleanup
    if os.path.isfile(exploits_file):
        os.remove(exploits_file)

    # prepare
    for cve in CVE_EXPLOITS_2:
        exploits = CVE_EXPLOITS_2[cve]
        if exploits != ['N/A']:
            for exploit in exploits:
                create_metasploit_files(cve, exploit)

    # function under test
    cve = 'CVE-2015-8660'
    exploit = 'exploits/linux/local/overlayfs_priv_esc'
    add_exploit_to_run_exploits_file(cve, exploit)

    # verify the file contents
    test_file = open(exploits_file, 'r')
    verified_file = open(verified_exploits_file, 'r')
    test_lines = test_file.readlines()
    verified_lines = verified_file.readlines()
    assert test_lines == verified_lines

    # cleanup
    if os.path.isfile(exploits_file):
        os.remove(exploits_file)
```

Bibliography

[1] M. Agarwal and A. Singh. *Metasploit penetration testing cookbook*. Packt Publishing Ltd, 2013.

[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, et al. Understanding the mirai botnet. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1093–1110, 2017.

[3] T. Beardsley. CVE 100K: A Big, Round Number. https://blog.rapid7.com/2018/04/30/cve-100k-a-big-round-number/. Accessed on 11 May 2020.

[4] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[5] M. Bordin, C. Comar, T. Gingold, J. Guitton, O. Hainque, and T. Quinot. Object and source coverage for critical applications with the c ouverture open analysis framework. 2010.

[6] T. by Shibby. About Tomato (ang.). https://tomato.groov.pl/?page_id=81. Accessed on 21 April 2020.

[7] D. Chen. Issue 121. https://github.com/firmadyne/firmadyne/issues/121. Accessed on 22 April 2020.

[8] D. D. Chen, M. Woo, D. Brumley, and M. Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, pages 1–16, 2016.

[9] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 95–110, 2014.

[10] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 437–448. ACM, 2016.

[11] A. Costin, A. Zarras, and A. Francillon. Towards automated classification of firmware images and identification of embedded devices. In *IFIP International*

Conference on ICT Systems Security and Privacy Protection, pages 233–247. Springer, 2017.

[12] Y. David, N. Partush, and E. Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. In *ACM SIGPLAN Notices*, volume 53, pages 392–404. ACM, 2018.

[13] J. V. Dorp. Improving your firmware security analysis process with FACT. https://passthesalt.ubicast.tv/videos/improving-your-firmware-security-analysis-process-with-fact/. Accessed on 21 October 2019.

[14] H. Felbinger, J. Sherrill, G. Bloom, and F. Wotawa. Test suite coverage measurement and reporting for testing an operating system without instrumentation. In *Proceedings of the 17th Real-Time Linux Workshop*, pages 13–22. ., 2015.

[15] fkie cad.

[16] fkie cad. analysis plugin development. https://github.com/fkie-cad/FACT_core/wiki/analysis-plugin-development/. Accessed on 16 May 2020.

[17] fkie cad. FACT - Developer's Manual. https://github.com/fkie-cad/FACT_core/wiki. Accessed on 25 February 2020.

[18] fkie cad. FACT screenshots. https://github.com/fkie-cad/FACT_core/blob/master/docs/FACT_screenshots/02_upload.png. Accessed on 25 May 2020.

[19] fkie cad. How to detect a firmware analysis completed. https://github.com/fkie-cad/FACT_core/issues/215. Accessed on 3 April 2020.

[20] Gartner. Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. https://www.gartner.com/en/newsroom/press-releases/2017-02-07-gartner-says-8-billion-connected-things-will-be-in-use-in-2017-up-3 Accessed on 7 November 2019.

[21] A. Gupta. *The IoT Hacker's Handbook*. Springer, 2019.

[22] G. Hernandez, F. Fowze, D. J. Tang, T. Yavuz, P. Traynor, and K. R. Butler. Toward automated firmware analysis in the iot era. *IEEE Security & Privacy*, 17(5):38–46, 2019.

[23] R. W. Jones. Tracing QEMU guest execution. https://rwmj.wordpress.com/2016/03/17/tracing-qemu-guest-execution/. Accessed on 16 March 2020.

[24] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Mit Press, 2016.

[25] J. Li, B. Zhao, and C. Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[26] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[27] D. Liu, Y. Tang, B. Wang, W. Xie, and B. Yu. Automated vulnerability detection in embedded devices. In *IFIP International Conference on Digital Forensics*, pages 313–329. Springer, 2018.

[28] MITRE. About CVE. https://cve.mitre.org/about/index.html. Accessed on 3 April 2020.

[29] MITRE. Frequently Asked Questions. https://cve.mitre.org/about/faqs.html. Accessed on 3 April 2020.

[30] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS*, 2018.

[31] Openwrt. Welcome to the OpenWrt Project. https://openwrt.org/. Accessed on 21 April 2020.

[32] S. Rahalkar, Rahalkar, and Karkal. *Quick Start Guide to Penetration Testing*. Springer, 2019.

[33] L. A. B. Sanguino and R. Uetz. Software vulnerability analysis using cpe and cve. *arXiv preprint arXiv:1705.05347*, 2017.

[34] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[35] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer. Firmfuzz: Automated iot firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 15–21, 2019.

[36] S. Vasile, D. Oswald, and T. Chothia. Breaking all the things-a systematic survey of firmware extraction techniques for iot devices. In *International Conference on Smart Card Research and Advanced Applications*, pages 171–185. Springer, 2018.

[37] W. Xie, Y. Jiang, Y. Tang, N. Ding, and Y. Gao. Vulnerability detection in iot firmware: A survey. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 769–772. IEEE, 2017.

[38] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, pages 1–16, 2014.

[39] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1099–1114, 2019.