eBPF-Based Approach to Tracing System Calls and Predicting Privilege Escalation Attacks

Fábio Junior Bertinatto, Daniel Arioza Almeida,
Jéferson Nobre, Lisandro Z Granville
Institute of Informatics - Federal University of Rio Grande do Sul
Porto Alegre, Brazil
{fabio.bertinatto, daniel.almeida, jcnobre, granville}@inf.ufrgs.br

***Abstract*—The extensive adoption of containerized applications significantly raises the criticality of managing potential vulnerabilities, including privilege escalation within these environments. While the Bag of System Calls (BoSC) is a common technique to detect such attacks, tracing system calls in containerized applications is often inefficient for real-world scenarios. This paper proposes an eBPF-based solution to trace system calls in containerized applications and apply the BoSC technique to identify privilege escalation attempts within containers. We analyzed the cost of different system call hooking methods and found that raw tracepoint programs have the least overhead. Furthermore, we observed a slight increase in overhead when tracing all executed operations within a containerized application. Finally, we confirmed that our solution successfully identifies user efforts to escape containers, concluding that eBPF can be a powerful tool for containerized system security.**

***Index Terms*—containers, ebpf, bag of system calls, privilege escalation**

I. Introduction

Linux containers have become increasingly popular in recent years. Much of this popularity is due to the ability to package and isolate applications in a portable manner in containers. In 2022, a survey conducted by CNCF (Cloud Native Computing Foundation) [1] showed that 44% of the respondents reported using containers for almost all production applications in their organizations. 35% of the respondents use containers for a few production systems and 9% are actively evaluating the technology. Recent data [2] suggest that containerization is used by almost half of organizations as their main tool for deploying and managing applications.

The widespread use of containerized applications has increased the urgency to manage vulnerabilities in these environments, especially the escalation of privileges. Recent data from a 2023 Aqua Security report indicates that approximately 50% of organizations that use containers faced a significant security breach in the previous year, with many incidents directly related to privilege escalation [3].

Privilege escalation, in which unauthorized access extends beyond initial permissions, severely threatens the security of the container and its host [4]. Especially concerning is the possibility that a breach in one container can cascade to a system-wide incident [5]. This underscores the vital importance of robust security mechanisms, complemented by continuous auditing and monitoring in containerized infrastructures [6].

Techniques such as the *Bag of System Calls* (BoSC) [7] have emerged as powerful tools to predict and prevent privilege escalation attacks. After collecting and analyzing system calls, the BoSC technique can be used to train classifiers to identify malicious actions, offering a strengthened security layer in containerized settings [8] [9] [10] [11]. A vital prerequisite to using the BoSC technique is to trace all system calls executed by a monitored application. However, collecting those system calls is known to be a costly task. This task is typically achieved with tools such as *strace*[1], which are known to cause considerable slowdowns in the execution of applications.

In this paper, we propose a solution to this problem using eBPF[2], a Linux technology utilized for kernel instrumentation. We compare various types of eBPF programs and evaluate their impact when tracing system calls. Afterward, present an approach to capture system calls executed within the container and apply the BoSC technique to identify potential privilege escalation attempts.

The remainder of this paper is structured as follows. Section II provides an overview of Linux containers and eBPF. Section III discusses the current state of the art in detecting the escalation of privileges in containers. Our proposed solution is presented in Section IV, while Section V present our implementation and evaluation, respectively. Finally, Section VI concludes the paper with final remarks and a discussion of future work.

II. Background

Containers are now an essential part of scalable application deployment due to their ability to isolate resources and simplify packaging and deployment. In addition, eBPF has emerged as a powerful tool for monitoring and securing containerized environments. This section provides a comprehensive understanding of these technologies and explains how they work together to offer robust, scalable, and secure solutions.

A. Linux Containers

A container is essentially a process that runs on a host machine but with enhanced isolation features [12]. Linux-based containers achieve this isolation through three primary kernel features: *namespaces*, *cGroups*, and *seccomp*. *Namespaces*

[1]https://strace.io/
[2]https://ebpf.io/

isolate various system resources, such as the network stack and process tree, ensuring that containers cannot interact with each other's processes [13]. *cGroups* control the allocation of hardware resources such as RAM and CPU to each container. *Seccomp*, or Secure Computing, restricts the system calls that a container can make, thus limiting its access to the host system.

Although containers offer numerous advantages regarding scalability and resource optimization, they are not without security challenges. One of the most critical security concerns in containerized environments is privilege escalation, where attackers gain unauthorized access to resources beyond their initial permission level. These vulnerabilities can compromise the affected container, the host system, and the entire container cluster. Given the severity and potential impact of privilege escalation attacks in containers, it is crucial to understand how to effectively prevent, detect, and mitigate such risks.

B. eBPF

The BSD Packet Filter (BPF) [14] was introduced in 1992 to enable packet filtering within the kernel of Unix BSD systems. BPF introduced a virtual machine (VM) equipped with a Just-In-Time (JIT) compilation engine and a straightforward instruction set. The Linux operating system kernel has supported BPF since version 2.5. In kernel version 3.15, a new variant of the BPF language was introduced. This variant expanded the number of available registers from two to ten, introduced additional instructions, enabled the invocation of a controlled set of kernel instructions, and included various other improvements. These advances transformed BPF into a versatile in-kernel virtual machine. This new variant was named Extended BPF (eBPF) [15], [16].

Initially, eBPF was mainly associated with fast packet processing [17] and network monitoring [18]. However, subsequent advances have made eBPF a valuable program tracing, profiling, and debugging tool. With the introduction of eBPF, the Linux operating system provides an extensive range of methods to carry out tracing operations [19]. eBPF programs are executed upon triggering predefined hook points within the kernel or an application. These hook points can encompass various events, such as network events, system calls, and kernel tracepoints, among others. Additionally, it is feasible to create kernel or user probes to associate eBPF programs with locations where a predefined hook does not already exist [20].

Figure 1 illustrates the typical flow of an eBPF program. Initially, the user writes an eBPF program in pseudo-C code format. The program is then compiled by a compiler, often LLVM *clang*, into bytecode. Subsequently, another program employs a system call to load the bytecode into the kernel. Within the kernel, the bytecode undergoes a verification process to ensure the safety of the eBPF program. After successful verification, the bytecode proceeds through a Just-in-Time (JIT) compilation process, ultimately preparing it for execution.

As of version 6.1 of the Linux kernel, there are 32 different types of eBPF programs [21, Version6.1, `include/uapi/linux/bpf.h`, Line 948]. For this work, the most relevant type of eBPF programs are **BPF_PROG_TYPE_KPROBE** (kprobe), **BPF_PROG_TYPE_TRACEPOINT** (tracepoint) and **BPF_PROG_TYPE_RAW_TRACEPOINT** (raw tracepoint). A kprobe program allows dynamic hooking into any kernel function. Similarly, tracepoint programs can be attached to statically predefined hooks (i.e., tracepoints) in the kernel. Finally, raw tracepoint programs are very similar to tracepoint programs but are more flexible and require manual parsing of parameter fields. For that reason, they often perform better.

III. Related Work

This section expands the discussion to address the existing literature that explores the application of eBPF and other related techniques to mitigate security challenges in Linux container environments. Specifically, the focus is on studies investigating the detection of privilege escalation and security in containers. Additionally, approaches employing anomaly detection techniques, such as the Bag of System Calls (BoSC), are examined, as they have demonstrated efficacy in identifying malicious behaviors in containerized contexts.

The Bag of System Calls (BoSC) methodology has gained traction as an effective approach to intrusion detection, particularly in identifying privilege escalation attempts within Linux containers [8]. Unlike traditional sequence-based approaches, which rely on the specific order of system calls [11], BoSC treats each system call as an individual entity within a *bag* [9]. This flexibility enables the creation of robust and adaptable models capable of capturing nuanced behavior patterns of applications [10]. By employing Machine Learning (ML) classifiers trained in these BoSCs, the system can effectively identify general anomalies and specific malicious activities aimed at elevating privileges [22]. This makes BoSC an invaluable asset for real-time security monitoring, as it can rapidly analyze system calls without being hindered by the need to maintain their sequential order [8].

Abed et al. [8] explored the use of the BoSCs technique to identify behavioral anomalies in Linux containers. By collecting system call traces through the *strace* tool and comparing them against a pre-established database of normal behavior, the study aimed to detect anomalies. However, a notable limitation is that this approach is reactive and identifies anomalies only after an attack.

Castanhel et al. [23] investigated the potential of analyzing system call sequences to detect security threats in containerized applications. The authors used ML algorithms to assess system call sequences against a custom dataset comprising normal and anomalous behaviors. Their findings suggest that filtering out harmless system calls, as classified by the framework proposed by [24], slightly improves detection accuracy.

Previous studies [8], [23] have highlighted various techniques and approaches for detecting security threats and malicious activities in containerized applications. Among these techniques, system call analysis, particularly the BoSC technique, has emerged as an effective strategy in different contexts. Our current proposal aims to address a gap in the existing literature

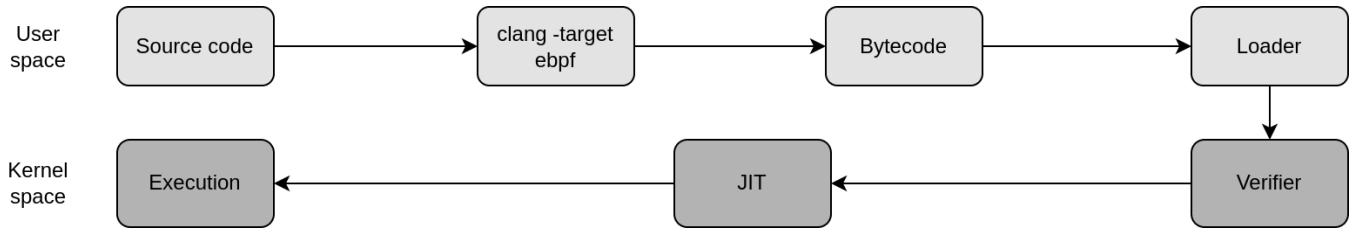

Fig. 1. eBPF program workflow.

by evaluating the most efficient method for tracing system calls in containers and using it to apply the BoSC technique.

IV. Proposal

Our solution encompasses two distinct components operating both in the kernel space and user space. The user space component serves as a communication channel between the kernel space component and the user. It detects and notifies the user any anomalies identified by the kernel space component.

The kernel space component forms the central part of our solution. It receives events for all system call executions by hooking into the *raw_tracepoint/sys_enter* raw tracepoint. In addition, this component will parse and construct system call parameter fields. The extra work of constructing the parameter fields is needed because raw tracepoint programs do not get that information ready for use. Even though this is an inconvenience for the programmer, this is the main reason raw tracepoint programs are the fastest type of eBPF program available. The following section, Section V, presents some data on this matter.

Furthermore, the program keeps track of the last *N* system calls executed to allow us to apply the BoSC technique and look for suspicious sets of system calls. This number must only be large enough to hold the bag of system calls. As a result, we chose to store the last 100 calls, as that number should be enough for our experiments. Algorithm1 presents our method for analyzing system calls in real-time, which is the subject of this section.

Algorithm 1 Search for Anomalous Set of System Calls

```
1: function SearchForAnomalousSyscallsSet
2:     syscall ← context.current_syscall()
3:     latest_syscalls[index mod 100] ← syscall
4:     index ← index + 1
5:     if latest_syscalls contains anomalous_set then
6:         return True
7:     else
8:         return False
9:     end if
10: end function
```

A. Implementation

The kernel space component is developed in C and is classified as a **BPF_PROG_TYPE_RAW_TRACEPOINT** type. The user space component is written in Go and relies on github.com/cilium/ebpf[3] library to load and attach the kernel space component to the *raw_tracepoint/sys_enter* tracepoint. In addition, we use the tooling provided by this library to compile the C source file from the kernel space component into eBPF bytecode and emit a Go file to be used by the user space component.

The successful operation of our solution depends on the ability of both kernel and user space components to exchange information with each other effectively. Therefore, communication between elements of our solution is done via a hash table implemented as an eBPF map of type *BPF_MAP_TYPE_ARRAY*. The kernel space component uses this map to notify the user space component that an anomalous set of system calls has been detected.

Figure 2 illustrates the workflow of our implementation. Each time a containerized user application executes a system call, our kernel space component receives an event and applies the algorithm presented in Algorithm 1. Once an anomalous set of system calls is detected, a signal is registered on the eBPF map, which is read and acted upon by the user space component. In our implementation, the user space component logs an event on the system's logging system. However, our solution could take on a more active role and terminate the suspicious process before it causes any harm to the system.

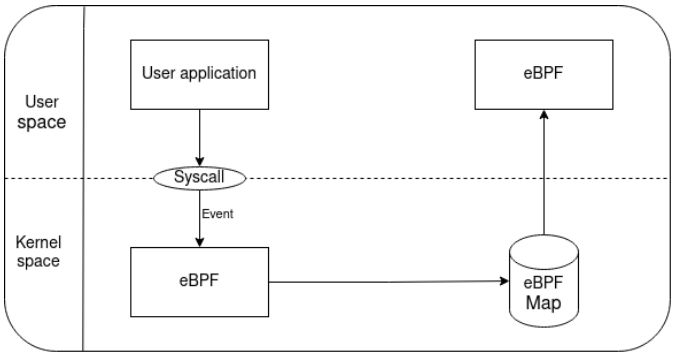

Fig. 2. Architecture of our implementation.

V. Evaluation

In this Section, we conducted a series of experiments to evaluate our proposal. Firstly, we analyze the overhead introduced by various kinds of eBPF programs while instrumenting two kernel functions. This information is critical in determining the appropriate type of eBPF program for our solution. Then, we assessed the overhead introduced by our proposal when

[3]https://github.com/cilium/ebpf/tree/main

tracing system calls. Finally, we simulate a scenario where a user tries to escape a container and verify whether our solution can detect this privilege escalation attempt.

A. eBPF Overhead

In our initial assessment, we measured the additional processing time caused by three different eBPF programs when they were tied into two kernel functions: ”task_rename” and ”fib_tab_lookup”. As our approach requires analyzing all system calls, it is crucial to keep any extra processing overhead to a minimum. We used a benchmarking program whose source code is available in the Linux kernel tree[4] to carry out this analysis.

The benchmark program performs operations that trigger the execution of the kernel functions ”task_rename” and ”fib_tab_lookup”. For example, the ”fib_table_lookup” test sends UDP packets to the localhost to invoke the homonymous kernel function. Similarly, the ”task_rename” test writes a string to the ”/proc/self/comm” file to trigger the ”task_rename” kernel function. In addition, the program attaches three types of eBPF programs capable of tracing system calls to each test and processor core at different times: *BPF_PROG_TYPE_KPROBE* (kprobe), *BPF_PROG_TYPE_TRACEPOINT* (tracepoint), and *BPF_PROG_TYPE_RAW_TRACEPOINT* (raw tracepoint).

The benchmark results are shown in Table I. The table has cells that display the total count of executions for a function under the Test column. A lower count means the function has a higher overhead due to tracing. The Base column displays a baseline count where system calls were not traced. The Kprobe, Tracepoint, and Raw Tracepoint columns display their respective eBPF type counts. Based on the results in the table, it was observed that raw tracepoints outperformed both tracepoint and kprobes. Therefore, we confirmed that the raw tracepoint is the most suitable type of eBPF program for our proposal. Based on this, we conducted the remaining experiments using raw tracepoints.

In our second experiment, we evaluated the latency of the *getpid()* syscall under two distinct conditions: traced with our raw tracepoint eBPF program and untraced. This approach allowed us to assess the impact of eBPF tracing on syscall performance. To do this, we developed a program that measures the execution time of the *getpid()* syscall, running it 100,000 times in each scenario. We selected the *getpid()* syscall to minimize the influence of disk and network I/O on our experiments. Moreover, this syscall does not require input and consistently returns a value of the same size (*pid_t*), making it an ideal candidate for our evaluation.

The comparison of execution times between the two scenarios is shown in Figure 3. It can be observed that the median execution time under eBPF tracing is only slightly higher than that of the baseline. The negligible increase in the median execution time suggests that our eBPF program monitoring the *getpid()* syscall had a minimal impact on overall performance.

The analysis of syscall execution times, captured through a Cumulative Distribution Function (CDF) plot, reveals distinct

TABLE I
eBPF OVERHEAD BENCHMARK.

Test	Base	Kprobe	Tracepoint	Raw Tracepoint
task_rename	3634077	2123379	2783954	3290377
	3575526	2119257	2775150	3269406
	3547105	2118399	2773609	3256166
	3552119	2114393	2772998	3238374
	3536425	2113806	2753713	3210991
	3535852	2108881	2741596	3210334
	3502616	2096884	2745746	3210268
	3481862	2095712	2734844	3204348
	3497963	2057025	2732783	3154480
	3436194	2058983	2721204	3126806
	3488200	2058618	2701266	3074301
	3370600	1942835	2556992	3064059
fib_table_lookup	227078	217006	219809	225775
	227418	216564	218820	224287
	224382	212639	218528	221524
	222373	211783	214378	220404
	222416	211558	213313	220570
	221204	210310	212369	219942
	219844	209461	212342	217033
	219352	209040	211847	213021
	217366	209247	205254	212151
	215330	208072	204731	211712
	212873	205315	205401	211606
	207209	202412	203225	208844

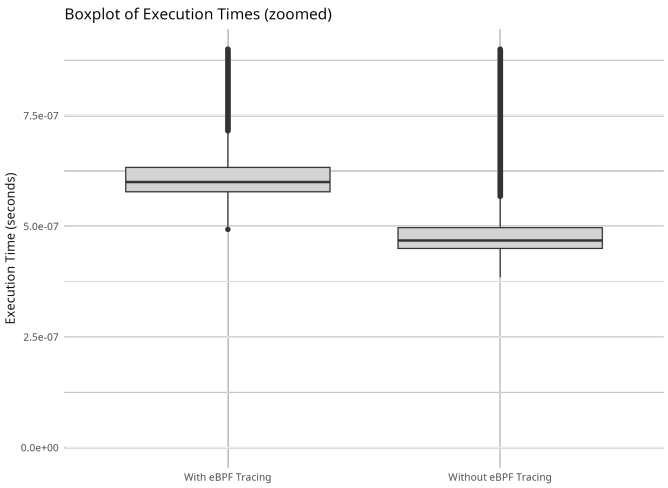

Fig. 3. Syscall execution times with and without eBPF tracing.

patterns when comparing scenarios with and without eBPF tracing. For around 70% of the data points, the execution times under eBPF tracing closely mirror those observed without tracing, albeit with a minor overhead. However, beyond around the 0.7 point in the y-axis, a noticeable shift occurs as the eBPF line diverges rightward, indicating that for the remaining 30% of the data, we observe longer execution times under the eBPF tracing. Figure 4 captures this distribution and the observed shifts.

Finally, we present the results of our third assessment, which aims to compare the results of *redis-benchmark*[5] when executed

[4]https://github.com/torvalds/linux

[5]https://redis.io/docs/management/optimization/benchmarks

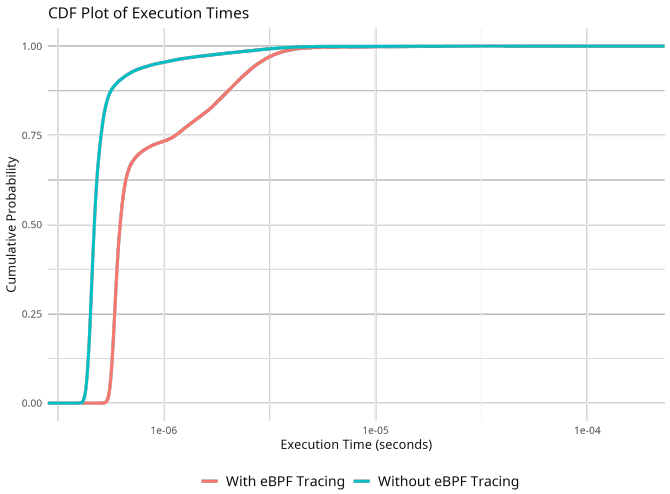

Fig. 4. Cumulative Distribution Function of execution times with and without eBPF tracing.

against a containerized *redis-server* in two scenarios. The first scenario involves running *redis-server* in a container as a standalone process. In this scenario, we aim to collect data on how the benchmark performs when no system calls are traced. The data collected in this scenario will serve as a baseline. In the second scenario, we examine the performance of *redis-server* while tracing its system calls with our eBPF program. By investigating these scenarios, we aim to understand how eBPF tracing affects the overall performance of *redis-server*. Figure 5 presents the number of requests per second (RPS) (y-axis) for all Redis operations (x-axis) achieved in both scenarios of this experiment. As evident from the data, the observed overhead has been minimal despite eBPF tracing introducing an overhead in all Redis operations.

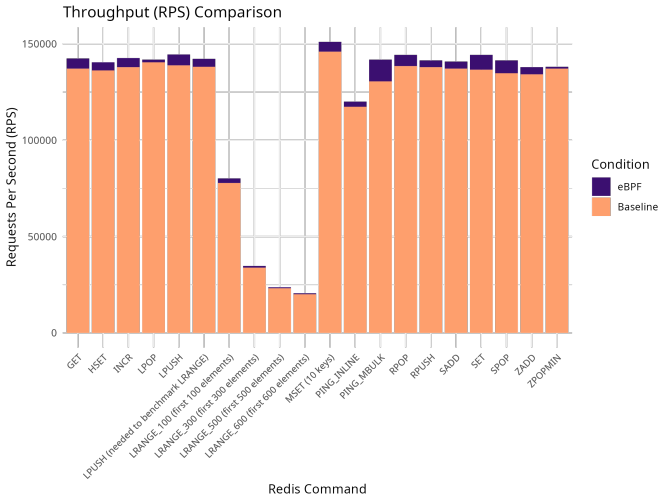

Fig. 5. Requests per Seconds (RPS) for all Redis operations with and without eBPF tracing.

The data presented in this subsection suggests that our eBPF solution introduces a small overhead as the cost of tracing all system calls executed in a containerized application. Consequently, we dedicate the following subsection to demonstrate the effectiveness of our solution in detecting a privilege escalation attempt.

B. Privilege escalation attempt

In this subsection, we simulate a user trying to escape a container by entering a different Linux namespace. This experiment evaluates whether our solution can apply the BoSC technique in a controlled environment. This experiment is an initial step toward testing our solution with real-world scenarios.

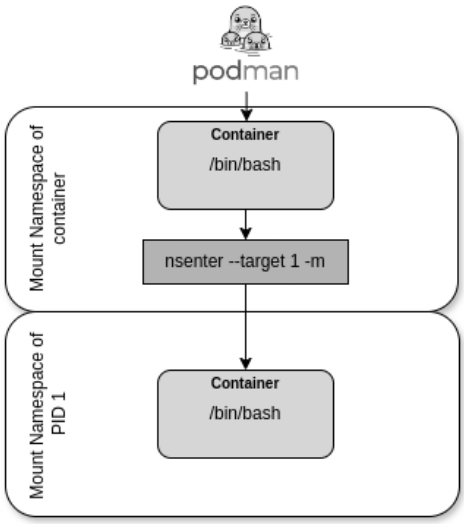

Fig. 6. Container escape by entering a different Linux namespace.

In this simulation, we use Podman[6], a popular container engine, to create a container in interactive mode and a pseudo-terminal (TTY). Once the container prompt starts, the user uses the *nsenter* tool to switch to the same *mount* namespace as (Process ID) PID number 1. In Linux, this PID typically belongs to the *init* program, which is the first process launched at the start of the system boot. This command gives the user access to the host's filesystem if successful. Figure 2 provides a visualization of these steps.

The subsequent task is to pinpoint critical system calls executed within the *nsenter* command and indicate an attempt to break out of the container. Our solution and *strace* identified the same sequence of system calls. Listing 1 shows an excerpt of the *strace* output. Consequently, the bag of syscalls used in this experiment includes the *openat* system call, with the second argument being "/proc/1/ns/mnt", and the *setns* system call, with the second argument being "CLONE_NEWNS" (0x00020000).

```
openat(AT_FDCWD, "/proc/1/ns/mnt", O_RDONLY) = 3
setns(3, CLONE_NEWNS)                        = 0
close(3)                                     = 0
```

Listing 1: Excerpt of strace output for nsenter command.

[6]https://podman.io

As the concluding phase of our experiment, we initiated our proposed eBPF program on the host system and executed the procedures delineated in Listing 2. The purpose of this was to determine the effectiveness of our solution in detecting a potential attempt by the user to escape the container using the *nsenter* tool.

```
$ podman run --interactive --tty fedora:38
# nsenter --target 1 -m
```

Listing 2: Simulation of the user trying to escape a container.

After executing the procedures as intended, we were pleased to confirm that our solution worked accurately. It successfully detected and alerted the attempt to escape the container, which is a testament to the effectiveness of our solution in improving the security and safety of the container environment.

VI. Conclusion

In this work, we evaluated the feasibility of using eBPF to monitor attempts to escalate privileges in containerized applications. We proposed an eBPF program to trace system calls in containerized applications, re-create the execution sequence of those system calls, and apply the BoSC technique to identify attempts to escalate container privileges. Our research indicates that eBPF serves as a robust tool for monitoring and auditing container behavior. It offers detailed visibility into container activity and improves the security of containerized systems.

In future work, several areas can be explored to enhance the eBPF-based solution for container security. First, future work can focus on developing rules and policies within the eBPF program to identify activities carried out by real-world exploits. This would mitigate the risk of privilege escalation attacks, improving the security of containerized environments. Additionally, our solution can be expanded to have a more active role in the system by taking proactive actions to prevent an escalation of privilege once it has been identified. An example of these actions would be to kill the offending process instead of just generating a passive alert. Lastly, a sequence-based system call filtering approach could be assessed in contrast to the BoSC technique used in this work. This would allow the solution to have a broader range of signatures of abnormal behaviors.

VII. Acknowledgments

This work was partially supported by the São Paulo Research Foundation (FAPESP) under grant number 2020/05152-7, the PROFISSA project, and is part of CNPq processes 316662/2021-6 and 315427/2023-0. It is also part of the INCT of Intelligent Communications Networks and the Internet of Things (ICoNIoT), funded by CNPq (proc. 405940/2022-0) and the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) Finance Codes 88887.954253/2024-00 and 001.

References

[1] C. N. C. Foundation, "Cncf annual survey 2022," mar 2023. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022

[2] I. CDatadog, "9 insights on real-world container usage," mar 2023. [Online]. Available: https://www.datadoghq.com/container-report

[3] A. Security, "The state of container security," 2023. [Online]. Available: https://www.aquasec.com/resources/container-security-report/

[4] B. Nikolova and D. Gollmann, "Privilege escalation vulnerabilities in the wild: a decade of attacks on the windows kernel," *Computers & Security*, vol. 100, p. 102033, 2021.

[5] I. Maidul, R. A. Shovon, and S. Hossain, "Vulnerabilities in containerized applications: A case study," in *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*. IEEE, 2020, pp. 255–260.

[6] J. Barnett and P. Chen, "Hardening kubernetes: Security from scratch," in *2021 IEEE/ACM 13th International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2021, pp. 305–310.

[7] D.-K. Kang, D. Fuller, and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, 2005, pp. 118–125.

[8] A. S. Abed, T. C. Clancy, and D. S. Levy, "Applying bag of system calls for anomalous behavior detection of applications in linux containers," in *2015 IEEE Globecom Workshops (GC Wkshps)*, 2015, pp. 1–5.

[9] D. Fuller and V. Honavar, "Learning classifiers for misuse and anomaly detection using a bag of system calls representation," *Journal Name*, 2007.

[10] S. Alarifi and S. Wolthusen, "Detecting anomalies in iaas environments through virtual machine host system call analysis," *Journal Name*, 2015.

[11] S. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal Name*, 1998.

[12] M. Zhan, Y. Li, H. Yang, G. Yu, B. Li, and W. Wang, "¡sc¿coda¡/sc¿: Runtime detection of application-layer cpu-exhaustion dos attacks in containers," *IEEE Transactions on Services Computing*, pp. 1–12, 2022.

[13] J. Simonsson, L. Zhang, B. Morin, B. Baudry, and M. Monperrus, "Observability and chaos engineering on system calls for containerized applications in docker," *Future Generation Computer Systems*, vol. 122, pp. 117–129, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21001163

[14] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," *Proceedings of the Winter 1993 USENIX Conference*, pp. 259–269, 1993.

[15] J. Corbet, "Bpf: the universal in-kernel virtual machine," mar 2023. [Online]. Available: https://lwn.net/Articles/599755

[16] J. Schulist, D. Borkmann, and A. Starovoitov, "Linux socket filtering aka berkeley packet filter (bpf)," feb 2023. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/filter.txt

[17] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, "The eXpress data path: Fast programmable packet processing in the operating system kernel," *CoNEXT 2018 - Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, pp. 54–66, 2018.

[18] M. Abranches, O. Michel, E. Keller, and S. Schmid, "Efficient network monitoring applications in the kernel with ebpf and xdp," in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2021, pp. 28–34.

[19] M. Gebai and M. R. Dagenais, "Survey and analysis of kernel and userspace tracers on linux: Design, implementation, and overhead," *ACM Computing Surveys*, vol. 51, no. 2, 2018.

[20] eBPF.io Authors, "ebpf - introduction, tutorials & community resources," feb 2023. [Online]. Available: https://ebpf.io

[21] L. Torvalds, "Linux operating system," feb 2023. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git

[22] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *Journal Name*, 2006.

[23] G. R. Castanhel, T. Heinrich, F. Ceschin, and C. Maziero, "Taking a peek: An evaluation of anomaly detection using system calls for containers," in *2021 IEEE Symposium on Computers and Communications (ISCC)*, 2021, pp. 1–6.

[24] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Remus: A security-enhanced operating system," *ACM Trans. Inf. Syst. Secur.*, vol. 5, no. 1, p. 36–61, feb 2002. [Online]. Available: https://doi.org/10.1145/504909.504911