Linux Kernel Vulnerabilities: Defence Against the Dark Arts

Blair Andrews – z5557932

Assessment 3 Project Report – Operating System Fundamentals

Abstract – The Linux kernel is a component of the core of a system, and it is imperative that this is protected, and that security is ensured. This report critically analyses an important paper that covers the many vulnerabilities found within the Linux kernel. There were 141 vulnerabilities found, and 83 of them were related to consequences of using C: an unsafe language. This paper conceptualises and follows recent research based on utilising type-safe languages to develop operating systems, such as Rust. Further analysis is conducted, evaluating Rust compared to C and the benefits and limitations of Rust being implemented into the Linux kernel. The findings of this paper recommend the best course of action for vastly improving security within the Linux kernel is by implementing type-safe languages into development of the Linux kernel.

I. Introduction

Linux has grown vastly in popularity for its customisability and modularity since its birth in 1994, and now 47% of professional developers use Linux-based operating systems [1]. Due to its vast usage, it is imperative to analyse its security, and any security vulnerabilities Linux may present. We are going to critically analyse and review a paper by Haogeng Chen et al. "Linux kernel vulnerabilities: State-of-the-art defenses and open problems" [2] which covers vulnerabilities found in the Linux kernel between 2010-2011. After reviewing and analysing the paper, this report is going to delve into researching the impact that our analysed paper had on future research on the security of the Linux kernel, and furthermore how this paper contributed to important discussion on otherwise previously undiscussed topics. Finally, this report will then discuss avenues for further research, as well as recommendations of solutions for the overarching question: "How can we improve the security of Linux without affecting its performance?".

II. Technical Review

A. Linux Kernel Vulnerabilities

This paper focused on the known research gap at the time – there were very few studies on Linux kernel vulnerabilities in practice, as well as a broader lack of understanding of mitigating those kernel vulnerabilities. A study by Arnold et al. [3], it was stated that every kernel bug should be treated as security-critical and to be patched as soon as possible, which is why it is concerning to learn that one-third of the 141 bugs that led to vulnerabilities were found in the core of the Linux kernel. The other two-thirds were found in kernel modules or drivers. As mentioned, the study found that there were 141 total bugs that led to vulnerabilities within the Linux kernel, which can be summarised in this list:

- Uninitialised data 29
- NULL dereference 20
- Integer overflow 19
- Missing permission checks 17
- Buffer overflow 15
- Memory mismanagement 10

- Missing pointer checks 8
- Data race / deadlock 8
- Miscellaneous (other types of vulnerabilities) 8
- Divide by zero 4
- Infinite loop 3

Uninitialised data describes bugs where the kernel copies contents of a kernel buffer without zeroing unused fields – which could leak sensitive information such as variables on the kernel stack. This vulnerability was found most frequently, and while is an attack itself could also lead to enabling other attacks such as side-channel attacks – where an attacker collects information outputted from processes to learn behaviours and randomness to learn sensitive information about a system [4].

NULL dereference is a bug that describes an application dereferencing (access data associated with a pointer [5]) a pointer that it expects to be a valid pointer, but is NULL, which typically causes a crash or forced exit [6]. This vulnerability can typically be associated with a DoS (Denial-of-Service) attack as an attacker can force-interrupt services that have this vulnerability.

Integer and buffer overflows are bugs that describe the kernel incorrectly checking the bounds in which a variable is set in when it is accessed by a buffer. An attacker could trick the kernel into executing malicious code by intentionally accessing memory outside of the bounds of variables within the kernel.

Missing permission checks is a bug that relates to the kernel performing privileged operations without checking if the process requesting the operation has the privilege to perform it. This vulnerability can result in privilege escalation attacks and malicious code execution.

Several more vulnerabilities are described in this paper that pertain to critical kernel vulnerabilities, and an important observation in this paper states that 78% of memory corruption exploits are caused by buffer and integer overflows – which make up 34 of the 114 critical bugs found in the kernel. From these observations we can identify the ramifications of using insecure languages – as Linux is programmed primarily in C [7] which is a universally unsafe language *especially* when it comes to, conveniently, all the bugs found in the Linux kernel – which is also described in this paper. Although, this comes with a challenge as contending with C and rewriting a whole operating system as new in a type-safe language such as Rust would prove to be a major and infeasible challenge. However, it is important to note that developers of the Linux kernel are experimenting with the production of Rust in the kernel which could see favourable results in terms of kernel security [7].

B. State-of-the-Art Prevention Tools

The paper then ventures into state-of-the-art prevention tools used to mitigate only a small subset of the vulnerabilities found – which is an open problem that the paper has addressed. The prevention tools are split into two categories being *runtime tools* and *compile-time tools*.

Runtime tools begin with *software fault isolation* which describes a tool known as BGI (Fast Byte-Granularity Software Fault Isolation) [8] that isolates kernel modules and ensures that processes do not have unnecessary privileges within the kernel – especially when writing or freeing memory. This tool prevents vulnerable modules from performing buffer overflow attacks on the kernel itself. However, the paper emphasises that BGI only handles vulnerabilities within a module and certain attacks that attempt to cross over to the kernel. It is a very specific tool and *is not* an all-encompassing tool for memory protection. We then venture into *code integrity* which describes a hypervisor known as SecVisor within the kernel that authenticates all code prior to execution [9], which as the paper describes, makes exploits more difficult to write, but does not prevent them entirely. *User-level drivers* describe the tool SUD which runs device drivers in the user space and prevents vulnerabilities from the driver affecting the kernel [10]. The paper addresses an important note with SUD that it turns most of the vulnerabilities found in the driver into DoS attacks, that crashes the driver itself – which is another important reason to modularise the kernel, and processes. Raksha is a *memory tagging* tool that detects when kernel code misuses trusted inputs which can prevent code injection attacks [11]. *Uninitialised memory tracking* is an issue described as uninitialised data, and the paper

describes two tools to mitigate this vulnerability: Kmemcheck which detects unitialised memory vulnerabilities [12], and SD (Secure Deallocation) which zeroes out the kernel stack to prevent information leaks [13]. The paper emphasises the fact that these tools do not *completely* mitigate the vulnerabilities listed but make them more difficult to exploit. Some of these tools have the potential to cause other vulnerabilities such as user-level drivers.

Compile-time tools describe code-analysis tools which can pinpoint vulnerabilities in operating system code. The paper describes how many tools like these described have been used in the development of the Linux kernel, but a main limitation is the large number of false positives that they result in. This results in programmers needing to filter out false positives from real vulnerabilities and introduces a whole new layer of potential error where a real vulnerability could be incorrectly mislabelled as a false positive. These tools also struggle to identify *semantic vulnerabilities*, which typically fall under DoS vulnerabilities, and missing permission checks, which this paper labels as a current open problem.

C. Open Problems

I. Addressed Concepts

The paper then identifies three key open concepts in the security of the Linux kernel that have been previously addressed in other research papers:

1. The vulnerabilities found are widely spread across the whole Linux kernel (drivers, kernel modules, core kernel code).
2. Many of the preventative runtime tools focus on only mitigating a small subset of vulnerabilities found – and most do not completely prevent that vulnerability from being exploited.
3. The Linux kernel is written in the unsafe language C. The concept of incrementally rewriting modules and drivers in the Linux kernel in a typesafe language is an open issue regarding availability of systems – and is purely in experimental stages.

II. Unaddressed Concepts

The paper then goes on to describe concepts that have not been addressed, or are rarely addressed, in other research papers.

1. Semantic vulnerabilities like missing permission checks within applications.
2. Denial-of-Service vulnerabilities.

These concepts are rarely addressed in previous research as they are not direct sources of harm toward the kernel. However, these vulnerabilities should be seen as a gateway toward kernel weakness, which in turn can directly affect the kernel, making it easier to attack. More importantly, it is difficult to mitigate these vulnerabilities as the prevention tools listed above do not have the capabilities of preventing semantic or DoS vulnerabilities, but also it is difficult to maintain availability and integrity of the kernel if these vulnerabilities exist. It is the programmer's responsibility to maintain policies and ensure that processes have necessary permissions added or revoked.

The paper concludes that the Linux kernel has a far way to go when discussing the security of the kernel. The main issues identified were that 141 critical vulnerabilities within the Linux kernel were discovered, and the state-of-the-art prevention tools provided to mitigate them only mitigated a small subset of the vulnerabilities found.

III. RESEARCH EVALUATION

This paper had a profound perspective on the security of the Linux kernel, especially for a paper written in 2011. There were three key research papers identified that followed similar topics – a paper by Kwangsun Ko et al. [14] touched on source-code level vulnerabilities within the kernel but did not go into detail on specific types of vulnerabilities, nor semantic vulnerabilities. As well as a paper published by Jaekwang Kim et al. [15] which described mechanisms for finding kernel vulnerabilities, but not vulnerabilities themselves. Finally, a paper published by R. Wita et al. [16] did go into levels of detail about Linux kernel vulnerabilities, but again, did not discuss the concepts of semantic vulnerabilities. This gives the paper that has been thoroughly analysed

in this report suitable novelty as it appropriately researched and discussed topics that had not been covered previously, as well as went into concise quantitative detail regarding specific exploits that occur due to specific vulnerabilities. The paper also discussed prevention tools and their limitations which were not previously covered in other research papers found to be published prior to 2011.

The paper is written concisely and discusses the honest limitations of the security of the Linux kernel at the time, which attributes to its quality. The paper's approach to focusing on the open problems of the Linux kernel security stimulated discussion and further research and left a large impact on future research regarding this.

IV. IMPACTS

Since this paper was published in 2011, it has been cited by 216 papers which shows its profound impact on future research. In a presentation by C. Guiffrida et al. [17] this paper is referenced in deeper discussion about kernel-level exploitation. This presentation delved into ASR (fine-grained address space randomisation) which mitigates against runtime attacks and increases the difficulty for performing exploits on vulnerabilities within the core kernel code. Another work that cites our analysed paper is a paper published by A. Balasubramanian et al. [18] which discussed the new type-safe programming language (at the time) Rust. As mentioned previously, Rust is an extremely similar language compared to C, which is what the Linux kernel is primarily written in – and Rust could be a valuable language to write OSes in as it protects against buffer overflow without runtime overhead [19]. This paper was important in the contribution to the discussion of the development of Linux in a type-safe language, and the paper we analysed gave a great contribution to it as it was one of the only papers at the time that discussed C being not type-safe and how that has adversely affected the security of the Linux kernel.

A major contribution the analysed paper made was specifically in the realm of semantic vulnerabilities, as mentioned previously other research papers at the time did not discuss semantic vulnerabilities that weren't direct programming errors, but more logical errors that would lead to DoS attacks or missing permission checks. A paper by Nezer J. Zaidenberg et al. [20] discusses LgDb, a proof-of-concept tool that can detect vulnerabilities during the development phase of the Linux kernel and would mitigate semantic vulnerabilities as discussed by the paper we analysed. This key contribution as well as the previously discussed highlights how much of a contribution to the discussion of kernel vulnerabilities the paper made to future avenues of research.

V. FUTURE RESEARCH

It is imperative that focus should be placed on security during the development phase of an operating system, as it appears that countless bugs are found after a release. OS updates need to be rigorously tested – especially any new code, and anything that interacts with it – to ensure that there are no new vulnerabilities being presented to the userbase. Along with that, current research suggests using type-safe languages such as Rust instead of C to develop operating systems, as noted previously there were 141 critical bugs found in the Linux kernel. Furthermore, 83 of those bugs are directly related to the usage of C and its poor memory management and synchronisation issues. The type-safety features of Rust as a programming language prevent critical bugs such as buffer overflow, NULL dereferences, integer overflow, and many others relating to critical errors in memory management.

The concept of developing Linux kernel modules in Rust has been a topic of recent discussion, beginning first in 2017 with a conceptualisation on improving security and reliability within Linux by using Rust, but also understanding the limitations – such as increased overhead, complexity, and high cost [18]. The concept then shifted into practicality, as RFL (Rust-for-Linux) began to be developed and implemented into Linux as a driver to allow for writing Rust to the Linux kernel [21]. This technology is a leap forward for the implementation of Rust in the Linux kernel, and for improving the security of the kernel altogether while maintaining performance and control. It should be emphasised that although Rust – and RFL as a subsidiary –

aims to bring memory integrity and further security with zero overhead compared to C. However, studies show that while Rust does provide substantial security improvements – which again, must be noted that it provides *substantial* security but not *total* security - it shows to be inefficient and has larger performance overhead compared to C when working with large configurations, and has further shown that Rust performs poorly in memory-intensive workloads [21]. The goal of Rust introducing zero performance overhead did seem infeasible, especially because of how Rust handles boundary checks in arrays which introduces extra performance cost. The latency increase is marginal, however, as it comes down to milliseconds. An argument must be made where scrutinising differences in performance when it's as marginal as this would prove to be detrimental, especially when discussing operating system security.

Future research and development within the Linux kernel has been slowly progressing with the use of RFL, but there has yet to be major implementation. The vulnerabilities found originally in 2011 by our analysed paper are still present in today's Linux kernel. Other avenues of improving security within the Linux kernel seem to be inefficient, as previously discussed, a large majority of the critical bugs in the Linux kernel that lead to vulnerabilities are related to memory mismanagement due to the C programming language, and so focus should be purely on finding an efficient, type-safe language that can be developed in Linux. Rust is the closest, but as we can ascertain it brings performance overhead. Potentially in the future a new programming language will reach the original goal of RFL – vastly improved security with *zero* overhead. An important discussion must be had on this topic: how much performance are we willing to sacrifice for maximisation of security? Is it purely dependent on scenario, or is there a fixed balance we must find between the two? How much do we *truly* value security?

VI. CONCLUSION

Through thorough critical analysis of the paper by Haogeng Chen et al. [2] we can determine that the security of the Linux kernel is lacklustre, with 141 critical bugs leading to vulnerabilities being discovered, and that state-of-the-art prevention tools are not exactly state-of-the-art – only covering a small subset of vulnerabilities found. This led us into the argument that Linux is written in a type-unsafe language – and noting that a large portion of the bugs found in the kernel are related to memory mismanagement bugs, we can ascertain that utilising a type-safe language would be highly beneficial to improving the security of the Linux kernel. Rust has been discussed as a potential suitor for the type-safe language to replace C, however, Rust provides consistent results with increased performance overhead compared to C. Focus should be placed on implementing a type-safe language into the Linux kernel, and further research should be undertaken to find the balance between security and performance – if there is an appropriate balance between both to begin with.

VII. REFLECTION

This research report was an opportunity to delve deeply into the concepts that we learned during the hexamester, and allowed me to perform critical research into the Linux kernel, as well as understand vulnerabilities that could affect the Linux kernel and acknowledge the main contributor to those vulnerabilities – which I found was the programming language that Linux was programmed in, C. I was provided a strong opportunity to view the capabilities and limitations of Rust as an 'improved' version of C and acknowledge the current research being conducted of this topic. Overall, I feel that I learned a lot from this assessment, and it boosted my passion for understanding more about Linux and its innerworkings.

VIII. REFERENCES

[1] L. S. Vailshery, "PC operating system distribution for software development worldwide in 2018 to 2022," Statista, 2022. [Online]. Available: https://www.statista.com/statistics/869211/w

orldwide-software-development-operating-system/. [Accessed 11 August 2024].

[2] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich and M. F. Kaashoek, “Linux kernel vulnerabilities: state-of-the-art defenses and open problems,” in *Proceedings of the Second Asia-Pacific Workshop on Systems*, New York, 2011.

[3] A. J, A. T, D. W, P. G, E. N, T. G and K. A, “Security impact ratings considered harmful.,” in *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, 2009.

[4] S. Chen, R. Wang, X. Wang and K. Zhang, “Side-Channel Leaks in Web Applications: a Reality Today, a Challenge Tomorrow,” Microsoft Research, Bloomington, 2016.

[5] GNU Manual, “Pointer Dereference (GNU C Language Manual),” [Online]. Available: https://www.gnu.org/software/c-intro-and-ref/manual/html_node/Pointer-Dereference.html. [Accessed 8 August 2024].

[6] OWASP Foundation, “Null Dereference | OWASP Foundation,” 20 April 2024. [Online]. Available: https://owasp.org/www-community/vulnerabilities/Null_Dereference. [Accessed 8 August 2024].

[7] kernel.org, “Programming Language - The Linux Kernel documentation,” [Online]. Available: https://www.kernel.org/doc/html/latest/process/programming-language.html. [Accessed 8 August 2024].

[8] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham and R. Black, “Fast Byte-Granularity Software Fault Isolation,” Microsoft Research, Cambridge, 2009.

[9] A. Seshadri, M. Luk, N. Qu and A. Perrig, “SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes,” 2019.

[10] s0ftproject, “sud - superuser daemon,” [Online]. Available: https://s0ftpj.org/projects/sud/index.htm. [Accessed 9 August 2024].

[11] M. Dalton, H. Kannan and C. Kozyrakis, “Real-world buffer overflow protection for userspace and kernelspace,” in *Proceedings of the 19th Usenix Security Symposium*, San Jose, 2008.

[12] V. Nossum, “Getting started with kmemcheck,” The Linux Kernel, [Online]. Available: https://www.kernel.org/doc/html/v4.12/dev-tools/kmemcheck.html. [Accessed 9 August 2024].

[13] C. Jim, P. Ben, G. Tal and R. Mendel, “Shredding your garbage: reducing data lifetime through secure deallocation,” in *Proceedings of the 14th Usenix Security Symposium*, Baltimore, 2005.

[14] K. Ko, I.-S. Jang, Y.-h. Kang, J.-S. Lee and Y. I. Eom, “Characteristic Classification and Correlational Analysis of Source-level Vulnerabilities in Linux Kernel,” Sungkyunkwan University, 2005.

[15] J. Kim and J.-H. Lee, “A methodology for finding source-level vulnerabilities of the Linux kernel variables,” in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, Hong Kong, 2008.

[16] W. R and T.-A. Y, “Vulnerability profile for Linux,” in *19th International Conference on Advanced Information Networking and Applications (AINA'05) Volume 1 (AINA papers)*, Taipei, 2005.

[17] C. Giuffrida, A. Kuijsten and A. S. Tanenbaum, “Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization,” in *21st USENIX Security Symposium*, Bellevue, 2012.

[18] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamaric and L. Ryzhyk, “System Programming in Rust: Beyond Safety,” *HotOS '17: Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pp. 156-161, 2017.

[19] The Rustonomicon, “Meet Safe and Unsafe - The Rustonomicon,” rust-lang.org, [Online]. Available: https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html. [Accessed 9 August 2024].

[20] N. J. Zaidenberg and E. Khen, “Detecting Kernel Vulnerabilities During the Development Phase,” in *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, New York, 2015.

[21] H. Li, L. Guo, Y. Yang, S. Wang and M. Xu, “An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise,” in *Proceedings of the 2024 USENIX Annual Technical Conference*, Santa Clara, 2024.