電子情報学中間報告資料 2024 年 5 月 10 日

Leveraging eBPF to Uncover the Characteristics of Shadow Attack

落合研究室 電子情報学専攻 修士課程 2 年 48-236427 手塚 尚哉

Abstract

In the complex arena of cybersecurity, evasive malware, such as shadow attacks that obscure malicious activities through multiple processes, challenge conventional detection methods. This paper introduces an innovative detection and analysis methodology using Extended Berkeley Packet Filter (eBPF) technology to counteract these threats. eBPF facilitates real-time, in-depth monitoring of system operations, enabling the identification of sophisticated malware behaviors.

We leverage eBPF to trace process interactions and system calls, identifying malicious patterns indicative of shadow attacks. This approach distinguishes between legitimate and malicious activities by analyzing process executions and network communications. Despite challenges like data volume and behavior differentiation, we apply smart filtering and machine learning to enhance detection accuracy.

Our research showcases eBPF's potential in detecting complex malware through case studies, emphasizing the need for advanced analysis techniques in cybersecurity. This work contributes significantly to understanding and mitigating advanced malware threats, proving eBPF as a vital tool in modern cybersecurity defenses.

1 Introduction

In the contemporary digital landscape, the sophistication of malware, particularly in its ability to evade detection and analysis, poses a significant challenge to cybersecurity efforts. Among these evasive tactics, shadow attacks [1], which cleverly distribute malicious activities across multiple processes, stand out as particularly insidious. These attacks exploit the inherent complexity of operating systems, mimicking benign multi-process behavior to obfuscate their malicious intent. Traditional detection mechanisms, reliant on static and dynamic analysis techniques, often fall short in identifying these distributed threats, necessitating the exploration of more advanced methodologies.

This paper introduces an innovative approach to tackling the challenge posed by shadow attacks through the use of Extended Berkeley Packet Filter (eBPF). eBPF, a technology that allows for the safe execution of custom code within the Linux kernel without changing kernel source code or loading kernel modules, offers a powerful mechanism for monitoring and tracing system-level operations. Our research leverages eBPF to analyze the interconnections between function calls, thereby revealing the execution patterns of processes involved in shadow attacks. By mapping these patterns, we aim to uncover the stealthy operations of evasive malware, providing insights that could lead to more effective detection and mitigation strategies.

We focus on the potential of eBPF to provide granular visibility into the behavior of systems at runtime, enabling the identification of the complex orchestration of processes characteristic of shadow attacks. Through the detailed analysis of function call chains, we can trace the flow of execution within malicious processes, identifying their strategies and mechanisms. This approach not only enhances our understanding of how such attacks are constructed and executed but also opens new avenues for developing countermeasures that can detect and neutralize these threats more efficiently.

By employing eBPF to dissect the intricacies of process execution and interaction in the context of shadow attacks, our research contributes a novel perspective to the field of cybersecurity. We demonstrate how eBPF's capabilities can be harnessed to advance our understanding of malicious process execution, offering a promising methodology for combatting evasive malware. Through this work, we aim to bolster the cybersecurity community's arsenal against the ever-evolving landscape of malware threats, ensuring a more secure digital environment for all users.

Our exploration into the use of eBPF against shadow attacks not only highlights the adaptability and complexity of modern malware but also underscores the necessity for innovative detection and analysis techniques. As we delve into the capabilities and applications of eBPF, we pave the way for future research and development in the domain of cybersecurity, seeking to establish more sophisticated defenses against the cunning and elusive nature of malware attacks.

2 Background

2.1 Shadow Attack [1]

Shadow Attack is one of the techniques used by malware writers to evade behavior-based detection systems, orchestrating multiple processes to stealthily carry out malicious activities.

Such detectors typically rely on comparing system call graph within a process under scrutiny with predefined malware specifications established on specific sequences or graphs of system calls [2]. For example, the malware specification of download-and-execute is expressed as follows [1]:

$$\texttt{recv} \wedge \texttt{open} \rightarrow \texttt{write} \rightarrow \texttt{exec} \quad (1)$$

, where $s_1 \wedge s_2$ denotes both of two system calls s_1, s_2 are executed and $s_1 \rightarrow s_2$ denotes s_1 is followed by s_2.

So more specifically, the goal of Shadow Attack is to bypass dynamic malware detection based on the analysis of system call graphs by exporting any critical system call included in malware specifications to other collaborating processes, which is called shadow processes. We call the Communication shadow processes do with each other as Shadow Process Communication (SPC). The concept of Shadow Attack is illustrated in Fig. 1.

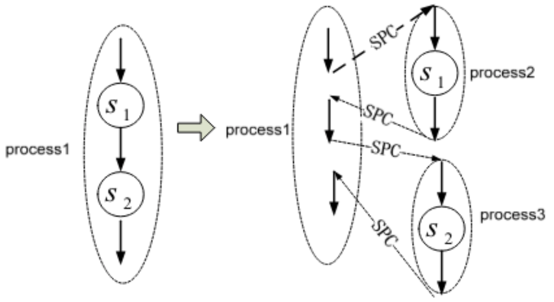

Fig. 1: Illustration of Shadow Attacks [1]

Shadow attacks can be categorized into in-host, remote-network-coordinated, and hybrid. In in-host shadow attacks, all shadow processes are executed on the same host and SPCs are conducted through unix domain socket or stream pipe, while remote-network-coordinated shadow attacks involve multiple remote hosts that are connected through network sockets. Prototype implementation is made by the authors only for in-host shadow attacks.

One countermeasure is extracting correlation between processes and reconstructing the original system call graph, but the authors conducted an evaluation experiment following this approach, whose result suggested that solution would encounter challenges of high overhead.

2.2 eBPF

2.2.1 Berkeley Packet Filter

Steven and Van [3] proposed the BSD Packet Filter architecture in 1993 for efficient packet capture on Unix-based operating systems. In the following, we refer to Berkeley Packet Filter as ”BPF.” At the time of paper publication, packet capture involved copying all packets acquired in the kernel space to the user space before filtering. This process resulted in unnecessary overhead. [3] devised a pseudo-machine (BPF pseudo-machine) that interprets programs written in special 32-bit instructions to perform filtering. By running this pseudo-machine in the kernel space, they addressed the issue. Compared to existing systems, BPF operated up to 20 times faster. The overview of BPF architecture is shown in Figure 2.

BPF was introduced in the Linux kernel as ”Linux Socket Filter” in version 2.1.75 and was used to accelerate the `tcpdump` command.

2.2.2 Extended Berkeley Packet Filter

BPF underwent significant improvements and extensions in the Linux kernel version 3.18, leading to

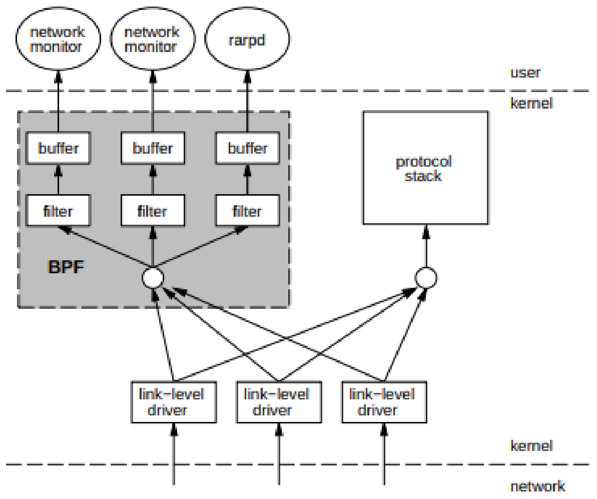

Fig. 2: The overview of BPF architecture. It accelerates packet capture by performing filtering in the kernel space. [3]

the emergence of extended BPF, commonly referred to as eBPF [4]. The enhancements cover various aspects, with notable additions summarized below [5]:

- 64-Bit BPF Instruction Set: The BPF instruction set was reworked from 32-bit to 64-bit, resulting in improved execution efficiency.
- eBPF Maps: The introduction of eBPF maps allowed data sharing between user space and kernel space. These maps serve as a mechanism for efficient communication.
- eBPF Verifier: To ensure safe execution of eBPF programs, an eBPF verifier was added. It validates the correctness and security of eBPF code.

The area covered by eBPF has also expanded. In the context of networking, it has become able to handle various layers of the Linux network stack, such as unix domain sockets and network devices. Additionally, eBPF programs can now be used for performance tracing and enhancing the security of Linux systems, leading to the term ”BPF” losing its original meaning of ”Berkeley Packet Filter” and being used as an independent term.

For convenience, BPF before the extension in v3.18 is sometimes referred to as classical BPF or cBPF.

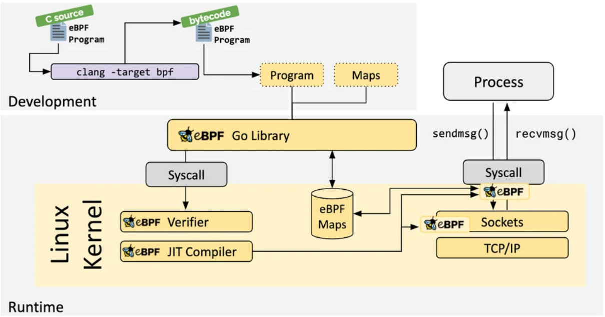

Fig. 3: The overview of eBPF architecture. This figure shows how eBPF programs are compiled, verified, and executed. [6]

2.2.3 Overview of eBPF Architecture

An overview of the eBPF architecture is shown in Fig. 3. Hereafter, we describe the important processing flows with reference to Fig. 3.

2.2.4 Event-Driven Architecture

eBPF has an event-driven architecture. The mechanism involves hooking an eBPF program to an event in the kernel, and then performing the specified processing when the event occurs. eBPF programs can be dynamically loaded or removed.

The events that eBPF can hook into are defined as Program Types in the kernel's source code [*1]. A few examples of Program Types are as follows:

- XDP: An event for manipulating packets before data is copied to the kernel space when a packet arrives at a network device.
- Tracing: An event for detecting kernel function calls and passing tracepoints.
- LSM: An event for applying security policies using the Linux Security Module.

When such events occur, the eBPF program executes the processing corresponding to the Program Type. For example, a program hooked to an XDP event can decide whether to accept or drop a packet.

2.2.5 eBPF Verifier

The eBPF verifier is a program that takes eBPF programs converted to bytecode as input and verifies that they can be executed safely on the kernel. The bytecode is loaded into the kernel via the `bpf()` sys-

[*1] `include/uapi/linux/bpf.h`

tem call (shown as "Syscall" in Fig. 3), but the program will not run unless it passes verification by the verifier. Specifically, it checks for things like avoiding memory access violations, ensuring the program exits normally, and that the program is not granted unnecessary privileges.

In this way, the eBPF verifier enhances security by imposing restrictions on eBPF programs. Since the verifier plays a crucial role in eBPF, research has been conducted to mathematically verify the logic of the verifier [7].

2.2.6 JIT Compilation

The eBPF bytecode that passes the verifier is converted by a JIT compiler into machine code that directly runs on the target CPU. This optimizes the execution speed, allowing it to operate as efficiently as the kernel and kernel modules directly compiled from source code [6].

3 Related Work

3.1 Malware Analysis

Malware analysis is a critical aspect of cybersecurity, enabling the identification and mitigation of malicious software threats. Analysis techniques can be roughly categorized into two classes: static and dynamic.

3.1.1 Static Analysis

Static analysis involves examining the structure and content of a suspected malicious file without executing it. Static analysis are often conducted as a preliminary step to grasp the general characteristics of the malware before more sophisticated analysis techniques are applied [8].

Signature matching is a common technique that compares the "signature", characteristics of a file against a database of malware such as VirusTotal [9]. The characteristics include file size, hash values, and byte sequences. Although this method has been a staple in malware detection, it is limited by its inability to detect only known malware.

Disassemble and decompile is another static analysis technique that involves converting the binary code of a malware into a human-readable format. From the decompiled code, analysts might be able to identify the malware's functionality and behavior as well as finding interesting strings like URLs, IP addresses, and encryption keys.

3.1.2 Dynamic Analysis [10]

Dynamic analysis entails examining the behavior of malware by executing it in a controlled environment. This approach enables analysts to witness the interactions between the malware and the system, providing insights into its true intentions and capabilities.

Function call analysis is a method that focuses on tracking functions issued by the malware and the parameters passed to them. One way to archive this is by code injection, in which analyzing code is hooked into a specific function call and various information is collected and notified when the function is called. Carsten et al. [11] created an automated malware analysis system that injects DLLs within CWSandbox, letting analysts monitor system calls.

Data flow tracking is another approach that tracks the flow of data through the malware, and data tainting is an established technique in this area. It involves marking (or "tainting") specific data components and then monitoring how this tainted data propagates through the system. Data tainting could be utilized in static analysis, but due to some evasion strategies like encryption and obfuscation it has endured challenges in practice [12]. SELECTIVE-TAINT [13] was invented to address performance overhead issue by employing static binary rewriting to selectively instrument only instructions related to taint analysis.

3.2 Kernel Modules

Kernel modules are a mechanism that allows for the extension of kernel functions without modifying the kernel's source code by loading object files during the execution of the Linux kernel. Kernel modules are not subject to constraints like the eBPF verifier, thus offering a high degree of program freedom. However, since kernel modules are executed with the same privileges as the kernel, it is necessary to develop carefully to avoid embedding vulnerabilities [14].

As Mayer et al. [15] point out, avoiding the creation of kernel modules can be considered an advantage of eBPF.

4 Proposed Method

Existing specification-based malware detectors can be utilized if we extract the correlation between shadow processes efficiently, because with that correlation we are able to reconstruct the original system call graph from system call sequences of shadow processes. In this section, we propose the design of a system method to extract the correlation.

4.1 Problem Scope

We focus on only shadow attacks that performs Inter-Process Communication, or IPC, via unix domain socket.

[1] showed prototype implementation of a compiler that takes existing malware as input and outputs the executable of malware of shadow-attack version. Therefore, it is reasonable to infer that shadow attacks based on IPC through unix domain socket are highly feasible, and they should be considered as a significant threat.

4.2 Key Concepts (tentative)

As mentioned before, shadow attacks are the strategy where malware exports its critical system calls to shadow processes and hyde its malicious behavior. [1] listed examples of system calls that are critical for malware's intent, shown in Table 1.

Among these system calls, file-related and network-related system calls handle file descriptors: `open` and `socket` create a new file descriptor, while others access the file tied to the file descriptor or perform network communication. So shadow processes need to transfer file descriptors to each other to perform the file-related and network-related system calls. This concept is shown in Fig. 4. To our best knowledge, file descriptor transfer through unix domain socket is a technology that, although not uncommon in cloud-native environments [16,17], is relatively rare in traditional Linux server environments (that would be because the technology introduces unnecessary complexity and overhead in the system). This situation makes the file descriptor transfer a unique characteristic of shadow attacks.

4.3 Design Overview

The overview of proposed method is shown in Fig. 5. Our system consists of the three components: the eBPF program, the eBPF map, and the user space program. Each of them are described in detail in the following subsections.

4.3.1 Details of eBPF Program

We use fentry/fexit [5], eBPF features that were introduced in kernel version 5.5, to efficiently attach the eBPF program to the entry of exit point of specific system calls.

In this system the eBPF program is attached to the file-related system calls such as `open` or `read`, and gets the value of file descriptor passed to the system call as an argument. It retrieves the pointer of `task_struct` [19] of the process that issued the system call, and dereferences it to read the file descriptor array in a way that meets the demand of eBPF verifier. Using this array, it reads the pointer to the open file object (named `struct file` [20] internally) corresponding to the specified file descriptor.

Then the program populates the eBPF map with the entry of data structure that contains pid (process id), ppid (parent pid), an integer that indicates which system call was issued, and the pointer value to the file structure.

4.3.2 Details of eBPF Map: Ring Buffer

eBPF map is a data structure that allows data sharing between user space and kernel space, and we select ring buffer as the type of eBPF map for this system because read from user space and write from kernel space happen at the same rate.

As described in the previous subsection, the eBPF program populates the ring buffer every time the file-related system call is issued.

4.3.3 Details of User Space Program

It needs to be noticed that some implementation details remain to be finalized, which could lead to further refinements in the future.

The user space program stores which process accesses a specific open file object (or `struct file`) as a key-value pair in a hash table, and infers that there is a correlation between the processes A and B if both of them access the same object. The `task_structs` of different processes have open file objects separately even if they are opening the same file. How-

Table 1: Examples of critical system calls. This table is reconstructed from [1] by the author from.

Function Category	**System Call**
File I/O operation	open, read, write
Network	socket, connect, recv, send, read, write
Process management	exec, execl

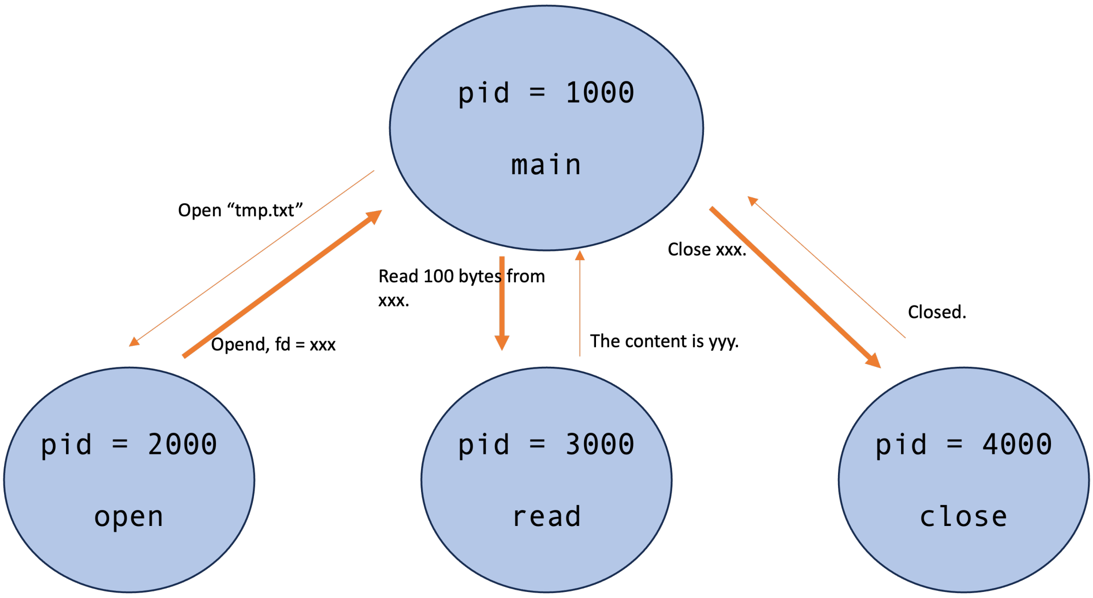

Fig. 4: Illustration of the concept of file descriptor transfer between shadow processes. The main process requests the shadow process to execute system calls, along with the required arguments. The thick arrows indicate the transfer of file descriptors over the control messages exchanged through IPC.

ever, when the file objects are shared between the processes explicitly via unix domain socket, or when they are in parent-child relationship, the open file objects are shared. Since the program knows whether given two processes are in parent-child relationship or not by checking pid and ppid, it can infer that the processes are transferring file descriptors through unix domain socket, and thus they are shadow processes if they access the same open file object and they are independent processes.

参考文献

[1] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. Shadow attacks: automatically evading system-call-behavior based malware detection. *Journal in Computer Virology and Hacking Techniques*, 8:1–13, 2012.

[2] Clemens Kolbitsch, Paolo Comparetti, Christopher Kruegel, Engin Kirda, Xiao-yong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. pages 351–366, 01 2009.

[3] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[4] Linux_3.18 - linux kernel newbies. `https://kernelnewbies.org/Linux_3.18`. (Accessed on 11/05/2023).

[5] Liz Rice. Learning ebpf. O'Reilly, March 2023.

[6] What is eBPF? An Introduction and Deep Dive into the eBPF Technology. `https://ebpf.io/what-is-ebpf/`. (Accessed on 11/05/2023).

[7] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Verifying the verifier: ebpf range analysis verification. In *International Conference on Computer Aided Verification*, pages 226–251. Springer, 2023.

[8] S Sibi Chakkaravarthy, D Sangeetha, and V Vaidehi. A survey on malware analysis and mitigation techniques. *Computer Science Review*, 32:1–23, 2019.

[9] Google. Virustotal. `https://www.virustotal.com`.

[10] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5), sep 2019.

[11] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2):32–39, 2007.

[12] Abdullah Mujawib Alashjee, Salahaldeen Duraibi, and

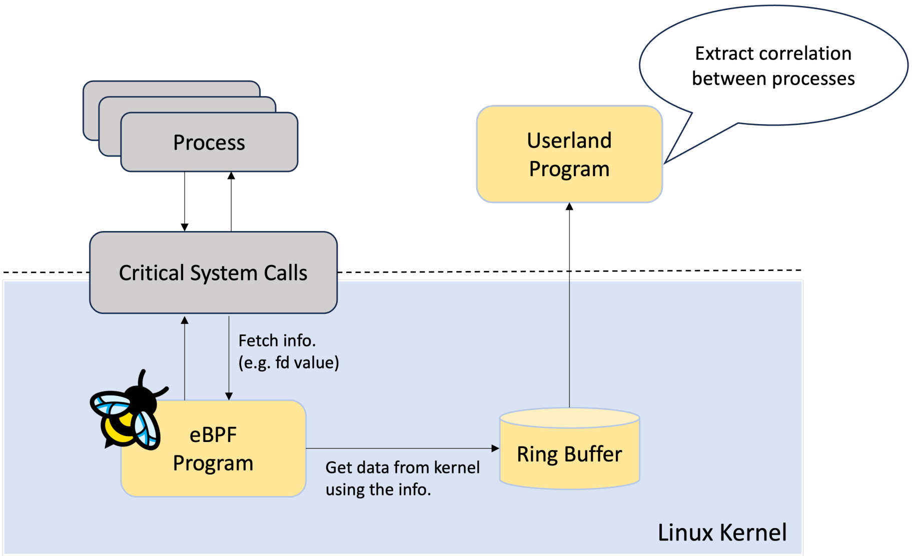
Extract correlation between processes
Process
Userland Program
Critical System Calls
Fetch info. (e.g. fd value)
eBPF Program
Get data from kernel using the info.
Ring Buffer
Linux Kernel

Fig. 5: Overview of the proposed method. Our research is not affiliated with or otherwise sponsored by eBPF Foundation [18].

Jia Song. Dynamic taint analysis tools: A review. *International Journal of Computer Science and Security (IJCSS)*, 13(6):231–244, 2019.

[13] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. {SelectiveTaint}: Efficient data flow tracking with static binary rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1665–1682, 2021.

[14] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, pages 1–5, 2011.

[15] Andrea Mayer, Pierpaolo Loreti, Lorenzo Bracciale, Paolo Lungaroni, Stefano Salsano, and Clarence Filsfils. Performance monitoring with hˆ2: Hybrid kernel/ebpf data plane for srv6 based hybrid sdn. *Computer Networks*, 185:107705, 2021.

[16] Envoy proxy - home. `https://www.envoyproxy.io/`.

[17] Haproxy - the reliable, high perf. tcp/http load balancer. `https://www.haproxy.org/`.

[18] Brand Guidelines – eBPF. `https://ebpf.foundation/brand-guidelines/`.

[19] Linux. sched.h - include/linux/sched.h - linux source code (v6.8.9) - bootlin. `https://elixir.bootlin.com/linux/latest/source/include/linux/sched.h#L748`.

[20] Linux. fs.h - include/linux/fs.h - linux source code (v6.8.9) - bootlin. `https://elixir.bootlin.com/linux/latest/source/include/linux/fs.h#L994`.