Firmware Entry Points in Memory

Mahmoud Dolah
NYU Tandon School of Engineering

Rafael De Los Santos
NYU Tandon School of Engineering

Abstract

Finding the entry points of firmware in memory is a problem that is featured prominently in many embedded systems. It is hard to find out because the method for it is not standardized and, since firmware works under the operating system, it does not follow the same rules that the OS uses. Also, very few tools exist that can find the entry points for any binary. In this paper, we present a static analysis approach for automating the process of finding the entry point of firmware in memory with a script called Basefind.py, which was created by Michael Coppola. We test his script against OpenWrt binaries compiled to MIPS. By analyzing the contents of a firmware binary, particularly the amount of strings that are referenced by each pointer, we can create a list of potential base addresses and find the entry point. By running Basefind.py against OpenWrt binaries, we will be able to tell if Coppola's technique will work as a generic tool to find the firmware entry points in memory. While Coppola's Basefind.py worked for his purposes, in this paper, we find that it appears to lack the results to be a universal tool and would require many tweaks to meet the standards of the industry.

1 Introduction

Unlike other areas of modern consumer computing, embedded systems suffer from a glaring lack of standardization. When shopping for a home pc or smartphone, for example, one could be relatively confident that most of the devices presented to them share similarities in hardware or software implementation. An analysis of any of these devices may provide the consumer with an intuition about other devices in their category. The same, unfortunately, could not be said for a device running an embedded system or system on a chip. Because of the variety in software and especially hardware implementation techniques, an analysis of, say, a router, fitness tracker, or bluetooth dongle, will often yield little useful information for any kind of large scale study. Whats worse, the analysis techniques and tools used for any particular embedded system often cannot be carried over to another, forcing the analyst to redo, usually from scratch, their work.

When conducting an analysis of a firmware image, we often start with trying to find the base address of the binary. Because no tools (or even methods) exist that apply to a wide range of firmware binaries, and considering it is quite possible that information on the architecture of the device is sparse or poorly documented, researches often have to rely on manual, static analysis. This, in part, also has to do with the fact that quite often the source hardware isn't available, so reliable dynamic analysis is usually out of the question. When attempting to do an analysis on a large body of firmware binaries, like for instance the Open Wrt library, it would almost be impossible to expect a researcher to obtain, and run some kind of dynamic analysis on, every single phsysical device.

What follows includes disassembly in some commercial tool (perhaps IDA Pro, Binary Ninja, or Objdump), followed by a manual analysis of the disassembly for potential base addresses. This is, of course, error prone and time consuming.

Locating the base address of a firmware image opens the door for a wide variety of more in-depth analysis techniques, so it is imperitive that a robust and wide spread solution is discovered. Coppola[2] proposes a method, basefind.py, that we believe has

much promise, and could one day we generalized to support images across a wide range of architectures. Currently, Coppola has used it to succesfully locate the base address of one image, so work will now turn to evaluating this technique on other images and other architures.

2 Related Work

Firmware entry point analysis can be considered one of the many steps in the overall study of embedded systems. As more and more objects are entering the realm of IOT and incorporate embedded systems of their own, the discussion of firmware analysis is already growing to include a wide variety of new topics.

Meier et al[10], Coppola[2], and Zaddach and Costello[15] touch upon the general idea of manually analyzing disassembled firmware images, mainly to extract information from headers and also for use in emulation. Manual analysis is, however, time consuming, considering the amount of information one needs to gather before they may begin to inspect actual code (which will be done with a tool such as IDA Pro).

Because of this, many tools have been developed to automate or at least assist in much of the analysis. Zaddach et al[14] and Costin et al[4] take the dynamic analysis approach, and run a given firmware image within an emulator. Costin et al[4] is of particular interest because its emulation process includes a partial emulation of the underlying hardware, meaning of course that only firmware for known hardware devices can be tested.

This lack of environmental knowledge (and often physical hardware) prompts many static approaches, such as Shoshitaishvili et al[11], which builds off of techniques in Kruegel et al[9] to reverse engineer firmware when only a binary-blob is present. This brand of analysis is especially useful, considering firmware entry point detection is usually a step in the process.

Similar efforts and challenges in static analysis are seen in Costin et al[3], but on a much larger scale. Tens of thousands of images were unpacked, requiring the analysis to be general enough for a large range of binaries. The results were compared to the popular firmware analysis tool, Binwalk[7], which is often used in manual static analysis. Heffner[7] uses known patterns to locate structures within executables, and can be particularly useful for locating firmware headers.

Heffner[8] and Viehbock[13] show examples of static analysis on firmware using Heffner[7] as well as other useful techniques for reverse engineering embedded systems.

Other techniques have been developed to target specific device vulnerabilities, rather than overarching analyses of firmware images. Meier[10] and Cui et al[5] focus on the exploitation of firmware updates to deliver malicious code (to fitness trackers and HP printers, respectively), and note a general lacking in the current state of firmware security. Similarly, FIE[6] is a tool developed to find bugs in the MSP430 family of micro controllers, building off of Cadar et al[1] for the purposes of conducting symbolic execution. While these analyses are limited in scope, they all shed light into interesting ways of inspecting firmware that can potentially be applied to a greater and more general range of devices.

3 Background

Firmware is a type of software that provides control over products and systems. In this paper, the firmware we analyzed is called OpenWrt, which is piece of firmware for routers. A router is an example of an embedded system. An embedded system is a computer system that acts with a dedicated function within a larger system.

Firmware is considered difficult to operate on because it acts under the operating system and does not have uniform standards and guidelines. Also, in some cases, firmware represents a security vulnerability because, unlike most modern operating systems, firmware rarely has an automatic mechanism for updating itself[12].

The base address is an absolute address that acts as a reference point for other addresses. Finding the base address is important for fully disassembling firmware in order to create a full analysis of a system.

OpenWrt is an embedded, Linux based, operating system that is used to route network traffic. It can be run on many different types of hardware architectures, such as ARM and MIPS. It is an open source project that has been under active development for over twelve years. The reason that OpenWrt was chosen for this paper is because, similar to the GNU Coreutils, it has been thoroughly tested and analyzed by other developers. Because of this, we can be confident that the results that we get from experimentation will not come from errors associated with the code.

Static program analysis is the analysis of software that is performed without actually executing the program. The information gained from static program analysis is useful and it can vary from highlighting potential code errors to formal methods that mathematically prove properties about a given program to ensure that its behaviour matches its specifications.

4 Design

Basefind.py, created by Coppola[2], is a relatively small python script that was successfully used to located the entry point of a firmware binary. As mentioned in Coppola's lecture, Basefind.py begins by locating all strings within a given executable. It then does another pass over the file, this time to locate every double word, which it will then consider as a potential base address. For each potential address, Basefind.py attempts to locate the number of strings referenced. The address that references the most strings is meant to be considered as the base address, but Basefind.py will output the 20 highest ranked addresses in case the first is incorrect.

Of course, this does not remove the need for manual analysis. A person still needs to evaluate each of these base address candidates to see if there is a match, and it is possible that all 20 could be negative. Assuming there is a match, Basefind.py does however drastically reduce the amount of time an analyst needs to spend on this first step.

5 Implementation

Coppola[2] originally tested Basefind.py on an arm binary. Our analysis attempts to run the same script on other binaries to evaluate whether or not this is a useful method towards generalizing Coppola's approach. Our sample space of binaries includes Coppola's original binary and MIPS binaries from Open WRT. Specifically, some of the MIPS binaries from Open WRT were compiled in ELF format, which is helpful for decompiling and checking the results of Basefind.py.

Each binary was scanned with Basefind.py. For binaries compiled to ELF, the resulting base addresses were checked against a readelf decompilation. Running Basefind.py on non-ELF binaries was still useful to evaluate the runtime of Basefind.py.

6 Evaluation

The following tables show the results of running Basefind.py on our selected files. The first table shows the file that was tested and general information about it. The table directly underneath it shows the results from Basefind.py.

File Name	ar71xxnboot.bin
File Size	1 MB
Execution Time	2.3 hrs
Base Address (if exists)	Cannot Get

Candidates	No. Pointers
0xfe0000	259
0x1fe0000	218
0xfde000	217
0x1fde000	203
0x210e0000	173
0x210dd000	167
0x2125d000	134
0x21260000	134
0x211e1000	67
0x211e4000	67
0xfdd000	63
0x211e0000	60
0x211dd000	59
0x217e5000	58
0x21861000	54
0x217e8000	50
0x21864000	46
0xfe8000	45
0x210e3000	45
0x212dd000	4

File Name	vmlinuxlzma.elf
File Size	2 MB
Execution Time	27.5 hrs
Base Address (if exists)	0x80050000

Candidates	No. Pointers
0xfef000	3
0xffd000	3
0x327af000	2
0x45a000	1
0x4da000	1
0x530000	1
0x5a6000	1
0x5ea000	1
0x648000	1
0x677000	1
0x67b000	1
0x769000	1
0x7ca000	1
0xd7a000	1
0xe1c000	1
0xff0000	1
0x1789000	1
0x19ba000	1

File Name	uImagelzma.bin
File Size	2 MB
Execution Time	13.4 hrs
Base Address (if exists)	Cannot Get

Candidates	No. Pointers
0x106d000	1
0x1448000	1
0x1ed4000	1
0x2c30000	1
0x32fd000	1
0x3da4000	1
0x43d5000	1
0x460d000	1
0x4f47000	1
0x5999000	1
0x6634000	1
0x75b7000	1
0x8ddb000	1
0x8f19000	1
0x9318000	1
0x93f9000	1
0x9a9f000	1
0x9e65000	1
0xa55b000	1
0xb01b000	1

File Name	vmlinux.elf
File Size	5 MB
Execution Time	11.3 hrs
Base Address (if exists)	0x8005f000

Candidates	No. Pointers
0x7ceb000	14036
0x7c6f000	13120
0x7cde000	13024
0x7cdb000	13005
0x7cfc000	12917
0x7c79000	12670
0x7c76000	12662
0x20eff000	8974
0x20e62000	8779
0xcb2000	6720
0x1bc65000	6438
0x20e01000	6214
0x20f01000	6202
0x1fcd1000	6067
0xfce5000	5846
0x17c70000	5733
0x17cf5000	5724
0x20dea000	5638
0x13c65000	5545
0x20dc0000	553

The results from running Basefind.py on our selected files were less useful than we originally expected. Notice that for two of the files, Basefind.py found candidate base addresses that referenced at most 3 strings. The results for ar71xxnboot.bin, however, more closely resembled Coppola's results, but because that file was not in ELF format we were not able to easily verify the validity of any of the candidate addresses.

The running time of Basefind.py is also of relative concern. While running Basefind.py on ar71xxnboot.bin took only 2 hours, executing it on all other files took over 10 hours, with vmlinuxlzma.elf taking almost 28 hours. We speculated originally that Basefind.py could be incorporated into other static analysis tools as an intial step in their process, but its slow running times would make it difficult for large scale analysis.

The final step of our evaluation of Basefind.py was to test the validity of its results. In the cases of vmlinuxlzma.elf and vmlinux.elf, analysing the files with readelf yielded different results for their potential base addresses.

7 Discussion

The results from our analysis with Basefind.py show unfortunate shortcomings when using it as an approach to determining the entry points of firmware images. While the runtime of Basefind.py is worrying, our true concern comes from the apparent inaccuracy of its results. We believe this may be because Basefind.py was created for determining the base address of a specific ARM binary, while we tested it on binaries of other architectures. Basefind.py may be relying on specific structures or artifacts found within binaries for ARM that simply do not exist in other formats. Because Basefind.py was successful in Coppola's case, it may be possible to generalize the approach.

8 Limitations and Future Work

In this paper, while we only tested ELF files due to the ease of finding the base address of an ELF file, we ran Basefind.py against other types of binary files. However, it is clear that the script only works on ELF files, regardless of the hardware architecture that it was compiled for.

In the future, we could perform more tests on the rest of the openwrt binaries, instead of just ELF files. The same tests can, also, be performed on binaries that were compiled for other architectures, aside from ARM and MIPS.

Another future improvement would be to optimized the script to decrease the run time. The average run time was considerably larger than predicted and lead to several setbacks. One way to optimize Basefind.py would be to use heuristics to determine attributes of the binary, which could speed up the run time. Also, after examining the results given from Basefind.py after testing it on other types of binary files, it is clear that it only works for ELF files. In the future, we would generalize the script to work on other types of binaries.

Finally, there are plans to turn this into a some type of module or library that can be integrated into currently existing software. It could be turned into a python package that could be called within other

static analysis programs.

9 Conclusion

Finding the entry points of firmware in memory is a very difficult problem and there lacks a simple universal solution. Coppola's Basefind.py is only able to find the base address for ARM binaries and, despite the method being very similar for other binary types and hardware architectures, it proved to be poor solution as a general tool which is sorely needed by the industry.

References

[1] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. 2008.

[2] M. Coppola. Weighing in on issues with cloud scale. 2013.

[3] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. 2014.

[4] A. Costin, A. Zarras, and A. Francillon. Automated dynamic firmware analysis at scale: A case study on embedded web interfaces. 2015.

[5] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack:a case study of embedded exploitation.

[6] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. 2013.

[7] C. Heffner. Binwalk.

[8] C. Heffner. Reversing the wrt120ns firmware obfuscation, 2014.

[9] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. 2002.

[10] M. Meier, D. Reinhardt, and S. WendZel. Attacks on fitness trackers revisited: A case-study of unfit firmware security. 2016.

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

[12] M. Shuttlesworth. Acpi, firmware, and you security, 2014.

[13] S. Viehbock. Reverse engineering an obfuscated firmware image e01 unpacking, 2011.

[14] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems firmwares. 2011.

[15] J. Zaddach and A. Costello. Embedded devices security firmware reverse engineering. 2013.