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

2. STATE OF THE ART 5

2.1. eBPF history - Classic 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 - *tcpdump* 10

2.2. Analysis of modern eBPF 11

2.2.1. eBPF instruction set 13

2.2.2. JIT compilation 13

2.2.3. eBPF architecture 14

3. METHODS?? 15

4. RESULTS 16

5. CONCLUSION AND FUTURE WORK 17

BIBLIOGRAPHY 18

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

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

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:

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

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 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].

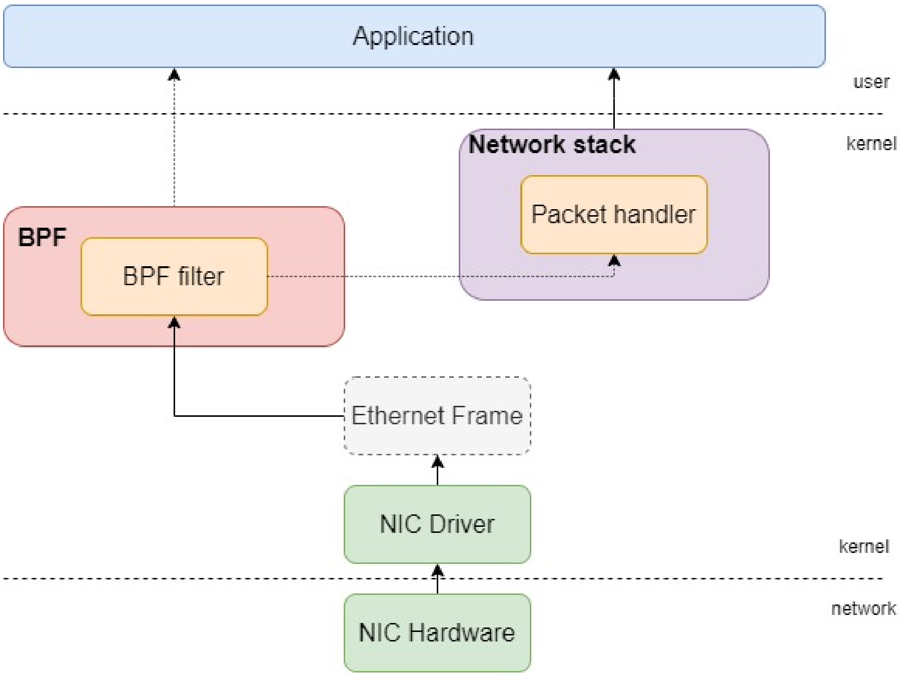


Fig. 2.1. Sketch of the functionality of classic BPF

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).

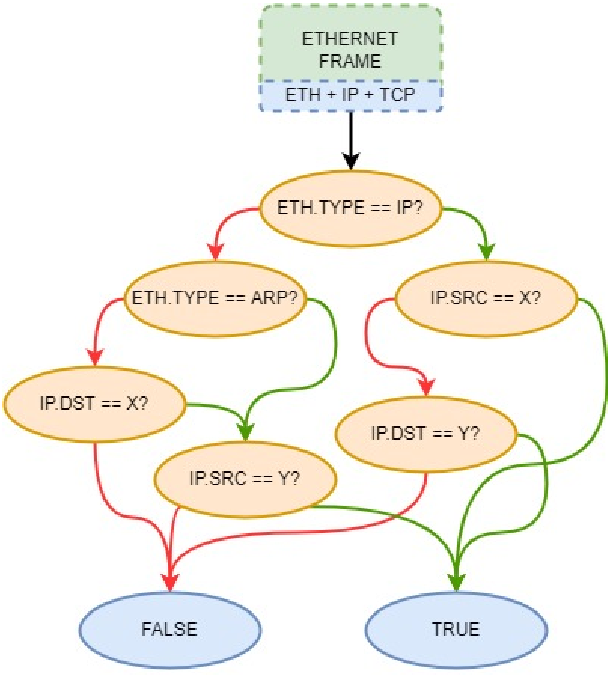


Fig. 2.2. Execution of a BPF filter.

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

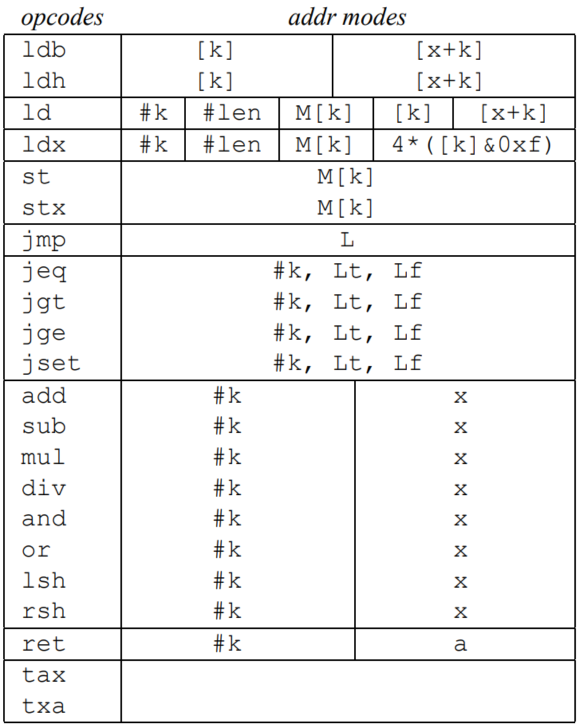
In order to implement the CFG to be run at the BPF VM, BPF filter programs are made up of BPF bytecode, which is defined by a new BPF instruction set. Therefore, a BPF filter program is an array of BPF bytecode instructions[17].

	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.

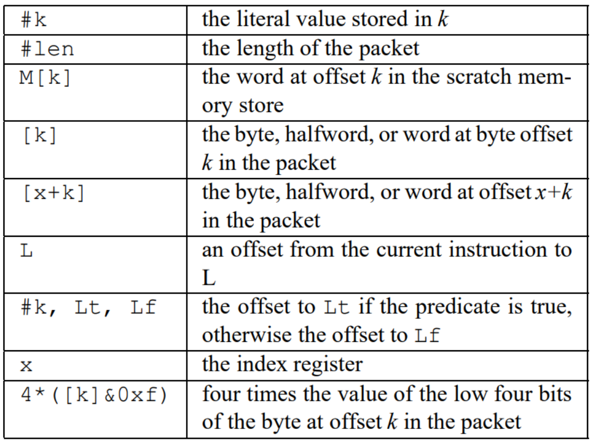


opcodes	*addr modes*				
ldb	[k]	[x+k]			
ldh	[k]	[x+k]			
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]&0xf)	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k	x			
sub	#k	x			
mul	#k	x			
div	#k	x			
and	#k	x			
or	#k	x			
lsh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

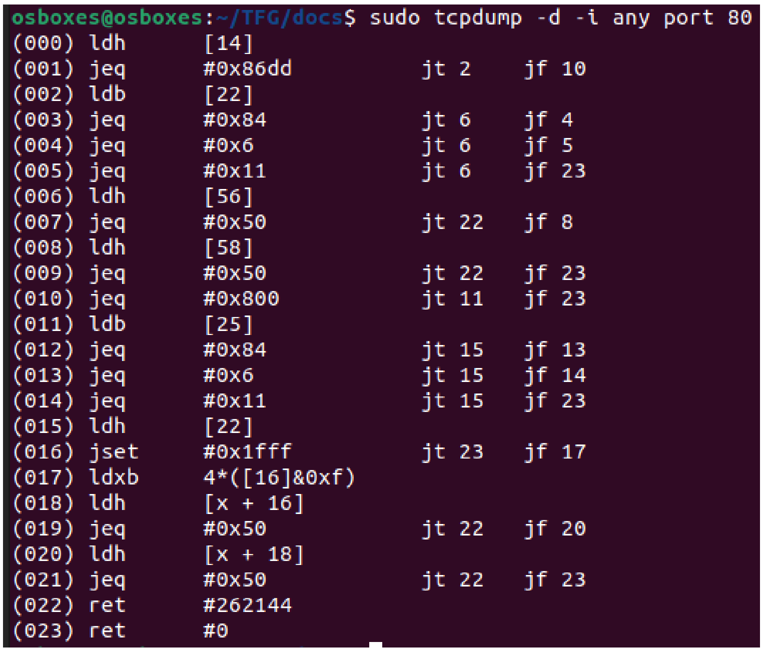
Fig. 2.4. Table explaining the column address modes in Figure2.3, as shown by McCanne and Jacobson[19]

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 - *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.

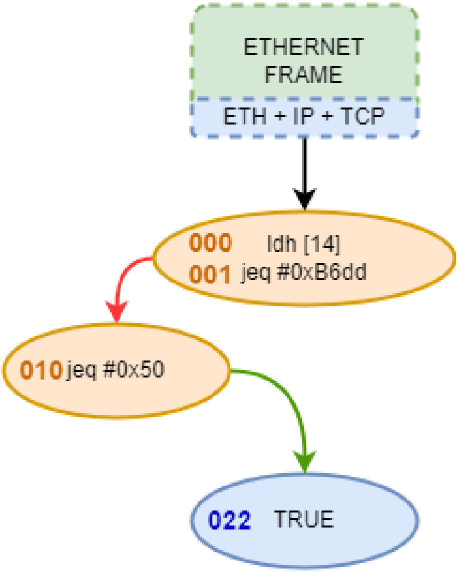


Fig. 2.6. Shortest path in the CFG described in the example of figure 2.5 that a packet needs to follow to be accepted by the BPF filter set with *tcpdump*.

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.

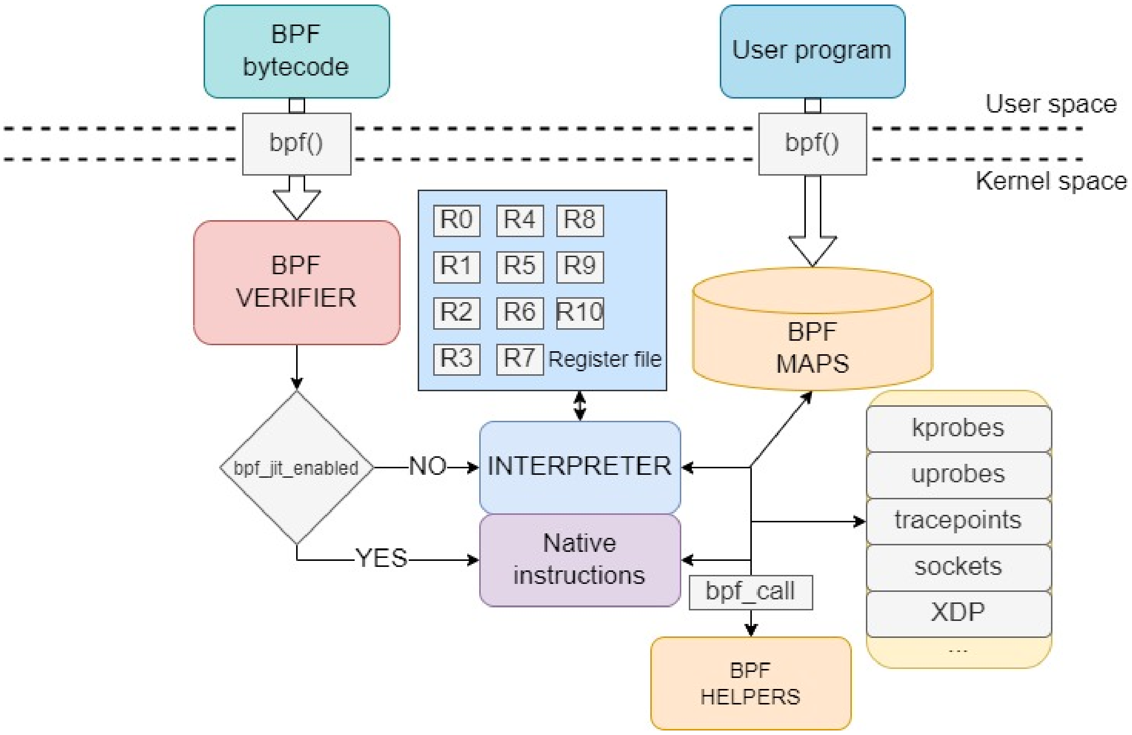

BPF bytecode
User program
User space
bpf()
bpf()
Kernel space
BPF VERIFIER
R0 R4 R8
R1 R5 R9
R2 R6 R10
R3 R7 Register file
BPF MAPS
bpf_jit_enabled
NO
YES
INTERPRETER
Native instructions
kprobes
uprobes
tracepoints
sockets
XDP
...
bpf_call
BPF HELPERS

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. eBPF architecture

Provided the instruction set architecture (ISA) described in section

3. METHODS??

4. RESULTS

5. 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.

APPENDIX A

APPENDIX B