MACH2: System for Root Cause Analysis of Kernel Vulnerabilities

Sidhesh Desai

Faculty Member 1 (Advisor):

Printed: Brendan Saltaformaggio

Signature: DocuSigned by: Brendan Saltaformaggio 69E014578DCB45E...

Faculty Member 2:

Printed: Frank Li

Signature: DocuSigned by: Frank Li 889E9685C85D429...

Introduction

Technology has become an increasingly integral part of everyday life, and though this increase in technology is seemingly purely beneficial, the software underneath it implies an underlying risk for vulnerabilities or malicious intent [9]. For this reason, it is crucial to ensure the security of modern day devices and software, including those within the realm of Internet of Things (IoT); keeping devices secure means that the users of these devices are safe from potential threats. The current study leverages dynamic analysis to find bugs and vulnerabilities in modern Linux kernels and device drivers and pinpoint the actual cause of these issues in order for developers to easily remediate them, allowing for an increase in the security level of the software.

One of the key methods for testing the software on hardware devices, also known as firmware, is dynamic analysis [2]. Dynamic analysis allows security researchers to modify code as it executes and use these changes to discover vulnerabilities and unknown parts of a given piece of source code [2]. This is one of the main ways fuzzing, which is the general name for the process by which tests are automatically generated to find vulnerabilities, can be done [7]. However, there exist other ways in which fuzzing can be done, including simply exploring the input space of a program [7].

Existing tools have handled firmware fuzzing through a variety of approaches but the current state-of-the-art tools have limitations with regard to tracing down the source of the vulnerability and indicating how it can be patched, both of which are necessary for ensuring that kernel/device security can be improved in a robust and efficient fashion. The current study addresses these issues by building upon the current standards of firmware emulation tools by allowing for root cause analysis of bugs and exploits in firmware. The second key step of the current study is to utilize this modified tool to test real-world firmware, not only in order to evaluate results of the tools, but also to discover the root cause of potential vulnerabilities, bugs, or other issues in kernel code that require remediation.

Building a framework for scalable and efficient kernel code testing and root-cause analysis is important in ensuring that kernels and kernel drivers can be reliably tested and the weaknesses of a given piece of software can be found. This process forms the basis through which these security issues can be fixed and through which both researchers and manufacturers can ensure that consumers are able to have a high level of security in their devices and products.

Literature Review

As an increasing number of devices begin to stay connected with the internet, the risks of firmware to both average and commercial users will only continue to increase [13]. This poses a serious threat because malicious or exploitable firmware can not only harm the hardware of the device it is in, but can also infect external systems through both wired and wireless methods [6][9]. Therefore, in order to ensure that firmware is safe, it needs to be extensively tested for vulnerabilities and harmful code, which has been done in a number of ways including static and dynamic analysis. There are a wide variety of tools that use these analysis methods in the context of firmware, but ultimately, in the context of the current project, the most important is kernel fuzzers.

Static analysis, which is based on analysis of code itself, is one of the most fundamental methods for firmware testing, but it has been proven to be limited in its ability to cover code and discover all possible vulnerabilities [11]. For example, Moser, Kreugel, and Kirda have created a method to encode constant values via the 3-SAT problem to allow for the creation of code that is easily runnable but which is almost impossible for a static analyzer to examine because of the time constraints and complications of its looping [11]. By not analyzing firmware during its execution and only examining the code itself, static analysis has a ceiling on its effectiveness for analyzing firmware [11].

Dynamic analysis through dynamic symbolic execution addresses these weaknesses by executing a program and viewing code as a directed graph through which a series of certain inputs can take you to a certain point in the code [2]. Then, from these symbolic expressions and their path conditions, new inputs are generated by substituting these values into automated theorem provers, which effectively evaluate slightly modified versions of the path conditions and expressions to determine the inputs necessary to access a new, unexplored point in the control flow of the program [2][14]. This dynamic exploration of code allows for more thorough analysis and coverage of unknown software [11]. One of the commonly used tools through which general dynamic analysis can be performed is Angr [1].

This can be applied in the context of firmware dynamic analysis, where the firmware needs to be scaled so that it can be executed on more powerful hardware and can be decoupled from peripherals. One of the initial steps in this direction was Firmadyne, which hosts the firmware on QEMU with a custom Linux kernel, from where the firmware can be emulated and dynamically analyzed [3][4]. From the hosted firmware, vulnerability scanners like Metasploit can be run in order to analyze the code safety [4]. However, this approach has a crucial flaw in that its execution is frequently halted by errors which drastically limit the scope of what tests it can run without failing [8]. To combat this, arbitrated execution, which introduces the concept of abstracting the firmware, can be used by pausing the emulated execution at times of errors rather than making the emulation completely accurate to the actual execution [8]. With this technique, key issues during booting, network access, and web availability which could not be handled through Firmadyne can be handled gracefully, opening up more areas for dynamic testing [8]. This shows the initial effectiveness of introducing abstraction into the process of firmware emulation.

One of the crucial subsequent problems with these hosting methods is that firmware frequently interacts with hardware and IO so the emulation of those components is necessary to scale the entirety of firmware for testing. One example of this issue is with DMA, which restricts the efficiency of emulation by serving as a limiting factor. An approach to this created by Mera, Feng, Lu, and Kirda emulates DMA channels instead of allowing for actual DMA by identifying DMA calls and routing them to an auxiliary program to handle inputs. In this way, custom inputs can be fed into the firmware in a scalable manner [10]. HALucinator is a more recent extension of this which utilizes breakpoints for methods requiring peripherals and then forwards these methods to peripheral servers which then return the IO output desired by the tester [5]. However, HALucinator brings in other benefits with regards to scalability by using HALs to implement these breakpoints, which previous iterations of similar emulation tools like Avatar2 have not done [5][12]. These HALs allow for generalized implementation of custom methods that do not depend on the exact nature of the hardware since they do not rely on the base-level firmware [5].

In addition to these tools for analysis via firmware emulation, dynamic and static analysis can be used in the context of fuzzing [7]. Current state-of-the-art kernel fuzzers such as Syzkaller use coverage-guided fuzzing with continuously generated syscalls that can explore the breadth of a given piece of kernel code, finding vulnerabilities in the process [15]. This process has proved to be quite successful in finding kernel vulnerabilities, with close to one hundred new bugs found each month [15]. However, the critical weakness of Syzkaller is that the bugs are purely provided in the form of an exploit, which provides little to no context to the developers or maintainers of a kernel or kernel driver to patch them. This weakness has led to the accumulation of many unpatched vulnerabilities that are still publicly accessible. It follows that there exist opportunities to build upon this research and address the weaknesses of current fuzzing approaches in order to create a more cohesive, usable solution.

The current study aims to specifically build upon the work of existing kernel fuzzing techniques, extending their capabilities to build a more capable system, particularly with regard to root cause analysis. With a completely optimal system, the ability to find and patch the source of exploits would be greatly improved, and firmware security would be able to be enforced to a much higher degree.

Materials and Methods

The process through which MACH2 achieves its root cause analysis of firmware involves multiple components. The two key phases of the process are the process of generating traces and the process of using these traces for root cause analysis. Finally, there is the integration of MACH2 into the overarching process of testing a given piece of firmware.

In order to generate traces, MACH2 utilizes QEMU with GDB. First, for a given exploit and corresponding piece of kernel code, the MACH2 system builds the given kernel or driver based on the relevant configuration file and executes the exploit at hand inside of QEMU. Then, as this execution occurs a trace of the program is generated with the help of GDB. These traces are not made with basic blocks, however, and are instead from translational blocks due to the many interrupts during a kernel startup process. Because the second phase of the process will reference these translational blocks, each is written to a separate file which can be referenced from memory later.

For the second phase of the process, MACH2 uses these traces for root cause analysis. Firstly, the initial inputs are dynamically represented, or tainted, so that they can be traced dynamically through the execution of the code. Then, the code is stepped through a dynamic symbolic execution engine, Angr, from where it can be dynamically analyzed. As opposed to traditional dynamic symbolic execution where multiple paths are explored, the MACH2 system only needs to explore the path given in the trace, leading through a significantly streamlined passthrough of the code. This is because the system already knows where the exploit occurs, so there is no need to use resources on unguided exploration of the remaining code. Throughout this dynamic execution process, whenever unseen addresses are hit, the relevant translational blocks are pulled from memory to be referenced by the engine and to allow the execution to continue. This process repeats until the end of the inputted trace is reached. Finally, based on what parts of memory and variables are tainted when the execution is completed, the MACH2 system works backwards to see what parts of the initial memory caused those parts of the output to be tainted. This effectively allows for the discovery of where in the initial kernel source code the bug is located. For example, with a buffer overflow, when the buffer is overflowed with tainted memory at the end of the execution, leading to a crash, the memory will be traced back to the input variable that allowed for the overflow.

Together, these two pieces of trace generation and root cause analysis form the core functionality for MACH2; however, the process of testing a firmware sample involves a few more steps. The first step of the overarching test process is to receive an exploit or bug that causes a crash; this can be generated manually or through a fuzzer. Then, this same exploit is run within MACH2's custom QEMU to generate the trace, as mentioned in the first step of the process. Next, this trace is fed into the second step of the process to find the root cause. Because we are only dynamically following the trace of the exploit that led to a crash, we have a known terminating point of our trace. Finally, the root cause discovered by the MACH2 system can be used to patch the bug at hand or inform relevant developers about where bugs should be patched.

Results

Though still in development, MACH2 is able to generate accurate traces from existing PoCs and vulnerabilities that have been documented. These traces are then able to be effectively analyzed through dynamic analysis to find the root cause of the given bug. The MACH2 root cause analysis has not yet been generalized to incorporate dynamic memory, but once it has done so, it will be able to pinpoint the root cause of a much wider suite of vulnerability classes.

In its current state, however, the MACH2 system has still been able to demonstrate its usability, for example with CVE-2020-15437, which is a null pointer dereference on the Linux Serial 8250 driver. On this CVE, MACH2 was able to successfully rerun the exploit up until it caused a kernel panic, and generate the trace to that point. The trace was then run within MACH2, which was able to find the instruction that caused the vulnerability, from where it is able to be patched. Overall this process is far simpler than dealing with the complications of manually finding the root cause from the kernel exploit.

MACH2 is also currently being tested against Syzkaller Bug #edf4cdf300eae7bc895016091520e5ca5fd2fb6a, which is a divide-by-zero error requiring dynamic memory for tracing and Syzkaller Bug #392ce929bb0e269e6782a4d0586e5f187b9e4d92, which is an index-out-of-bounds error also requiring dynamic memory. Strong results on these two vulnerabilities would indicate that MACH2 is applicable in the full context of dynamic tracing and can therefore be generalized to many of the bug classes which can be detected through dynamic symbolic execution. Thus, MACH2 would be able to pinpoint the root cause of a variety of open vulnerabilities and allow them to finally be patched efficiently.

Conclusion / Next Steps

MACH2 is a promising step forward to finding the root cause of kernel POCs and vulnerabilities and finding zero-day bugs in kernels or kernel drivers. With MACH2, many of the currently unpatched vulnerabilities that have been found for kernels and drivers can get simplified solutions pinpointing the location of the issue and the fix to be made, which could drastically streamline the bug-fixing process in modern firmware. The primary next steps for MACH2 is to achieve general applicability among a wider suite of bugs so that a larger number of CVEs can be traced back to their root cause and patched. In the future, MACH2 aims to be able to find new kernel bugs and their root causes simultaneously, primarily through the integration of a custom dynamic symbolic execution tool.

References

[1] Angr Documentation. (n.d.). https://angr.io/

[2] Ball, T., Daniel, J., & Ball, T. (2015). Deconstructing dynamic symbolic execution. Proceedings of the 2014 Marktoberdorf Summer School on Dependable Software Systems Engineering. https://www.microsoft.com/en-us/research/publication/deconstructing-dynamic-symbolic-execution/

[3] Bellard, F. (2005). QEMU, a fast and portable dynamic translator. Proceedings of the Annual Conference on USENIX Annual Technical Conference, 41.

[4] Chen, D. D., Egele, M., Woo, M., & Brumley, D. (2016). Towards automated dynamic analysis for Linux-based embedded firmware. Proceedings 2016 Network and Distributed System Security Symposium. Network and Distributed System Security Symposium, San Diego, CA. https://doi.org/10.14722/ndss.2016.23415

[5] Clements, A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., & Payer, M. (2020). HALucinator: Firmware re-hosting through abstraction layer emulation. 29th USENIX Security Symposium (USENIX Security 20) (pp. 1201–1218). USENIX Association.

[6] Costin, Andrei & Zaddach, Jonas. (2013). Embedded devices security and firmware reverse engineering. BH13US Workshop. http://s3.eurecom.fr/docs/bh13us_zaddach.pdf

[7] Godefroid, P. (2020, March 04). Fuzzing: Using automated testing to identify security bugs in software.
https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/

[8] Kim, M., Kim, D., Kim, E., Kim, S., Jang, Y., & Kim, Y. (2020). FirmAE: Towards large-scale emulation of IOT firmware for dynamic analysis. Annual Computer Security Applications Conference, 733–745. https://doi.org/10.1145/3427228.3427294

[9] Maskiewicz, J., Ellis, B., Mouradian, J., & Shacham, H. (2014). Mouse trap: Exploiting firmware updates in USB peripherals. 8th USENIX Workshop on Offensive Technologies (WOOT 14).
https://www.usenix.org/conference/woot14/workshop-program/presentation/maskiewicz

[10] Mera, A., Feng, B., Lu, L., & Kirda, E. (2021). DICE: Automatic emulation of DMA input channels for dynamic firmware analysis. ArXiv:2007.01502 [Cs].
http://arxiv.org/abs/2007.01502

[11] Moser, A., Kruegel, C., & Kirda, E. (2007). Limits of static analysis for malware detection. Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), 421–430. https://doi.org/10.1109/ACSAC.2007.21

[12] Muench, M., Nisi, D., Francillon, A., & Balzarotti, D. (2018). Avatar2: A multi-target orchestration platform. Proceedings 2018 Workshop on Binary Analysis Research. Workshop on Binary Analysis Research, San Diego, CA. https://doi.org/10.14722/bar.2018.23017

[13] Neagu, C. (2021, January 29). What is firmware? What does firmware do? Digital Citizen. https://www.digitalcitizen.life/simple-questions-what-firmware-what-does-it-do

[14] Salwan, J., & Saudel, F. (2015). Triton: Concolic Execution Framework. Symposium sur la sécurité des technologies de l'information et des communications (SSTIC). http://shell-storm.org/talks/SSTIC2015_French_Paper_Triton_Framework_dexecution_Concolique_FSaudel_JSalwan.pdf

[15] Vyukov, D. (2020). Syzkaller: Adventures in continuous coverage-guided kernel fuzzing. Microsoft BlueHat.