uc3m | Universidad Carlos III de Madrid

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

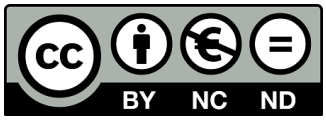
Bachelor Thesis

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

Marcos Sánchez Bajo

Juan Manuel Estévez Tapiador

Leganés, 2022



SUMMARY

Keywords:

DEDICATION

ABSTRACT

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

As the efforts of the computer security community grow to protect increasingly critical devices and networks from malware infections, 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

2. STATE OF THE ART

This chapter is dedicated to an study of the eBPF technology. Firstly, we will analyse its origins, 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. Finally, an study of the existing alternatives for developing eBPF applications will be also included.

Although during our discussion of the offensive capabilities of eBPF in section**??** we will use 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.

2.1. eBPF history - Classic 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].

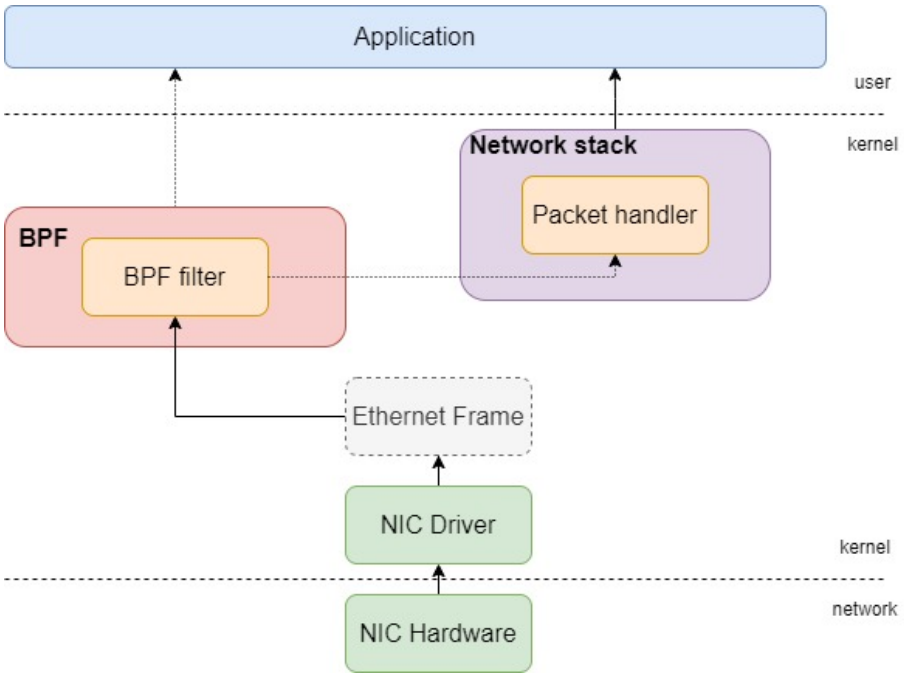


Fig. 2.1. Sketch of the functionality of classic BPF

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).

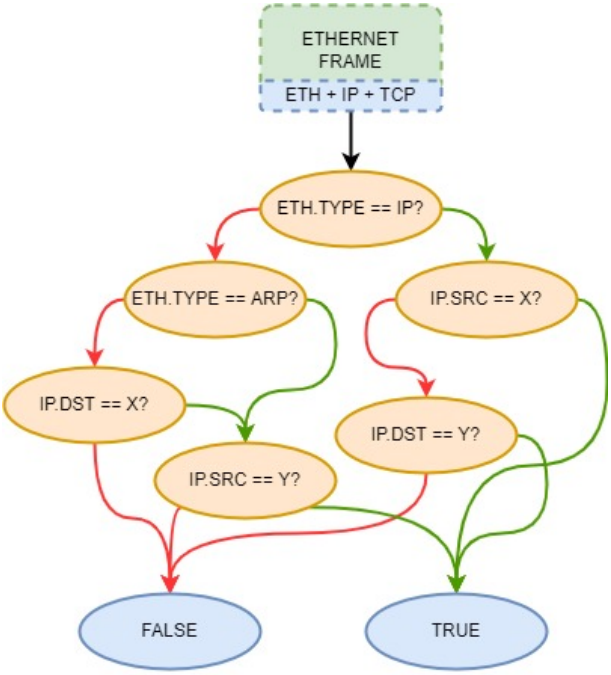


Fig. 2.2. Execution of a BPF filter.

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.

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

Fig. 2.3. 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.

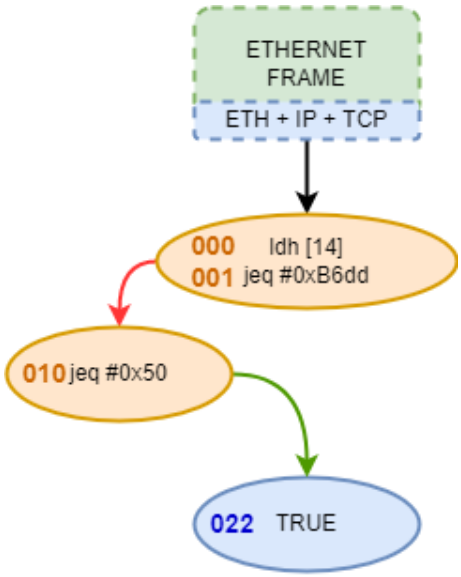


Fig. 2.6. Shortest path in the CFG described in the example of figure 2.5 that a packet needs to follow to be accepted by the BPF filter set with *tcpdump*.

2.2. Analysis of modern eBPF

This section discusses the current state of modern 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
*N*ew bpf() syscall	3.18	2014
*I*ntroduction of eBPF maps	3.19	2015
*e*BPF attached to kprobes	4.1	2015
*I*ntroduction of Traffic Control	4.5	2016
*e*BPF attached to tracepoints	4.7	2016
*I*ntroduction of XDP	4.8	2016

Table 2.2. 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.

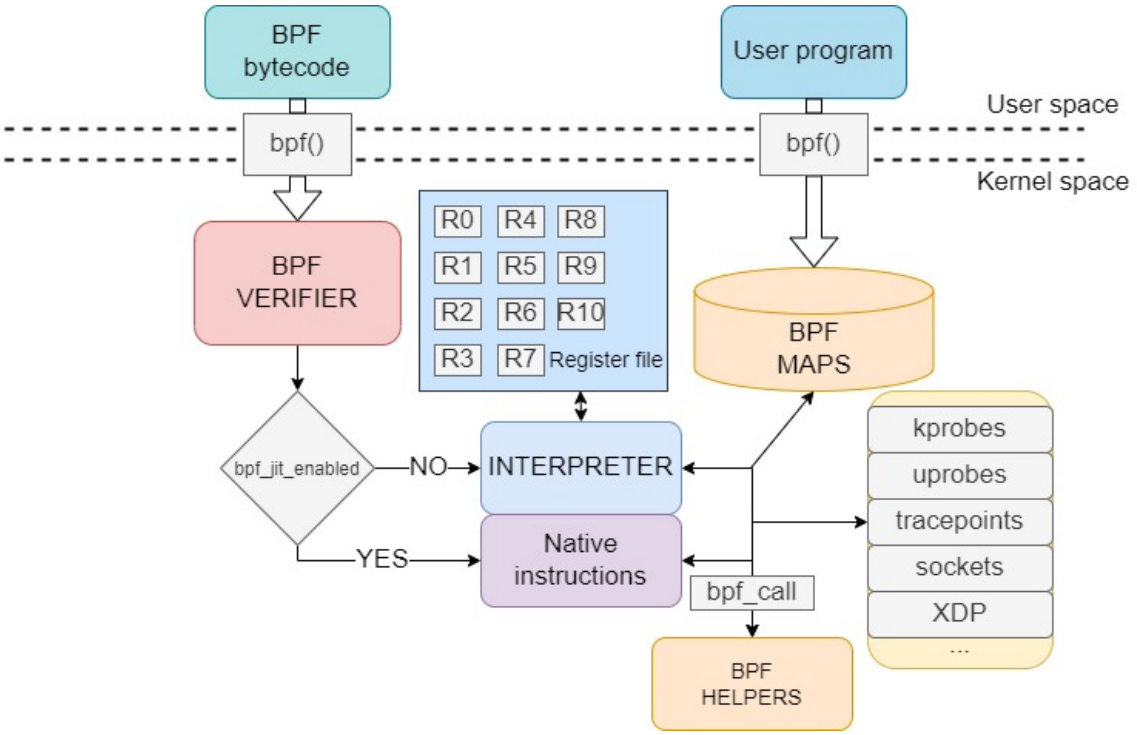


Fig. 2.7. Figure showing overall eBPF architecture in the Linux kernel and the process of loading an eBPF program. Based on[22] and [23].

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.

- 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].

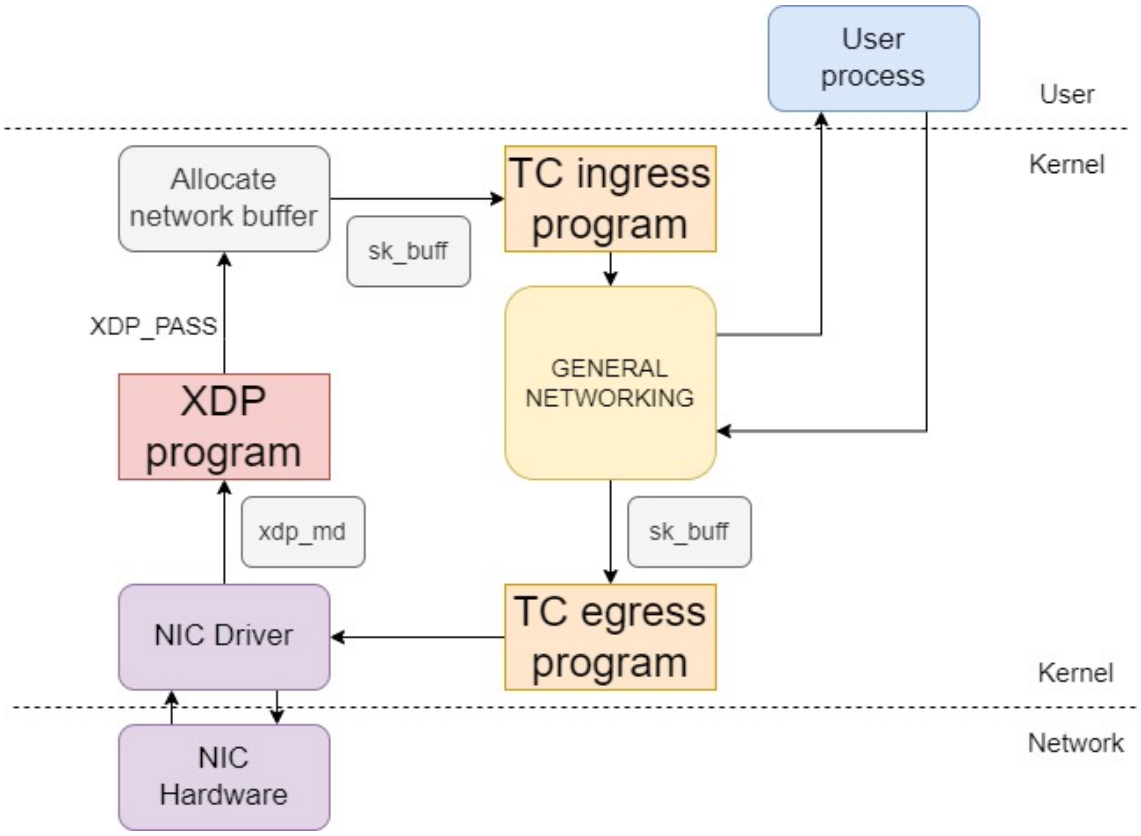


Fig. 2.8. Figure showing how the eBPF XDP and TC modules are integrated in the network processing in the Linux kernel.

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).

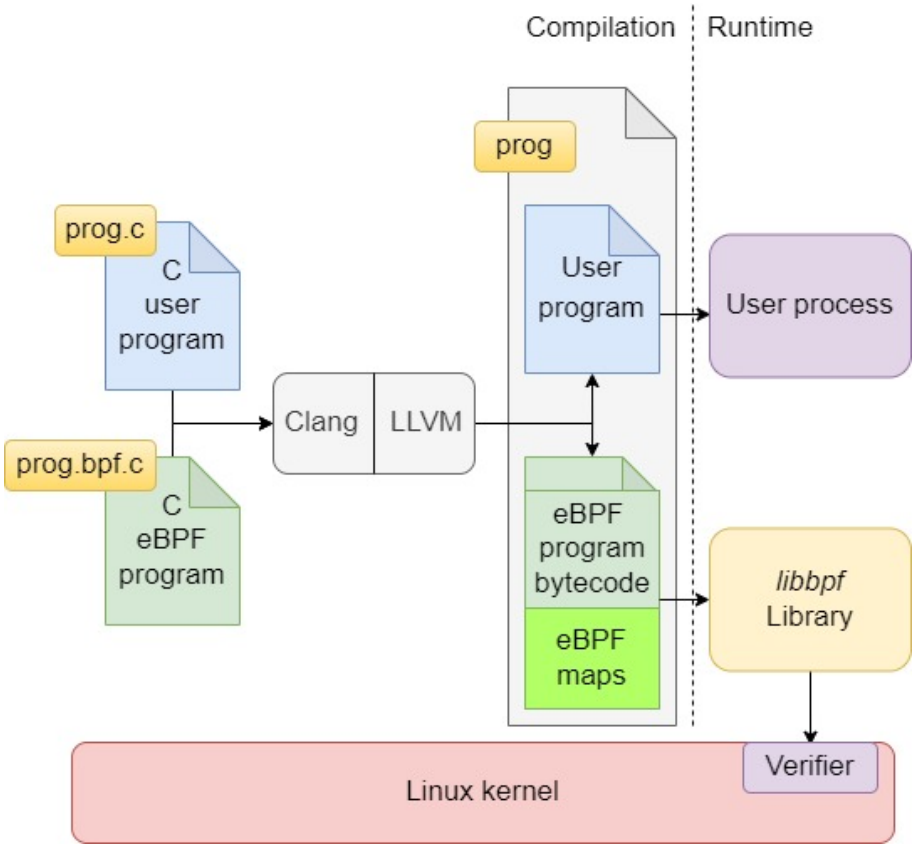


Fig. 2.9. Sketch of the compilation and loading process of a program developed with libbpf.

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 **??**.

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. Security features in eBPF

As we 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 3.1.

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 3.1. Kernel compilation flags for eBPF.

The above table is based on BCC's documentation3.1, 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.

3.1.1. 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[52] 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[54][55]. The current system is therefore described in table 3.2.

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 3.2. 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[56]. 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[57].

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 3.3.

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 3.3. 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[58], Suse Linux[59] or Red Hat Linux[60], between others.

3.1.2. eBPF maps security

In table 3.2, 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[61], the

calling convention used in Linux. Depending on whether we are in the kernel or in user space, the registers used are different to store the values of the function arguments. Table 3.4 summarizes these two interfaces. Some other relevant registers are also displayed as a reference in table 3.5.

Register	Purpose
rdi	1st argument
rsi	2nd argument
rdx	3rd argument
rcx	4th argument
r8	5th argument
r9	6th argument
rax	Return value

Register	Purpose
rdi	1st argument
rsi	2nd argument
rdx	3rd argument
r10	4th argument
r8	5th argument
r9	6th argument
rax	Return value

Table 3.4. Argument passing convention of registers for function calls in user and kernel space respectively.

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 3.5. Other relevant registers in x86_64 and their purpose.

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 3.5 shows the format of the struct.

CODE 3.5. 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, and a first attribute containing the address pointing to

another *struct pt_regs* as in kprobes and uprobes, so that we will 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 3.4, 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.

In general, the eBPF verifier attempts to reject illegal memory accesses, however 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 read any address in user or kernel space. 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.

3.3. Memory corruption

Privileged malicious eBPF programs (or those with the CAP_BPF + CAP_PERFMON capabilities) have the potential to get:

- Read and write access in user memory.
- Read-only access in kernel memory.

3.3.1. Accessing user memory

4. METHODS??

5. RESULTS

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

[53] 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.

[54] “[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/.

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

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

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

[58] "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.

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

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

[61] 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.

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