Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces [*]

Andrei Costin
EURECOM
costin@eurecom.fr

Apostolis Zarras
Technical University of Munich
zarras@sec.in.tum.de

Aurélien Francillon
EURECOM
francill@eurecom.fr

ABSTRACT

Embedded devices are becoming more widespread, interconnected, and web-enabled than ever. However, recent studies showed that embedded devices are far from being secure. Moreover, many embedded systems rely on web interfaces for user interaction or administration. Web security is still difficult and therefore the web interfaces of embedded systems represent a considerable attack surface.

In this paper, we present the *first fully automated framework* that applies dynamic firmware analysis techniques to achieve, in a scalable manner, automated vulnerability discovery within embedded firmware images. We apply our framework to study the security of embedded web interfaces running in Commercial Off-The-Shelf (COTS) embedded devices, such as routers, DSL/cable modems, VoIP phones, IP/CCTV cameras. We introduce a methodology and implement a scalable framework for discovery of vulnerabilities in embedded web interfaces regardless of the devices' vendor, type, or architecture. To reach this goal, we perform full system emulation to achieve the execution of firmware images in a software-only environment, i.e., without involving any physical embedded devices. Then, we automatically analyze the web interfaces within the firmware using both static and dynamic analysis tools. We also present some interesting case-studies and discuss the main challenges associated with the dynamic analysis of firmware images and their web interfaces and network services. The observations we make in this paper shed light on an important aspect of embedded devices which was not previously studied at a large scale.

1. INTRODUCTION

Embedded devices are present in many complex systems like cars, airplanes, and programmable logic controllers. Such devices also appear massively in customer products such as network gateways and IP cameras. Those devices are becoming more pervasive and "invade" our lives under many different forms (e.g., home automation, smart TVs). Thus, their proper and secure functioning is of great importance.

Embedded systems, in particular Small Office/Home Office (SOHO) devices, are often insecure [54]. Their lack of security may be the consequence of the harsh market competition. For instance, the time to market is crucial and the competition puts high pressure on the design and production costs, and enforces short release timelines. Vendors try to provide as many features as possible to differentiate products, while customers do not necessarily look for the most secure products.

Some embedded systems have clear and well-defined security goals, such as the pay-TV smart cards and the Hardware Security Modules (HSM). However, many embedded systems are not designed with a clear threat model in mind. This gives little motivation to manufacturers to invest time and money in securing them. This fact motivated several researchers to evaluate the state of security of such embedded devices [11,17]. Moreover, during the past few years, embedded devices became more connected forming what is called the Internet of Things (IoT). Such devices are often put online by composition; attaching a communication interface to an existing (insecure) device. Most of these devices lack the user interface of desktop computers (e.g., keyboard, video, mouse), but nevertheless need to be configured and maintained. Albeit some devices rely on custom protocols used by "thick" clients or even legacy interfaces (i.e., telnet), the web quickly became the universal administration interface. Thus, the firmware of these devices often embed a web server running web applications; for the rest of this paper, we will refer to these as *embedded web interfaces*.

It is well known that making secure web applications is not a trivial task. In fact, researchers showed that more than 70% of vulnerabilities are hosted in the (web) application layer [51]. Attackers use various techniques to exploit these web applications. Well known vulnerabilities, such as SQL injection or Cross Site Scripting (XSS), constitute a significant portion of the total amount of vulnerabilities discovered each year [15] and are frequently used in real-world attacks [29]. Additionally, vulnerabilities such as Cross Site Request Forgery (CSRF), command injection, and HTTP response splitting are also often present in web applications. Given such a track record of security problems in both embedded systems and web applications, it is natural to expect the worse from *embedded web interfaces*. However, those vulnerabilities are neither easy to discover, analyze, and confirm, nor do the vendors perform the necessary security quality assurance of their released firmware images.

[*]An extended version of this paper is available at [18].

ASIA CCS '16, May 30-June 03, 2016, Xi'an, China
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4233-9/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2897845.2897900

While there are solutions that can be used during the design phase of the software [52], it is also important to discover and patch existing vulnerabilities before they are found and exploited "in the wild". This can be done by static analysis on their source code [8, 22], or by dynamic analysis where their compiled code or web interface is typically exercised against a number of known attack patterns [9, 11].

Unfortunately, these techniques can be inefficient or difficult to use for detecting vulnerabilities inside embedded web interfaces [9, 30]. For instance, performing static analysis on embedded web interfaces seems to be a rather simple task once the firmware has been unpacked. One main limitation of this approach is that the web interfaces often rely on various technologies (e.g., PHP, CGIs, custom server-side languages). However, the static analysis tools are usually designed for a particular technology and many of them are concentrated around some trendy environment (e.g., PHP) leaving the others "uncovered". In addition, though sound static analysis tools exist, many others are merely "glorified greps" and have a large number of *false positives*, which make them problematic to reliably use in an automated large scale study. On the other hand, dynamic analysis tools [28] are more generic as they are less sensitive to the server-side language. Yet, they require the system or the web interface to be functional. Unfortunately, it is challenging to create an environment that can perfectly emulate firmware images for a broad range of devices based on a variety of computing architectures and hardware designs.

The easiest way to perform dynamic analysis is to do it on a live device. However, acquiring devices to dynamically analyze them is expensive and does not scale. At the same time, it is ethically questionable, if not illegal, to test devices one does not own (e.g., devices on the Internet). Another option is to extract the web interface files from a device and load them to a test environment, like an Apache web server. Unfortunately, a large majority of the embedded web interfaces use native CGIs, bindings to local architecture-dependent tools or custom web server features which cannot be *easily* reproduced in a different environment (Section 2.4.1).

Emulating the firmware is an elegant method to perform dynamic analysis of an embedded system, since it does not require the physical device to be present and can be completely performed in a controlled environment while being easy to scale. But emulation of unknown devices is not easy because an embedded firmware expects specific hardware to be fully present (i.e., peripherals or memory layouts). Previous attempts were made at improving emulation of firmware images by forwarding hardware I/O or `ioctl` to the hardware [45, 57]. These techniques achieve a rather good emulation, but require the presence of the original device and custom manual setup, which does not scale. One observation is that in Linux-based embedded systems the interaction with the hardware is usually performed from the kernel, and their web interfaces often do not interact with the hardware directly.

To perform scalable security testing of embedded web interfaces we developed a distributed framework for automated analysis (Figure 1) and evaluated it in a cloud setup. We started our analysis with a dataset of 1925 unpacked firmware images that contain embedded web interfaces. Then, for each unpacked firmware we identify any potential web document root present inside the firmware. At this point we make a pass with static analysis tools on the modules of the web service under test. Next, we perform a system emulation of firmware images by replacing their kernel with a stock kernel (for the same CPU architecture) and emulating the whole userland of the firmware using the QEMU emulator [27]. We then `chroot` to the unpacked firmware and start the `init` program, the init scripts, or directly the web server in some cases. Once (and if) the web service under test is up and operational, we perform dynamic analysis on it. Finally, we analyze the results, and whenever applicable we perform manual analysis and investigate the failures. Overall, in this paper we present a completely automated framework to perform scalable dynamic firmware analysis and demonstrate its effectiveness by testing the security of embedded web interfaces. Essentially, our framework relies on the emulation of the firmware images, which allows to test the embedded web interfaces using off-the-shelf dynamic analysis tools.

In summary, we make the following main contributions:

- We present the first framework that achieves scalable and automated dynamic analysis of firmwares for discovering vulnerabilities in embedded devices using the software-only approach.
- We highlight the challenges in emulating firmware images and dynamically testing (web interfaces of) embedded systems, and describe the techniques we used.
- We perform the first large scale security study on web interfaces of embedded systems and automatically discover 225 previously unknown serious vulnerabilities in 45 firmware images.

2. EXPLORING METHODS TO ANALYZE EMBEDDED WEB INTERFACES

In this section, we summarize the different possibilities for static and dynamic analysis of embedded web interfaces, their limitations, and motivate our final choices.

2.1 Static Analysis

Static analysis tools have many practical advantages as they are often automated and do not require setting up too complex test environments. In general, they only need the source code (or the application) to be provided to generate an analysis report. It is also relatively easy to plug new static analysis tools for increased coverage or wider support of file formats and source code languages. Finally, as a result of all the above, such tools are scalable and easy to automate.

Static analysis, however, has well understood limitations. Although it cannot find all the vulnerabilities, i.e., *false negatives (FN)*, it may also alert on non-vulnerabilities, i.e., *false positives (FP)*. Additionally, we found that embedded devices' firmware often rely on uncommon technologies for which security static analysis tools do not exist (e.g., *Haserl*, *Lua*, *binary CGIs*). Albeit there exist a number of static analysis tools for PHP [22, 44], in our dataset only 8% of embedded firmware images contain PHP code in their web interfaces. This is not really a surprise since PHP is not primarily designed for embedded systems. We nevertheless analyze these cases with *RIPS* [22]. Finally, *binary static analysis* could be applied to binary CGIs to find vulnerabilities such as buffer overflow, (remote) code execution or command injection [53]. In particular, techniques that aim at supporting the diversity of CPU architectures found in embedded systems start to appear [50].

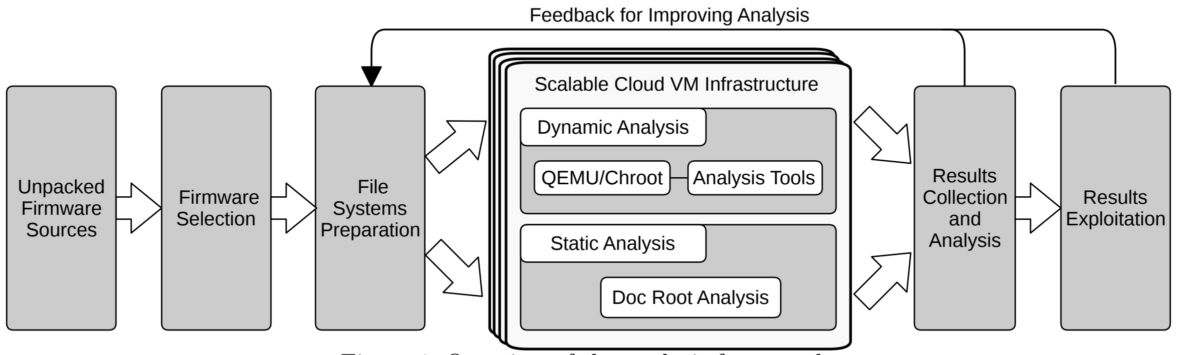

Figure 1: Overview of the analysis framework.

2.2 Dynamic Analysis

Dynamic analysis—an analysis that relies on testing an application by running it—has many benefits. First, dynamic analysis of web interfaces is mostly independent from the server-side technology that is used. For instance, the very same tool can test web interfaces that are implemented in PHP, native CGIs, or custom web scripting engines. Second, it can be used to confirm vulnerabilities found in the static analysis phase. Although many dynamic analysis tools for security testing of web applications exist [9], unfortunately they often require significant effort to setup and customize, by writing new modules for scanning, testing, or validation.

For this study, we focused on web penetration tools that are open source so that we can easily adapt and integrate them in our framework, and fix their defects when required. Based on this, we selected *Arachni* [1], *Zed Attack Proxy (ZAP)* [2], and *w3af* [3]. However, our approach and framework are designed in a way that allows great flexibility. As Figure 3 depicts, other tools such as *Metasploit*, *Nessus* and *Nmap* can supplement or replace the tools mentioned above. In this way, we can achieve additional security and vulnerability testing that can help us increase the surface of vulnerability discovery for known and unknown vulnerabilities.

2.3 Limitations of Analysis Tools

Our framework relies on existing web analysis tools, which have their own limitations. For instance, the number of FPs and FNs of this study is a direct consequence of the tools we rely on. An example of such limitation is their ability to detect or not *command injection* vulnerabilities. Those are frequently missed because such flaws are nontrivial to discover via automated testing [4, 9]. For example, tools may try to inject commands such as `ping <ip>`, assuming that the network is functional and that the target system provides the `ping` tool. We overcome some of these limitations by taking advantage of our “white box” approach (Section 3.4.1).

In addition, the tools we use were not particularly designed to target vulnerabilities in embedded web interfaces or to be integrated in automated frameworks. Therefore, we faced various problems using these tools and this impacted the success rate of the vulnerability discovery. We were able to improve or fix many of them at the cost of a significant engineering effort. Nevertheless, fixing these bugs proved necessary to obtain better results. This highlights that better web application analysis tools are needed, especially ones that are particularly adapted to test embedded web interfaces.

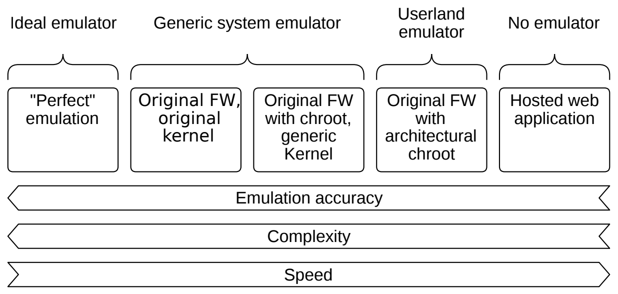

Figure 2: Ways to emulate embedded web interfaces: from perfect emulation of a hardware platform to hosting the web interface. (The arrows show a general increasing trend, actual evolution of the properties may not be linear.)

2.4 Running Web Interfaces

Dynamic analysis of web applications requires a functioning web interface. There are different ways to launch the web interface that is present in the firmware of an embedded system, however, none of them are perfect. Some methods are very accurate but infeasible in our setup, such as emulating the firmware in a perfect emulator—which is not available. Other methods are much less accurate, like extracting the web application files and serving them from a generic web server. Therefore, we evaluated different approaches (Figure 2) and describe their advantages and drawbacks.

2.4.1 Hosting Web Interfaces Non-Natively

A straightforward way to launch a web interface from a firmware is to extract and then launch it under a web server on an analysis environment, without trying to emulate the original web server and firmware. The web application is located (i.e., the document root, as described in Section 3.2.2), extracted, and “transplanted” to the *hosting environment*. The main advantage of this technique is that it does not require emulation, which dramatically simplifies the deployment and thus is easy to automate and scale.

However, this approach has many limitations. For example, it is not possible to handle platform dependent binaries and CGIs. We analyzed the document roots within 1580 firmware candidates for emulation and found that 57% out of these were using binary CGIs or were in some way bound to the hardware platform. In essence, this is a lower bound as we did not count web scripts calling local system utilities, for instance, using the `system()` call. In addition to this, the firmware images often use either customized web servers, or versions which are not available on normal systems, thus a generic web server (e.g., Apache) has to be used in the hosting environment. We evaluated this technique and we present results of its evaluation in Section 5.4, where we also compare its performance to other techniques we used.

2.4.2 *Firmware and Web Interface Emulation*

A preliminary step to emulate a firmware is to know its architecture. While this may seem straightforward, it is in fact a nontrivial step to perform automatically at scale. For instance, some firmware images contain files for various architectures (e.g., ARM and MIPS), where the vendors package two different firmware blobs into a single firmware update package. The firmware updater then picks the right architecture during the upgrade based on the detected hardware. In such cases, we try to emulate this filesystem with each detected architecture. We detect the architecture of each executable in a firmware either using ELF headers (if available) or statistical opcode distribution for raw binaries. We decide on the target architecture by counting the number of architecture-specific binaries and then we use the QEMU version for that particular architecture. Different possibilities for emulating firmware images exists (Figure 2), and we compare them hereafter.

Perfect emulation: Ideally, the firmware would be complete (including the bootloader, kernel, etc.) and a QEMU configuration which emulates the original hardware would be available. However, QEMU emulates only few platforms for each supported CPU architecture. Therefore, emulating unknown hardware is impossible in practice, especially considering that hardware devices can be arbitrarily complex. Furthermore, hardware in embedded devices is often custom and its documentation is not available. Thus, it is infeasible to adapt the emulator, let alone applying it at a large scale.

Original kernel and filesystem on a generic emulator: Reusing the kernel of the firmware could lead to a more accurate emulation, in particular because it may export interfaces for some custom devices that are needed to properly emulate the whole system. Unfortunately, kernels for embedded systems are often customized and hence do not support a wide range of peripherals. Therefore, using the original kernel is unlikely to work very well on a generic emulator. Additionally, in our dataset only 5% of the firmware images contain a kernel, making this approach infeasible.

Firmware chroot with a generic kernel and filesystem: Lacking the original kernel, it is still possible to rely on a complete and generic system (targeting the same CPU architecture), which is then used as a base for the firmware analysis. In our framework, we use the pre-compiled releases of Debian Squeeze [43]. From this generic system we `chroot` to the unpacked firmware and execute the shell (e.g., `/bin/sh`) or the init binary (e.g., `/sbin/init`). Finally, we start the web server's binary along with the web interface document root and web configuration.

It should be possible to directly boot the firmware filesystem instead. However, using a generic file system provides a consistent environment to control the QEMU Virtual Machines (VM), to perform our analysis, and to monitor the emulated system. As an advantage, this approach allows the emulation of web servers and interfaces in their original file system structure, and can also execute native programs.

This approach, however, has some drawbacks. First, emulating the system is not very fast (i.e., the emulation is one order of magnitude slower than native execution). Additionally, the emulator environment setup and cleanup introduces a significant overhead. Furthermore, with this approach we cannot fully emulate the peripherals and specific kernel extensions of the embedded devices. Even so, few firmware images and a limited part of embedded web interfaces actually interact directly with the peripherals. One such example is a web page that performs a firmware upgrade which in turn requires access to `flash` or `NVRAM` memory peripherals.

Overall, we found this approach to offer the best tradeoff between emulation accuracy, complexity, and speed (see Figure 2). It is also scalable and provided the best results in analyzing dynamically the web interfaces (see Section 5.4).

Architectural chroot: One way to improve the performance and emulation management aspects of our framework is by using *architectural chroot* [5] (also known as *QEMU static chroot*). This technique uses `chroot` to emulate an environment for architectures other than the architecture of the running host itself. This basically relies on the Linux kernel's ability to call an interpreter to execute an ELF executable for a foreign architecture. Registering the userland QEMU as an interpreter allows to transparently execute `ARM` Linux ELF executables on an `x86_64` Linux system. However, we found this approach to be quite unstable, making it a challenge to use it at a large scale. Finally, while this approach has the advantage of improving emulation speed, in essence, it is unlikely to improve the number of firmware images we can finally emulate. Therefore, we did not use this technique in our setup, and we leave this for future work.

3. ANALYSIS FRAMEWORK DETAILS

To perform a large scale and automatic analysis of firmware images we designed a scalable framework (Figure 1). First, we obtain a set of unpacked firmware images, which we analyze and filter (Section 3.1). Next, we perform some pre-processing of the selected unpacked firmware images, as some of them are incompletely unpacked or the location of the document root is not obvious (Section 3.2). We then perform static and dynamic analysis (Section 3.3). Finally, we collect and analyze the reported vulnerabilities (Section 3.4) and exploit these results (Section 3.5).

3.1 Firmware Selection

The firmware selection works as follows. First, we select the firmware images that are successfully unpacked and are Linux-based systems which we can natively emulate and `chroot` with QEMU (Section 3.2). Second, we select firmware instances that clearly contain web server binaries (e.g., `httpd`, `lighttpd`) and typical web configuration files (e.g., `boa.conf`, `lighttpd.conf`). In addition to these, we select firmware images that include server-side or client-side code related to web interfaces (e.g., `HTML`, `JavaScript`, `PHP`, `Perl`). Our dataset is detailed in Section 4.

3.2 Filesystem Preparation

To emulate a firmware the emulator requires its root filesystem. In the simplest case the unpacked firmware directly contains the root filesystem. However, in many cases the firmware images are packed in different and complex ways. For instance, a firmware can contain two root filesystems, one for the upgrade and one for the factory restore, or it can be packed in multiple layers of archives along with other resources. Thus, we first need to detect the potential candidates for root filesystems. We can achieve this by searching for key directories (e.g., `/bin/`, `/sbin/`, `/etc/`, `/usr/`) and for key files (e.g., `/init`, `/linuxrc`, `/bin/sh`, `/bin/bash`, `/bin/dash`, `/bin/busybox`). Once we discover such files and folders relative to a directory within the unpacked firmware, we select that particular directory as the *root filesystem* point. There are also cases where it is hard or impossible to detect the root filesystem. A possible reason is that some firmware updates are just partial and do not provide a complete system. We extract each detected root filesystem and pack it as a standalone root filesystem that is a candidate for emulation. But unpacking firmware images can produce "broken" root filesystems which we attempt to fix. Additionally, in order to start the web server within the root filesystem, we need to detect the web server type, its configuration, and its document root. For these reasons, we have to use heuristics on the candidate root filesystems and apply transformations before we can use them for final emulation and analysis.

3.2.1 Filesystem Sanitization

Unpacking firmware images is not a perfect procedure. First, unpacking tools sometimes have defects. Second, some firmware images have to be unpacked using an imperfect "brute force" approach [17]. Finally, some vendors customize archives or filesystem formats. For instance, some filesystems have symbolic links that are incorrectly unpacked because they were represented as text files containing the target of the link (e.g., the symbolic link `/usr/bin/boa -> /bin/busybox` is represented with a single text file named `/usr/bin/boa` and containing the string `/bin/busybox`). All these lead to an incorrect unpacking and thus the unpacked firmware image differs from the filesystem representation intended to be on the device. This reduces the chances of successful emulation and thus we need a sanitization phase.

This sanitization phase is performed by scripts that traverse unpacked firmware filesystems and fix such problems. Sometimes, there are multiple ways to fix a single unpacked firmware, which results in multiple root filesystems submitted for emulation and increases the chances of proper emulation for a given firmware. Implementing these heuristics added a 13% processing overhead. At the same time, it allowed us to increase the successful emulations by 2% and the successful web server launches by 11%.

3.2.2 Web Server Heuristics

Within the firmware, we additionally locate web server binaries and their related configuration files (e.g., `boa.conf`, `lighttpd.conf`). The path of the web server and its configuration file is sufficient to start the web server using a command such as `/bin/boa -f /etc/boa/boa.conf` (i.e., a real example from our dataset). We also extract important settings from the configuration files (e.g., the document root).

Sometimes, we miss important parameters which are required to properly start the web server, such as the document root path or the CGI path. Often this happens because of a missing configuration file (e.g., partial firmware update) or because the parameters are supplied via the command line from a script which is not available. In these cases, we experiment with all the potential document roots found inside the

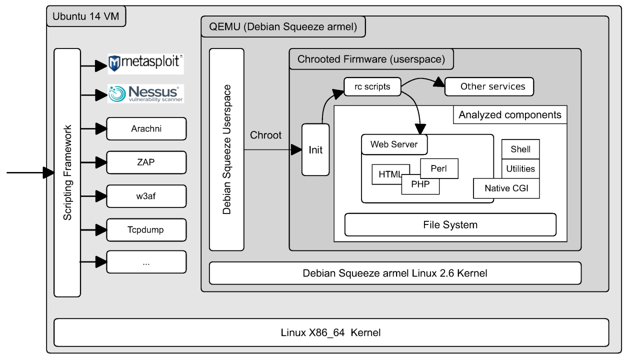

Figure 3: Overview of one analysis environment for Linux armel with a 2.6 kernel.

firmware. To find a potential document root (within the root filesystem) we first search for index files (e.g., `index.html`, `default.html`) with possible file extensions (`.html`, `.shtml`, `.php`, `.asp`, `.cgi`). Then, we build a set of *longest common prefix directories* of these files. This can result in multiple document root directories, for example, a second document root can be found in a recovery partition. Once we discover the document roots, we prepare the possible commands to start the web server. These allow us to increase the chances of bringing the web server up and operational.

We also build an optimized site map for each such document root directory. We use the site maps to hint the dynamic analysis tools which URLs they have to analyze. In general, dynamic analysis tools crawl the web application to discover its site map. However, this is inefficient and can easily miss some pages and even whole sets of vulnerabilities [24]. Thus, for multiple reasons, we instruct the tools to derive their analysis based on the supplied site map. First, it significantly lowers the time required to complete the dynamic analysis. No time is wasted to analyze uninteresting files, such as image files, or to (inefficiently) crawl the web application [24]. Second, it reduces the chances for the web interface or the emulator to crash by limiting the resource load (e.g., number of requested URIs). Third, it increases the chances that the files that are reported as vulnerable by static analysis will also undergo dynamic analysis.

There are several possible improvements to our approach. Restricting the site map may miss URLs when content is dynamically generated or monolithic web server binaries are used. Therefore, we could use the site map as crawling seeds for the tools. Additionally, using a tool like *ConfigRE* [55] could improve the automatic inference for configuration files.

3.3 Analysis Phase

Once we prepare the filesystems, we then emulate each of them in an analysis VM where dynamic testing is performed (Figure 3 and Section 2.2). We also submit the document roots to the static analyzers (Section 2.1). This phase is completely automated and scales well as each firmware image can be analyzed independently.

3.4 Results Collection and Analysis

After dynamic and static analysis phases are completed, we obtain the analysis reports from the analysis tools. We also collect log files that can help us make further analysis and improve our framework. These are typically required to debug the analysis tools or our emulation environment. For instance, we collect SSH communication logs with the emulator host, changes in the firmware filesystem, and network traffic of interactions with the web interfaces. Below we discuss these in slightly more details.

3.4.1 Filesystem Changes

We capture a snapshot of the emulated filesystem at several different points in time. We do this (*i*) before starting the emulation, (*ii*) after emulation is started, and (*iii*) after dynamic analysis is completed. Then, we perform a filesystem `diff` among these snapshots. Interesting changes are usually observed when new files are created or within log files of the emulated system. Log files are interesting to collect in case a manual investigation is needed. New files can be the consequence of a *OS command injection* or more generally of a *Remote Code Execution (RCE)* vulnerability triggered during the dynamic analysis phase. This often occurs when dynamic testing tools try to inject commands (e.g., `touch <filename>`). Sometimes, the command injection can be successful but not detected by the analysis tools. However, it is easy to detect such cases with the filesystem `diff`.

3.4.2 Capturing Communications

Performing dynamic analysis involves a lot of input and output data between the (emulated) embedded system and the dynamic analysis tool. Capturing the raw input and output of the communication allows to increase accountability in case of emulation problem. For instance, a successful *OS command injection* can go undetected by the tools. Also, such vulnerability can be difficult to verify, even in a “white box” testing approach (see Section 2.3). Once the testing phase is over, it can be discovered that a command injection was, in fact, successful. In such case, we need to rewind through all HTTP transactions to find the input triggering the particular vulnerability and afterward we can look for incriminating inputs and parameters (e.g., a `touch` command). The testing tools often behave like fuzzers as they try many malformed inputs one after the other. Because of this, a detected vulnerability may not be a direct result of the last input, but the combination of several previous inputs. It is therefore important to recover all these previous inputs in order to successfully reproduce the vulnerability.

3.5 Results Exploitation

After collecting all the details of the analysis phase, we perform several steps to exploit these results. First, we validate the high impact vulnerabilities by hand and try to create a proof-of-concept exploit. We could fully automate this in the future, as performed in other fields of vulnerability research [7]. Unfortunately, none of the tools we currently use provide such functionality. Additionally, from the static analysis reports we manually select the high impact vulnerabilities (e.g., command injection, XSS, CSRF) and the files they impact. We then use these to explicitly drive the dynamic analysis tools and aim mainly at two things: (*i*) get the dynamic analysis tools to find the vulnerabilities they missed (if they did) and (*ii*) find the bugs or limitations that prevented the dynamic tools to discover these vulnerabilities in the first place. Even though manual analysis does not scale, it can help uncover additional nontrivial vulnerabilities. Finally, we summarize all our findings in vulnerability reports to be submitted as CVEs.

Dataset phase	**# of FWs (unique)**	**# of root FS**	**# of vendors (unique)**
Original dataset	1925	–	54
Candidates for chroot and web interface emulation	1580	1754	49
Improved by heuristics	1580	1982	49
Chroot OK	488	–	17
Web server OK	246	–	11
High impact vulnerabilities (static & dynamic)	185	–	13

Table 1: Number of firmware images and corresponding vendors at each phase of the experiment.

Web server	**% among started web servers**
minihttpd	37%
lighttpd	30%
boa	4%
thttpd	3%
empty banner	26%

Table 2: Distribution of web server types among the 246 started web server.

4. DATASET

We started our study with a set of firmware images that we collected over time from publicly available sources. Table 1 presents details about the counts of the firmware images for each of the phases in our framework. First, we chose the firmware instances which were successfully unpacked and which were Linux-based embedded systems (1925). These are the systems which seemed the easiest to emulate. Then, we selected firmware instances that clearly contained a web server binary (e.g., `httpd`, `lighttpd`) and typical configuration files (e.g., `lighttpd.conf`, `boa.conf`). In addition, we chose images that included server-side or client-side code associated with web interfaces (e.g., `HTML`, `JS`, `PHP`, `CGI`). Once we applied all the heuristics to the firmware candidates 1580), we tried to `chroot` them and start their web interface emulation. Unfortunately, we were able to `chroot` only 488 firmware candidates. On top of that, we managed to start the embedded web interfaces for only 246 firmware images which successfully chrooted. Table 2 shows the distribution of the web servers among the successfully chrooted images, while Table 3 exhibits the web technologies these servers use. Finally, we discovered high impact vulnerabilities only in 185 web interfaces that were successfully emulated.

Challenges and Limitations: Inevitably, our dataset and the heuristics we apply lead to a bias. First, the dataset contains firmware images that are publicly available online. Second, Linux-based devices only account for a portion of all embedded systems. Third, because we use pre-compiled Debian Squeeze images, we performed our tests mainly on ARM, MIPS, and MIPSel firmware images. However, as we present in Table 4, adding support for additional architecture should be straightforward, and requires mainly engineering effort. For example, our framework could ideally support the emulation of $\approx 97\%$ of the firmware images in our dataset when pre-compiling Debian for all architectures in Table 4 and using mainline QEMU version with additional patches. Finally, there exist images running as monolithic software or embedding web servers which we currently do not detect or support. In essence, these choices were needed to perform this study and it will be an interesting future work to extend the study to more diverse firmware images.

Web interface contains	**% of started web servers**
HTML	98%
CGI	57%
PHP	2%
Perl	3%
POSIX shell	11%

Table 3: Web technologies used by the started web servers (combinations possible).

Architecture	**QEMU support**	**Original firmware**	**Chroot OK**	**Web server OK**
ARM	mainline	35%	53%	55%
MIPS	mainline	19%	21%	17%
MIPSel	mainline	17%	26%	28%
Axis CRIS	patch [40,41]	16%	–	–
bFLT	mainline	5%	–	–
PowerPC	mainline	3%	–	–
Intel 80386	mainline	2%	–	–
DLink Specific	no	$\approx 1\%$	–	–
Unknown	no	$\approx 1\%$	–	–
Altera Nios II	patch [56]	$\ll 1\%$	–	–
ARC Tangent-A5	no	$\ll 1\%$	–	–
Total	–	**1925**	**488**	**246**

Table 4: Distribution of CPU architectures, QEMU support of those CPUs, and the success rates of `chroot` and web launch for each architecture. (The failure analysis is detailed in Section 5.6.)

5. EVALUATION

In this section, we present the findings from both static and dynamic analysis, and study different aspects of embedded web interfaces that attackers could potentially exploit.

5.1 Summary of Discovered Vulnerabilities

Our automated system performed both static and dynamic analysis of embedded web interfaces inside 1925 firmware images from 54 vendors. We found serious vulnerabilities in at least 45 firmware images out of those 246 for which we were able to emulate the web server. These include 225 *high impact* vulnerabilities found and verified by dynamic analysis. Static analysis reported 145 unique firmware images to expose 9046 possible vulnerabilities. Aggregating static and dynamic analysis reports, a total of 185 firmware images are responsible for 9271 vulnerabilities, affecting nearly a quarter of vendors in our dataset.

5.2 Static Analysis Vulnerabilities

PHP is one of the most used server-side web programming languages [25]. Over the past years, many researchers focused on investigating vulnerabilities in PHP applications and creating static analysis tools [22, 44]. However, to the best of our knowledge, we are the first to study the prevalence of PHP in embedded web interfaces and their security. In our dataset 8% of the embedded firmware images contain PHP code in their server-side. We extracted the PHP source code from these firmware images and analyzed the code using RIPS. More specifically, RIPS reported 145 unique firmware images to contain at least one vulnerability and a total of 9046 reported issues. The detailed breakdown is presented in Table 5. We observe that cross-site scripting and file manipulation constitute the majority of the discovered vulnerabilities, while command injection (one of the most serious vulnerability class) ranks third.

Vulnerability type	**# of issues**	**# of affected FWs**
Cross-site scripting	5000	143
File manipulation	1129	98
Command execution	938	41
File inclusion	513	40
File disclosure	461	87
SQL injection	442	10
Possible flow control	171	56
Code execution	141	21
HTTP response splitting	127	27
Unserialize	119	15
POP gadgets	4	4
HTTP header injection	1	1
Total	**9046**	**145 (unique)**

Table 5: Distribution of PHP vulnerabilities reported by RIPS static analysis. (The typical error rates of each type of vulnerability can be found in [22].)

Vulnerability type	**# of issues**	**# of affected FWs**
Command execution	*51*	*21*
Cross-site scripting	*90*	*32*
CSRF	*84*	*37*
Sub-total HIGH impact	*225*	*45 (unique)*
Cookies w/o HttpOnly †	9	9
No X-Content-Type-Options †	2938	23
No X-Frame-Options †	2893	23
Backup files †	2	1
Application error info †	1	1
Sub-total low impact †	5843	23 (unique)
Total	**6068**	**58 (unique)**

Table 6: Distribution of dynamic analysis vulnerabilities. (The vulnerabilities followed by a "†" sign have low severity and are known to be reported with a very high false positive rate, therefore they are not mentioned elsewhere in this paper, including when we mention a total number of vulnerabilities found.)

5.3 Dynamic Analysis Vulnerabilities

Our framework was able to perform dynamic security testing on 246 distinct web interfaces, and the general results are presented in Table 6. In particular, we discovered 21 firmware images which are vulnerable to command injection. The impact of such vulnerabilities can be significant as a large number of devices may be running these firmware images. Additionally, we found that 32 firmware images are affected by XSS and 37 are vulnerable to CSRF. Even though XSS and CSRF are usually not considered to be critical vulnerabilities, they can have a high impact. For example, Bencsáth et al. [10] were able to completely compromise an embedded device only by using XSS and CSRF flaws.

The above vulnerabilities affect the firmware of multiple type of devices in our dataset, such as SOHO routers, CCTV cameras, smaller WiFi devices (e.g., SD-cards). Leveraging tools such as Shodan [6] or ZMap [26], it is possible to correlate these firmware images to populations of online devices using multiple correlation techniques [17], which we leave for future work.

In summary, we found vulnerabilities in roughly one out of four (24%) of the dynamically tested firmware images, which demonstrates the viability of our approach. In other words, these findings reveal that embedded devices are vulnerable and easy to exploit by the attackers.

	Emulated (unique FWs)	Hosted (unique FWs)	Hosted Contribution (added unique FWs)
Command execution	*21*	*15* †	0
Cross-site scripting	*32*	*13* †	0
CSRF	*37*	*307* †	269
Tested FWs	**246**	**515** †	269
Vulnerable FWs	**45**	**307** †	262

Table 7: Comparison of firmware images affected by high impact vulnerabilities found with the *Emulated and Chroot* method and the ones found with the *Hosted* technique. The firmware and vulnerabilities marked with a "†" are found using the *Hosted* technique, which is not yet integrated in the fully automated framework. Therefore, they are not aggregated elsewhere in this paper, including when we mention a total number of vulnerabilities and affected firmware.

5.4 Evaluating the Hosting Web Interfaces

"Hosting" embedded web interfaces seems a promising approach as it allows testing a web interface without emulating the complete firmware. Indeed, many firmware images are difficult (or even impossible) to emulate. We therefore evaluated the "hosting" approach on all the firmware images from our dataset where our web server heuristics tools could extract a document root (Section 3.2.2). These document roots are "transplanted" into testing hosts running a generic web server (i.e., *Ubuntu 14.10, Kernel 3.13, Apache2, PHP 5.5.9, Perl 5.18.2.*). Then we perform dynamic analysis with the same methodology as for the emulated case.

Table 7 shows the high impact vulnerabilities found in this experiment and also presents a comparison with the emulation approach. We can immediately see that unsurprisingly the "hosted" analysis allows to test web interfaces from many more firmware images, but surprisingly it almost only reports CSRF vulnerabilities. In fact the technique did not allow to detect any new command injection or XSS vulnerabilities. We expect that the lack of results for some categories of vulnerabilities is due to the fact that using the "hosted" approach with a static web server configuration has some undesired side effects. For instance, the headers of the HTTP responses will be different from those of the real device's web server, while these headers may have an important security role (e.g., Cookies w/o HttpOnly, No X-Content-Type-Options, No X-Frame-Options) [59]. In fact, for command execution and XSS vulnerabilities we had to perform manual interventions into the hosting environment to make the web interface more functional. Then, we had to rerun the dynamic analysis to discover a part of vulnerabilities already reported by the fully automated emulation approach. In few instances we had to install additional `apache2` modules, while in some others we had to disable `.htaccess` configuration files which came with the transplanted document roots. Yet in several other cases we had to adjust a variety of `shebang (#!)` paths in the interpreted scripts' headers to point to the correct interpreter path of the hosting environment. These manual interventions clearly do not scale and limit the "hosted" approach. In the future, we plan to address such limitations with approaches to automatically reconfigure the "hosting" environment based on the semantics of the transplanted document root.

Overall, we conclude that both firmware emulation and "hosting" web interfaces are useful and complementary techniques. Moreover, whenever the emulation is possible, it finds more vulnerabilities.

Port type	Port number	Service name	# of FWs
TCP	554	RTSP	91
TCP	555	RTSP	84
TCP	23	Telnet	60
TCP	53	DNS	23
TCP	22	SSH	15
TCP	Others	Others	58
Total			**207 (unique)**

Table 8: Distribution of network services opened by 207 firmware instances out of 488 successfully emulated ones. The last entry summarizes the 16 unusual port numbers opened by services such as web, telnetd, ftp or upnp servers.

5.5 HTTPS and Other Network Services

We also explored how often embedded devices provide HTTPS support. In our dataset, nearly 19% of the original firmware images contained at least one HTTPS certificate. This provides a lower bound estimate of firmware images that could provide a web server with HTTPS support, as some devices may generate certificates dynamically. Similarly, around 24% of the firmware instances starting an HTTP web server, also started an HTTPS one. We also expect this number to be a lower bound estimate as an HTTPS web server might not start for multiple reasons. Still, it is unfortunate that so few devices provide HTTPS support.

While in this paper we focus on the security of web interfaces, we found it interesting to also report on the network services that are automatically started during the dynamic analysis. Indeed, these additional services which we detected using `netstat` may be vulnerable on their own (e.g., `CVE-2007-1435/CVE-2011-4821` for TFTP, `CVE-2010-2965/CVE-2014-0659` for TR-069, `CVE-2014-4880/CVE-2013-1606` for RTSP, `CVE-2014-9222` for Debug). Therefore, we compare the `netstat` output before and after starting the chroot and the init scripts. This provides a precise information on which program is listening and its corresponding port (Table 8).

5.6 Analysis of the Failures

The failures at various stages limit the coverage of the tested firmware images. For example, Table 4 shows that chroot failed for around 69% of the original firmware images, and around 50% of the successfully chrooted firmware images failed to start the embedded web interface. To increase the coverage and hence the chances of finding more vulnerabilities in more firmware images, we have to perform analysis of the failures and improve our framework. Such a *failure diagnosis* is very time consuming as it requires the exploration of the failure symptoms such as message patterns, error codes, unstructured or inconsistent logs. Nevertheless, this information differs from one system (i.e., device, firmware) to another. Ideally, such fixes should resolve the failures permanently. However, in practice failures often reoccur. One reason is that the corrective maintenance activities, failure diagnosis and solution development can take a long effort. Another reason is that the deployed solution is not completely effective. Additionally, failures can become more recurrent in older systems (e.g., old devices and firmware). This is often linked with the program's resistance to change being maximal, also known as *software fatigue* [33]. Moreover, once the firmware complexity grows, human analysts become overloaded with the logging or failure information. Therefore, scalable failure analysis approaches are required.

5.6.1 Failures Analysis

In our experiments, we encountered 1092 firmware images where *chroot* failed and 242 firmware emulations where *web interface* launch failed. However, this is too many failures to analyze manually and the diversity of the systems makes automated log analysis challenging. Hence, we perform the analysis on a sample of failure data. We apply statistical methods with confidence intervals to reason about failures and their root causes, and to find ways to avoid them. For this, we consider a 95% confidence level and a ± 10% confidence interval for the accuracy of estimations. Those parameters allow to provide coarse grained results while remaining within a reasonable number of failures to manually analyze.

Overall, we analyzed the log samples of 88 randomly selected (out of 1092) firmware files that *failed to chroot*. Among them we found actual chroot failures, and cases where chroot was in fact successful but we failed to exploit. In 36 cases the chroot was the actual cause of the failure (which we extrapolate to 40.9% ± 9.8% cases out of 1092). In these sampled logs we found two main reasons of chroot failure.

First, chroot failed for 10 firmware images because of *exec format errors* (extrapolated to 11.3% ± 6.3% out of 1092 firmware images). These failures are the consequence of an incorrect guess of the CPU architecture or due to a `/bin/sh` that contain *illegal instructions*. We believe that these error cases should be relatively *easy* to fix, for instance, by changing the QEMU architecture (e.g., because architecture was improperly detected in the first place) or improving QEMU (e.g., to support or ignore non-standard instructions).

Second, chroot failed for 26 samples because the firmware images were only *partial firmware updates* (which we extrapolate to 29.5% ± 9.1% out of 1092). Such firmware images do not contain any shell or busybox binary that our framework could chroot to. These cases can also be fixed by replacing missing utilities, yet, at this point the firmware under analysis will start to diverge from the actual device's firmware.

The remaining 52 chroot failures were found to be *false positives*. In fact, these firmware images did chroot successfully but our framework failed to detect this. This can occur in systems that set the environment in unusual ways or take long time to respond to chroot and environment queries (timeout). We therefore extrapolate that 59.1% ± 9.8% of 1092 chroot failure cases were in fact successfully chrooted. We estimate that these cases should be relatively easy to fix. The fixes could include more adaptive timeouts, and more robust handling of various shells and environments.

In summary, for the *chroot failed* failures we estimate that 62 samples should be relatively *easy* to fix, meaning that 70.4% (± 9.1%) of the failures would allow the emulation to advance one step further.

Similarly, we analyzed log samples of 69 randomly selected (out of 242) firmware files that successfully chrooted but failed to start the web interface. We found 45 instances where *missing device* were the cause of the failure. Some examples of the missing devices are `eth1`, `br0`, `/dev/gpio`, `/dev/mtdblock0`. We estimate that fixing the *missing devices* in the emulator is generally *hard* (or even impossible) due to the lack of specification datasheets. This means that 65.2% ± 9.5% of these 242 web server failures are in general *hard* to fix. We also found 15 firmware samples that failed or hung during the initialization of the emulation. Some of these errors were *"Init is the parent of all processes"* and *"init: must be run as PID 1"*. The reason for such errors could be the chrooted nature of the emulation. However, we expect this not to be extremely difficult to fix. This translates to 21.8% ± 8.2% of original 242 web interface failure will be *likely easy* to fix. Finally, we identified 9 firmware samples that reached the web interface launch but failed to launch. We estimate that this is the case for 13.0% ± 6.7% of firmware images that produced *web server* failures. Examples of errors are *"(server.c.621) loading plugins finally failed"* and *"(log.c.118) opening errorlog /tmp/log/lighttpd/error.log failed: No such file or directory"*. However, we estimate this failure category can be relatively *easy* to fix as well.

In summary, for the *web server* failures we estimate that 24 samples should be relatively *easy* to fix, meaning that 34.8% (± 9.6%) of the failures would eventually allow the launch of embedded web interfaces.

5.6.2 Failures Correction and Further Improvements

The failure analysis and determination in large-scale deployments [14] can be improved and automated in several ways. One way is to perform log pre-processing, log mining, and analysis. This method often uses clustering and machine learning techniques to classify an unknown execution of the system based on its logs and based on previously seen logs of that system [16, 49]. Filesystem instrumentation is another approach to automate the failure analysis [38]. Such an approach is generic because it does not assume that the system is based on particular components or existing log files. The failure causes are determined by looking at differences between file accesses (e.g., which file, when) under both normal and abnormal executions. However, these approaches assume that there are samples of the analyzed system that run under normal and abnormal conditions. Also, some of these approaches require domain-specific knowledge [48]. These techniques are not trivial to apply in our case, as we aim at emulating unknown firmware images regardless of the type or application domain of the device for which the firmware is intended, and often we do not have samples of a non-failure run of the firmware. Another way to trace and analyze the failures is to use tracing. For example, `strace` is a debugging tool that provides useful diagnostics by tracing system calls and signals. Unfortunately, `strace` is broken for stock kernels 2.6, which also affects the builds for embedded architectures (e.g., ARM, MIPS). Finally, kernel level instrumentation and analysis could be a reliable approach to monitor and detect the failures and their root cause. For example, `Kprobes` [46] can be used to dynamically monitor any kernel routine and collect debugging and performance information non-disruptively. Sadly, `Kprobes` is often not enabled in default kernels we used, but more importantly its support for various CPU architectures is not stable in some old kernel versions that we need to use for emulation.

These limitations do not represent themselves research challenges. However, it takes more than a trivial engineering effort to properly address them, and overcome their effect in a systematic and generic manner. We leave the resolution of such limitations as an engineering challenge for future work. Nevertheless, we plan to make the logs and their manual analysis available at `http://firmware.re`.

6. DISCUSSION

As with any other detection system, our approach has some restrictions. In this section, we discuss the limitations we confronted during our study and propose solutions.

6.1 Limitations of the Emulation Techniques

Although our approach can discover vulnerabilities in embedded web interfaces that run inside an emulated environment, setting up such environments is not always straightforward. We discuss several limitations and outline how they could be handled in the future. In fact, many of these limitations are the results of the failures analyzed in Section 5.6.

6.1.1 Forced Emulation

Even though most of the firmware instances in our dataset are for Linux-based devices, they are quite heterogeneous and their actual binaries vary. Examples include `init` programs having different set of command parameters or strictly requiring to run as PID 0 (which does not work by default in a chrooted environment). Ideally, there should be a simple and uniform way to start the firmware, but this is not the case in practice as devices are very heterogeneous. In addition to this, unless we have access to the bootloader of each individual device, there is no reliable way to reproduce the boot sequence. Obtaining and reverse-engineering the bootloaders themselves is not trivial. This usually requires access to the device, use of physical memory dumping techniques, and manual reverse-engineering, which is outside the scope of this paper. We emulate firmware images by forcefully invoking its default initialization scripts (e.g., `/etc/init`, `/etc/rc`), however, sometimes, these scripts do not exist or fail to execute correctly leading to an incomplete system configuration. For instance, it may fail to mount the `/etc_ro` partition at the `/etc` mount point, and then, the web server is missing some required files (e.g., `/etc/passwd`).

6.1.2 Emulated Web Server Environment

Even if the basic emulation was successful, other problems with the emulated web server environment are common. For example, an emulated web interface return for many requests the HTTP response codes `500 Internal Server Error` or `404 Not Found`. Manual inspection of the cases when code `500` is returned suggests that some scripts or binaries are either missing from the root filesystem or do not have proper permissions. Code `404` was often returned due to the wrong web server configuration file being loaded, which lead to the document root pointing at a wrong directory. To overcome this, we try to emulate the web interface of a firmware using all combinations of the configuration files and document roots we find within the firmware.

6.1.3 Imperfect Emulation

The ability to emulate embedded software in QEMU is incredibly valuable, but comes at a price. One big drawback is that some very basic peripheral devices are missing in the emulated environments. A very common emulation failure is related to the lack of non volatile memories (e.g., NVRAM). Such memories are used by embedded devices to store boot and configuration information. One solution is to have an universal or on-the-fly NVRAM emulator plugged into QEMU, for example instrumented at kernel-level or implemented using *Avatar* [57]. Another approach is to intercept calls to the commonly used `libnvram` functions (such as `nvram_get` and `nvram_set`) and return fake data [21, 34]. While these tools are easy to compile and use, it is not trivial to automatically generate meaningful application data without producing false alerts or breaking the emulation. We plan to integrate these techniques in our future versions.

6.2 Outdated Firmware Versions

The firmware files in our experiments were including both older and newer versions of the firmware images. Therefore, the vulnerabilities that were found may be found on older firmware versions. It is nevertheless important to know and understand how many embedded devices that are vulnerable or have outdated firmware will update their firmware in such a case. On the one hand, many embedded devices are SOHO devices on which the users decide *if and when* they will upgrade their firmware version. On the other hand, researchers showed that even simple improvements, such as changing the default credentials of the embedded devices, are not always applied by the users during long period of times [31]. For example, it was found that 96% of accessible devices having factory default root passwords still remain vulnerable after a period of 4 months [20]. However, downloading a firmware update and updating a device is a more complex task than changing the default credentials. Therefore, unless the devices are connected to the Internet and have an automatic firmware update mechanism in place and enabled, it is reasonable to expect that in practice the firmware updates are applied far less than it would be desirable.

Second, even though the embedded devices should keep their firmware updated, this is not always feasible (e.g., for field-deployed devices). Such devices often cannot be remotely updated and require the physical access of an operator in the field to do so. However, even in such cases the upgrade of the firmware is not always straightforward. Cerrudo [12] showed that in some cases embedded devices could be buried in the roadway, making firmware updates that require physical access very challenging, if not impossible.

Third, even the latest firmware releases could still contain the very same vulnerabilities as the older versions [19]. Therefore, vulnerabilities discovered in older firmware versions can prove extremely useful as direct input or mutation template for testing the latest firmware versions.

In summary, we believe that a security study performed only on the latest firmware releases could provide important details for securing embedded devices (e.g., critical vulnerability discovery, patching 0-days). At the same time, however, such a study would not be completely accurate as many existing devices run outdated firmware versions. Ultimately, the goal of this work is not to find (all) the vulnerabilities in (all) the latest firmware versions. The main goal is to provide a methodology and insights that can be applied on any firmware version of any embedded device in order to automatically discover and verify vulnerabilities inside them, and in particular inside their embedded web interfaces.

6.3 Manual Interventions

Our framework is designed to be as automated as possible. However, manual interventions are sometimes necessary or even desirable. First, for each web server type never seen before we write (as a one time activity) a small tool which will automatically detect, parse, and launch instances of this particular web server. Automation of such a step could be improved, for example, using *ConfigRE* [55]. Second, manual inspection of the results and of the affected software allows to confirm vulnerabilities and sometimes leads to finding new ones. This is part of the power of our methodology, for instance, pointing the finger on likely vulnerable software. In our experience this last phase was very productive as there were only a few FPs left after the dynamic analysis phase.

7. RELATED WORK

Analysis of embedded devices is an active topic. Costin et al. [17] performed a large scale firmware analysis, but they mainly focused on simple static analysis. Zaddach and Costin [58] demonstrated the feasibility of dynamic analysis and vulnerability discovery in embedded devices via firmware emulation using QEMU and custom-built kernels. Bojinov et al. [11] studied the security of embedded management interfaces, but they performed the analysis manually and focused only on 21 devices. Similar studies were recently preformed on popular SOHO devices [37, 42] each performing manual analysis on about ten devices and uncovering flaws in almost all of them. In contrast, we show that by automating the analysis we can scale the analysis to hundreds of devices and find thousands of vulnerabilities.

In addition, several projects scanned the Internet, or parts of it, to discover vulnerabilities in embedded systems [6, 20, 35]. In most cases these approaches lead to discovery of devices with known vulnerabilities such as default passwords or keys, and in several notable cases helped the discovery of new flaws [35]. However, such approaches raise serious ethical problems and in general only allow to find devices that are vulnerable to known or manually found bugs.

Web static analysis is a very active field of research, where Huang et al. [39] were the first to statically search for web vulnerabilities in the context of PHP applications. Balzarotti et al. [8] showed that even if the developer performs certain sanitization on input data, XSS attacks are still possible due to the deficiencies in the sanitization routines. Pixy [44] proposed a technique based on data flow analysis for detecting XSS, SQL, or command injections. RIPS [22], on the other hand, is a static code analysis tool that detects multiple types of injection vulnerabilities. While we could use any of those detection mechanisms we only used RIPS which has low false positives and is still openly available.

Several recent works rely on emulation to discover or verify vulnerabilities in embedded systems. Avatar [57] is a dynamic analysis framework for firmware security testing of embedded devices that executes embedded code inside a QEMU emulator, while the I/O requests to the peripherals are forwarded to the real device attached to the framework. Kammerstetter et al. [45] targeted Linux-based embedded systems that are emulated with a custom kernel which forwards `ioctl` requests to the embedded device that runs the normal kernel. Li et al. [47] proposed a hybrid firmware/hardware emulation framework to achieve confident SoC verification by using a transplanted QEMU at BIOS level to directly emulate devices upon hardware. Unfortunately, these approaches require access to the physical devices, which does not scale as our approach does. Independently and at the same time, Chen et al. [13] proposed an approach for firmware emulation and dynamic analysis similar to ours. Although they build and use instrumented stock kernels, they do not perform extensive web interfaces testing and do not automatically discover new vulnerabilities as they only test for known vulnerabilities, for example using Metasploit. Meanwhile, Firmalice [53] is a static binary analysis framework that supports the analysis of firmware files for embedded devices. It was shown to detect three *known backdoors* in real devices, but it requires manual annotations and is therefore challenging to use in a large scale analysis.

Fong and Okun [30] took a closer look at web application scanners, their functions and definitions, and proposed a taxonomy of software security tools. Bau et al. [9] conducted an evaluation of the state of the art of tools for automated "black box" web application vulnerability testing. While results have shown the promise and effectiveness of such tools, they also uncovered many limitations of existing tools. Similarly, Doupé et al. [24] performed an evaluation of eleven "black box" web pen-testing tools, both open-source and commercial. Authors found that crawling ability is as important and challenging as vulnerability detection techniques and many classes of vulnerabilities are completely overlooked. Holm et al. [36] performed a quantitative evaluation of vulnerability scanning. The authors showed that automated scanning is unable to accurately identify all vulnerabilities, and that the scans of Linux-based hosts (i.e., many embedded systems are known to be Linux-based) are less accurate than of the Windows-based ones. Doupé et al. [23] proposed improvements to such "black box" tools by observing the web application state *from the outside*, which allows them to extend their testing coverage and to precisely control the "black box" web vulnerability scanner. They implemented the technique in a crawler linked to a fuzzing component of an open-source web pen-testing tool. Such improvements to the analysis tools may improve our framework as we can integrate them in our analysis phase.

Finally, Gourdin et al. [32] addressed the challenges of building secure embedded web interfaces with *WebDroid*, the first framework specifically dedicated to this purpose. Such frameworks can be used by the vendors of embedded systems to provide secure web interfaces within their devices.

8. CONCLUSION

In this work, we presented a new methodology to perform large scale security analysis of web interfaces within embedded devices. For this, we designed the first framework that achieves *scalable and automated dynamic analysis of firmwares*, and that was precisely developed to discover vulnerabilities in embedded devices using the software-only approach. Our framework leverages off-the-shelf static and dynamic analysis tools. Because of the limitations in static analysis tools, we created a mechanism for automatic emulation of firmware images. While perfectly emulating unknown hardware will probably remain an open problem, we were able to emulate systems well enough to test the web interfaces of 246 firmware images. Our framework found serious vulnerabilities in at least 24% of the web interfaces we were able to emulate, including 225 *high impact* vulnerabilities found and verified by dynamic analysis. When counting static analysis, 9271 issues were found in 185 firmware images, affecting nearly a quarter of vendors in our dataset. These results show that some embedded systems manufacturers need to start considering security in their software life-cycle, for instance, by using off-the-shelf security scanners as part of their product quality assurance.

ACKNOWLEDGMENTS

The research was partially supported by the German Federal Ministry of Education and Research under grant 16KIS0328 (IUNO).

9. REFERENCES

[1] http://www.arachni-scanner.com/.
[2] https://github.com/zaproxy/zaproxy/.
[3] http://w3af.org/.

[4] http://owasp.org/index.php/Top_10_2013-A1-Injection.
[5] http://www.darrinhodges.com/chroot-voodoo/.
[6] http://www.shodan.io.
[7] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2011.
[8] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy*, 2008.
[9] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *IEEE Symposium on Security and Privacy*, 2010.
[10] B. Bencsáth, L. Buttyán, and T. Paulik. XCS Based Hidden Firmware Modification on Embedded Dievices. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2011.
[11] H. Bojinov, E. Bursztein, E. Lovett, and D. Boneh. Embedded Management Interfaces: Emerging Massive Insecurity. *BlackHat USA*, 2009.
[12] C. Cerrudo. Hacking US (And UK, Australia, France, Etc.) – Traffic Control Systems. http://blog.ioactive.com/2014/04/hacking-us-and-uk-australia-france-etc.html, Apr 2014.
[13] D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.
[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2002.
[15] S. Christey and R. A. Martin. Vulnerability Type Distributions in CVE. *Mitre Report*, 2007.
[16] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, 2004.
[17] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*, 2014.
[18] A. Costin, A. Zarras, and A. Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces (Extended Version). *arXiv*, (arXiv:1511.03609), 2015.
[19] A. Cui, M. Costello, and S. J. Stolfo. When Firmware Modifications Attack: A Case Study of Embedded Exploitation. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.
[20] A. Cui and S. J. Stolfo. A Quantitative Analysis of the Insecurity of Embedded Network Devices: Results of a Wide-Area Scan. In *Annual Computer Security Applications Conference (ACSAC)*, 2010.
[21] Z. Cutlip. Emulating and Debugging Workspace. http://shadow-file.blogspot.fr/2013/12/emulating-and-debugging-workspace.html.
[22] J. Dahse and T. Holz. Simulation of Built-In PHP Features for Precise Static Code Analysis. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.
[23] A. Doupé, L. Cavedon, C. Kruegel, and G. Vigna. Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner. In *USENIX Security Symposium*, 2012.
[24] A. Doupé, M. Cova, and G. Vigna. Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2010.
[25] L. Duflot, Y.-A. Perez, and B. Morin. Netcraft. PHP Usage Stats. http://www.php.net/usage.php, June 2007.
[26] Z. Durumeric, E. Wustrow, and J. A. Halderman. ZMap: Fast Internet-Wide Scanning and Its Security Applications. In *USENIX Security Symposium*, 2013.
[27] F. B. et al. QEMU – Quick EMUlator. http://www.qemu.org.
[28] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *USENIX Security Symposium*, 2010.
[29] Firehost. The Superfecta Report Special Edition. https://www.firehost.com/media/1657954/firehost-superfecta-2013-year-in-review.pdf, 2013.
[30] E. Fong and V. Okun. Web Application Scanners: Definitions and Functions. In *Annual Hawaii International Conference on System Sciences (HICSS)*, 2007.
[31] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J. A. Halderman. Green Lights Forever: Analyzing the Security of Traffic Infrastructure. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2014.
[32] B. Gourdin, C. Soman, H. Bojinov, and E. Bursztein. Toward Secure Embedded Web Interfaces. In *USENIX Security Symposium*, 2011.
[33] P. Grubb and A. A. Takang. *Software Maintenance: Concepts and Practice*. World Scientific, 2003.
[34] C. Heffner. Emulating NVRAM in Qemu. http://www.devttys0.com/2012/03/emulating-nvram-in-qemu/.
[35] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. In *USENIX Security Symposium*, 2012.
[36] H. Holm, T. Sommestad, J. Almroth, and M. Persson. A Quantitative Evaluation of Vulnerability Scanning. *Information Management & Computer Security*, 19(4):231–247, 2011.
[37] HP-Fortify-ShadowLabs. Report: Internet of Things Research Study. http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA5-4759ENW, 2014.
[38] L. Huang and K. Wong. Assisting Failure Diagnosis Through Filesystem Instrumentation. In *Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research*, 2011.
[39] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *International Conference on World Wide Web (WWW)*, 2004.
[40] E. Iglesial. CRIS Target Port of Qemu. http://repo.or.cz/qemu/cris-port.git.
[41] E. Iglesial. Status of CRIS Architecture Support in Linux Kernel. https://lkml.org/lkml/2014/9/15/1082.
[42] Independent Security Evaluators. SOHO Network Equipment (Technical Report), 2013.
[43] A. Jarno. Debian Pre-Compiled Images for QEMU. https://people.debian.org/~aurel32/qemu/.
[44] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy*, 2006.
[45] M. Kammerstetter, C. Platzer, and W. Kastner. PROSPECT – Peripheral Proxying Supported Embedded Code Testing. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2014.
[46] R. Krishnakumar. Kernel Korner: Kprobes-A Kernel Debugger. *Linux Journal*, 2005(133):11, 2005.
[47] H. Li, D. Tong, K. Huang, and X. Cheng. FEMU: A Firmware-Based Emulation Framework for SoC Verification. In *International Conference on Hardware/Software Codesign and System Synthesis*, 2010.
[48] C. Lim, N. Singh, and S. Yajnik. A Log Mining Approach to Failure Analysis of Enterprise Telephony Systems. In *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2008.
[49] T.-T. Y. Lin and D. P. Siewiorek. Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis. *Reliability, IEEE Transactions on*, 39(4):419–432, 1990.
[50] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-Architecture Bug Search in Binary Executables . In *IEEE Symposium on Security and Privacy*, 2015.
[51] J. Prescatore. Gartner, Quoted in ComputerWorld, 2005.
[52] P. Saxena, D. Molnar, and B. Livshits. SCRIPTGARD: Automatic Context-Sensitive Sanitization for Large-Scale Legacy Web Applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.
[53] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice: Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2015.
[54] J. Viega and H. Thompson. The State of Embedded-Device Security (Spoiler Alert: It's Bad). *IEEE Security & Privacy*, 10(5):68–70, 2012.
[55] R. Wang, X. Wang, K. Zhang, and Z. Li. Towards Automatic Reverse Engineering of Software Security Configurations. In *ACM Conference on Computer and Communications Security (CCS)*, 2008.
[56] C. Wulff. Altera NiosII Support. https://lists.gnu.org/archive/html/qemu-devel/2012-09/msg01229.html.
[57] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2014.
[58] J. Zaddach and A. Costin. Embedded Devices Security and Firmware Reverse Engineering. *BlackHat USA*, 2013.
[59] A. Zarras, A. Papadogiannakis, R. Gawlik, and T. Holz. Automated Generation of Models for Fast and Precise Detection of HTTP-Based Malware. In *Annual Conference on Privacy, Security and Trust (PST)*, 2014.