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. 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. TO DO: EVALUATION AND TAKEAWAY

1 Introduction

Unlike other areas of modern consumer computing, embeded 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, unfortunatly, could not be said for a device running an embeded 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 embeded 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), followed by a manual analysis of the disasembly 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.

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

7 Discussion

OPTIONAL Place to explain if there is something weird about the result

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 can 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 large setbacks. 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.

9 Conclusion

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.