Detecting eBPF Rootkits Using Virtualization and Memory Forensics

Nezer Jacob Zaidenberg[1][a], Michael Kiperberg[2][b], Eliav Menachi[3][c] and Asaf Eitani[3]
[1]*Department of Computer Science, Ariel University, Ariel, Israel*
[2]*Department of Software Engineering, Shamoon College of Engineering, Bear Sheva, Israel*
[3]*Faculty of Computer Science, College of Management Academic Studies, Rishon Le Zion, Israel*

Keywords: eBPF, Rootkit, Virtualization, Forensics.

Abstract: There is a constant increase in the sophistication of cyber threats. Areas considered immune to malicious code, such as eBPF, are shown to be perfectly suitable for malware. Initially, the eBPF mechanism was devised to inject small programs into the kernel, assisting in network routing and filtering. Recently, it was demonstrated that malicious eBPF programs can be used to construct rootkits. The previously proposed countermeasures need to be revised against rootkits that attempt to hide their presence. We propose a novel detection scheme that divides the detection process into two phases. In the first phase, the memory image of the potentially infected system is acquired using a hypervisor. In the second phase, the image is analyzed. The analysis includes extraction and classification of the eBPF programs. The classifier's decision is based on the set of helper functions used by each eBPF program. Our study revealed a set of helper functions used only by malicious eBPF programs. The proposed scheme achieves optimal precision while suffering only a minor performance penalty for each additional eBPF program.

1 INTRODUCTION

There is a constant increase in the sophistication of cyber threats. Areas considered immune to malicious code are shown to be perfectly suitable for advanced, evasive, and persistent malware. One such example is eBPF programs. eBPF (extended Berkley Packet Filter) is a kernel mechanism devised for injecting small programs into the kernel without recompiling the kernel or introducing new kernel modules. These small programs can be used in networking, auditing, and security (Ben-Yair et al., 2019).

eBPF programs are written in a C-like language and compiled into a bytecode. The kernel can interpret these programs or compile them into a native code. During the loading of an eBPF program, the kernel verifies that the program is correct. eBPF programs are limited. In particular, eBPF programs cannot call arbitrary functions or access arbitrary memory locations. Instead, they should rely on *helper functions* provided by the kernel. Due to all these limitations, it was believed that eBPF programs cannot be used as malware.

Unfortunately, recently several advanced malicious eBPF programs were demonstrated: *ebpfkit* (Fournier, 2023), *bad-bpf* (PatH, 2022), *boopkit* (Nóva, 2023), and *TripleCross* (Bajo, 2022). As eBPF programs execute in kernel mode, these programs can be used as components of a rootkit, an advanced malware with high privileges. After penetrating the kernel, the rootkit can hide its presence from any possible detector executing with similar or lower privileges. The countermeasures proposed by the authors of the rootkits either assumed that the rootkit would not attempt to hide itself or that the verification could be performed before loading the eBPF programs.

We propose a different detection system for the eBPF rootkit that does not assume anything about the abilities of the kernel. The system consists of four components:

- a memory acquisition tool,
- a memory image analysis framework,
- an eBPF program extraction tool,
- an eBPF program classifier.

Figure 1 depicts the design of the proposed system. Our system uses a thin hypervisor for reliable memory acquisition (Kiperberg et al., 2019). Using a thin hypervisor guarantees that the resulting memory im-

[a] https://orcid.org/0000-0003-3496-7925
[b] https://orcid.org/0000-0001-8906-5940
[c] https://orcid.org/0000-0001-5149-9209

Zaidenberg, N., Kiperberg, M., Menachi, E. and Eitani, A.
Detecting eBPF Rootkits Using Virtualization and Memory Forensics.
DOI: 10.5220/0012470800003648
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 10th International Conference on Information Systems Security and Privacy (ICISSP 2024)*, pages 254-261
ISBN: 978-989-758-683-5; ISSN: 2184-4356
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

age will be trustworthy even in the presence of an active rootkit. The analysis is performed on a separate machine, which is believed to be secure. For memory image analysis, our system uses Volatility (Mohanta et al., 2020), equipped with our plugin for eBPF program extraction. Finally, our classifier notifies the user about suspicious programs based on the set of helper functions they use.

Our evaluation shows that the detection accuracy of our system is optimal, and the detection latency increases slowly with the number of loaded eBPF programs. These results suggest that the proposed system can be used in real-world scenarios.

The main contributions of this paper are:

- We describe a novel system that can detect active eBPF malware.
- We evaluate the accuracy and the performance of the proposed system.
- We present a list of suspicious eBPF helper functions.

2 BACKGROUND

2.1 Virtualization

The underlying technology that enables the memory acquisition component of the described system is virtualization. Virtualization defines a new execution mode with higher privileges than the operating system, whose purpose is to supervise the execution of the operating system. The software executing in this privileged mode is called a hypervisor. This section provides a short overview of hypervisors and virtualization technology in general. There are two types of hypervisors: full hypervisors and thin hypervisors. Full hypervisors like Xen (Barham et al., 2003), VMware Workstation, and Oracle VirtualBox can execute several operating systems concurrently.

VT-x, also known as Intel Virtualization Technology, is a hardware virtualization technology provided by Intel for IA-32 processors. Its purpose is to simplify virtualization and enhance the performance of virtual machine monitors (VMMs) (Karvandi et al., 2022). VT-x allows processors to act like independent processors, enabling multiple operating systems to run simultaneously on the same machine (Karvandi et al., 2022).

Other processor vendors also extended their architectures with hardware-assisted virtualization. In addition to Intel VT-x, AMD introduced AMD-v and ARM introduced ARM-VE. The main goal of hardware-assisted virtualization is to provide software developers with the means to construct efficient full hypervisors.

This paper focuses on Intel VT-x. Hardware-assisted virtualization is not cross-platform. Implementation for other architectures requires similar additional re-implementation.

To assist developers with hypervisor development, VT-x introduces new data structures and instructions to the instruction set architecture (ISA) (Karvandi et al., 2022). These additions enable the processor to efficiently support virtualization by providing architectural support for processor virtualization (Ganesan et al., 2013). VT-x allows the virtual machine monitor to create and manage virtual machines (VMs) by intercepting and handling privileged instructions and events (Neiger, 2006).

When a virtual machine is running on a system with VT-x enabled, the VMM can allocate resources and manage the execution of the VM. The VMM can intercept and handle privileged instructions, such as those related to memory management and I/O operations, to ensure proper isolation and control of the VM (Neiger, 2006). VT-x provides mechanisms for virtualizing the CPU, memory, and I/O devices, allowing the VMM to provide a virtualized environment for the guest operating systems (Neiger, 2006).

VT-x also includes features such as Extended Page Tables (EPT) and Virtual Machine Control Structure (VMCS) that enhance the performance and efficiency of virtualization (Neiger, 2006). EPT is Intel's implementation for Second Level Address Translation (SLAT). EPT improves memory virtualization by reducing the overhead of address translation, while VMCS provides a data structure that holds the state of a VM and controls its execution (Neiger, 2006).

Thin hypervisors, in contrast, can execute only a single operating system. Thin hypervisors' purpose is not to execute multiple operating systems. Instead, the purpose of thin hypervisors is to enrich the functionality of an operating system. The main benefit of a hypervisor over kernel modules (device drivers) is the hypervisor's ability to create an isolated environment, which is essential in some cases. In general, because thin hypervisors are much smaller than full hypervisors, they are superior in performance, security, and reliability. A well-known example of a commonly used thin hypervisor is Microsoft Device Guard (Durve and Bouridane, 2017), available since Windows 10. The hypervisor used in the memory acquisition component described in this paper is thin.

Similarly to an operating system, a hypervisor does not execute voluntarily but responds to events, e.g., execution of special instructions, generation of exceptions, access to memory locations, etc. The hypervisor can configure interception of (almost) each event. The interception of an event (a VM exit) is similar to the handling of an interrupt, i.e., the processor executes a predefined function. Another similarity with an operating system is the hypervisor's ability to configure the access rights to each memory page through a data structure named the second-level address translation table (SLAT). The SLAT resembles the virtual page table in the operating system and is, in fact, a page table for operating systems instead of processes. An attempt to write to a non-writable (according to SLAT) page induces a VM exit and allows the hypervisor to act. SLAT exists in virtually all CPUs that are in use today under various brand names (such as Intel EPT, AMD RVI, ARM VE, etc.)

2.2 Memory Forensics

Memory forensics is the art of acquiring a full image of the system's memory during execution and analyzing it in a secondary program. It may be performed by a computer on itself (for example, scanning the memory for viruses) or by a known good computer inspecting a possibly infected computer's memory. Since a virus or rootkit will attempt to mask its existence from the operating system it runs on, forensics by a secondary station is often required.

Memory Forensics by a secondary host consists of two separate tasks. The first task is acquiring a reliable image of the memory, preferably in a non-intrusive manner. It is easy to get a reliable, consistent memory image in an intrusive manner, for example, by getting the system to a sleep state and copying the system's memory file to a separate computer. It is also possible to get an unreliable system image by copying memory pages to disk while the system operates.

The problem is that many artifacts may be created as the memory content changes during the system execution. For example, the process table may point to memory pages of processes that no longer exist. Memory inconsistencies are a fundamental problem in security forensics as artifacts in memory may result from a malicious virus or rootkits but may also exist due to inconsistent memory dump (Palutke et al., 2020). It was previously shown how hypervisors can overcome both problems and create reliable memory images in a non-intrusive way (Kiperberg et al., 2019).

Once an accurate and reliable memory image has been captured, one can analyze the memory using many possible tools such as Yara (Cohen, 2017), Rekall (Stadlinger et al., 2018), and Volatility(Case and Richard III, 2017). In this work, we focus on Volatility, which is the most common tool.

2.3 eBPF

eBPF, or extended Berkeley Packet Filter, is a powerful technology in modern operating systems like Linux. eBPF was initially designed for packet filtering (Gowtham et al., 2021). eBPF has evolved into a versatile framework that allows custom code to load and run in the operating system kernel without requiring modifications or recompilation.

eBPF enables the creation of small programs (BPF programs) that can be loaded into the kernel to perform various tasks like packet filtering, tracing, monitoring, and more. One of eBPF's significant features is its safety; eBPF programs undergo rigorous verification before running in the kernel, ensuring they cannot harm the system (Findlay et al., 2020). eBPF provides high-performance execution due to its JIT (Just-In-Time)(Van Geffen et al., 2020) compilation to native machine code, enabling efficient execution of complex tasks in the kernel space(Scholz et al., 2018).

eBPF has many use cases, including network-related operations like packet filtering, tracing system calls(De Giorgi, 2023), performance analysis(Cassagnes et al., 2020), security monitoring(Fournier et al., 2021), and even extending Kubernetes capabilities(Miano et al., 2021).

2.4 Rootkits

The first task of an attacker is gaining access to a remote computer without the owner's permission. The next task, after gaining the initial access, is performing permission elevation to gain root access. The attacker's third task is gaining persistence. Persistence means allowing the attacker to re-access the remote host after the current session was terminated or after rebooting and patching the vulnerable software exploited in the original access. Persistence usually also means masking the traces of the attack, i.e., hiding the attacker's processes and network connections from a potential observer. It is assumed that if the victim is aware of the attacker's activities, she will reinstall the infected system or at least turn it off, locking the attacker outside of the system.

To achieve persistence, the attacker installs a rootkit on the victim system. A rootkit is malicious software designed to gain unauthorized access to a computer system while remaining hidden from the user and most security software. The rootkit operates deep within the operating system, often exploiting vulnerabilities or using advanced techniques to conceal its presence and evade detection by antivirus programs or system scans.

Rootkits can provide attackers with privileged access to a system, enabling them to execute various malicious activities, such as:

- Rookits can hide files, processes, network connections, and registry keys, making them difficult to detect and remove. Rootkits often establish persistence on a system, ensuring they remain active even after reboots or system updates.
- Some rootkits exploit system vulnerabilities to gain administrative or root-level privileges, granting the attacker extensive control over the system.
- Rootkits can create backdoors or remote access mechanisms, allowing attackers to control the infected system remotely, execute commands, or exfiltrate sensitive data.
- Some rootkits might be used to spy, monitor user activities and keystrokes, or capture sensitive information without the user's knowledge.

Rootkits can be challenging to detect and remove because of their ability to embed themselves deeply within the system. Therefore, software that aims to detect rootkits must execute with higher privileges than the rootkit.

There are multiple types of rootkits using various technologies. Rootkits can be created using boot loaders that boot before the operating system boots (Li et al., 2011), Hypervisors such as the blue pill (Rutkowska, 2006), kernel modules, and even rootkits that exist in the BIOS or hardware itself (Hili et al., 2014). In this paper, we focus only on eBPF rootkits.

3 SYSTEM DESIGN

This paper proposes a novel detection system for eBPF rootkits. The system consists of four components:

- a memory acquisition tool,
- a memory image analysis framework,
- an eBPF program extraction tool,
- an eBPF program classifier.

Figure 1 depicts the design of the proposed system. There are multiple possible approaches for memory acquisition, ranging from dedicated hardware devices (Carrier and Grand, 2004) to leveraging system calls (Stüttgen and Cohen, 2014). Our system uses a thin hypervisor for reliable memory acquisition (Kiperberg et al., 2019). Upon a request of an external entity, the hypervisor begins the process of memory acquisition. Most memory pages are sent to the external entity one by one sequentially. However, when the operating system or a user mode application attempts to modify some page, the hypervisor sends the page to an external entity out of order before the modification occurs. The resulting image represents an atomic memory snapshot obtained without freezing the target machine.

Our system uses Volatility (Mohanta et al., 2020) for memory image analysis. The Volatility framework is a widely recognized tool in digital forensics, particularly for volatile memory forensics. It has been utilized in various contexts, showcasing its adaptability and effectiveness. Volatility is an extendable framework. Volatility supports multiple plugins to be added. These plugins assist in locating, extracting, and analyzing data. For example, it was extended to live memory address spaces, enabling the dynamic recreation of kernel data structures for live forensics, thereby enhancing its applicability in digital investigations (Tobin et al., 2017).

Unfortunately, Volatility cannot locate and extract the compiled eBPF programs. We have developed a plugin for Volatility that closes this gap (Mohanta et al., 2020). The plugin operates as follows:

1. The data structure that stores all the compiled eBPF programs is located.
2. The data structure is traversed, and the details of each program are extracted: name, length, loading time, type, and code.
3. The code is disassembled, and the called functions are identified.
4. The function addresses are translated to helper function names.

The plugin output lists the helper functions invoked by each currently loaded eBPF program.

The kernel stores all the compiled eBPF programs in a variable named `prog_idr` of type `struct idr`. This type generally represents an associative array that maps small integers to arbitrary pointers or objects. The array itself is implemented as a radix tree. In the `prog_idr` 's case, the pointees are objects of type `struct prog` representing eBPF programs. This object's `func` field points to the compiled program.

The plugin leverages the infrastructure of Volatility to locate the `prog_idr` variable and traverse the radix tree to acquire all the eBPF program objects. Then, the plugin extracts the compiled program from each object and disassembles it using the ”capstone” library (Anh, 2014).

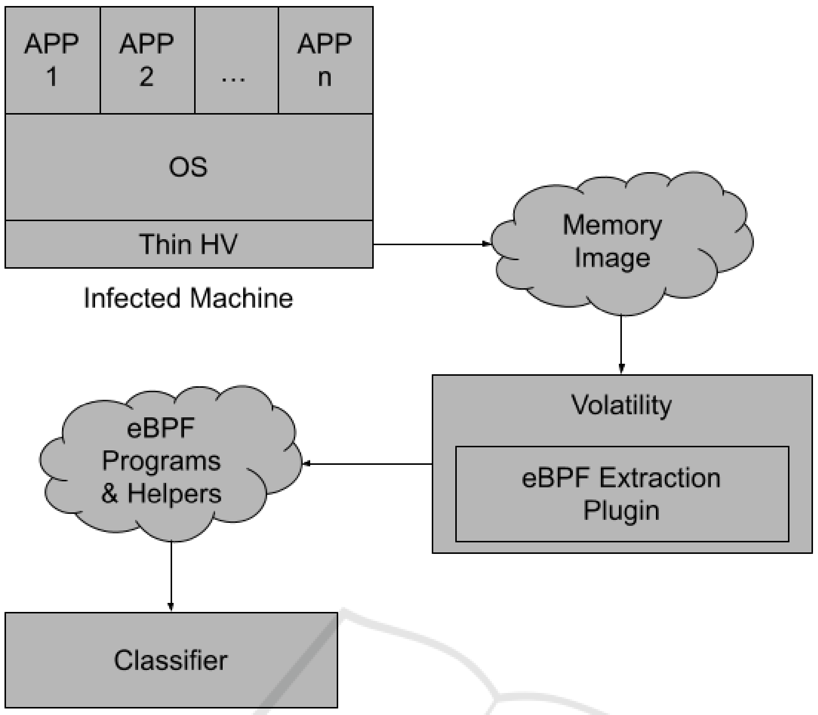

Figure 1: The design of the eBPF malware detection system. The thin hypervisor responsible for memory acquisition is the only component on the infected machine. The hypervisor has higher privileges than the applications and the operating system, making it resilient to rootkits. The other three components run on a separate secure machine. The memory image analysis framework, Volatility, is equipped with our eBPF program extraction plugin. The plugin output is a list of eBPF programs and their helper functions. The classifier uses this list to output the potentially malicious programs.

To identify the helper functions used by the eBPF program, the plugin scans the disassembled code and locates all the "call" instructions. The argument of each such instruction represents the address of the called helper function. The address is converted to a symbolic name using Volatility's infrastructure.

In the final step, the plugin tests whether the set of helper functions is suspicious. In its current implementation, the plugin checks whether the set of helper functions includes at least one of the functions in Table 1. If so, a warning message is displayed.

Table 1 lists the suspicious eBPF helper functions. It was constructed by reviewing the open-source eBPF rootkits. Each helper function can be used to accomplish a specific task by the rootkit:

- The `probe_write_user` is used to alter the results returned by the current system call. For example, it can be used to change the actual data read from a file.
- The `override_return` is used to falsify an error or success by replacing the return code from a function. For example, it can prevent the deletion of the rootkit files.
- The `skb_store_bytes` and `skb_pull_data` create a covert communication channel between the rootkit and the C&C server.
- The `send_signal` is used to terminate processes that attempt to reveal the rootkit's presence.

4 EVALUATION

We have evaluated the proposed system from two perspectives: detection precision and detection latency. We ran a virtual machine with an Ubuntu 22.04 operating system to evaluate the detection precision. We have pre-installed 134 benign and nine malicious open-source benign eBPF programs. Then, we took a snapshot of the virtual machine memory. Finally, we analyzed the memory image using Volatility and our plugin. All the malicious and none of the benign eBPF programs were reported. The malicious programs we tested are part of the *bad-bpf* project (PatH, 2022). In addition, we have verified that all publicly available eBPF rootkits use one of the helper functions listed in Table 1. We have tested the following rootkits: ebpfkit (Fournier, 2023), bad-bpf (PatH, 2022), boopkit (Nóva, 2023), and TripleCross (Bajo, 2022).

Table 1: Suspicious eBPF helper functions.

Function Name	Description
`probe_write_user`	Writes to a user-supplied memory region
`override_return`	Specified the return value of the current system call
`skb_store_bytes`	Writes to a network packet
`skb_pull_data`	Reads from a network packet
`send_signal`	Sends process a signal

We have measured the execution time of Volatility equipped with our plugin when analyzing a memory image containing 134 benign eBPF programs and a memory image containing nine malicious programs. The execution time was 8.68 seconds in the first case and 7.35 seconds in the second case. Assume that the execution time can be expressed using the formula $T = A + B \cdot n$ (i.e., the dependence on n is linear) where T is the total execution time, A is the execution time that does not depend on the number and size of the programs and B is the handling time of a single program. From our measurements, we can deduce that $B = 0.11s$ and $A = 7.25$. Therefore, the execution time increases very slowly with n.

5 RELATED WORK

Rootkits based on eBPF programs are relatively new. Only a few examples of open-source rootkits were previously discussed: *ebpfkit* (Fournier, 2023), *bad-bpf* (PatH, 2022), *boopkit* (Nóva, 2023), and *Triple-Cross* (Bajo, 2022). The authors presented not only the rootkits but also countermeasures. Some countermeasures, such as eBPF program signing and verification, are unrelated to the current work. Therefore, in this review, we concentrate on countermeasures that aim to detect malicious eBPF programs.

In the current landscape of eBPF rootkit detection, two tools primarily stand out: Tracee (Security, 2023) and ebpfkit-monitor (Fournier, 2023). Tracee, developed by Aqua Security, is a runtime security and forensics tool for Linux built on eBPF technology. By leveraging eBPF's capabilities, Tracee can trace system and application calls directly from the Linux kernel without requiring any prior instrumentation. Specifically, Tracee employs an event named `bpf_attach` to mark the instance of an eBPF program being attached to a probe in the system. This event occurs whenever an eBPF program is attached to a performance event of the types: *kprobe*, *uprobe*, or *tracepoint*. The event's purpose is to provide information about the eBPF program and the probe itself. The information includes the eBPF program type, name, ID, and list of all eBPF helper functions used by the program.

On the other hand, *ebpfkit-monitor* is a utility that can either statically analyze eBPF bytecode or monitor suspicious eBPF program loading at runtime. The tool was specifically designed to detect *ebpfkit*, a rootkit leveraging eBPF.

While these two tools provide valuable capabilities in detecting eBPF rootkits during runtime or before execution, they have limitations. Both tools are ineffective at detecting rootkits that have already penetrated the system. Such rootkits can hide themselves from an internal observer. In contrast, the method proposed in this paper relies on a hypervisor that extracts a reliable memory image even in the presence of an active rootkit. The analysis is performed on a separate machine, which is believed to be secure.

Memory acquisition is an essential task in forensic analysis. It can be done using means provided by the operating system, a more privileged software, or hardware devices. While methods based on operating system capabilities are the simplest, they can easily be circumvented by an active rootkit.

Another method of memory acquisition is based on generic or dedicated hardware. Previous works show how a generic FireWire card can acquire memory remotely (Zhang et al., 2010). A dedicated PCI card, named Tribble, works in a similar manner (Carrier and Grand, 2004). The main advantage of a hardware solution is resistance to even the most advanced rootkits and the reliable memory image it provides. The two main disadvantages are:

- If the rootkit installed a hypervisor with IO-MMU (Amit et al., 2010) capability, it can mask certain memory pages from a specific PCI interface.
- Hardware solutions suffer from high costs and complexity. A shutdown is required to add hardware to a running system, which may be complicated in many environments. If the hardware is installed beforehand, it needs to be determined which systems are to be inspected, and specific hardware needs to be installed on these systems. Also, memory forensics using physical hardware may require reaching the inspected system, connecting cables, and a secondary inspecting system to perform the inspection. This process may be too complicated to be carried out in practice.

A hypervisor can detect rootkit behavior (Zaidenberg and Khen, 2015), but detecting rootkits in memory dumps is less intrusive and requires fewer VM exits. Hypervisor-based methods of memory acquisition have low cost and the same degree of resistance to rootkits. HyperSleuth (Martignoni et al., 2010) is a driver with an embedded hypervisor. Its hypervisor is capable of performing atomic and lazy memory acquisition. The laziness is expressed in the ability of the hypervisor to continue the normal execution while the memory is acquired. ForenVisor (Qi et al., 2016) is a similar hypervisor with additional features that allow it to log keyboard strokes and hard-drive activity. Kiperberg et al. (Kiperberg et al., 2019) showed how HyperSleuth and ForenVisor can be adapted to multiprocessor systems executing Windows 10. This tool was selected for memory acquisition in the proposed system. A similar solution is possible in the ARM architecture as well(Yehuda et al., 2021).

6 CONCLUSIONS

We described a system for detecting eBPF rootkits. The system is resilient to active rootkits. The system achieves optimal precision while suffering only a minor performance penalty for each additional eBPF program. The classifier was developed after studying the available eBPF rootkits and extracting the potentially malicious helper functions.

In its current implementation, the system requires the hypervisor to transmit a complete memory image, which may take considerable time. This aspect can be optimized by transmitting only those memory pages required to extract the eBPF programs. Moreover, it is possible to embed the Volatility framework and our eBPF extraction plugin in the hypervisor itself, thus eliminating the need for network transmission.

The system presented is not limited to thin hypervisors. Future implementation can include the system in a full hypervisor. By building the system in a full hypervisor, it would be possible to provide eBPF monitoring service to all guest operating systems.

7 SOURCE CODE AVAILABILITY AND STATISTICS

The Volatility plug-in described in this paper is open-sourced and available for download under https://github.com/AsafEitani/volatility3/tree/ebpf_plugin The virtual machine we used to infect and detect the eBPF malware is available for download at https://drive.google.com/drive/folders/1_lvtwV0J9608vP0nbxQ9SEwyd7xy5TPF?usp=sharing username/password is eitani/a

The User home directory has two memory dumps (one with and one without malicious eBPF malware).

Our plug-in also displays statistics about usage: the number of eBPF programs and helper functions. Running the plugin on a dump without eBPF rootkits produces the following output:

```
$python3 ./vol.py -f ~/legit_bpf.vmem
linux.ebpf_programs
Execution took 8.684305429458618 seconds.
134 ebpf programs were detected.
1707 total helper functions were used -
28 unique.
12.738805970149254 average helper
per ebpf program.
eBPF Malware not detected.
```

Running the plug-in on a malicious dump produces the following output:

```
$python3 ./vol.py -f ~/bad-bpf.vmem
linux.ebpf_programs
Execution took 7.350746393203735 seconds.
9 ebpf programs were detected.
6 total helper functions were used -
6 unique.
0.6666666666666666 average helper
per ebpf program.
eBPF Malware detected.
```

REFERENCES

Amit, N., Ben-Yehuda, M., and Yassour, B.-A. (2010). Iommu: Strategies for mitigating the iotlb bottleneck. In *International Symposium on Computer Architecture*, pages 256–274. Springer.

Anh, Q. N. (2014). Capstone: Next generation disassembly framework. *Proceedings of the 2014 Black Hat USA, Black Hat USA*, 14.

Bajo, M. S. (2022). An analysis of offensive capabilities of ebpf and implementation of a rootkit. *Bachelor Thesis of Charles III University of Madrid*.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177.

Ben-Yair, I., Rogovoy, P., and Zaidenberg, N. (2019). Ai & ebpf based performance anomaly detection system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 180–180.

Carrier, B. D. and Grand, J. (2004). A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60.

Case, A. and Richard III, G. G. (2017). Memory forensics: The path forward. *Digital investigation*, 20:23–33.

Cassagnes, C., Trestioreanu, L., Joly, C., and State, R. (2020). The rise of ebpf for non-intrusive performance monitoring. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE.

Cohen, M. (2017). Scanning memory with yara. *Digital Investigation*, 20:34–43.

De Giorgi, M. (2023). System calls monitoring in android: An approach to detect debuggers, anomalies and privacy issues.

Durve, R. and Bouridane, A. (2017). Windows 10 security hardening using device guard whitelisting and applocker blacklisting. In *2017 Seventh International Conference on Emerging Security Technologies (EST)*, pages 56–61. IEEE.

Findlay, W., Somayaji, A., and Barrera, D. (2020). Bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 91–103.

Fournier, G. (2023). ebpfkit. https://github.com/Gui774ume/ebpfkit. Accessed: 21.11.2023.

Fournier, G., Afchain, S., and Baubeau, S. (2021). Runtime security monitoring with ebpf. In *17th SSTIC Symposium sur la Sécurité des Technologies de l'Information et de la Communication*.

Ganesan, R., Murarka, Y., Sarkar, S., and Frey, K. (2013). Empirical study of performance benefits of hardware assisted virtualization. In *Proceedings of the 6th ACM India Computing Convention*, pages 1–8.

Gowtham, V., Keil, O., Yeole, A., Schreiner, F., Tschöke, S., and Willner, A. (2021). Determining edge node real-time capabilities. In *2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–9. IEEE.

Hili, G., Mayes, K., and Markantonakis, K. (2014). The bios and rootkits. *Secure Smart Embedded Devices, Platforms and Applications*, pages 369–381.

Karvandi, M. S., Gholamrezaei, M., Khalaj Monfared, S., Meghdadizanjani, S., Abbassi, B., Amini, A., Mortazavi, R., Gorgin, S., Rahmati, D., and Schwarz, M. (2022). Hyperdbg: Reinventing hardware-assisted debugging. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1709–1723.

Kiperberg, M., Leon, R., Resh, A., Algawi, A., and Zaidenberg, N. (2019). Hypervisor-assisted atomic memory acquisition in modern systems. In *International Conference on Information Systems Security and Privacy*. SCITEPRESS Science And Technology Publications.

Li, X., Wen, Y., Huang, M. H., and Liu, Q. (2011). An overview of bootkit attacking approaches. In *2011 Seventh International Conference on Mobile Ad-hoc and Sensor Networks*, pages 428–431. IEEE.

Martignoni, L., Fattori, A., Paleari, R., and Cavallaro, L. (2010). Live and trustworthy forensic analysis of commodity production systems. In *Recent Advances in Intrusion Detection: 13th International Symposium, RAID 2010, Ottawa, Ontario, Canada, September 15-17, 2010. Proceedings 13*, pages 297–316. Springer.

Miano, S., Risso, F., Bernal, M. V., Bertrone, M., and Lu, Y. (2021). A framework for ebpf-based network functions in an era of microservices. *IEEE Transactions on Network and Service Management*, 18(1):133–151.

Mohanta, A., Saldanha, A., Mohanta, A., and Saldanha, A. (2020). Memory forensics with volatility. *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*, pages 433–476.

Neiger, G. (2006). Intel$\hat{W}$virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*.

Nóva, K. (2023). Boopkit. https://github.com/krisnova/boopkit. Accessed: 21.11.2023.

Palutke, R., Block, F., Reichenberger, P., and Stripeika, D. (2020). Hiding process memory via anti-forensic techniques. *Forensic Science International: Digital Investigation*, 33:301012.

PatH (2022). Bad bpf. https://github.com/pathtofile/bad-bpf. Accessed: 21.11.2023.

Qi, Z., Xiang, C., Ma, R., Li, J., Guan, H., and Wei, D. S. (2016). Forenvisor: A tool for acquiring and preserving reliable data in cloud live forensics. *IEEE Transactions on Cloud Computing*, 5(3):443–456.

Rutkowska, J. (2006). Introducing blue pill. *The official blog of the invisiblethings. org*, 22:23.

Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., and Carle, G. (2018). Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 1, pages 209–217. IEEE.

Security, A. (2023). Tracee. https://github.com/aquasecurity/tracee. Accessed: 21.11.2023.

Stadlinger, J., Dewald, A., and Block, F. (2018). Linux memory forensics: Expanding rekall for userland investigation. In *2018 11th International Conference on IT Security Incident Management & IT Forensics (IMF)*, pages 27–46. IEEE.

Stüttgen, J. and Cohen, M. (2014). Robust linux memory acquisition with minimal target impact. *Digital Investigation*, 11:S112–S119.

Tobin, P. C., Le-Khac, N., and Kechadi, T. (2017). Forensic analysis of virtual hard drives. *Journal of Digital Forensics, Security and Law*.

Van Geffen, J., Nelson, L., Dillig, I., Wang, X., and Torlak, E. (2020). Synthesizing jit compilers for in-kernel dsls. In *International Conference on Computer Aided Verification*, pages 564–586. Springer.

Yehuda, R. B., Shlingbaum, E., Gershfeld, Y., Tayouri, S., and Zaidenberg, N. J. (2021). Hypervisor memory acquisition for arm. *Forensic Science International: Digital Investigation*, 37:301106.

Zaidenberg, N. J. and Khen, E. (2015). Detecting kernel vulnerabilities during the development phase. In *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*, pages 224–230.

Zhang, L., Wang, L., Zhang, R., Zhang, S., and Zhou, Y. (2010). Live memory acquisition through firewire. In *International Conference on Forensics in Telecommunications, Information, and Multimedia*, pages 159–167. Springer.