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

BY NC ND

This work is licensed under Creative Commons **Attribution – Non Commercial – Non Derivatives**

SUMMARY

Keywords:

DEDICATION

ABSTRACT

CONTENTS

1. INTRODUCTION 1
 1.1. Motivation 1
 1.2. Project objectives 3
 1.3. Regulatory framework 4
  1.3.1. Social and economic environment 4
  1.3.2. Budget 4
 1.4. Structure of the document 4
 1.5. Code availability 4
2. BACKGROUND 5
 2.1. BPF 5
  2.1.1. Introduction to the BPF system 5
  2.1.2. The BPF virtual machine 6
  2.1.3. Analysis of a BPF filter program 7
  2.1.4. BPF bytecode instruction format 8
  2.1.5. An example of BPF filter with tcpdump 10
 2.2. Modern eBPF 11
  2.2.1. eBPF instruction set 13
  2.2.2. JIT compilation 13
  2.2.3. The eBPF verifier 14
  2.2.4. eBPF maps 15
  2.2.5. The eBPF ring buffer 16
  2.2.6. The bpf() syscall 16
  2.2.7. eBPF helpers 17
 2.3. eBPF program types 18
  2.3.1. XDP 18
  2.3.2. Traffic Control 20
  2.3.3. Tracepoints 21
  2.3.4. Kprobes 22

2.3.5. Uprobes 22

2.4. Developing eBPF programs 23

2.4.1. BCC 23

2.4.2. Bpftool 24

2.4.3. Libbpf 24

2.5. Security features in eBPF 26

2.5.1. Access control 26

2.6. Memory management in Linux 28

2.6.1. Memory pages and faults 28

2.6.2. Process virtual memory 30

2.6.3. The process stack. 32

2.7. Attacks at the stack. 35

2.7.1. Buffer overflow. 36

2.7.2. Return oriented programming attacks 39

2.8. Networking fundamentals in Linux. 41

2.8.1. An overview on the network layer 41

2.8.2. Introduction to the TCP protocol 43

2.9. ELF binaries 45

2.9.1. The ELF format and Lazy Binding 45

2.9.2. Hardening ELF binaries 49

3. ANALYSIS OF OFFENSIVE CAPABILITIES 52

3.1. eBPF maps security 52

3.2. Abusing tracing programs 53

3.2.1. Access to function arguments 53

3.2.2. Reading memory out of bounds. 56

3.2.3. Overriding function return values. 57

3.2.4. Sending signals to user programs 58

3.2.5. Conclusion 58

3.3. Memory corruption 58

3.3.1. Attacks and limitations of bpf_probe_write_user() 59

3.3.2. Conclusion 61

3.4. Abusing networking programs 62
  3.4.1. Attacks and limitations of networking programs 62
  3.4.2. Conclusion 65
4. DESIGN OF A MALICIOUS EBPF ROOTKIT 66
4.1. Library injection attacks 66
  4.1.1. ROP with eBPF. 67
  4.1.2. Bypassing hardening features in ELFs 69
  4.1.3. Library injection via GOT hijacking 71
5. EVALUATION 73
6. RELATED WORK 74
BIBLIOGRAPHY. 75

LIST OF FIGURES

2.1 Sketch of the functionality of classic BPF 6

2.2 Execution of a BPF filter. 7

2.3 Table of supported classic BPF instructions, as shown by McCanne and Jacobson[18] 8

2.4 Table explaining the column address modes in Figure2.3, as shown by McCanne and Jacobson[19] 9

2.5 BPF bytecode tcpdump needs to set a filter to display packets directed to port 80. 10

2.6 Shortest path in the CFG described in the example of figure 2.5 that a packet needs to follow to be accepted by the BPF filter set with *tcpdump*. . 11

2.7 Figure showing overall eBPF architecture in the Linux kernel and the process of loading an eBPF program. Based on[22] and [23]. 12

2.8 Figure showing how the eBPF XDP and TC modules are integrated in the network processing in the Linux kernel. 19

2.9 Sketch of the compilation and loading process of a program developed with libbpf. 25

2.10 Memory translation of virtual pages to physical pages. 28

2.11 Major page fault after a page was removed from RAM. 29

2.12 Minor page fault after a fork() in which the page table was not copied completely. 30

2.13 Virtual memory architecture of a process[54]. 31

2.14 Simplified stack representation showing only stack frames. 32

2.15 Representation of push and pop operations in the stack. 33

2.16 Stack representation right before starting the function call process. . . . 34

2.17 Stack representation right after the function preamble. 34

2.18 Execution hijack overwriting saved rip value. 36

2.19 Stack buffer overflow overwriting ret value. 37

2.20 Executing arbitrary code exploiting a buffer overflow vulnerability. . . . 38

2.21 Steps for executing code sample using ROP. 40

2.22 Ethernet frame with TCP/IP packet. 42

2.23 TCP 3-way handshake. 44
2.24 TCP packet retransmission on timeout. 45
2.25 PLT stub for timerfd_settime, seen from gdb-peda. 48
2.26 Inspecting address stored in GOT section before dynamic linking, seen from gdb-peda. 48
2.27 Inspecting address stored in GOT section after dynamic linking, seen from gdb-peda. 48
2.28 Glibc function to which PLT jumps using address stored at GOT, seen from gdb-peda. 48

3.1 Overview of stack scanning and writing technique. 60
3.2 Technique to duplicate a packet for exfiltrating data. 64

4.1 Initial setup for the ROP with eBPF technique. 67
4.2 Process memory after syscall exits and ROP code overwrites the stack. . . 68
4.3 Stack data is restored and program continues its execution. 69
4.4 Two runs of the same executable using ASLR, showing a library and two symbols. 70
4.5 Call to the glibc function, using objdump 72

LIST OF TABLES

2.1 Table showing BPF instruction format. It is a fixed-length 64 bit instruction, the number of bits used by each field are indicated. 8

2.2 Table showing relevant eBPF updates. Note that only those relevant for our research objectives are shown. This is a selection of the official complete table at [21]. 12

2.3 Table showing eBPF instruction format. It is a fixed-length 64 bit instruction, the number of bits used by each field are indicated. 13

2.4 Table showing eBPF registers and their purpose in the BPF VM.[24][26]. 13

2.5 Table showing common fields for creating an eBPF map. 15

2.6 Table showing types of eBPF maps. Only those used in our rootkit are displayed, the full list can be consulted in the man page [37] 15

2.7 Table showing types of syscall actions. Only those relevant to our research are shown the full list and attribute details can be consulted in the man page [37] 16

2.8 Table showing types of eBPF programs. Only those relevant to our research are shown. The full list and attribute details can be consulted in the man page [37]. 17

2.9 Table showing common eBPF helpers. Only those relevant to our research are shown. Those helpers exclusive to an specific program type are not listed. The full list and attribute details can be consulted in the man page [38]. 18

2.10 Table showing XDP relevant return values. 19

2.11 Table showing relevant XDP-exclusive eBPF helpers. 20

2.12 Table showing TC relevant return values. Full list can be consulted at [44]. 21

2.13 Table showing relevant TC-exclusive eBPF helpers. 21

2.14 Table showing BPF skeleton functions. 25

2.15 Kernel compilation flags for eBPF. 26

2.16 Capabilities needed for eBPF. 27

2.17 Values for unprivileged eBPF kernel parameter. 27

2.18 Relevant registers in x86_64 for the stack and control flow and their purpose. 32

2.19 Relevant TCP flags and their purpose. 43

2.20 Tools used for analysis of ELF programs. 46

2.21 Tools used for analysis of ELF programs. 47

2.22 Security features in C compilers used in the study. 49

3.1 Argument passing convention of registers for function calls in user and kernel space respectively. 54

1. INTRODUCTION

1.1. Motivation

As the efforts of the computer security community grow to protect increasingly critical devices and networks from malware infections, so do the techniques used by malicious actors become more sophisticated. Following the incorporation of ever more capable firewalls 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 on 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 functionalities depend on the type of rootkit developed, a classification usually made depending on the level of privileges on which the rootkit operates in the system.

- **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.
- **Kernel-mode** rootkits run at the same level of privilege as the operating system, thus enjoying unrestricted access to the whole computer. 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.
>
> These rootkits are usually the most attractive (and difficult to build) option for a malicious actor, but the installation of a kernel rootkit requires of 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 and research institutes 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. However, during the last years, a new technology called eBPF has been found to be 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], which provides the possibility of 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 APTs. On February 2022, the Chinese security team Pangu Lab reported about a NSA backdoor that remained unnoticed since 2013 that used eBPF for its networking functionality and that 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].

Moreover, there currently exists official efforts to extend the eBPF technology into Windows[6] and Android systems[7], which spreads the mentioned risks to new platforms. Therefore, we can confidently claim that there is a growing interest on researching the capabilities of eBPF in the context of offensive security, in particular given its potential on becoming a common component found of modern rootkits. 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 defences.

1.2. Project objectives

The main objective of this project is to compile a comprehensive report of the capabilities in the eBPF technology that could be weaponized by a malicious actor. In particular, we will be focusing 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 an specific feature, and also by building a realistic rootkit system which weaponizes these PoCs and operates malicious activities.

Before narrowing down our objectives and selecting an specific list of rootkit capabilities to emulate using eBPF, we needed to consider previous research. The work on this matter by Jeff Dileo from NCC Group at DEFCON 27[8] is particularly relevant, setting the first basis of eBPF ability 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 Afchainthe from Datadog at DEFCON 29[10], research deeper on eBPF's ability to behave like a rootkit. 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, whilst Fournier and Afchainthe 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 the previous research into account, and on the basis of common functionality we described to be usually incorporated at rootkits, the objectives of our research on eBPF is set to be on the following topics:

- Analysing eBPF's possibilities when hooking system calls and kernel functions.
- Learning eBPF's potential to read/write arbitrary memory.
- Exploring 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.

1.3. Regulatory framework

1.3.1. Social and economic environment

1.3.2. Budget

1.4. Structure of the document

1.5. Code availability

2. BACKGROUND

This chapter is dedicated to an study of 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[11], 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"[12], 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[13].

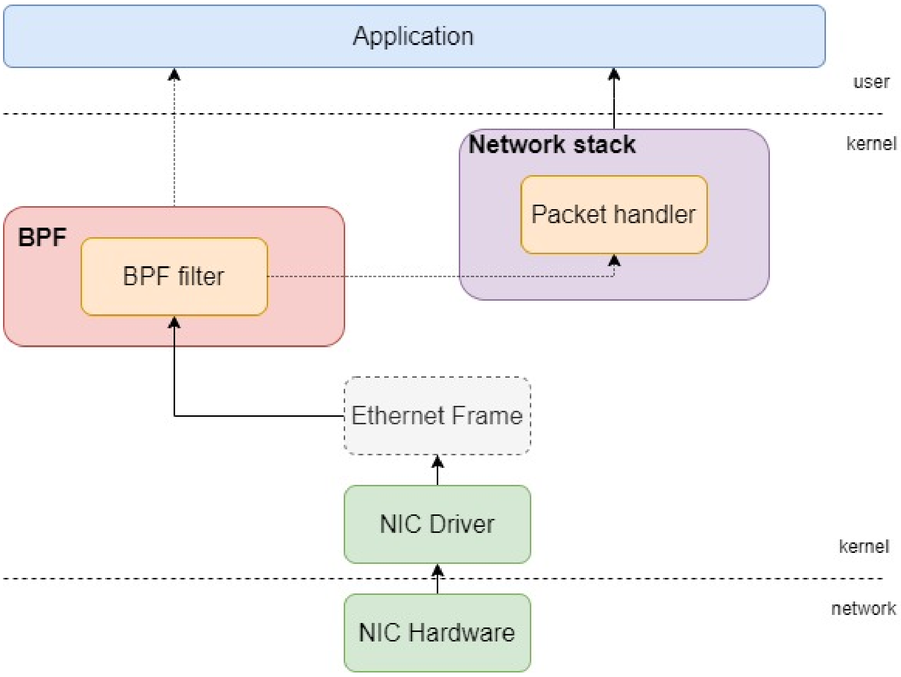


Fig. 2.1. Sketch of the functionality of classic BPF

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, 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[14], 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[15].
- **An 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).

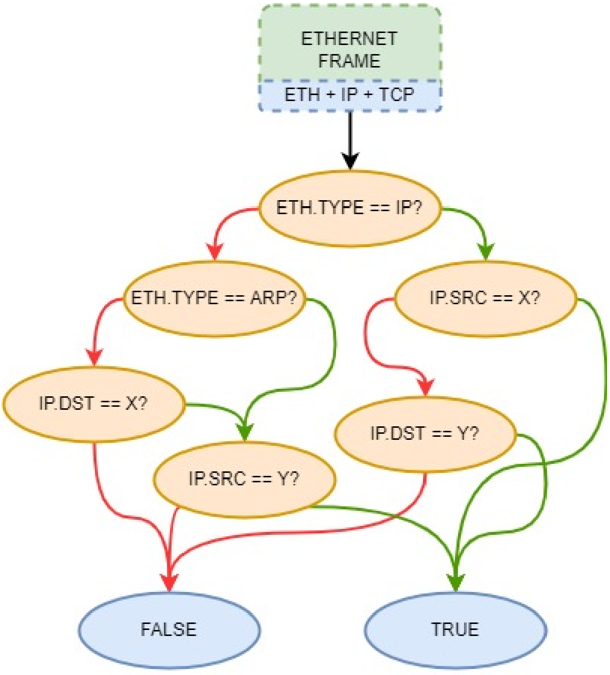


Fig. 2.2. Execution of a BPF filter.

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[16]. 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.

2.1.4. BPF bytecode instruction format

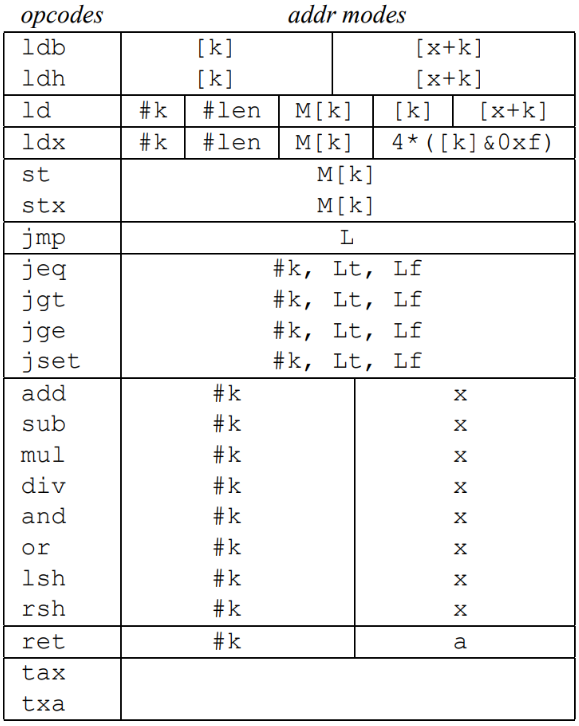
In order to implement the CFG to be run at the BPF VM, BPF filter programs are made up of BPF bytecode, which is defined by a new BPF instruction set. Therefore, a BPF filter program is an array of BPF bytecode instructions[17].

	OPCODE	JT	JF	K
BITS	16	8	8	32

Table 2.1. Table showing BPF instruction format. It is a fixed-length 64 bit instruction, the number of bits used by each field are indicated.

Table 2.1 shows the format of a BPF bytecode instruction. As it can be observed, it is a compound of:

- 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.

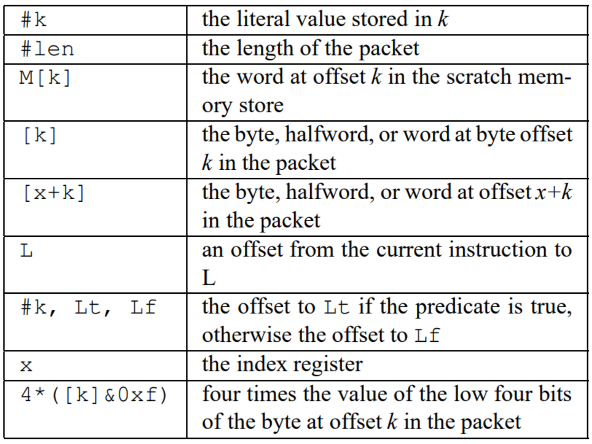


<table>
<tr><th><i>opcodes</i></th><th colspan="5"><i>addr modes</i></th></tr>
<tr><td>ldb
ldh</td><td colspan="3">[k]
[k]</td><td colspan="2">[x+k]
[x+k]</td></tr>
<tr><td>ld</td><td>#k</td><td>#len</td><td>M[k]</td><td>[k]</td><td>[x+k]</td></tr>
<tr><td>ldx</td><td>#k</td><td>#len</td><td>M[k]</td><td colspan="2">4*([k]&0xf)</td></tr>
<tr><td>st
stx</td><td colspan="5">M[k]
M[k]</td></tr>
<tr><td>jmp</td><td colspan="5">L</td></tr>
<tr><td>jeq
jgt
jge
jset</td><td colspan="5">#k, Lt, Lf
#k, Lt, Lf
#k, Lt, Lf
#k, Lt, Lf</td></tr>
<tr><td>add
sub
mul
div
and
or
lsh
rsh</td><td colspan="3">#k
#k
#k
#k
#k
#k
#k
#k</td><td colspan="2">x
x
x
x
x
x
x
x</td></tr>
<tr><td>ret</td><td colspan="3">#k</td><td colspan="2">a</td></tr>
<tr><td>tax
txa</td><td colspan="5"></td></tr>
</table>

Fig. 2.3. Table of supported classic BPF instructions, as shown by McCanne and Jacobson[18]

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[19]:

- 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).



`#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

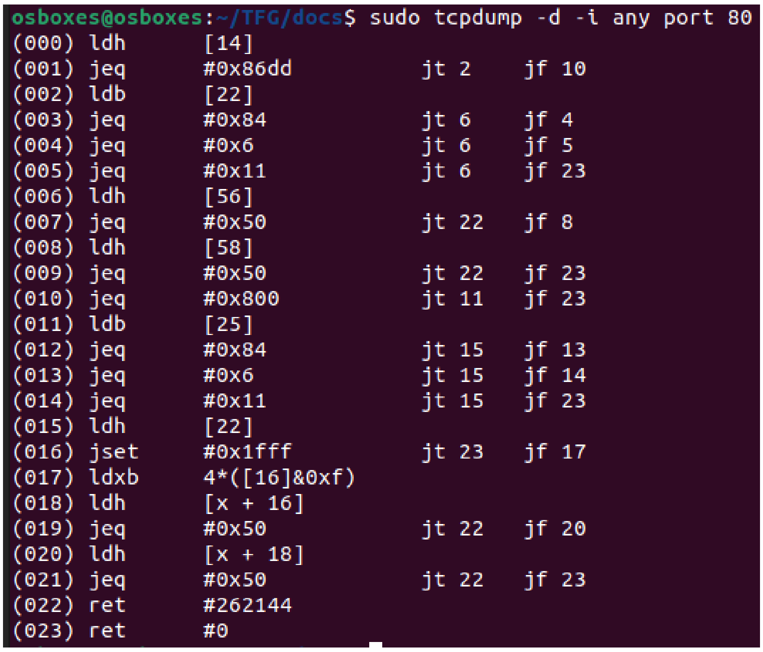
Fig. 2.4. Table explaining the column address modes in Figure2.3, as shown by McCanne and Jacobson[19]

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, paremeters 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 an user-level application, BPF offered a performance improvement between 10 and 150 times the state-of-the art technologies of the moment[14]. Since then, multiple popular tools began to use BPF, such as the network tracing tool *tcpdump*[20].

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. We will now show 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.

Figure 2.5 shows 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.

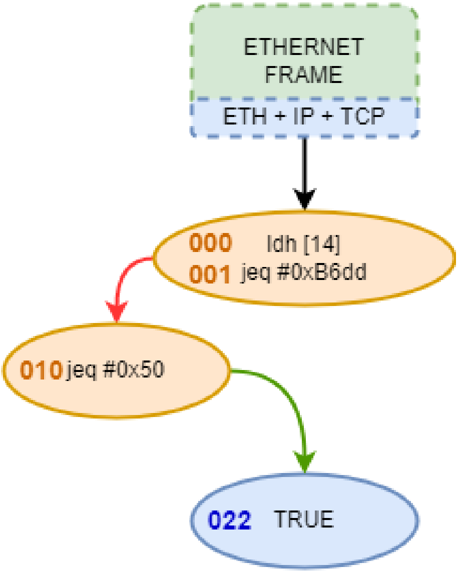


Fig. 2.6. Shortest path in the CFG described in the example of figure 2.5 that a packet needs to follow to be accepted by the BPF filter set with *tcpdump*.

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 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
New 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. Table showing relevant eBPF updates. Note that only those relevant for our research objectives are shown. This is a selection of the official complete table at [21].

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[22].

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

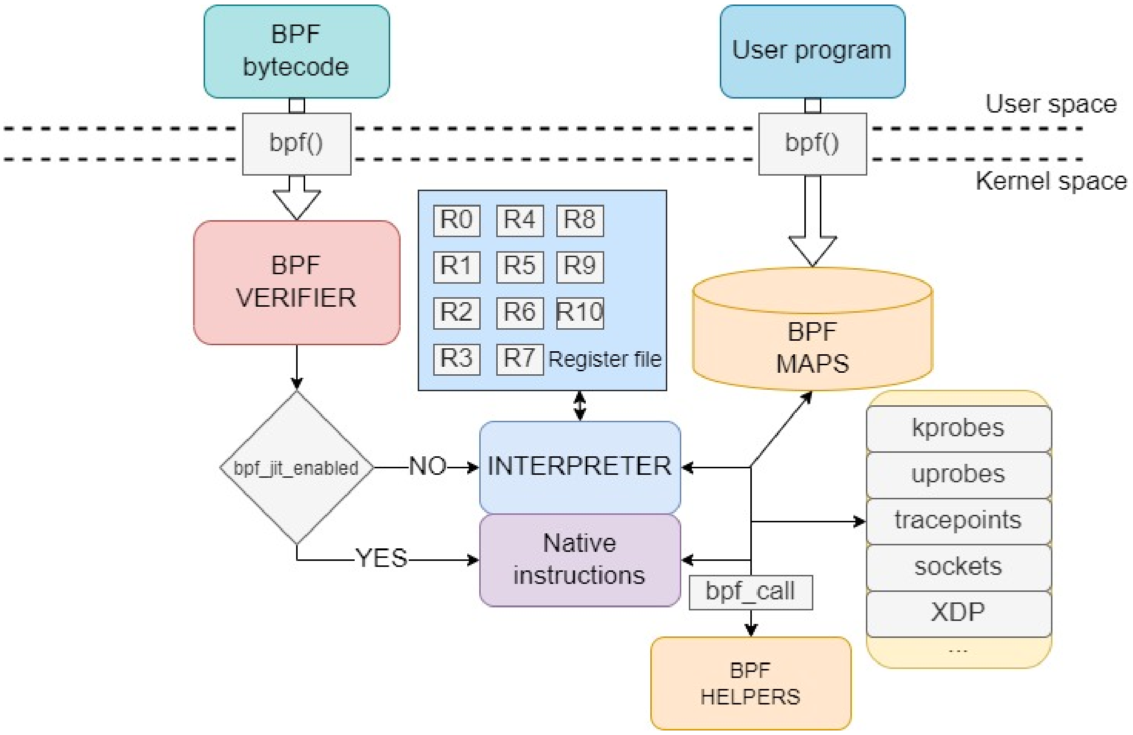

BPF bytecode
User program
User space
bpf()
bpf()
Kernel space
R0 R4 R8
R1 R5 R9
R2 R6 R10
R3 R7 Register file
BPF VERIFIER
BPF MAPS
bpf_jit_enabled
NO
YES
INTERPRETER
Native instructions
bpf_call
BPF HELPERS
kprobes
uprobes
tracepoints
sockets
XDP
...

Fig. 2.7. Figure showing overall eBPF architecture in the Linux kernel and the process of loading an eBPF program. Based on[22] and [23].

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.

	IMM	OFF	SRC	DST	OPCODE
BITS	32	16	4	4	8

Table 2.3. Table showing eBPF instruction format. It is a fixed-length 64 bit instruction, the number of bits used by each field are indicated.

Table 2.3 shows the new instruction format for eBPF programs[24]. The new fields are similar to x86_64 assembly, incorporating the typically found immediate and offset fields, and source and destination registers[25]. 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[24].

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. Table showing eBPF registers and their purpose in the BPF VM.[24][26].

2.2.2. JIT compilation

We mentioned in subsection 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[27][28].

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[29].

Therefore, when using JIT compiling (a setting defined by the variable *bpf_jit_enable*[30], BPF registers are translated into machine-specific registers following their one-to-one mapping and bytecode instructions are translated into machine-specific instructions[31]. There no longer exists an interpretation step by the BPF VM, since we can execute the code directly[32].

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[33][34]:

- Tests for ensuring overall control flow safety:

    No loops allowed (bounded loops accepted since kernel version 5.3[35].

    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[36].

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[37]. 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 following fields:

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. Table showing common fields for creating an eBPF map.

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 subsection 2.2.5
BPF_MAP_TYPE_PROG_ARRAY	Stores descriptors of eBPF programs

Table 2.6. Table showing types of eBPF maps. Only those used in our rootkit are displayed, the full list can be consulted in the man page [37]

Table **??** 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.

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 an 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 **??**:

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 an 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. Table showing types of syscall actions. Only those relevant to our research are shown the full list and attribute details can be consulted in the man page [37]

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.

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. Table showing types of eBPF programs. Only those relevant to our research are shown. The full list and attribute details can be consulted in the man page [37].

In section **??**, 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[38], 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 eBPF program type in the sections on which they are studied.

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. Table showing common eBPF helpers. Only those relevant to our research are shown. Those helpers exclusive to an specific program type are not listed. The full list and attribute details can be consulted in the man page [38].

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[39].

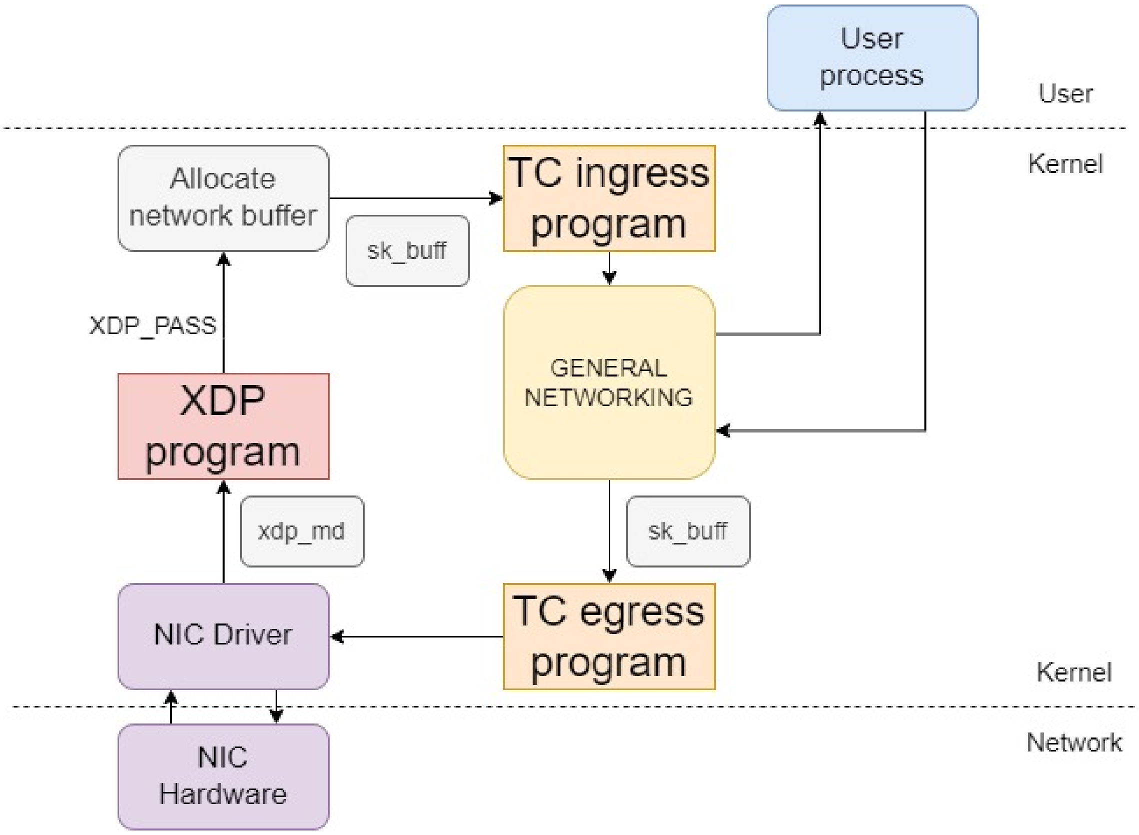

User process
User
Kernel
Allocate network buffer
TC ingress program
sk_buff
XDP_PASS
GENERAL NETWORKING
XDP program
xdp_md
sk_buff
NIC Driver
TC egress program
Kernel
Network
NIC Hardware

Fig. 2.8. Figure showing how the eBPF XDP and TC modules are integrated in the network processing in the Linux kernel.

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[40]:

- Analyse the data between the packet buffer bounds.
- Modify the packet contents, and modify the packet length.
- Decide between one of the actions displayed in table 2.10.

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. Table showing XDP relevant return values.

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

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. Table showing relevant XDP-exclusive eBPF helpers.

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:

- 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[41].
- 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 [42]. For this document, we will explain the overall process needed to load a TC program[43]:

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 an 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. Table showing TC relevant return values. Full list can be consulted at [44].

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. Table showing 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[45]. 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[46]. 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[47].

Also similarly, since tracepoints could be found in their *enter* and *exit* variations, kprobes have their counterpart, name 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 exeuction flow reaches a hooked instruction, a probe function is run.

For setting an uprobe on an 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 also exist, 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[48]. 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. An example of a BCC program is included in

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. This was in particular 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[49] is a library for loading and interacting with eBPF programs, which is currently maintained in the Linux kernel source tree[50]. 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[51]. 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).

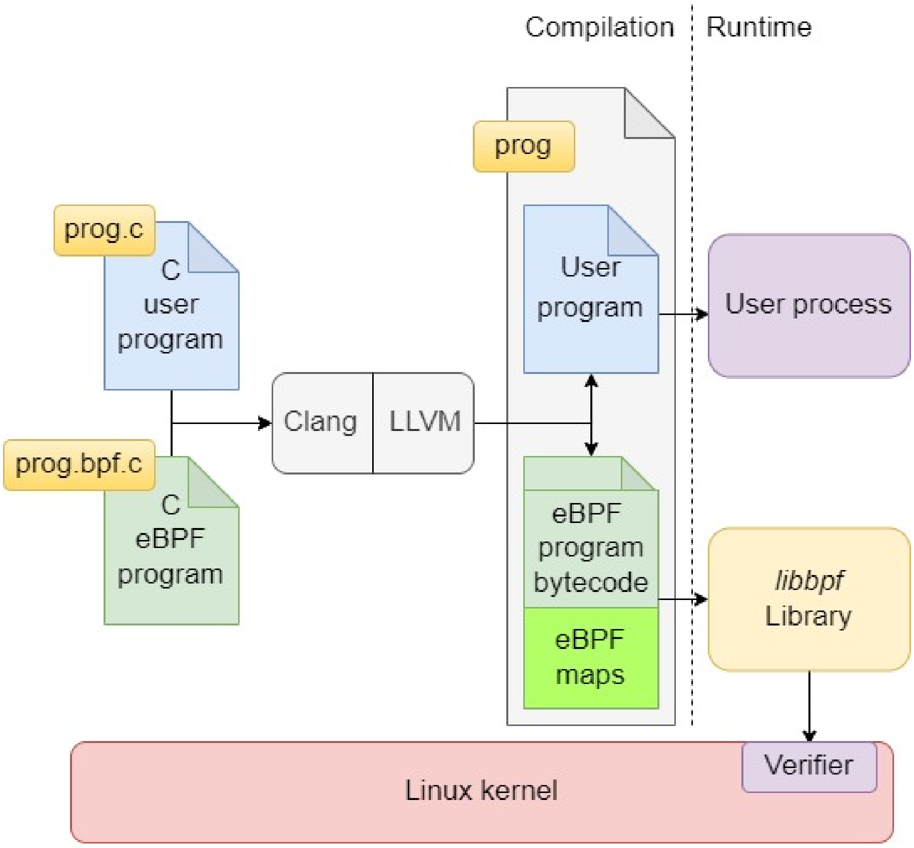

Compilation
Runtime
prog
prog.c
C
user
program
User
program
User process
Clang
LLVM
prog.bpf.c
C
eBPF
program
eBPF
program
bytecode
eBPF
maps
libbpf
Library
Verifier
Linux kernel

Fig. 2.9. Sketch of the compilation and loading process of a program developed with libbpf.

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 an 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.

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. Table showing BPF skeleton functions.

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 will be explained in section **??**.

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.

The above table is based on BCC's documentation2.15, but the full list of eBPF-related flags can be extracted in a live system via bpftool, as detailed in Annex 6. 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 an user having full root permissions, or having the Linux capability[67] 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_F 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[69][70]. 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 still 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[71]. 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[72].

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[73], Suse Linux[74] or Red Hat Linux[75], 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[52]. Figure 2.10 shows a diagram of the described architecture.

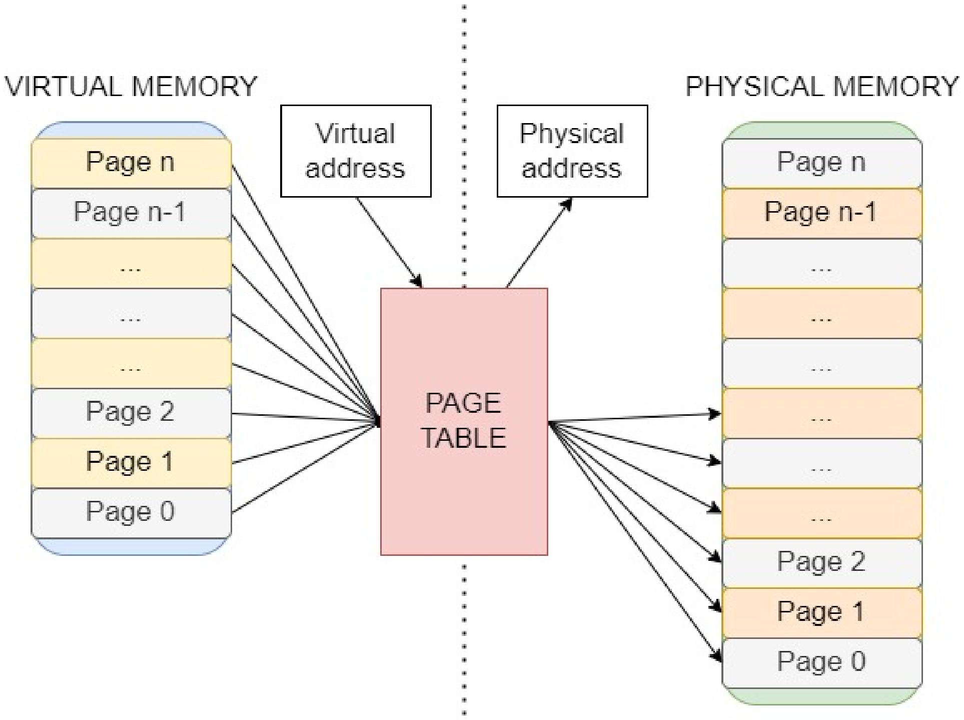


Fig. 2.10. Memory translation of virtual pages to physical pages.

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[53]:

- **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.

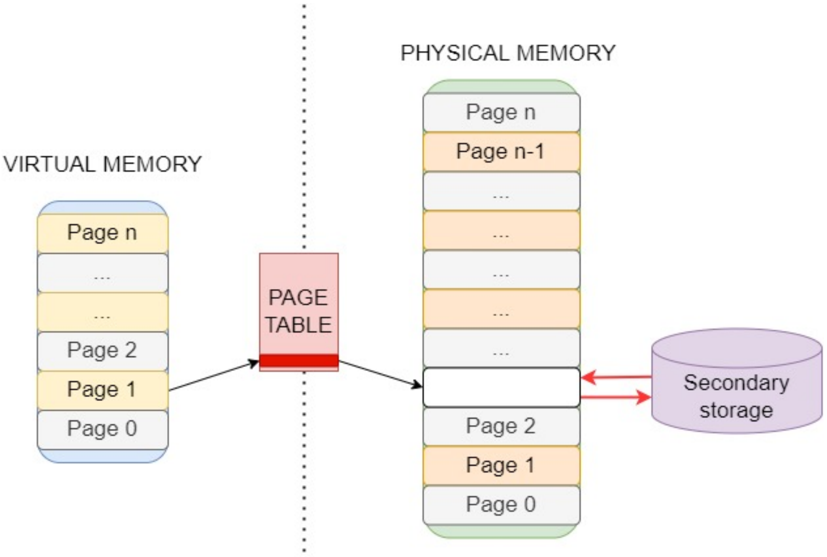


Fig. 2.11. Major page fault after a page was removed from RAM.

- **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.

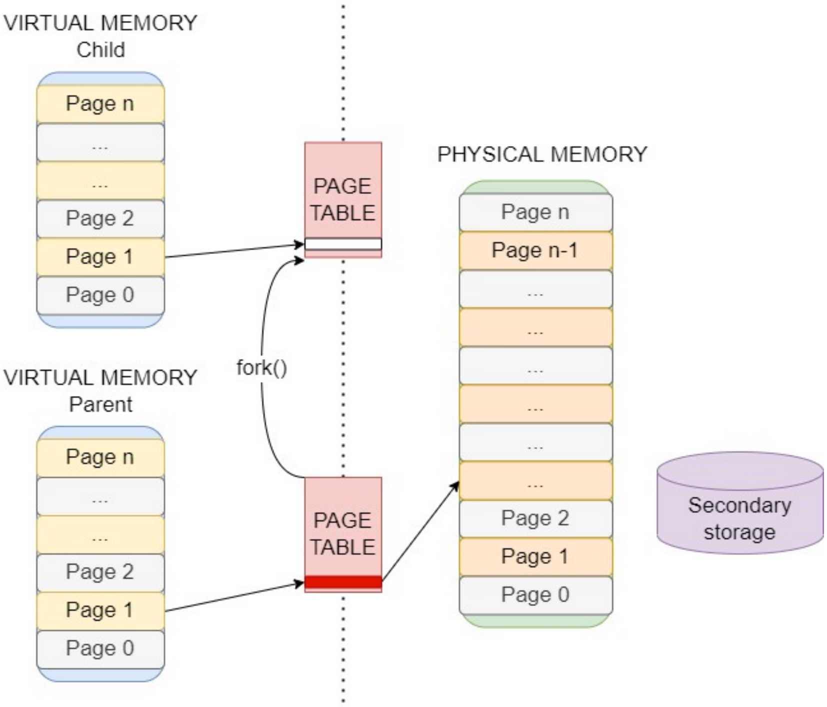

VIRTUAL MEMORY
Child
Page n
...
...
Page 2
Page 1
Page 0
VIRTUAL MEMORY
Parent
Page n
...
...
Page 2
Page 1
Page 0
fork()
PAGE
TABLE
PAGE
TABLE
PHYSICAL MEMORY
Page n
Page n-1
...
...
...
...
...
...
Page 2
Page 1
Page 0
Secondary
storage

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

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.

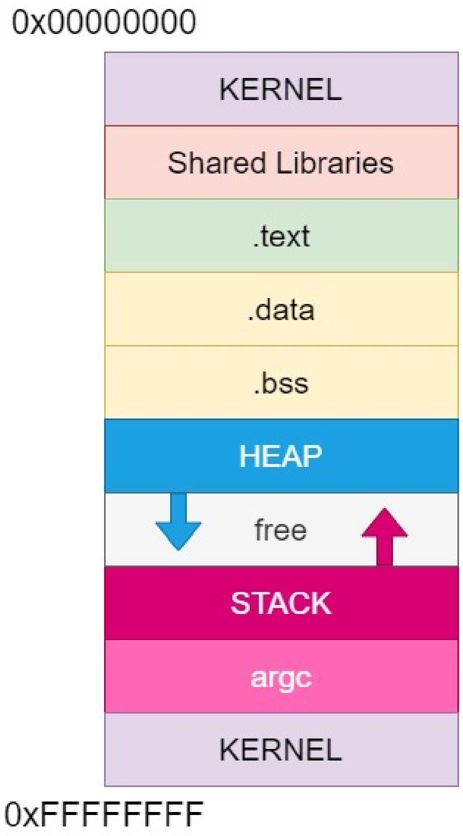


Fig. 2.13. Virtual memory architecture of a process[54].

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:

- 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 unitialized 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

Between 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:

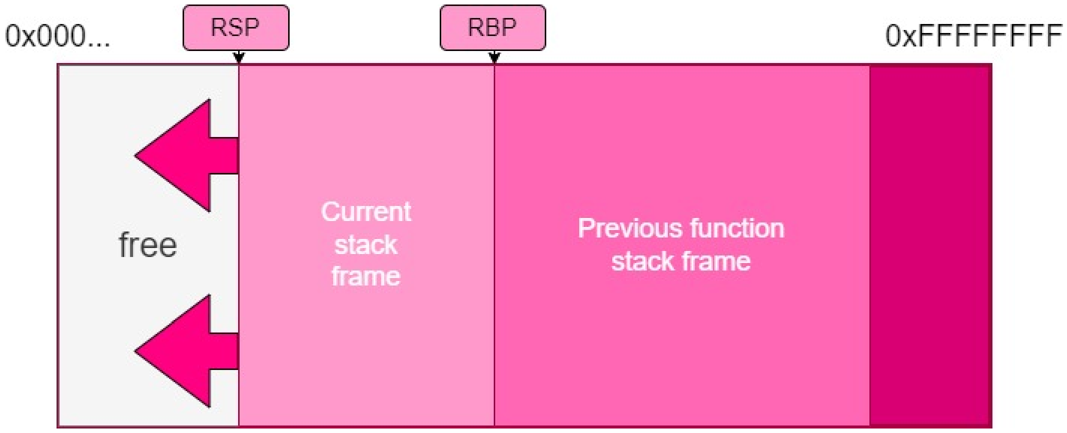


Fig. 2.14. Simplified stack representation showing only stack frames.

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.

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 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.

- 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.

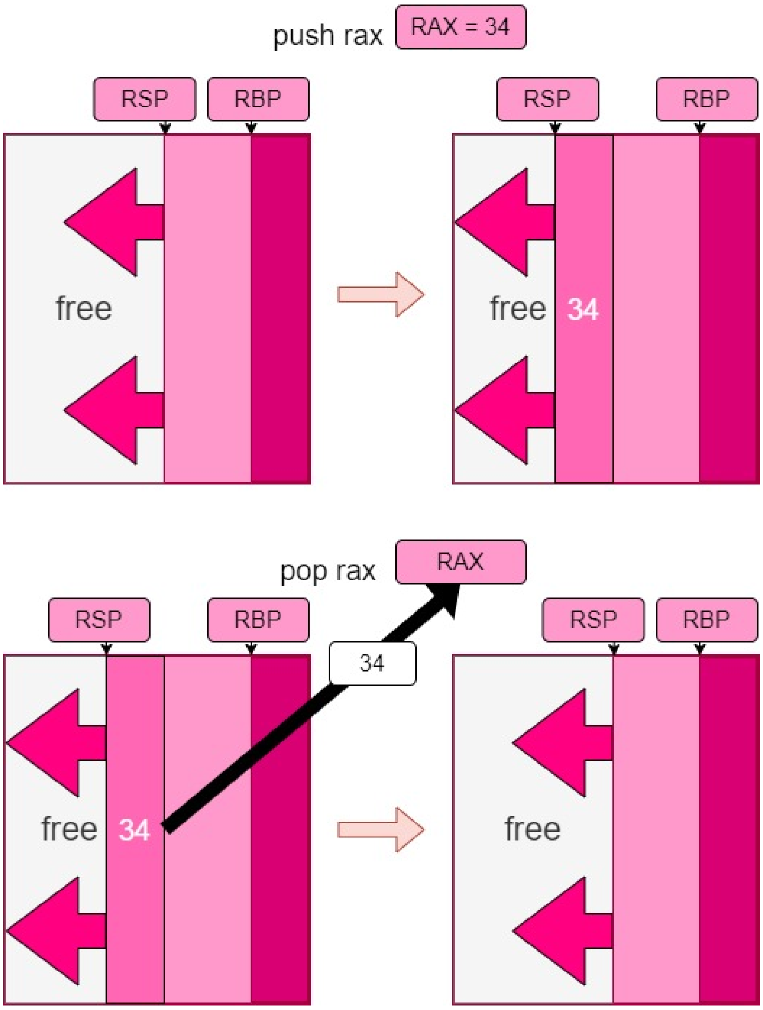


Fig. 2.15. Representation of push and pop operations in the stack.

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)` :

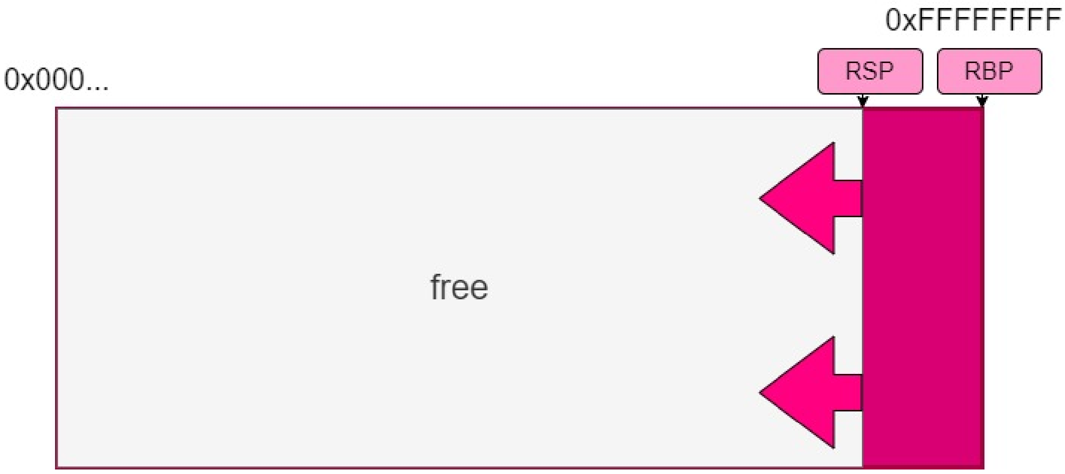

0xFFFFFFFF
0x000...
RSP
RBP
free

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

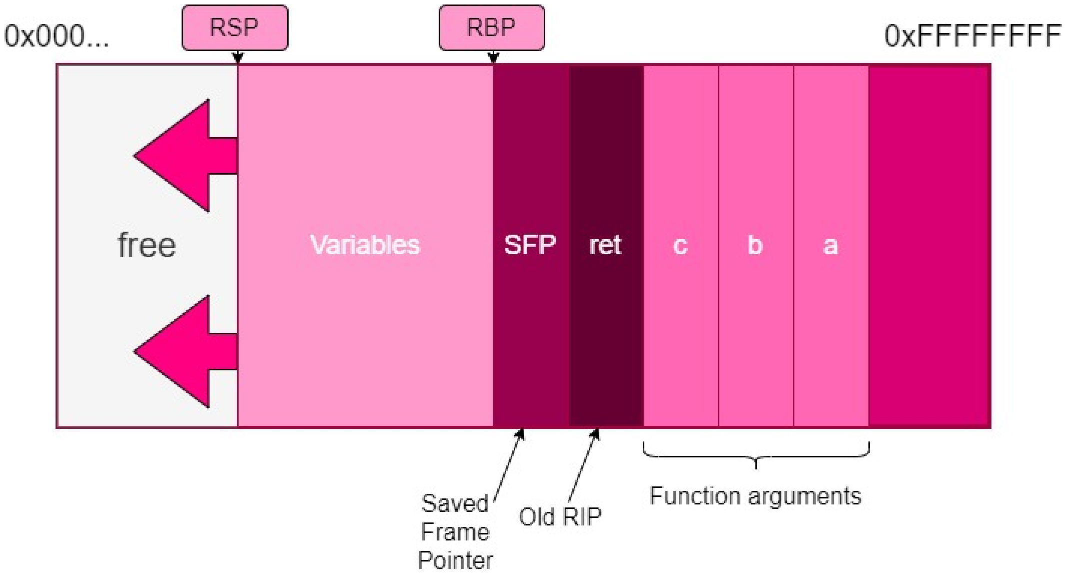

0x000...
RSP
RBP
0xFFFFFFFF
free
Variables
SFP
ret
c
b
a
Saved Frame Pointer
Old RIP
Function arguments

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

1. The function arguments are pushed into the stack. We can see them in the stack in reverse order.
2. The function is called:
   (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 the figure 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*[55], 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 the figure 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:

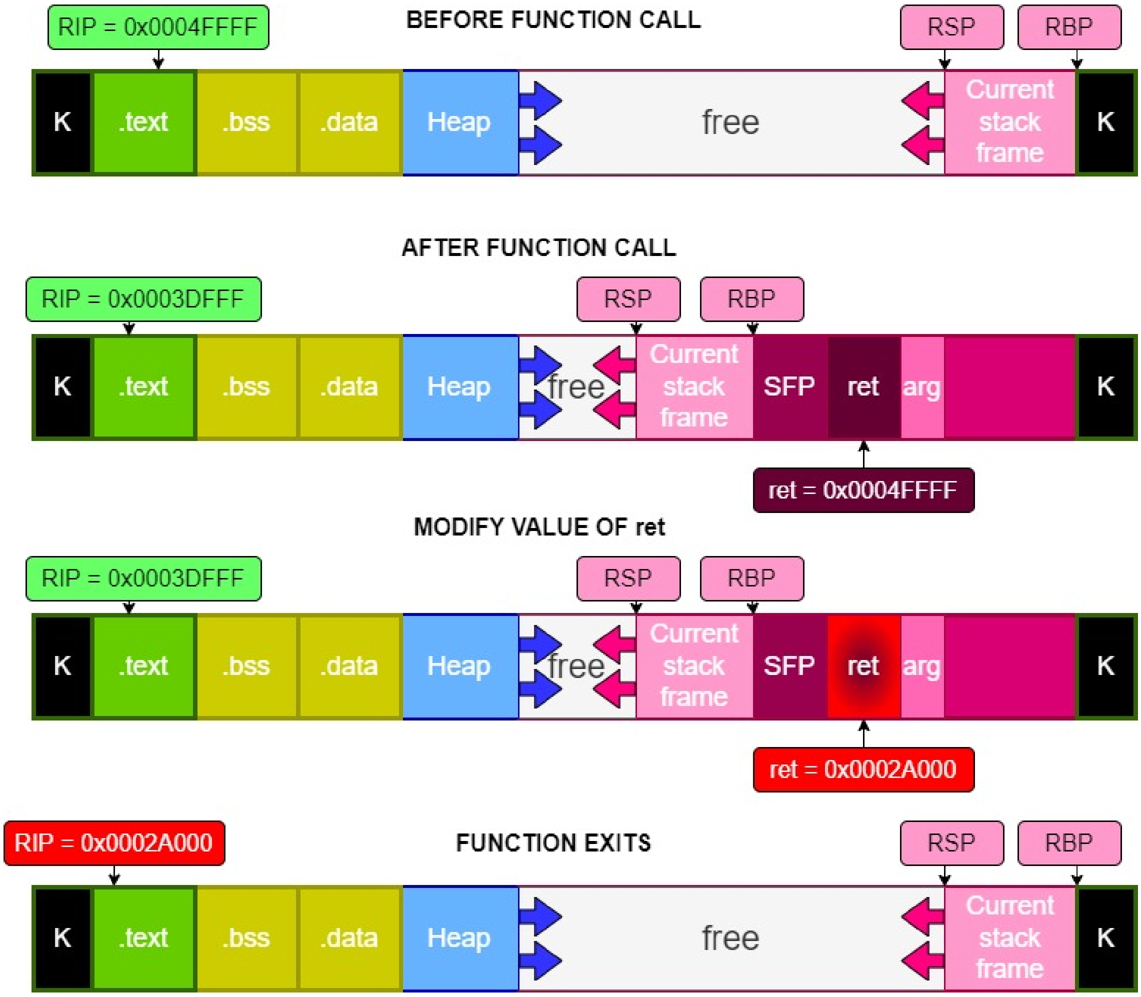

BEFORE FUNCTION CALL
RIP = 0x0004FFFF
RSP
RBP
K
.text
.bss
.data
Heap
free
Current stack frame
K
AFTER FUNCTION CALL
RIP = 0x0003DFFF
RSP
RBP
K
.text
.bss
.data
Heap
free
Current stack frame
SFP
ret
arg
K
ret = 0x0004FFFF
MODIFY VALUE OF ret
RIP = 0x0003DFFF
RSP
RBP
K
.text
.bss
.data
Heap
free
Current stack frame
SFP
ret
arg
K
ret = 0x0002A000
RIP = 0x0002A000
FUNCTION EXITS
RSP
RBP
K
.text
.bss
.data
Heap
free
Current stack frame
K

Fig. 2.18. Execution hijack overwriting saved rip value.

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 an 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.

```
void foo(char *bar){ // bar may be larger than 12 characters
   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 does not check whether the array fits in the buffer. Figure 2.19 shows how the overflow happens in the stack.

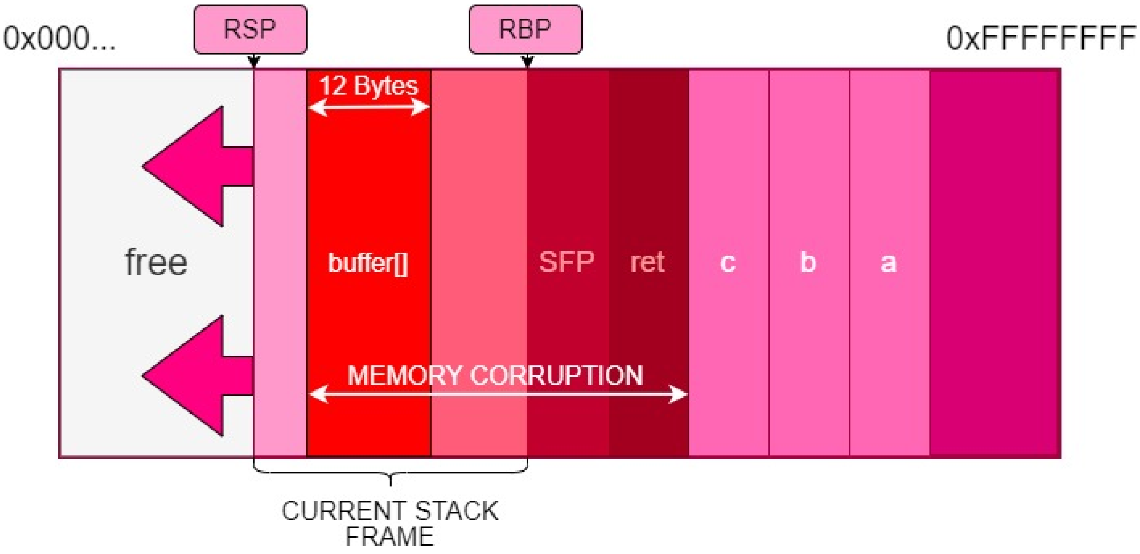


Fig. 2.19. Stack buffer overflow overwriting ret value.

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:

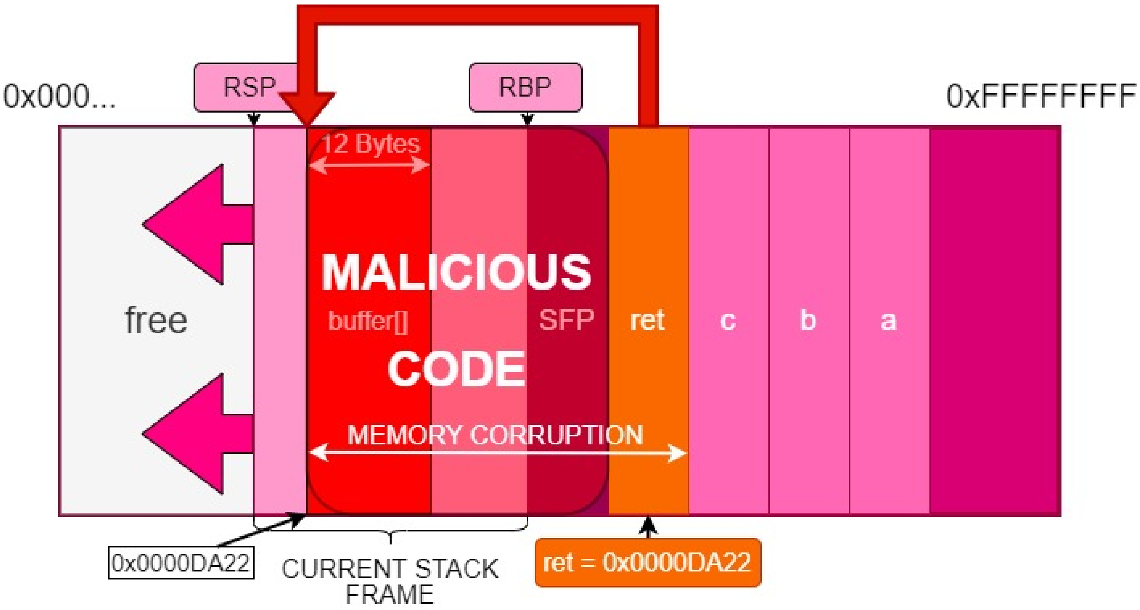


Fig. 2.20. Executing arbitrary code exploiting a buffer overflow vulnerability.

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 on instruction opcodes (machine assembly instructions translated to their representation in hexadecimal values) which the processor will execute. We will briefly explain how to write shellcode in section **??**. Therefore, in this technique the attacker will:

- Introduce a byte array that overflows the buffer, consisting on SHELLCODE + the 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 these type of exploits. This is why the attack presented here does not work work in a modern system any more.

The reason is that one of the protections consits on 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 **??**.

2.7.2. 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[56].

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 snippet 2.2:

CODE 2.2. Sample program to run using ROP.

```
mov rdx, 10
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 vale 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:

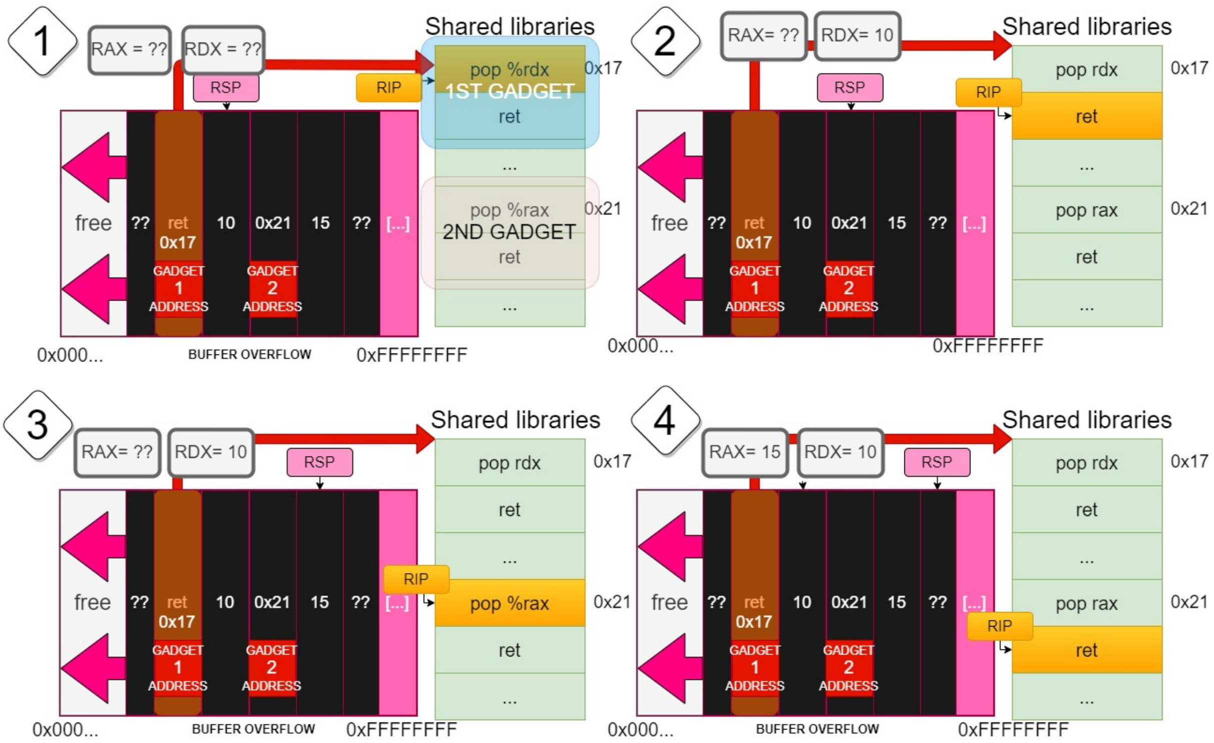

1
RAX = ??
RDX = ??
RSP
RIP
Shared libraries
pop %rdx
1ST GADGET
ret
...
pop %rax
2ND GADGET
ret
...
0x17
0x21
free
??
ret
0x17
10
0x21
15
??
[...]
GADGET
1
ADDRESS
GADGET
2
ADDRESS
0x000...
BUFFER OVERFLOW
0xFFFFFFFF
2
RAX= ??
RDX= 10
RSP
RIP
Shared libraries
pop rdx
ret
...
pop rax
ret
...
0x17
0x21
free
0x000...
0xFFFFFFFF
3
RAX= ??
RDX= 10
RSP
RIP
Shared libraries
pop rdx
ret
...
pop %rax
ret
...
0x17
0x21
free
0x000...
BUFFER OVERFLOW
0xFFFFFFFF
4
RAX= 15
RDX= 10
RSP
RIP
Shared libraries
pop rdx
ret
...
pop rax
ret
...
0x17
0x21
free
0x000...
BUFFER OVERFLOW
0xFFFFFFFF

Fig. 2.21. Steps for executing code sample 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). 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 **??**.

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:

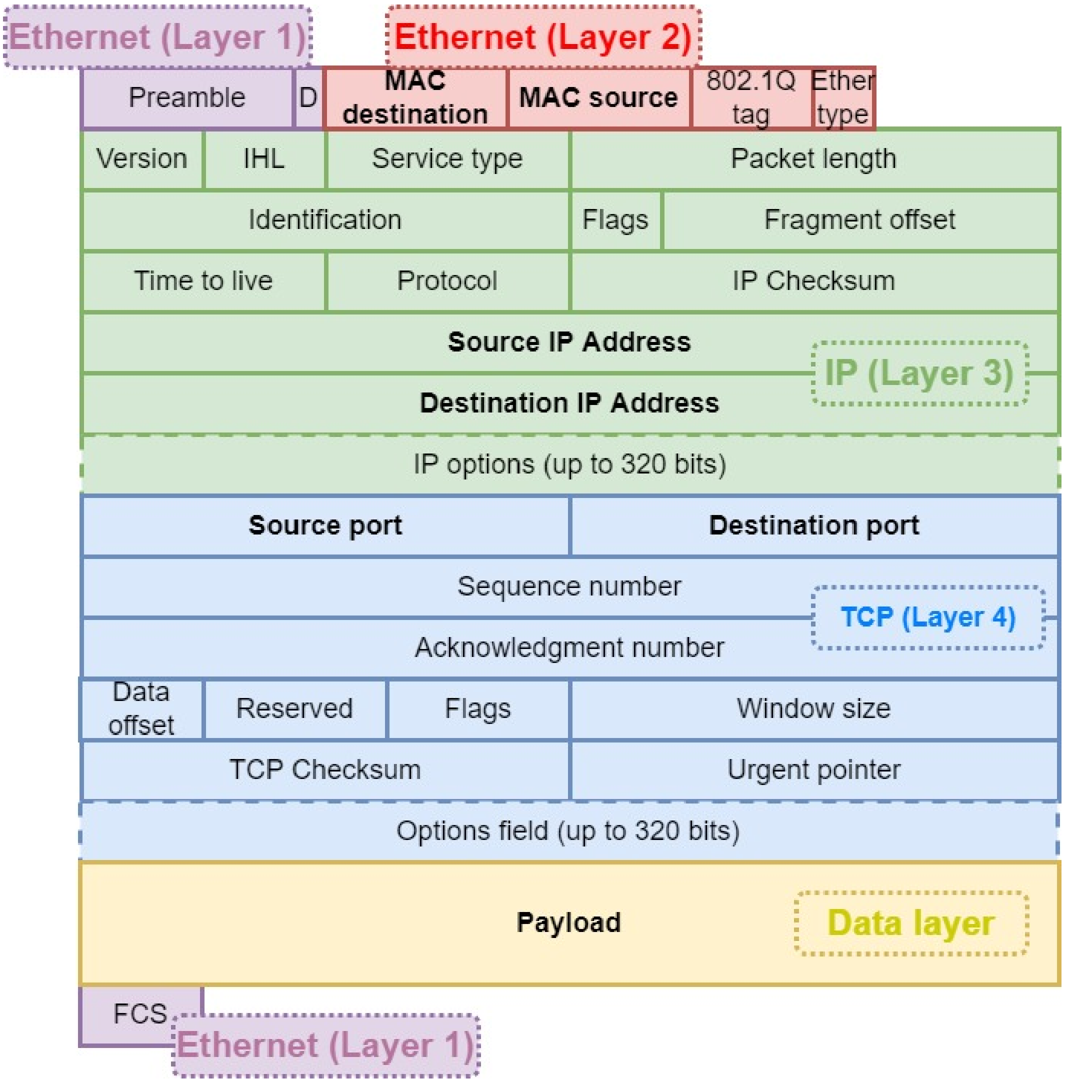

Ethernet (Layer 1)
Ethernet (Layer 2)
Preamble
D
MAC destination
MAC source
802.1Q tag
Ether type
Version
IHL
Service type
Packet length
Identification
Flags
Fragment offset
Time to live
Protocol
IP Checksum
Source IP Address
Destination IP Address
IP (Layer 3)
IP options (up to 320 bits)
Source port
Destination port
Sequence number
Acknowledgment number
TCP (Layer 4)
Data offset
Reserved
Flags
Window size
TCP Checksum
Urgent pointer
Options field (up to 320 bits)
Payload
Data layer
FCS
Ethernet (Layer 1)

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[57]:

- 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 performing 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 application 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[58], 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.

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.

Taking the above into account, figure 2.23 shows a depiction of the 3-way handshake[59]:

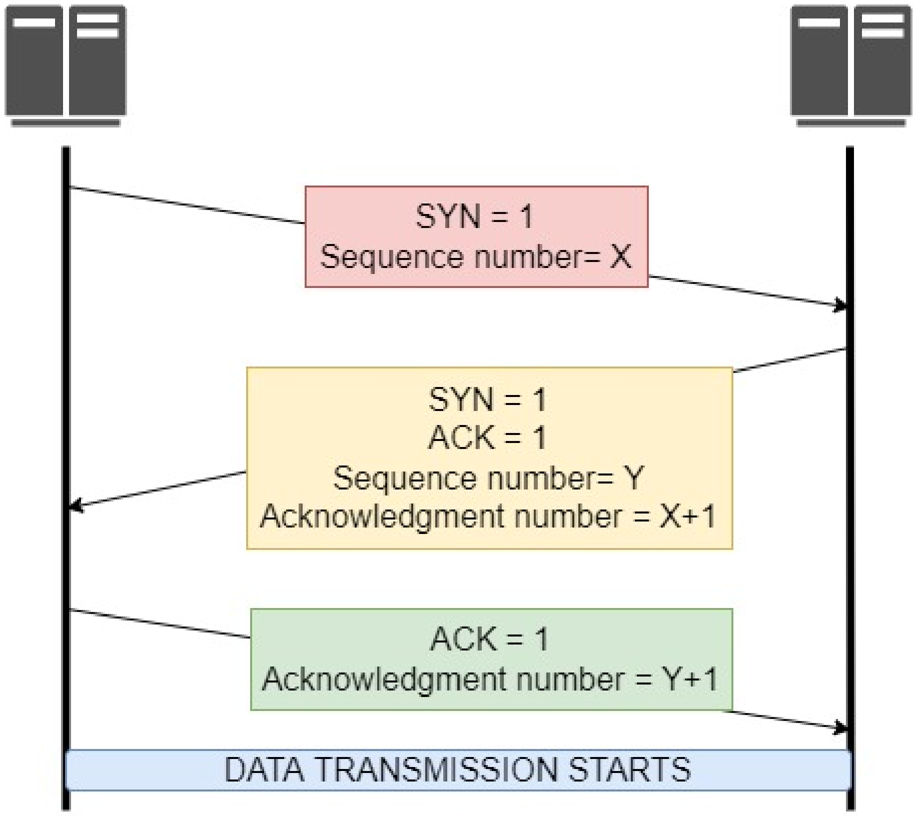

SYN = 1
Sequence number= X
SYN = 1
ACK = 1
Sequence number= Y
Acknowledgment number = X+1
ACK = 1
Acknowledgment number = Y+1
DATA TRANSMISSION STARTS

Fig. 2.23. TCP 3-way handshake.

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 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.

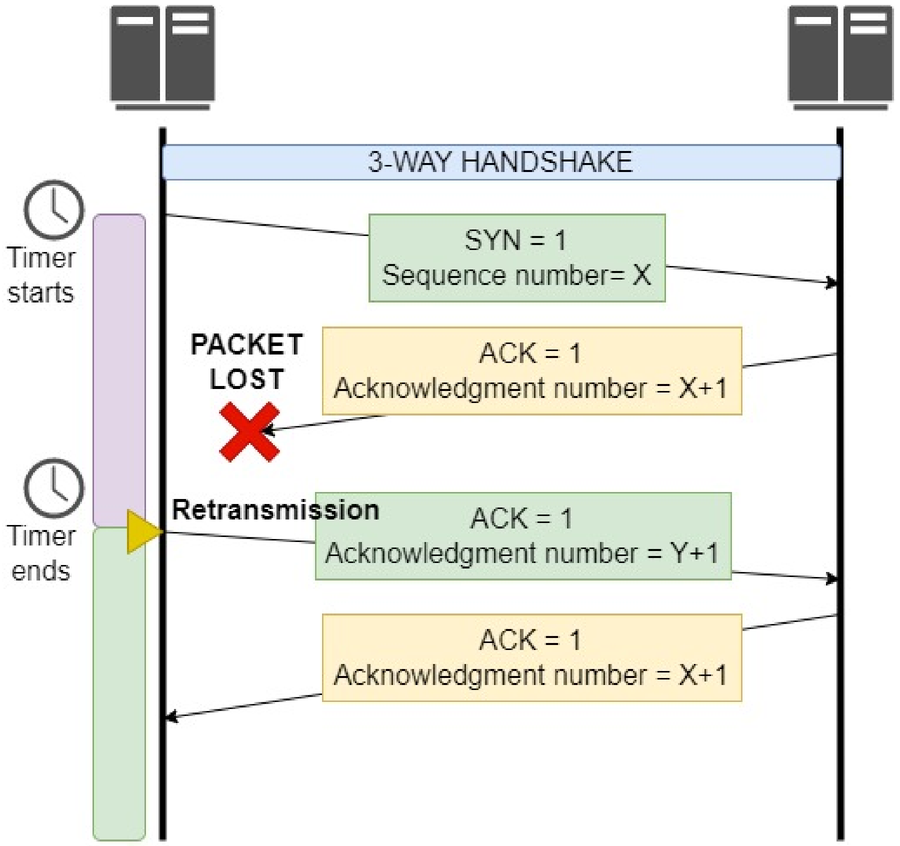

3-WAY HANDSHAKE
Timer starts
SYN = 1
Sequence number= X
PACKET LOST
ACK = 1
Acknowledgment number = X+1
Timer ends
Retransmission
ACK = 1
Acknowledgment number = Y+1
ACK = 1
Acknowledgment number = X+1

Fig. 2.24. TCP packet retransmission on timeout.

2.9. ELF binaries

This section details the Executable and Linkable Format (ELF)[60], 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	Purposes
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: TODO

The commands used and complete list of headers can be found in Annex 6. The most relevant sections are described in table 2.21:

Tool	Purpose	Permissions
.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 unitialized or initialized to zero.	Alloc, Writable

Table 2.21. Tools used for analysis of ELF programs.

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 (see table **??**, they have not been shown in previous diagrams for simpleness).

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[61].

Therefore, in order to call a function of a shared library, the dynamic linker follows a process called 'Lazy binding'[62]:

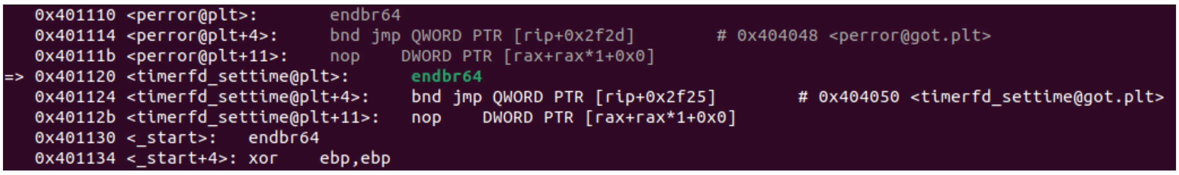
1. From the .text section, instead of calling a direct absolute address as usual, a PLT stub (in the .plt section) is called. Snippet 2.3 shows a call to the function timerfd_settime, implemented by the shared library glibc and thus using a PLT:

CODE 2.3. Call to PLT stub seen from objdump.

```
$ objdump -d simple_timer
```

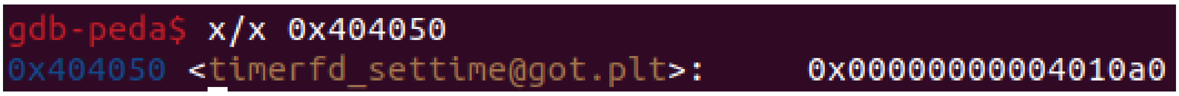
```
4014cb: b9 00 00 00 00          mov    $0x0,%ecx
4014d0: be 01 00 00 00          mov    $0x1,%esi
4014d5: 89 c7                   mov    %eax,%edi
4014d7: e8 44 fc ff ff          call   401120 <timerfd_settime@plt
    >
```

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
```

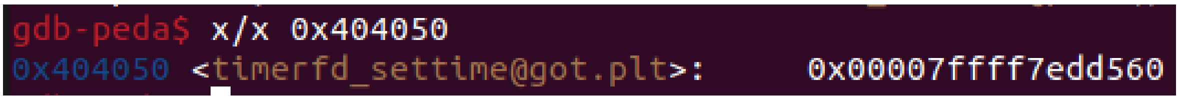
Fig. 2.25. PLT stub for timerfd_settime, seen from gdb-peda.



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

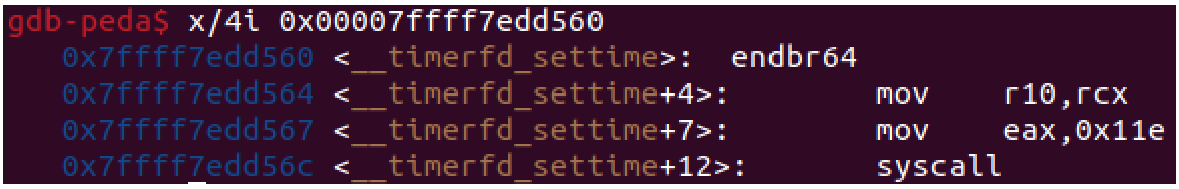
Fig. 2.26. Inspecting address stored in GOT section before dynamic linking, seen from gdb-peda.

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.



```
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.

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 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 4.1.3, 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
GCC 10.3.0 (2021)	Stack canaries, DEP/NX, 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)[63].

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 [63].

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) [64].

Intel CET

Intel Control-flow Enforcement Technology is a hardening feature fully incorporated in Windows 10 systems [65] and a work in progress in Linux[66]. 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.

3. ANALYSIS OF OFFENSIVE CAPABILITIES

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:

- Analysing eBPF's possibilities when hooking system calls and kernel functions.
- Learning eBPF's potential to read/write arbitrary memory.
- Exploring networking capabilities with eBPF packet filters.

Finally, we will study in detail some of the malicious applications that previous researchers have proposed to take advantage of these capabilities of eBPF. In the next chapter, we will proceed to elaborate on these ideas, find new purposes and design our own rootkit.

3.1. eBPF maps security

In table 2.16, we observed 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.

In subsection 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.

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

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

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

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

```
SEC("tp/syscalls/sys_enter_read")
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.

```
struct pt_regs {
        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 are able to extract the parameters of the original hooked function. This can be done by using the System V AMD64 ABI[76], 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 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 an 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 subsection, 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 **??**.

3.2.3. Overriding function return values

A potentially dangerous functionality in eBPF tracing programs is the ability to modify the return value of kernel functions[77][78]. 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 include this macro, but most system calls can be found to implement it. The following code snippets 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 [79]

```
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 [80]

```
#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 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[81], 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 viceversa. 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. Conclusion

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 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 [38]) 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. 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[82], 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 will be particularly relevant since we will be able to bypass this alert by taking advantage of this.

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 in a nested call of the system call sys_read receives an user buffer:

CODE 3.9. Definition of kernel function vfs_read. [83]

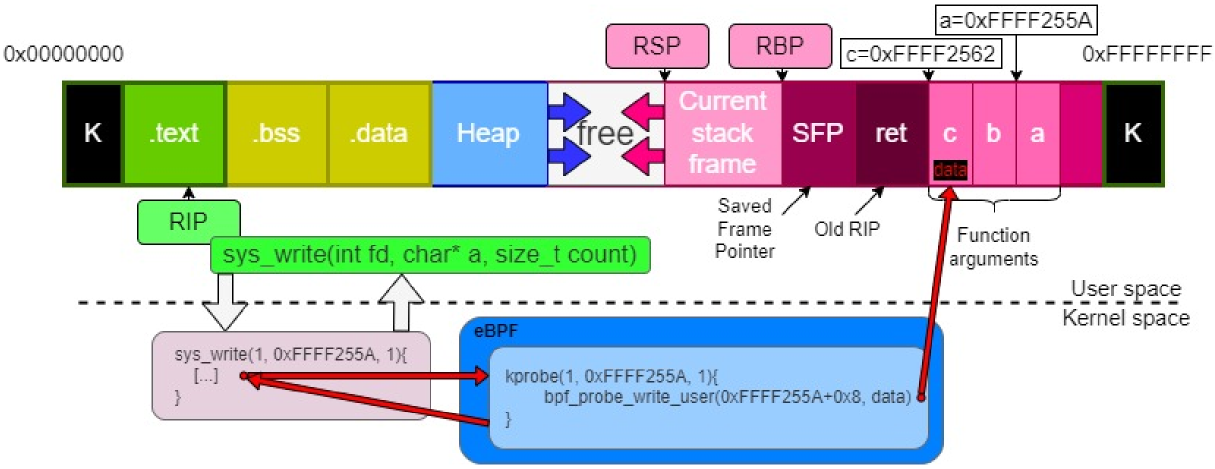
```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count
    , loff_t *pos)
```

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[84], 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[85], so we will receive an 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:


0x00000000
RSP
RBP
a=0xFFFF255A
c=0xFFFF2562
0xFFFFFFFF
K
.text
.bss
.data
Heap
free
Current stack frame
SFP
ret
c
b
a
data
K
RIP
sys_write(int fd, char* a, size_t count)
Saved Frame Pointer
Old RIP
Function arguments
User space
Kernel space
eBPF

```
sys_write(1, 0xFFFF255A, 1){
    [...]
}
```

```
kprobe(1, 0xFFFF255A, 1){
    bpf_probe_write_user(0xFFFF255A+0x8, data)
}
```

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

The above figure shows process memory executing a program similar to the following:

CODE 3.10. Sample program being executed on figure 3.1.

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

int main(){
	char a[] = "AAA";
	char b[] = "BBB";
	char c[] = "CCC";
	func(a, b, c);
}
```

In the figure, we can clearly observe how the technique is used to overwrite an 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 (we will see how this process works in section **??**) 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. Conclusion

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 on the basis of 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 sections 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.
- External 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 tables 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 again 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:

- **Spy 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 sectionTODO, 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 Afchainthe in their DEFCON talk, consists of taking advantage of TCP retransmissions we described on section 2.8.2. Figure 3.2 shows this process:

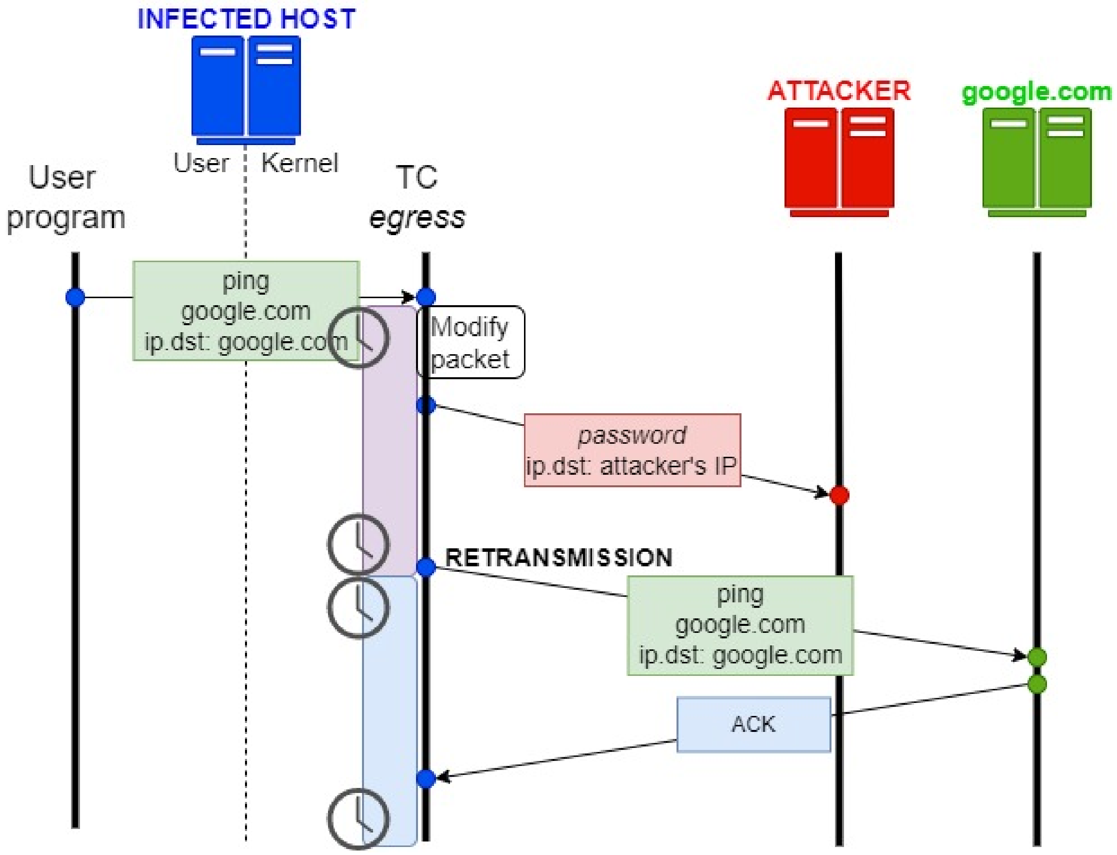

INFECTED HOST
ATTACKER
google.com
User
Kernel
User program
TC egress
ping google.com ip.dst: google.com
Modify packet
password ip.dst: attacker's IP
RETRANSMISSION
ping google.com ip.dst: google.com
ACK

Fig. 3.2. 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. Conclusion

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 or 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 functionality 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 restrictions of eBPF and find new attack vectors.

Taking as a basis these capabilities, this chapter is now dedicated to a comprehensive description of the advanced techniques and functionalities implemented in our eBPF rootkit, which show 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 be exploring each functionality individually, presenting the necessary background on each of them, and offering a final comprehensive view on how each of the systems work.

4.1. Library injection attacks

In this section, we will discuss how to hijack an 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).

This technique has some advantages and disadvantages to the one described by Jeff Dileo at DEFCON 27[84], which we will briefly cover before presenting ours. Both techniques will be later compared in section **??**.

4.1.1. ROP with eBPF

In 2019, Jeff Dileo presented in DEFCON 27 the first technique to achieve arbitrary code execution using eBPF[84]. 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.

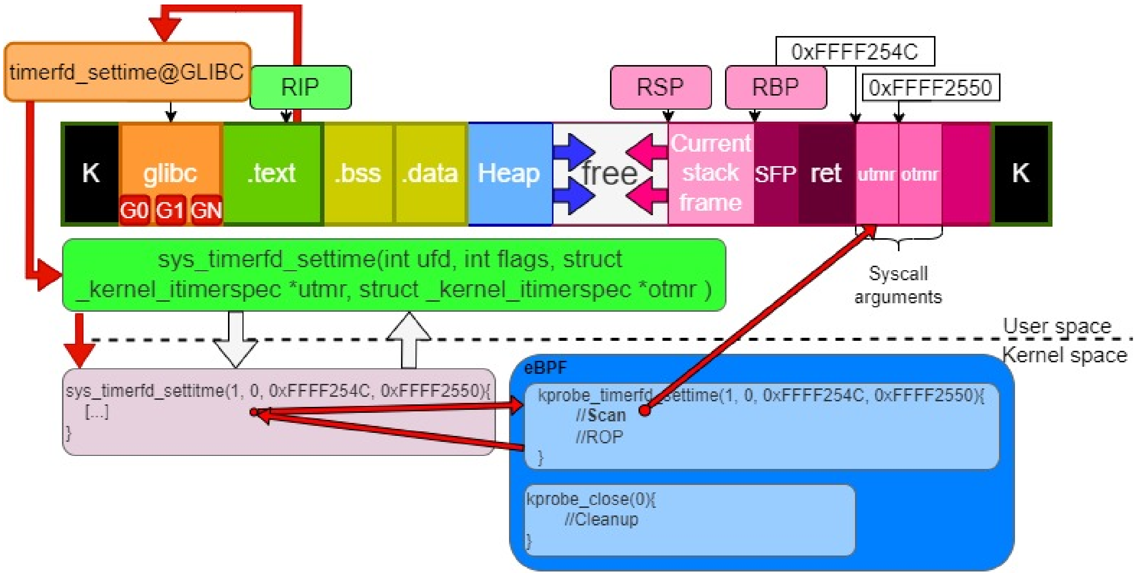

timerfd_settime@GLIBC
RIP
RSP
RBP
0xFFFF254C
0xFFFF2550
K
glibc
G0
G1
GN
.text
.bss
.data
Heap
free
Current stack frame
SFP
ret
utmr
otmr
K
sys_timerfd_settime(int ufd, int flags, struct _kernel_itimerspec *utmr, struct _kernel_itimerspec *otmr)
Syscall arguments
User space
Kernel space
sys_timerfd_settitme(1, 0, 0xFFFF254C, 0xFFFF2550){
[...]
}
eBPF
kprobe_timerfd_settime(1, 0, 0xFFFF254C, 0xFFFF2550){
//Scan
//ROP
}
kprobe_close(0){
//Cleanup
}

Fig. 4.1. Initial setup for the ROP with eBPF technique.

Figure 4.1 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).

An additional aspect must be introduced now (we will cover it more in detail in section **??**): 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)[86]. 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 function of 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 glibc. 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 old rip), then the instruction that called the function in glibc must be the previous instruction (rip - 1).
2. We now have a *call* instruction, that directs us to the function at glibc. We check in the instruction to which address it moves the flow of execution, that is the address of timerfd_settime in 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 4.2 shows process memory after this overwrite:

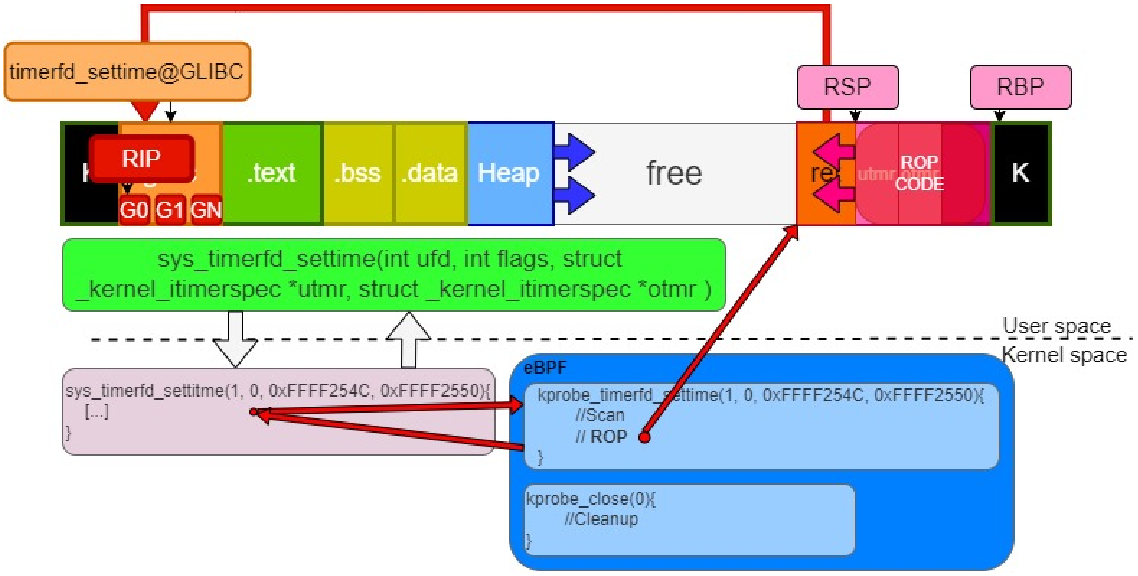

timerfd_settime@GLIBC
RSP
RBP
RIP
G0
G1
GN
.text
.bss
.data
Heap
free
re
utmr
ROP CODE
K
sys_timerfd_settime(int ufd, int flags, struct _kernel_itimerspec *utmr, struct _kernel_itimerspec *otmr)
User space
Kernel space
sys_timerfd_settitme(1, 0, 0xFFFF254C, 0xFFFF2550){
[...]
}
eBPF
kprobe_timerfd_settime(1, 0, 0xFFFF254C, 0xFFFF2550){
//Scan
// ROP
}
kprobe_close(0){
//Cleanup
}

Fig. 4.2. 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 4.3 shows this final step:

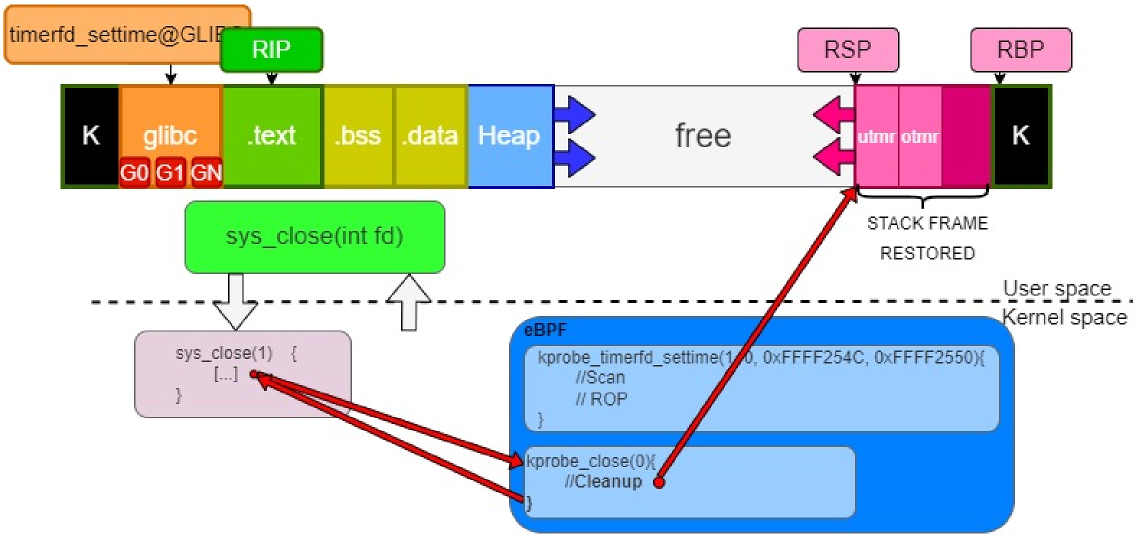

timerfd_settime@GLIB
RIP
RSP
RBP
K
glibc
G0
G1
GN
.text
.bss
.data
Heap
free
utmr
otmr
K
STACK FRAME
RESTORED
sys_close(int fd)
User space
Kernel space
sys_close(1) {
[...]
}
eBPF
kprobe_timerfd_settime(1, 0, 0xFFFF254C, 0xFFFF2550){
//Scan
// ROP
}
kprobe_close(0){
//Cleanup
}

Fig. 4.3. 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.

4.1.2. 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 later 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[87].

In our rootkit, unlike in the ROP technique presented in section 4.1.1, 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 **??** shows.

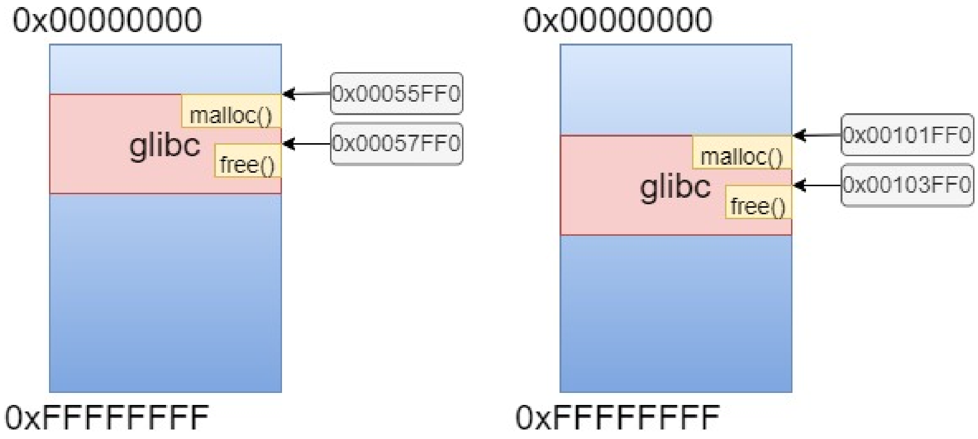


Fig. 4.4. Two runs of the same executable using ASLR, showing a library and two symbols.

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[88]. 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.

4.1.3. Library injection via GOT hijacking

Taking into account the background 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 only after the second time that an specific syscall is called by a process, since the first time we will 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.

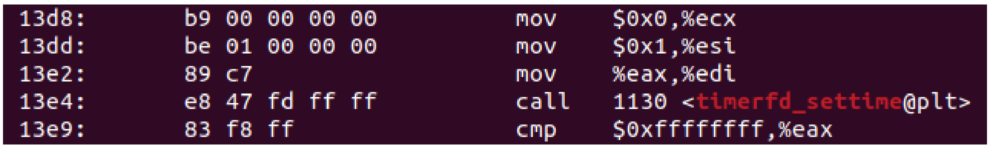
For this research work, the rootkit is prepared to perform this attack on any process that makes use of either the system call sys_openat or sys_timerfd_settime, which are called by the standard library glibc.

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, the algorithm will go as follows:

1. Take one of the syscall parameters and scan forward in the scan. For each iteration, we must check if the data at the stack corresponds to the saved rip:
    (a) 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 4.5).



```
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. 4.5. Call to the glibc function, using objdump

    (b) 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.
    (c) We analyze the instruction at the PLT stub. If the program was compiled with GCC, it will be an *endbr64* instruction followed by the PLT jump instruction using the address at GOT (since it generates Intel CET-compatible programs, see table 2.22). Otherwise, if using Clang, the first instruction is the PLT jump.

5. EVALUATION

6. RELATED WORK

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.

[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] (Dec. 7, 2014), [Online]. Available: https://kernelnewbies.org/Linux_3.18.

[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.

[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.

[6] "Ebpf incorporation in the linux kernel 3.18." (Dec. 7, 2014), [Online]. Available: https://kernelnewbies.org/Linux_3.18.

[7] "Ebpf for windows." (), [Online]. Available: https://source.android.com/devices/architecture/kernel/bpf.

[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.

[9] P. Hogan, DEFCON 27. (), [Online]. Available: https://www.youtube.com/watch?v=g6SKWT7sROQ.

[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.

[11] *Ebpf documentation*. [Online]. Available: https://ebpf.io/what-is-ebpf/.

[12] 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.

[13] "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.

[14] ——, "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.

[15] *Index register*. [Online]. Available: https://gunkies.org/wiki/Index_register.

[16] ——, "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.

[17] "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/.

[18] ——, "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.

[19] ——, "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.

[20] *Tcpdump and libpcap*. [Online]. Available: https://www.tcpdump.org.

[21] *Bpf features by linux kernel version*, iovisor. [Online]. Available: https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md.

[22] B. Gregg, *BPF performance tools*. [Online]. Available: https://www.oreilly.com/library/view/bpf-performance-tools/9780136588870/.

[23] *Ebpf documentation: Loader and verification architecture*. [Online]. Available: https://ebpf.io/what-is-ebpf/#loader--verification-architecture.

[24] *Ebpf instruction set*. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/instruction-set.html.

[25] 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 05/13/2022).

[26] *BPF – in-kernel virtual machine*, Feb. 20, 2015. [Online]. Available: http://vger.kernel.org/netconf2015Starovoitov-bpf_collabsummit_2015feb20.pdf.

[27] J. Corbet, *A jit for packet filters*, Apr. 12, 2011. [Online]. Available: https://lwn.net/Articles/437981/.

[28] *Demystify eBPF JIT Compiler*, Sep. 11, 2018, p. 13. [Online]. Available: https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf.

[29] *Demystify eBPF JIT Compiler*, Sep. 11, 2018, p. 14. [Online]. Available: https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf.

[30] *Bpf_jit_enable*. [Online]. Available: https://sysctl-explorer.net/net/core/bpf_jit_enable/.

[31] *BPF – in-kernel virtual machine*, Feb. 20, 2015, p. 23. [Online]. Available: http://vger.kernel.org/netconf2015Starovoitov-bpf_collabsummit_2015feb20.pdf.

[32] 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.

[33] *Ebpf verifier*. [Online]. Available: https://kernel.org/doc/html/latest/bpf/verifier.html.

[34] *Demystify eBPF JIT Compiler*, Sep. 11, 2018, pp. 17–22. [Online]. Available: https://www.netronome.com/media/documents/demystify-ebpf-jit-compiler.pdf.

[35] M. Rybczynska. “Bounded loops in bpf for the 5.3 kernel.” (Jun. 30, 2019), [Online]. Available: https://lwn.net/Articles/794934/.

[36] *Ebpf maps*. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/maps.html.

[37] *Bpf(2)- linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man2/bpf.2.html.

[38] *Bpf-helpers(7)- linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[39] D. Lavie. “A gentle introduction to xdp.” (Feb. 3, 2022), [Online]. Available: https://www.seekret.io/blog/a-gentle-introduction-to-xdp/.

[40] *Xdp actions*. [Online]. Available: https://prototype-kernel.readthedocs.io/en/latest/networking/XDP/implementation/xdp_actions.html.

[41] Hangbin. “Tc/bpf and xdp/bpf.” (Mar. 13, 2019), [Online]. Available: https://liuhangbin.netlify.app/post/ebpf-and-xdp/.

[42] M. A. Brown. “Traffic control howto.” (Oct. 1, 2006), [Online]. Available: http://linux-ip.net/articles/Traffic-Control-HOWTO/.

[43] 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/.

[44] “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.

[45] M. Desnoyers, *Using the linux kernel tracepoints*. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/tracepoints.html.

[46] M. H. Jim Keniston Prasanna S Panchamukhi, *Kernel probes (kprobes)*. [Online]. Available: https://www.kernel.org/doc/html/latest/trace/kprobes.html.

[47] N. Alcock. “Kallsyms: New /proc/kallmodsyms with builtin modules and symbol sizes.” (Jun. 6, 2021), [Online]. Available: https://lwn.net/Articles/862021/.

[48] “Bpf compiler collection (bcc).” (), [Online]. Available: https://github.com/iovisor/bcc.

[49] (), [Online]. Available: https://github.com/libbpf/libbpf.

[50] “Bpf next kernel tree.” (), [Online]. Available: https://kernel.googlesource.com/pub/scm/linux/kernel/git/bpf/bpf-next.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] “Ropgadget tool.” (), [Online]. Available: https://github.com/JonathanSalwan/ROPgadget.

[57] Alienor. “The network layers explained [with examples].” (Nov. 28, 2018), [Online]. Available: https://www.plixer.com/blog/network-layers-explained/.

[58] “Transmission control protocol,” IBM. (Apr. 19, 2022), [Online]. Available: https://www.ibm.com/docs/en/aix/7.2?topic=protocols-transmission-control-protocol.

[59] “Three-way handshake.” (), [Online]. Available: https://www.sciencedirect.com/topics/computer-science/three-way-handshake.

[60] *Elf*. [Online]. Available: https://wiki.osdev.org/ELF.

[61] D. Tomaschik. “Got and plt for pwning.” (Mar. 19, 2017), [Online]. Available: https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html.

[62] 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.

[63] “Aslr/pie intro.” (), [Online]. Available: https://guyinatuxedo.github.io/5.1-mitigation_aslr_pie/index.html#aslrpie-intro.

[64] 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.

[65] 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/.

[66] 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.

[67] *Capabilities - overview of linux capabilities*. [Online]. Available: http://manpages.ubuntu.com/manpages/trusty/man7/capabilities.7.html.

[68] 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.

[69] “[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/.

[70] “Capability: Introduce cap_bpf and cap_tracing.” (), [Online]. Available: https://lwn.net/Articles/797807/.

[71] “Reconsidering unprivileged bpf.” (), [Online]. Available: https://lwn.net/Articles/796328/.

[72] "Cve-2021-4204: Linux kernel ebpf improper input validation vulnerability." (), [Online]. Available: https://www.openwall.com/lists/oss-security/2022/01/11/4.

[73] "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.

[74] "Security hardening: Use of ebpf by unprivileged users has been disabled by default." (), [Online]. Available: https://www.suse.com/support/kb/doc/?id=000020545.

[75] "Cve-2022-0002." (), [Online]. Available: https://access.redhat.com/security/cve/cve-2021-4001.

[76] 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.

[77] 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.

[78] "Bpf-based error injection for the kernel." (), [Online]. Available: https://lwn.net/Articles/740146/.

[79] (), [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/fs/open.c#L1192.

[80] (), [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/include/linux/syscalls.h#L233.

[81] "Injecting faults into the kernel." (Nov. 4, 2006), [Online]. Available: https://lwn.net/Articles/209257/.

[82] "Probe_write_common_error." (), [Online]. Available: https://www.spinics.net/lists/bpf/msg16795.html.

[83] (), [Online]. Available: https://elixir.bootlin.com/linux/v5.11/source/fs/read_write.c#L476.

[84] 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.

[85] ——, *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.

[86] “The gnu c library.” (), [Online]. Available: https://www.gnu.org/software/libc/.

[87] “Stack canaries.” (), [Online]. Available: https://ir0nstone.gitbook.io/notes/types/stack/canaries.

[88] “Position independent code.” (), [Online]. Available: https://ir0nstone.gitbook.io/notes/types/stack/pie.

APPENDIX A - BPFTOOL COMMANDS

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 - READELF COMMANDS

Section headers in ELF file

CODE 6.1. 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)
```