

**BAE, YOUR COMPANION TO DETECT KERNEL ATTACKS BY A
BEHAVIOUR ANALYSIS**

by

QUENTIN AUDINET

(Singapore, National University of Singapore)

**A THESIS SUBMITTED FOR THE DEGREE OF
MASTER OF SCIENCE (BY RESEARCH)**

in

COMPUTER SCIENCE

in the

RESEARCH SECURITY LABORATORY

of the

NATIONAL UNIVERSITY OF SINGAPORE

2024

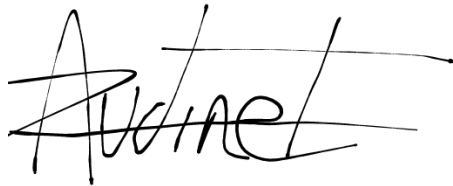
Supervisor:

Professor LIANG Zhenkai

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

A handwritten signature in black ink, appearing to read 'Audinet', with a horizontal line extending from the end of the signature.

Quentin AUDINET

05 July 2024

Acknowledgments

I would like to extend my deepest gratitude to all those who have supported me throughout the journey of completing this Master Thesis.

First and foremost, I am immensely grateful to my advisor, Professor Liang Zhenkai, for his invaluable guidance, patience, and encouragement. His expertise and insights were crucial in shaping the direction and quality of this work.

I would also like to thank my colleagues from the NUS Research Security Lab, Ruan Bonan and Zhang Chuqi. Their collaboration, thoughtful feedback, and camaraderie made the research process both productive and enjoyable. The discussions and brainstorming sessions we shared significantly contributed to the progress and success of this project.

A heartfelt thank you goes to Télécom Paris and the French Double Degree Program for offering me the incredible opportunity to study in Singapore. This experience has not only enriched my academic pursuits but also broadened my cultural and professional horizons.

Finally, I am deeply indebted to my family and friends for their constant support and encouragement even when I was 10'000km away. Their understanding, patience, and love provided me with the strength and motivation to persevere through the challenges of this journey.

Thank you all.

Contents

Acknowledgments	i
Abstract	v
List of Figures	vi
List of Tables	vii
1 Introduction	1
1.1 Overview	1
1.2 Issue	2
1.3 Approach	2
1.4 Outline	3
2 Background	4
2.1 How exploits work	4
2.1.1 Goal of an exploitation	4
2.1.2 Different attacks	5
2.1.3 What exploits do	6
2.2 General Work	7
2.2.1 eBPF and Rust	7
2.2.2 Hot-Patching	8
2.3 Function Context	8
2.3.1 Calling a function	8
2.3.2 Signature detection	9
2.4 Contribution	9

3	Methods and Limitations	11
3.1	Approaches and their limitations	11
3.1.1	First approach: Android devices	11
3.1.2	Second approach: Kernel modification	12
3.1.3	Third approach: Rust eBPF	13
3.2	Hooking Kernel Functions	14
3.2.1	Detecting calls at the entry point	15
3.2.2	Detecting calls at the function entry	15
3.3	Analysing the Context	16
3.4	Measures to Prevent an Attack	17
4	Monitor Malicious Behaviours	19
4.1	Program behaviour	19
4.1.1	Logical structure	19
4.1.2	Application to an exploit	20
4.2	Pattern recognition	21
4.2.1	Graph of Conditions	21
4.2.2	Recognise a pattern	22
4.2.3	From the GoC to the detector	25
4.3	A language to describe the graph	25
4.3.1	Intuitively represent dependencies	26
4.3.2	Describe conditions	26
4.4	Kernel-land and Userland	28
4.4.1	Kernel tasks	28
4.4.2	Delegate tasks to Userland	30
4.4.3	Userland tasks	31
4.5	Use of Rust	31
4.5.1	Advantages	31
4.5.2	Challenges	32
4.6	Limitations	33
5	Experiments	35
5.1	Detection Efficiency	35

5.1.1	Known Vulnerability Detection	35
5.1.2	False Positives and False Negatives	38
5.2	Performance Overhead	40
5.2.1	System Performance Impact	40
5.2.2	Scalability	42
5.3	Deployment Time	47
6	Related Work	49
6.1	Kernel Patching	49
6.2	eBPF Security	49
6.3	Kernel Exploitation	50
6.4	Runtime Enforcement	50
7	Conclusion and Future Work	52
7.1	Conclusion	52
7.2	Future Research Directions	52
7.2.1	Acting after detecting	53
7.2.2	The signature problem	53
7.2.3	Determinism	53
7.2.4	Hooking optimization	54
7.2.5	Language improvements	54
7.2.6	Malicious usage	54
	Bibliography	55

Abstract

BAE, your companion to detect kernel attacks by a Behaviour Analysis

by

Quentin AUDINET

Master of Science (By Research) in Computer Science

National University of Singapore

Modern computers run multiple programs simultaneously, coordinated by the Operating System through its kernel. The kernel operates in Kernel-land, where it has full system control, making it a critical target for attackers. Vulnerabilities in the kernel can compromise the entire system, with patches often delayed due to rigorous testing and approval processes. This delay leaves systems exposed, especially to 0-day exploits—vulnerabilities unknown to the software’s developers but exploited by attackers.

Temporary solutions such as hot fixing and runtime enforcement have emerged to mitigate these risks. These methods apply temporary patches without requiring a full kernel update, thus quickly countering potential attacks.

The introduction of eBPF revolutionised kernel security and hot-patching by enabling dynamic programming and real-time kernel monitoring. Running in a privileged, sandboxed environment, eBPF programs have become the versatile Swiss Army knife of kernel hot-patching, offering a robust solution for various security tasks.

In this thesis, I present **BAE** , a new Linux hot-patch software written entirely in Rust, designed to prevent exploits by analysing their behaviour. **BAE** leverages the eBPF framework and requires only the Rust toolchain, making it compatible with any kernel version supporting eBPF. Results indicate that **BAE** achieves near 100% detection accuracy with minimal false positives.

List of Figures

3.1	Kernel vulnerability triggered	14
4.1	Vulnerability in the CFG representation	20
4.2	Insertion of a hook using KProbes	22
4.3	Representation of the Graph of Condition for a given process	23
4.4	Processing the call to a hooked function between Kernel Land and User Land	29
5.1	System slow down for increasing number of hooked functions	41
5.2	System slow down for different functions	42
5.3	Time to fill different buffer size with function <code>try_to_wake_up</code>	44
5.4	Trend line after scaling Figure 5.3 on both axes with \log_2	45
5.5	Time to fill using different functions	46

List of Tables

4.1	Graph of Condition in Figure 4.3 representation inside the memory . .	23
5.1	Accuracy of the detector for given CVEs	38
5.2	Number of processes reaching conditions for different CVEs	38

Chapter 1

Introduction

In order to prevent attacks over the Linux Kernel, I have used the eBPF technology which allows to hook kernel functions and execute some code inside the kernel before the function execution. When a function is hooked, the environment is analysed to detect any malicious behaviour through the call.

1.1 Overview

Linux Kernel hot patching is nowadays a hot topic. Numerous studies try to develop their own way to fix kernel vulnerabilities without requiring any system update. **PET** presented by Zicheng Wang et al. in [38] and **Tetragon** [35] have been references for developing **BAE**. They leverage eBPF technology with sanitizer reports and policies in order to detect malicious programs and prevent their execution.

eBPF being a recent technology, further research must be pursued to continuously increase Kernel protection. Through this master thesis would like to propose **BAE**, in the scope of eBPF kernel security and control flow analysis as described in [2]. Control Flow graph is an inherent part of any program and I leveraged this feature to detect specific behaviours with **BAE**.

Contrary to some hot-fixing projects [41][22][39], **BAE** doesn't really hot-fix lying vulnerabilities but ensures that no processes follow the path leading to the vulnerability trigger while waiting for the security patch to come up.

1.2 Issue

The goal of this master thesis is to face the issue that a vulnerability has almost always a time to be exploitable before being patched. When a vulnerability has been discovered, the patch needs to be written and then approved by the community to finally be released in one of the Linux kernel updates. A study calculates that a vulnerability lives in average 5 years [5]. However, this duration doesn't mean that the vulnerability is exploitable for 5 years. The Project Zero leaded by Google [18] calculates that it takes around 25 days to fix a Linux vulnerability. Vulnerabilities will still exist and so, exploitation will still be feasible. 0-day is also a very harmful exploitation, and **BAE** tries also to offer some way to prevent them.

1.3 Approach

In order to detect malicious calls inside the Kernel, I have to load my own program inside it. I have used Aya, an eBPF crate for Rust which brings multiple features for Kernel development such as Kernel Probes (KProbes), Linux Security Modules (LSM), Express Data Path (XDP)... To use this library, two or three programs are necessary. One is running into UL and launches the eBPF program inside the Kernel. The third program is optional and permits communication between KL and UL programs (through mapping as an example). In this thesis, the eBPF program waits for kernel functions (kfunctions) to be called. The program won't wait for any kfunction call but only the ones specified by the user program. When such a function is called, the control flow will be redirected to the eBPF program. By analysing the environment in which this function has been called, **BAE** determines if the call belongs to a path leading to a vulnerability exploitation. This path is a graph I called *Graph of Conditions* (**GoC**) describing each steps and the condition to go to the next step for a vulnerability exploitation. Later in § 4.3, I propose a new representation language **BFFL** the Behaviour of Functions Formatting Language, to offer an intuitive way to write the **GoC**. The detection being one of the main part of this master thesis, we'll come back to this later in Chapter 4.

1.4 Outline

The remaining parts of this thesis are organised as follows. In Chapter 2, we first explain the background necessary for the understanding of the following. Chapter 3 summarises the steps I have followed and their limitations. Chapter 4 provides a way to detect malicious calls. Chapter 5 presents experiments I have conducted to demonstrate the feasibility of the project but also its limitations. Chapter 6 focuses on current and previous work related to the topic. The last chapter, Chapter 7 is destined to conclude this thesis and presents some future research directions.

Chapter 2

Background

2.1 How exploits work

Exploits are programs used, as they're named, to exploit a vulnerability on a vulnerable system. This system can be various and not always as sensitive. In this work, we'll focus on kernel exploitation, and thus related exploits.

2.1.1 Goal of an exploitation

Depending on the type of vulnerability, an attacker can have different goals. One of the first steps for an exploit is to corrupt the memory. Memory corruption happens when a process memory has been tampered modifying the expected behaviour of the program. Memory corruption can be a read, write, or even both corruption, one corruption that could lead to another one, such as based-address leaking. When the process memory is controlled by an untrusted entity, this process can perform harmful actions on the system, such as privilege escalation (PrivEsc), remote code execution (RCE), and denial of service (DoS)... The privilege escalation consists of elevating the current privileges of a user into another one with higher privileges on the system, the best case being having system privileges. This step is often essential for an attacker since the first interaction with the system can be extremely restricted (apache user, normal employee...)

2.1.2 Different attacks

If a great number of exploits and vulnerabilities exist, there are not that many attack vectors, and most security bypasses differ and require more imagination. Much of the vulnerabilities rely on the same code error family. The most famous ones are Out-of-bounds errors, also known as buffer overflow, Use-after-free and Double-free errors, format string errors, and race condition [24]. Each of these vectors is different from each other but can be combined in order to achieve the exploitation. We will focus quickly on each of them to understand the pattern used during an attack.

Out-of-Bond (OOB) error. This attack is one of the most common. The principle is to overwrite or read some unauthorised memory by overflowing a buffer. For this reason, this attack is commonly called Buffer Overflow Error. This error happens when data is written into some buffer, but the length isn't correctly checked. In C, if data is written outside the targeted object, the program will overwrite the following data. A classic exploitation is to overwrite the return address of a function to redirect the execution flow of a process into some instructions written by an attacker. Thus, in order to happen, such an attack requires writing a non-usual number of bytes in the memory. Knowing where this unusual writing occurs in the memory, by intercepting each writing, an abnormal write could be detected.

Use-after-free (UAF) error. A process can allocate memory dynamically in a region called the heap. If the memory required is available, a pointer to the start of this region is returned to the program. When the program doesn't need this memory anymore, it has to free it. However, it can happen that this memory is freed, but the pointer is not destroyed. Later in the code, the memory can be reused for a completely different purpose, but with two different pointers representing two different objects pointing at it. Then, using the first object with the data of the second one has undetermined behaviour and can be dangerous when controlled by an attacker. Knowing a program implements a UAF error, one can monitor pointer accesses and check if one is used after having been freed.

Double-free (DF) error. This error is similar to the previous one. However, this time a same memory region is freed two times instead of just once. Because of how the OS manages the memory, when a section is freed, it is added to a list of available sections. Thus, if a same region is freed two times, it will be added two times in the list. An attacker can then allocate some memory to a first object and then allocate the same memory region to a second object, overwriting the first object data. Nonetheless, the first object is still available but with new data. As in the UAF error, if this data is controlled, the object can be used for malicious purposes. Once again, when this attack occurs, memory accesses can be monitored in order to detect a double allocation without free.

Format string (FS) error. This error takes advantage of the format functions (`printf` and all associated functions) to leak or overwrite data. These functions take the format string and replace each format symbol with the relevant data that is set in the stack. The normal behaviour is to give the arguments to the function so that they can be added to the stack and used by the function. However, the function doesn't check that enough arguments have been written, and for example the format symbol `%100$p` will read the 100th value on the stack as an address, even if this value hasn't been provided by the user. The format string can also write data into the stack using the format `%20$n` that writes, in this example, the numbers of characters printed before the symbol at the address provided in the 20th argument. In this attack, the attacker provides an own-built format string with arguments that often don't match the format.

2.1.3 What exploits do

The objective of an exploit is most of the time to manage to perform a memory corruption using one of the errors presented above. Such errors are unexpected behaviours, resulting from programming errors. Such an attack cannot occur if a vulnerability hasn't been introduced beforehand. Attacks are not easy to set up even if vulnerabilities are found. Many Common Vulnerabilities and Exposures (CVEs) are discovered each month [29][13]; however, only a few of them have been proven exploitable. This is because modern systems already have protections against these attacks. Such additional security layers are as an example *canaries* to

detect stack overflow, *Address Space Layout Randomisation* (ASLR) and *Position Independent Executable* (PIE) to prevent the attacker from predicting program and kernel addresses, *Non Executable Stack* (NX) to prevent code stored on the stack to be executed. In order to still exploit the vulnerability, the attackers have to find new ways to bypass these protections.

Thus, an exploitation requires a succession of steps well-ordered to finally trigger the vulnerability [24]. If one step is missing, the vulnerability could never be triggered. In recent programs, thanks to all security layers, attackers have little room for exploitation and have few choices to exploit a vulnerability. This means that knowing the succession of steps required to trigger the vulnerability, an exploitation can be detected and stopped just before all the steps are reached.

This is exactly how **BAE** works, and what is developed in later sections relies on this assertion. Knowing the behaviour of an exploit, by intercepting only few but required instructions, such a behaviour can be detected and the exploitation prevented.

2.2 General Work

2.2.1 eBPF and Rust

eBPF (extended Berkeley Packet Filter) offers a way to program the kernel with high performance, security, and flexibility [15]. eBPF programs are run inside a *Virtual Machine* (VM) in the kernel and compiled so that the program won't crash the kernel. Those programs are loaded from UL inside the kernel only by privileged users. All those features simplify kernel development and allow hotfix, that is to say that a temporary kernel patch can be created and used immediately without waiting for the official Kernel update. eBPF also offers a way to place some probes over kernel functions (KProbes) in order to intercept calls and manage them. Although mainly used along C language in the kernel development, the game-changing apparition of Rust led to the development of Aya, an eBPF library (crate) for the Rust programming language.

Rust is a recent language offering safety features, more than relevant in the Kernel context. Because of the recent addition of Rust support, there are still very

few patches written in Rust, especially using eBPF technology. However, both offer a way to manage and monitor the kernel safer, if well used, than previous works in C. Some kernel modules have already been developed in Rust [31][37]; however, a total rewriting of the kernel isn't feasible now, as explained by Linus Torvalds on LKML, the Linux Kernel Mailing List.

2.2.2 Hot-Patching

Hot-patching isn't something really new in the security research field. The first important work about it was published in 2009 by Jeff Arnold and M. Frans Kaashoek in [4] to mitigate new discovered vulnerabilities. Those hot-patch rely on different techniques to prevent the exploitation [38][2][10][35].

2.3 Function Context

In a programming language, we say a given function has signature to describe the arguments it takes and what it returns. The context is the memory related to a specific call. It includes the registers, which themselves describe the stack segment to use and the location of all arguments.

2.3.1 Calling a function

In the process of calling a function, a exchange of information occurs between the caller and the function itself. This exchange is given by the function's signature, specifying the types and order of arguments it expects. Upon invocation, the caller provides the requisite arguments to the function, which then undertakes processing based on these inputs. Thus, the function may produce a result, which could either be returned to the calling environment or utilised for further operations within the function's scope.

However, even if the act of calling a function seems straightforward, it is not without its pitfalls. Occasionally, coding errors or oversights may inadvertently introduce vulnerabilities within a function's implementation. These vulnerabilities, if exploited, can potentially allow malicious actors to hijack the control flow of the process, inducing unauthorised influence over its execution.

2.3.2 Signature detection

The concept of signature detection relies on the recognition that vulnerabilities within a function are often never triggered under normal usage conditions. In other words, if a function is utilised in accordance with its intended design and specifications, the underlying error may not manifest. The vulnerability may then only be triggered when the function is invoked in an atypical or unexpected context.

Signature detection follow this principle by discerning normal and abnormal contexts in which a function is called. By establishing some criteria to identify typical usage patterns or contexts, it becomes feasible to detect cases where the function is invoked in a manner that deviates from its normal usage. This approach enables measures to be taken to mitigate the potential malicious behaviour or exploitation when the function is executed in an unexpected environment.

Signature detection could thus serve as a preventive mechanism, allowing for the identification of potentially dangerous scenarios where a function is invoked outside its intended parameters or under conditions that may trigger some vulnerabilities. By establishing a baseline of expected behaviour and monitoring deviations from this norm or on the contrary monitoring known deviant behaviours, signature detection could help against unforeseen threats and exploitation.

2.4 Contribution

The primary aim of this project is not to introduce groundbreaking innovations in the field of hot patching and kernel security, with already many research publications [38][41][21][28][39]. Rather, my objective is to demonstrate the feasibility and potential of leveraging the Rust programming language associated with eBPF technology to achieve comparable outcomes. By taking profit of these emerging technologies, I seek to demonstrate that Rust can effectively replace the functionalities traditionally written with C in the context of kernel security.

Moreover with **BAE** , I endeavour to contribute a novel approach in the field of exploit detection through behaviour analysis. This innovative method involves the transformation of observed behaviours into graphical representations, facilitating a deeper understanding of exploits dynamics and relationships between exploitation steps. Thus, I outline how kernel monitoring techniques can be employed to

CHAPTER 2. BACKGROUND

detect abnormal behaviours using these graphs, and so revealing potential exploit execution.

Chapter 3

Methods and Limitations

3.1 Approaches and their limitations

Among all the existing kernels, the Linux Kernel is the most used. Not only Linux users rely on this kernel but also Android. Android is actually the most spread OS [33], because of all smartphones based on Android and all other devices such as Android TV, Android auto, Wear OS... Android is in fact based on the Linux Kernel, so vulnerabilities inside the Linux Kernel can also be present on Android devices.

3.1.1 First approach: Android devices

The decision to initially focus on the Android Kernel was driven by the user base of over 3.9 billion active Android users [8], highlighting the widespread significance of securing this platform against cyber threats. Android applications, serving as primary vectors for attacks, underscore the critical importance of protecting the Android Kernel against vulnerabilities. In addition, Android users are for most of them not enough aware of cyber threats. The open-source nature of the Android Kernel [17] facilitated the development and testing of custom kernels, enabling experimentation within a controlled virtual environment.

However, the effort encountered significant challenges inherent to the Android Kernel ecosystem. The substantial disk space requirement of approximately 300GB [17] and the prolonged compilation time of 4 to 5 hours with 16GB of RAM imposed restrictive constraints on the development and testing process. These limitations

severely restricted the frequency and scope of testing, slowing down progress and efficiency.

Moreover, Android Kernel is based on Linux Kernel what means that the kernel used by Android isn't the current one used by Linux. The integration of a new Linux Kernel into the Android OS starts the release date of this new kernel. This process takes about one year [17][11]. Because Rust support has been released for the Linux Kernel only since the Kernel 6.1 deployed on December 2022, Android Kernel still didn't support Rust. This means that the kernel program should have been written in C, losing all safety measures and benefits provided by Rust, which was one of the main conditions in this project. So, I decided to stop focusing on Android and continue with the Linux Kernel.

3.1.2 Second approach: Kernel modification

Linux Kernel development was much easier compared to Android Kernel. First of all, the source code is smaller - only 8.6G after compilation - and by using the same amount of resources, the compilation time reduces to about 10 minutes. In addition, the last version of the kernel is always available and so Rust is. During this second approach, the goal wasn't yet to write an eBPF program, but to modify the kernel in order to hook each call made from UL to it.

Since Rust has been added to the Linux kernel, only some new modules are written in Rust but all the previous code is still in C. It would take too much time to replace all the Kernel by a Rust one version. Because Rust for Linux is very recent [31], it is difficult to have full control over the kernel using only Rust. The first thing to do was to create new Rust kernel modules which are launched at boot time and allowed to run code inside KL. One more time, this method appeared to be very challenging and not efficient.

Moreover, if the program is launched among all kernel processes, it means that it can't be easily loaded or removed. The program is launched automatically at boot and so consume resources, what means that the program has to be extremely optimised in order to not disturb other kernel processes. This solution wasn't good enough and would have figured only as a Proof of Concept (PoC) interesting for the research but never deployed.

3.1.3 Third approach: Rust eBPF

According to the eBPF documentation, "eBPF is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in a privileged context such as the operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules." [15]. eBPF library was at first written in C when Rust wasn't yet available. However, since Rust for Linux, a Rust library to create eBPF programs has been developed : **Aya** [12]. Aya programs are split into two or three programs. The first one is run by a privileged user in UL in order to load the second program (the eBPF one) inside the Kernel.

This first program specifies what kind of program will run inside the Kernel, its name and the kfunction it will attach. It defines also the life duration of the eBPF program, which will end for **BAE** when the UL program stops it.

The second program being the eBPF one, has to interact with the Kernel as a purpose. In **BAE** , this program adds pieces of code that will be run before the monitored kfunction, just after the call. It acts like a new function called with the same context as the kfunction. Thus, the program cannot directly prevent the kfunction code to be executed and some techniques we will discuss in § 3.4 have to be performed. Because the eBPF program is executed in the same context as the kfunction, all arguments and registers can be analysed. This feature is the key to detect the malicious calls § 3.2.

The third program acts as an interface between the eBPF program and the UL program. It allows to create shared objects and mapping that can be used by both the eBPF program and the UL one. It offers a way to communicate information from KL to UL, like PIDs, IP addresses... This last program isn't compulsory in order to use eBPF but it facilitates some action.

eBPF and particularly **Aya** have many features helping in this master thesis. First of all, eBPF program are run inside a Virtual Machine (VM) so that if an error occurs, it stays far away from the Kernel. The eBPF program is also checked

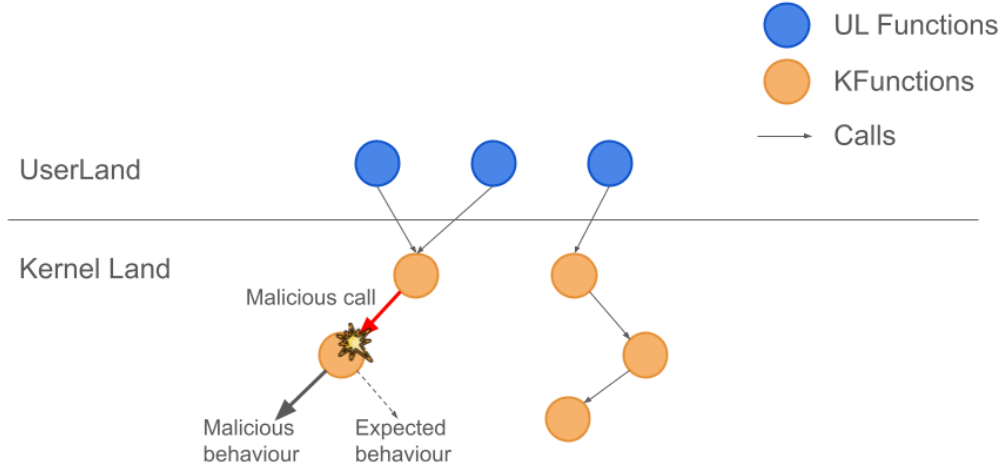


Figure 3.1: Kernel vulnerability triggered

at compilation time to ensure that it will end and not crash the Kernel. Combined with the memory-safety of Rust, **Aya** becomes a powerful library to write kernel programs. Another advantage of eBPF programs is that they can be inserted or removed dynamically inside the Kernel and so updated quickly. The kernel doesn't require to be recompiled to load the program and when this one is updated, it can be released instantly to all users using it, without waiting for the next kernel update. This is what we call a hot-fix. **Aya** also offers a powerful tool simply named **aya-tool** [12] allowing the conversion between C structures and Rust. Together, all these features offer the possibility to hook kfunctions, analyse their context and react accordingly, through KL and/or UL.

3.2 Hooking Kernel Functions

Hooking Kernel functions was the first action to achieve to ensure the feasibility of the idea. There were two ways to achieve this hooking at different levels in the Kernel. The hook can be performed either at the Kernel entry or just before the execution of the kfunction. Both have their pros and cons. We'll now focus over each of these techniques.

3.2.1 Detecting calls at the entry point

The first option is to hook every call at the Kernel entry point. It means that each time a call to the Kernel is performed, it will be hooked and analysed. On Figure 3.1, the hook happens on the calls crossing the line between UL and KL. This idea isn't the one I have chosen to implement for different reasons that I will expose later in this section. We can describe a call as a path followed in a graph whose nodes are all kernel functions (Figure 3.1), driving information through each node crossed. The goal is here to predict the path that the call will follow inside the kernel and determine if at one node, the control flow will change maliciously that-is-to-say, conducing to a malicious behaviour instead of the expected one. In order to do that, the kernel has to simulate the successions of calls before allowing them to be executed. Before reaching a new node (i.e. executing a new function), the context can be analysed to determine if the call is potentially malicious. This method is extremely powerful in the case where each case can be treated efficiently and accurately.

However this method also comes with its set of hurdles. One of the trickiest issue is to hook each call at kernel entry, what cannot be achieved without modifying some of the main features of the kernel. In the case where this hook is feasible, the next step would have been to predict the path of the call inside the kernel given only the initial context. If we assume again that this second issue is solved, the final step is to determine when a call is malicious. All these three steps are extremely complex tasks that cannot be performed easily on top of the existing Linux Kernel. Moreover, some of them need huge calculations slowing down drastically the kernel. Because the kernel is a sensitive environment, we cannot allow a too important loss of computation power. The method is powerful theoretically, but not really applicable.

3.2.2 Detecting calls at the function entry

The second option is to hook some well chosen kernel functions and analyse the call just before executing the function code. Relying on Figure 3.1, the hook happens on some well chosen kernel nodes, and particularly the one where the vulnerability is triggered. This is the approach I have chosen in this master thesis. Unlike hooking

at the kernel entry, calls are intercepted when a function is called, that is to say when an instruction ordering to call the hooked function is executed. This means that all the context (registers and all the memory) in which the function is executed is known at hook time. Thus, we don't need anymore to predict the path followed by the call and how the memory will change. This save a lot of computations and so the kernel is far less slowed down. However each function has to be hooked independently, but the same hook can be applied to different functions. It means that if given a context and a function name a check function can determine if the call is malicious or not, this check function could be applied to all kernel functions hook. This check function is actually the hardest part of this method, and is the one we'll elaborate in Chapter 4. Hooking and some of the argument analysis is performed using **Aya**. Using Figure 3.1, the analysis is now for each function to determine if the result of the function will be an undetermined (potentially malicious) behaviour or an expected one.

The Aya Rust library, used to develop eBPF programs written in Rust for the Linux Kernel offers the tool to detect function's call and hook them. This tool is called KProbe which stands for Kernel Probe. The goal of a KProbe is to wait for a kernel function to be called and when this happens, place a piece of code before executing the function. When the added code is executed, the control flow returns to the kernel function. It means that those hook cannot prevent directly the function to be executed in case of a malicious call. However, it remains possible to stop the execution using some tricks on which we'll focus on in § 3.4.

3.3 Analysing the Context

No matter the method chosen, in order to determine if a kfunction is called in a malicious way or not, the arguments provided through the registers and the other memory set (stack, heap, environment...) have to be analysed. All this memory is what we call the context in which the function is executed. This context determines the behaviour of the function and if an exploit uses ones function vulnerability, the attacker has to modify the context in order to exploit this vulnerability. In this case, I will assume that only the arguments can be used to determine the behaviour of the call. This assumption can be done because in order to modify the stack, the

heap or the environment, another function has to be called with specific arguments. So we can simply move the place where the attack happens at a higher level in the succession of calls, making the exploit still detectable through a meticulous arguments analysis.

Using **Aya**, the context can be analysed as a `ProbeContext` and all arguments and registers can be read. The knowledge of the functions arguments and registers offers a way to know the stack segment used as well using the stack and frame pointers. Often, arguments provided to the `kfunction` are complex C structures that can be hard to parse in another language. Fortunately, **aya-tool** can help to read C kernel structures. **aya-tool** is a function taking as arguments the structures that have to be converted and provide as an output a Rust file that can be imported in the eBPF program to convert C memory describing a structure into a Rust object representing this given C structure. However, all conversion are not supported by **aya-tool** and sometimes it can be necessary to make this conversion by hand as explained later in § 4.6.

3.4 Measures to Prevent an Attack

When an attack is detected, some actions have to be taken in order to prevent it to happen. As stated before, hooking a function with aya eBPF cannot stop the call to happen. However, because the program has a control over some kernel resources, some tricks can be used to circumvent this issue.

Inside the Kernel, the program doesn't have a direct control over what happens in UL, but both KL and UL program can communicate through mappings declared in the common program [15]. If the eBPF discovers a malicious call about to happen, it can share the malicious PID to the User program that will then kill it. However this solution isn't perfect due to race conditions. Multiple programs have to be executed at the same time, but in a simple architecture there is only one processor that can execute one instruction at a time. The goal of the OS is then to execute processes instruction in turn with different states to be simple: Run, Ready and Wait. Because the order of instruction execution cannot be determined in advance, when two programs share some data and are executed simultaneously, some weird results can happen. This phenomenon is called race condition. In our case, a

CHAPTER 3. METHODS AND LIMITATIONS

malicious call is executed at the same time the User program wants to kill it so, if the OS gives priority to the malicious call, the bad action will happen before being killed. Moreover, the eBPF program can't wait the malicious call while it isn't killed because eBPF programs have to terminate in order not to introduce waiting in the Kernel. Thus, terminating a program would require further control, but it remains feasible as demonstrates Tetragon [35].

Another option would have been to redirect the control flow to an exit point as presented in [38]. The eBPF program can access the registers and particularly the instruction pointer. This pointer is specific for each process and determines where the following instruction to execute is located inside the memory. By editing this pointer, the next instruction that would have to be executed can be changed. As an example, the context can be modified in order to execute the exit function to terminate the current process. This method requires the use of other technologies since the eBPF VM has only read access over most of the kernel memory.

Nonetheless, we have to be careful when modifying Kernel processes' life. The kernel is a very sensitive region and if it crashes, all the system has to be rebooted with possible data loss. For this reason, we can't allow false positives that would stop normal kernel processes. The prevent part is not implemented in this project as we focus on attack detection, but this section is here to provide some trail to complete the project.

Chapter 4

Monitor Malicious Behaviours

Hooking function calls of interest is one thing but determining if this call is part of an exploit is another task. In order to properly define a way to detect a malicious program, an understanding of programs behaviour is mandatory.

4.1 Program behaviour

This section aims to understand the logical structure of a program and how this could be used in order to detect a known malicious behaviour.

4.1.1 Logical structure

In essence, a program operates as a structured sequence of instructions, each designed to execute a specific task or achieve a particular objective. This sequential flow of instructions forms the logical structure of the program, wherein computations are performed, and conditions are verified to progress toward the desired outcomes.

To comprehensively understand the behaviour of a program, it can be abstracted into a graphical representation known as a Control Flow Graph (CFG) [3]. The CFG represents all possible paths that the program may traverse during its execution. Each node in the graph corresponds to a specific instruction or block of code, while the edges denote the flow of control between these instructions. By traversing the CFG, one can visualise and analyse the various execution paths within the program, identifying decision points, loops, and branches that dictate its behaviour.

By mapping out the logical structure of a program through a Control Flow Graph, we gain valuable insights into its behaviour, facilitating the understanding

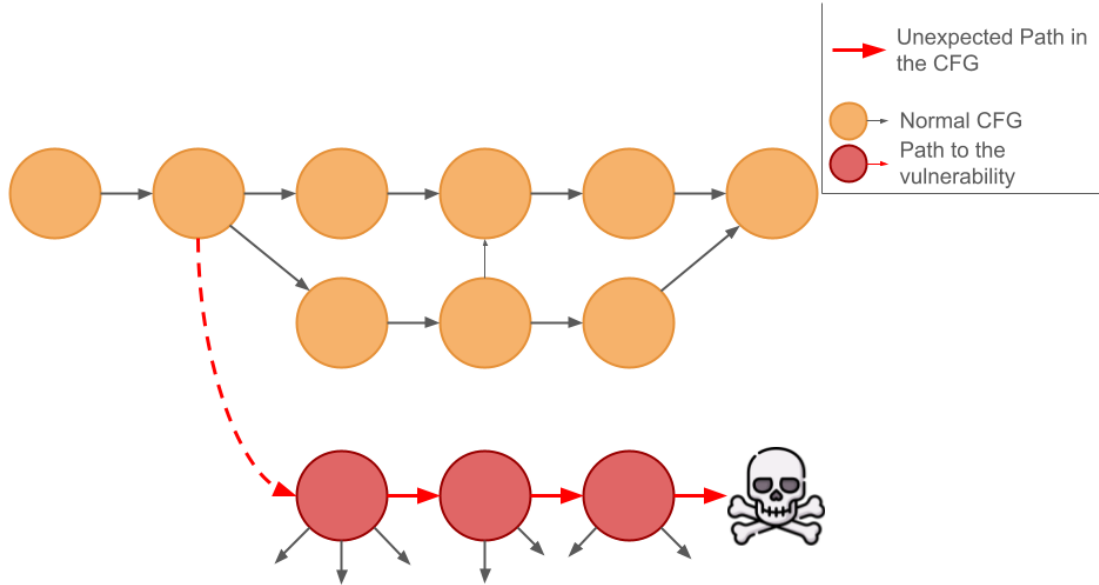


Figure 4.1: Vulnerability in the CFG representation

of program execution as in [2].

4.1.2 Application to an exploit

An exploit is a program as another except its purpose is to trigger a vulnerability. In order to do so, the malicious program has to follow a succession of instructions. Indeed, a vulnerability is rarely triggered randomly, or it provokes a crash. A normal user will never see a root shell appears in a conventional usage of a program whose purpose has nothing related to this event. However, when a program contains a vulnerability, all events can happen since the vulnerability can lead to the redirection of the control flow to other instructions, chosen by the attacker. It should be noticed that this control flow redirection, as a possible event, belongs to the initial CFG as shown in Figure 4.1. Even if the next steps depend of the payload, the CFG must contain a path leading to an error, not caught by the program. Those path are very sensitive and specifics instruction must be follow in order to achieve the exploitation. It means that if one step is missing, the vulnerability won't be triggered. On Figure 4.1, red arrows have to be followed to avoid going out of the path leading to the vulnerability.

BAE relies on this feature. First of all, even when a vulnerability exists, it isn't

necessarily exploitable, until a proof of concept is discovered [13][29]. Secondly, when a vulnerability is exploitable, the exploit cannot be changed easily and should still follow some unavoidable instructions [7]. Some exploits even work only on specific systems. Thus, the overall idea here is to detect when a program follows the path leading to the vulnerability triggered in the CFG. The objective is then to map the exploit behaviour and identify the instructions leading to the exploitation.

4.2 Pattern recognition

Having now a better understanding of exploit's behaviour, the focus is set on recognising a pattern used by these exploits to trigger a vulnerability. We don't need to know the later purpose of the exploit, that-is-to-say if it will open a shell, connect to a command and control server or leak some data... If the attacker reaches this point, it is already too late. The detection operates before this happens.

4.2.1 Graph of Conditions

In order to detect some behaviour, it is required to feed the detector program with some data representing the behaviour to monitor as **PET** is fed with sanitizer reports in [38]. Since the data to detect is nothing but a sub graph of the CFG, the best representation is a graph whose node are the instructions to follow to trigger the vulnerability. Most particularly, the graph is a directed acyclic graph. The graph is:

Directed because a program is a succession of instructions and operations destined to be executed by the processor in order. Some instructions can be executed multiple times but this is due to the logic flow of the program, represented in the CFG.

Acyclic because instructions have to be executed in a certain order so that the vulnerability is correctly triggered. Thus in the graph, it means that to reach a node, all its parents have to be reached before. If there is a cycle among the graph, it means that one node (belonging to the cycle) is also its own ancestor, and thus, will never be reached and stay in a blocked state.

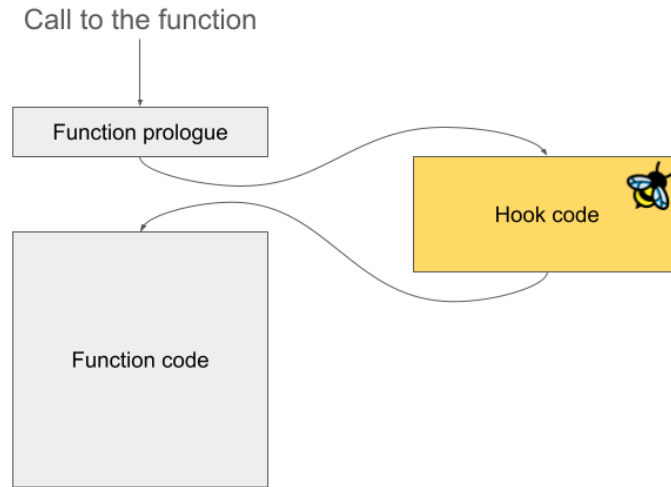


Figure 4.2: Insertion of a hook using KProbes

Some conditions are required to go from one node to the following ones. These conditions are the essence of the work, and maybe the most important part of this research. How we gather, represent and detect these conditions determines the ability of the program to prevent the focused exploit. Those conditions will be described in a later section § 4.3.2. If we compare with Figure 4.1, a Graph of Conditions can be the graph formed with the nodes linked by red arrows.

In order to build this Graph of Conditions (**GoC**), a deep analysis of the exploit behaviour has to be realised. Though this topic won't be detailed in this work, some tools such as **ftrace** can be used to enlighten what calls a program is sending to the kernel as explained in § 5.3. Another method would be to inspect the source code and determine the subsequent kernel calls following a C function (memory allocation, TCP connection establishment...). In this work, data required to build the conditions are mostly supposed to be already gathered.

4.2.2 Recognise a pattern

A graph's condition relies on two main information : the **kernel function** called by the program and the actual **condition** that should be validated in the context of this current kfunction.

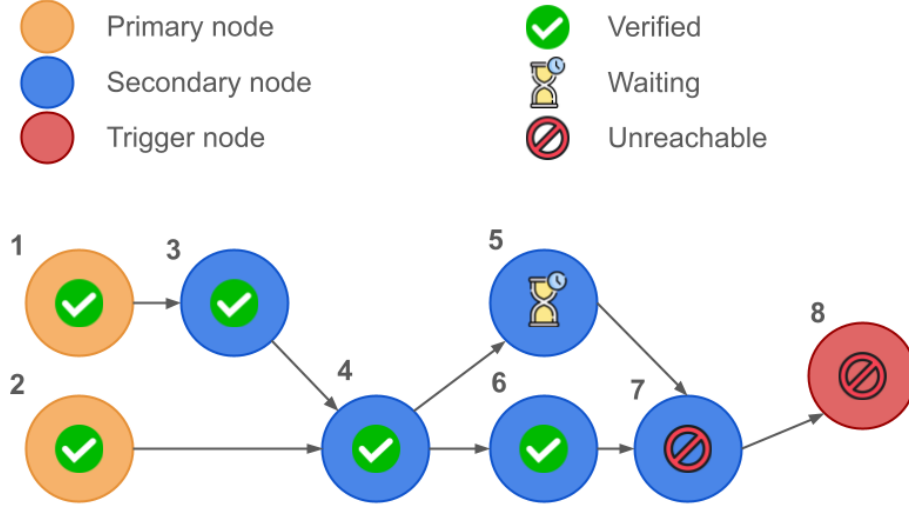


Figure 4.3: Representation of the Graph of Condition for a given process

Node ID	1	2	3	4	5	6	7	8
State	V	V	V	V	W	V	U	U

Table 4.1: Graph of Condition in Figure 4.3 representation inside the memory

In order to recognise a pattern, that-is-to-say a sequence of conditions satisfied by a monitored actor, every kfunctions involved in the graph of condition are involved and must be tracked. In order to do so, the eBPF feature named *KProbe* allowing kernel function hooking was the great candidate. Basically, the hook is placed at the entry of the hooked kfunction, just after the function prologue, so with all the context in which the call has been executed (arguments, environment, process id...). The control flow will be redirected to the hook, allowing collecting all information needed, and then will return to the caller as represented in Figure 4.2.

When a probed function is hooked, checks and logging have to be performed to determine if the call is a new step in one vulnerability exploitation. Each *Process Identifier* (PID) is mapped to its current **GoC**. If the current PID and kfunction matches a condition to verify, the context is sent to a checker that will perform the check. If the check is positive, the condition is satisfied and the position in the graph is updated.

CHAPTER 4. MONITOR MALICIOUS BEHAVIOURS

The graph is stored as a list of states as shown in Table 4.1, where each node has one unique state. These states are the three following:

Waiting The node is one of the the following in a vulnerability exploitation and the condition attached to this node has to be verified. If the result is negative, the node remains in a Waiting state, otherwise it becomes a Verified node.

Verified The node has already been verified. This status indicates the action already happened and doesn't need any new check, since the graph is directed. When all parents of one node are verified, this node is no longer Unreachable and becomes Waiting.

Unreachable Some other instructions have to be performed before. Any check at this point is useless and the node shouldn't be inspected. This status indicates that at least one parents of the node is still not verified, as **Node 7** in Figure 4.3, translating that some conditions have to be met before.

In addition to these states, each node is declared with one type, which is fixed by the **GoC** and will never change. Three types represent the three classes of nodes:

Primary Those nodes are the entries in the **GoC**. They have no parents and at least one child. Their status is initialised as Waiting since they are the first condition to meet in order to exploit one vulnerability. One vulnerability can have multiple nodes declared Primary.

Trigger Unlike Primary nodes, Trigger nodes are the ones leading to the vulnerability triggering. If this node is reached, one pattern has been executed and an exploitation could occur. Trigger nodes must have at least one parent and have no child. Multiple Trigger nodes can be present in the **GoC**. Their status is Unreachable and if it were about to change, it means the vulnerability is about to be triggered.

Secondary The most common nodes in the **GoC**. They are the nodes between the Primary and Trigger ones. They have at least one parent and one child and are all along the path from a Primary node to a Trigger one. Their status is initialised as

Unreachable.

Using the combination of types and states, one can monitor processes behaviour and determine if a vulnerability is about to be exploited by watching the Trigger nodes' state. It must be noticed that different vulnerabilities can share a same sequence of instructions at one point and thus their respective **GoC** could be merged. Moreover, the **GoC** is a list of the states for each node (the complete graph being immutable) and thus, the minimal size required is *2 bits * the number of nodes in the graph*, which is $O(n)$ over the number of conditions for one process. The total space required to store all the graphs is $O(nm)$ with n the number of conditions and m the number of processes monitored (increasing over time). These requirements will be analysed in the section about scalability in § 5.2.2.

4.2.3 From the GoC to the detector

The detector runs in the kernel what makes dynamic memory allocation a challenge [12]. Data structure must have a fixed size during compilation and thus the **GoC** must be known at the compilation time. This means that the files to compile must be modified and recompiled after each change in the **GoC**. In order to apply changes smoothly and conveniently, a loading script is run before compiling the detector to apply the current behaviours to monitor. This loader takes the graph as an argument and complete the detector source files using a set of templates. This allows a fast and secure way to load a new graph in the detector as the only task the user must do is modifying the graph and running the loader. Thanks to the use of templates, the formatted graph, and the pre-checking of the graph, we are ensured of the correctness of the resulting code, and avoid any human interaction with the source code.

4.3 A language to describe the graph

Since the **GoC** is the core of the detector and must be completed for each new vulnerability the user wants to hot-fix, it should be described in a simple and efficient way. Thus, I have implemented a new representation language to intuitively describe the **GoC** : **BFFL** .

4.3.1 Intuitively represent dependencies

Each node is represented on a line by line basis so that it remains easy to add and remove nodes. Each line follows the following model:

<TYPE> <ID> SATISFIES <CONDITION> DEPENDS <PARENTS>

- **TYPE** is the type of the node: either *PRIMARY*, *SECONDARY* or *TRIGGER*
- **ID** is the node ID used to declare parents. It must obviously be unique but not necessarily in order.
- **CONDITION** is the condition to satisfy, detailed in the next section.
- **PARENTS** is a list of IDs surrounded by brackets

This language offers an intuitive way to represent a node and all its characteristics. In addition, the representation doesn't depend on the final language used. The parser, written with Python3 recreates the final **GoC**, generates required information and performs checks to assert the correctness of the graph. These checks are:

1. Nodes don't have parents if and only if they're *PRIMARY*
2. Nodes don't have children if and only if they're *TRIGGER*
3. The graph contains no loop
4. Each node is reachable.

All these information, present in Figure 4.3, permit to describe the structure and the dependencies of the **GoC** and ensure its correctness, preventing any non-terminating loop or unreachable condition. The structure being completed, the logic remains to explain.

4.3.2 Describe conditions

This section is destined to explain how to describe the conditions that processes must pass to go one step further in the **GoC**. The focus is made on the *<CONDITION>* part of the node description. Here relies the main part of the behaviour analysis

CHAPTER 4. MONITOR MALICIOUS BEHAVIOURS

performed by **BAE** . Each condition belongs to one kfunction, that-is-to-say one call, and so the involved kfunction must always be specified. Then, the condition to check differs widely but can be classified in sub groups. The three groups I present in this work are the conditions based on **Arguments**, **Count** of calls and **PID**. The first one is the trickiest one and will be explained later in this section.

The **Count** and **PID** groups are, in term of logic, quite intuitive. In the case of the **Count** group, the kfunction has to be called a certain amount of times. Regarding the **PID** group, the process ID is required to satisfy some value. A case in point would be having been called before or after another process. The language representation adopted is the following one:

<KFUNCTION> <CALLED|PID> <OPERATOR> <VALUE> [AND|OR <CONDITION>]

The *OPERATOR* belongs to the common operators (`==`, `!=`, `>`, `>=`, `<`, `<=`) and the *VALUE* is an unsigned integer over 32 bits. This allows an intuitive way to represent the condition.

However most of the time, in order to trigger a vulnerability the exploit won't rely on how many times a function has been called or if other events happened before. The condition will meet some criteria over the context in which the function is executed, its arguments. This is the last group and the more complex one.

When the function is hooked, the types and name of arguments are unknown. The only accessible data is the block of memory with pointers to each argument data. The first argument could be a structure describing a TCP connection or a simple integer. Moreover, the condition could rely on a specific value inside a complex structure and the path leading to the value could involve new memory readings due to pointers. As an example, let's focus on the `tcp_connect` kfunction whose signature is the following:

```
int tcp_connect(struct sock *sk);
```

Let's suppose we want to know the port used by the connection. The port is accessible through the following path : `sk->__sk_common.skc_dport`. The only argument given by the `tcp_connect` function is a pointer to the structure `sock`. Thus to access the required value, it is required to dereference the address of `sk`, get

the address of `__sk_common`, dereference it and then pick the value of the `dport`. All those operations will be operated by the **BAE** dynamically, and the actions to execute to gather the correct final values must be described properly and intuitively in the graph description, provided by the user.

Since all kernel structures are for now almost all written in C, the C representation of the argument to extract is convenient. Thus, an arrow (`->`) induces memory dereferencing and a dot (`.`) a memory reading. We also suppose that given the name of the field, we are able to determine its position inside the memory (with Rust, we can use `aya_tool`). The type should be specified for typed languages. Thus the language representation of the argument's condition is the following

```
ARGS ARG_N[< -> | . >(type)field...] <OPERATOR> \
<VALUE> [AND|OR <CONDITION>]
```

The N in `ARG_N` represents the argument position in the signature (starting with 0) the *OPERATOR* belongs to the common operators (`==`, `!=`, `>`, `>=`, `<`, `<=`) and the *VALUE* depends on the final value, but must be a primary type. Here again, it offers an intuitive way to describe the condition over the arguments with all required information.

4.4 Kernel-land and Userland

When dealing with the kernel, an important feature to have in mind is the location of data and code instructions. The kernel-land is a sensitive environment far more controlled than the userland. Only tasks specific to the kernel should be operated by it. Thus, **BAE** is made of two joined programs, one in KL and the other one in UL that operate together in order to detect threats.

4.4.1 Kernel tasks

Kernel memory accesses are very limited and the code running inside must be optimised to not slow down the entire system and OS processes. The KL has far less resources than UL and thus it cannot store too much data. Moreover as a sensitive environment, the kernel cannot be paused, as for waiting for a result for example.

CHAPTER 4. MONITOR MALICIOUS BEHAVIOURS

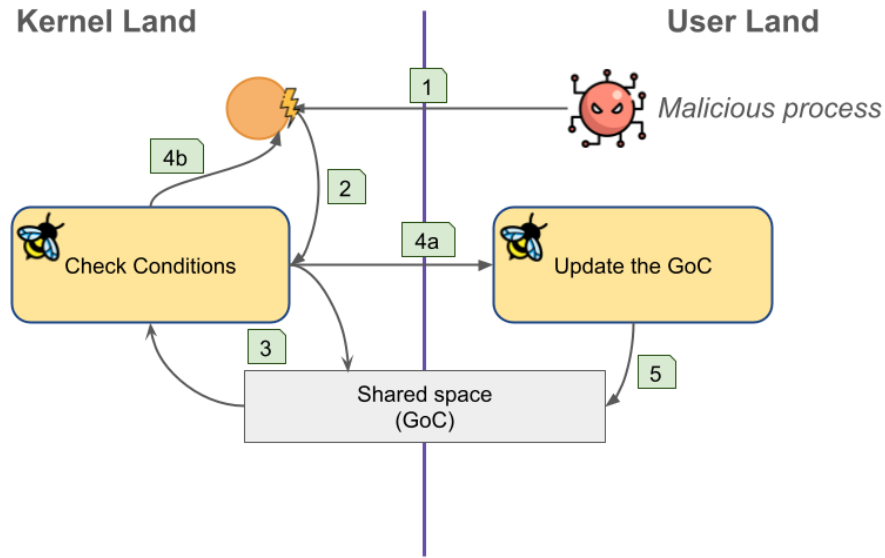


Figure 4.4: Processing the call to a hooked function between Kernel Land and User Land

1. The malicious process makes a call to a hooked function
2. The call is intercepted by the **KProbe** and the control flow redirected to the eBPF program
3. The program checks if a condition is met using the **GoC** in the shared space
- 4a. If the condition is met, an update is required to Userland
- 4b. By the same time the control flow returns to the function
5. Userland may update the **GoC** for future checks

CHAPTER 4. MONITOR MALICIOUS BEHAVIOURS

However, since we deal with kernel exploitation, most of the events take place in KL and should be inspected. Thus the goal of the kernel program is to:

Hook kernel functions The kfunction being called inside the kernel, so this task belongs to KL. This is the second step in Figure 4.4.

Read memory When checking the conditions, most of the arguments are stored inside the kernel memory and must then be read in KL. Conditions must then be verified in the kernel since communication between KL and UL is unidirectional as we'll explain. To verify if a condition must be checked, the KL program must first verify the current status of the **GoC** provided in a shared space between UL and KL (step 3 in Figure 4.4).

4.4.2 Delegate tasks to Userland

Sometimes, tasks must be delegated to the user to optimise performances. However, the communication can only be achieved from KL to UL as KL has more privileges than UL. Moreover, since the kernel cannot be paused and so cannot execute pseudo infinite loop, there is no simple way to wait for any UL answer. This implies that when a task is delegated to UL, it cannot be given back to KL. All kernel operations must then be completed before sending the data to the UL to finish the treatment of the event.

Fortunately, eBPF offers a feature to facilitate the communication: a shared space. This common space offers a way to manipulate data structure accessible from both KL and UL. It avoids declaring huge data in KL and permits to both party to modify the same data without giving it through some calls.

A last feature for the communication is the **RingBuffer**. The **RingBuffer** is a buffer in a ring shape, with a fixed size. This data structure can be fed by the Kernel and then collected by User (step 4a. in Figure 4.4), which can wait for new data to appear in a loop. Thanks to this features, when the task in KL is finished, it can be signalled to UL with the relevant data through the **RingBuffer**, so that the task can finish being processed in UL.

4.4.3 Userland tasks

Userland is a far less sensitive area than Kernel-land. Thus, we can allow executing the more tasks we can in it. The only limitation is the lack of power of UL over kernel data. Nonetheless, some tasks don't require such privileges and can easily be performed in UL.

Initialise environment At the far beginning of the program, all the environment must be set up and the graph loaded inside the memory. As the memory can be shared between KL and UL, this task can be executed in UL at runtime. Basically, functions to hook are declared and the **GoC** is loaded. Then the program waits for further tasks, should it be a program interruption or new events from the kernel.

Update the graph After KL processed an event, if a condition have been verified, it notifies UL with the new detected event. This event is as simple as a *PID* and a *Node ID*. From that, the graph can be updated as well as are the status of affected nodes (step 5. in Figure 4.4). The graph being up to date, if a **TRIGGER** node is now reachable, it means a pattern has been detected.

Prevent the attack When an attack is detected, the final decision belongs to UL which knows the PID and can decide to kill the malicious or do any other action. We'll come back later in Chapter 7 on this particular point.

4.5 Use of Rust

This section is apart from the rest of the work since here, I justify my choice of having used the Rust language to implement the project.

4.5.1 Advantages

Rust is a very recent language maintained by the Mozilla team [32] that offers plenty of advantages compared to other languages.

Performance Rust is a compiled language, very optimised as efficient as C. This makes Rust an appropriate language to develop Kernel features.

CHAPTER 4. MONITOR MALICIOUS BEHAVIOURS

Security One important point emphasised by Rust maintainers is the safety of the language. Thanks to an extremely powerful compiler, compiled Rust programs are ensured to avoid most of the memory issues such as memory leaks, short variable lifetime or multiple ownership. Since the purpose of this project is to improve security in the system, a memory safe language as Rust is a perfect candidate.

Versatility Rust gather a strong community and offers the possibility to extends its features with the use of crates. A large choice of crates is available in the marketplace [30] and among these choices relies **Aya**, the eBPF one. This feature was essential to ease the development of kernel monitoring.

Portability Rust can be used on multiple platforms should it be different hardware or levels of programming. Rust offers a way to develop Object Oriented Programming software, bare-metal program or in this case, kernel function hooking.

All these features make Rust an innovative language and a good candidate to replace old C. Moreover, Rust is the first language after C being integrated in the Linux kernel [31].

4.5.2 Challenges

All these benefits come with some challenges yet.

Conversion with C As mentioned in previous sections, analysed data come mostly from C kernel functions. Thus, all read structures are C ones. Because of the use of Rust, a straightforward conversion isn't feasible and some more steps are required. These steps involve the use of a rewrite of the structures in Rust language. Fortunately, **aya-tool**, the tool developed by the Aya team already offers this conversion for a large set of C structures.

Strict compilation Rust is a safe language thanks to its strict rules during compilation. However, these checks can sometimes be too strict and induce weird compilation errors leading sometimes to writing a more complex piece of code. This comes most of the time when dealing with dynamic data in the kernel which doesn't

allow easy dynamic memory allocation.

Protected areas Rust doesn't allow multiple references in write mode on a single object. It can be challenging while using the same data in both KL and UL without easy communication between both parties. Thus, write access to shared structures should be delegated to only one side, UL in this case as it is the part managing and updating changes.

Too recent features Because Rust is a very recent language, many features are still under development. This is the case for the Aya library, used in this project to access eBPF powerful capabilities. However, the project isn't totally released yet and thus the documentation is still incomplete [12]. This leads to a lack of resources related to specific features of the library, although counterbalanced by an active and efficient online community.

In a nutshell, Rust language isn't the easier choice to develop such specific kind of programs but all challenges are not blocking and not pitfalls. The overall benefits are worth the time spent solving difficulties encountered.

4.6 Limitations

Theoretically, this project should fix most of the new discovered vulnerabilities and prevent future attacks. Though it mostly achieves its objectives as we'll explain in Chapter 5, I discovered while implementing the project some new challenges and limitations that I would write down at the end of this chapter.

Parallelism A single event is processed by the kernel and then, if an update is required, processed by the user program. Thus, if two monitored steps are extremely closed (timing speaking), due to the system parallelisation, it can happen than the second event is processed by the kernel before the user updates following the first event. Thus, the same event happening very often can be processed multiple times or two closed events could remain undetected.

CHAPTER 4. MONITOR MALICIOUS BEHAVIOURS

Split attack Since the GoC applies to a single process, an easy attack to bypass the security would be to split the exploit (when possible) into different processes running in order. A solution to mitigate this issue would be to use the condition over PID to check if some processes run under the same parent or to use a Global Graph of Condition with no dependencies to any PID.

Over and under "fitting" When detailing an exploit behaviour, one has to take care to monitor only required steps, but not missing important ones. Indeed in one hand, since each step has to be followed to be detected by **BAE**, any changes in the exploit code could impede the detection. On the other hand if not enough steps are monitored, it could result in a larger rate of false positives. Independently, one exploit's instruction isn't malicious and could be executed by many other processes innocently. The monitored steps have to represent only the essential steps leading to vulnerability exploitation.

Memory required One of the main hurdles encountered is the physical memory required. The memory needs to be allocated statically at compilation time, and the given space has to be not too short nor too large. If allocated memory is too short, the logs buffer will be full after a short time preventing the log of future processes or in opposite stopping the detection of an older process. Conversely, a too large memory allocation isn't a solution as it would slow down considerably system performances. To avoid these issues, the size of the buffer should be chosen so that it is always full but when a new process needs some space, another can be removed with enough confidence. We'll come back on this part in the section about scalability § 5.2.2.

Stopping the attack One sensitive question discussed later in Chapter 7 is the action taken after detecting a malicious process. This question has not been treated in this work as it doesn't impact the overall project but remains an import point to discuss. In such a sensitive environment as the kernel, stopping a process can lead to some systems' dysfunctions.

Chapter 5

Experiments

BAE has been tested through different experiments in order to evaluate the real world feasibility and use of such a program. The system used for running the tests is an x86_64 ArchLinux 6.8.6. As **BAE** operates in a sensitive environment, the detection efficiency is one of the main features to put to test. A known vulnerability has to be detected efficiently and on the contrary, false positives shouldn't be raised to not compromise the system's stability. Beyond the logical accuracy, it remains important to consider **BAE**'s performances, should it be on the system's overall impact or its scalability. Last but not least, the deployment and update time is a major consideration to evaluate as it represents the real world complexity required to set up the tool and respond to new vulnerabilities.

5.1 Detection Efficiency

In this section, the logical efficiency of the detection is put to the proof. Different CVEs and their PoC has been used to test **BAE** with concrete examples. We'll consider different CVEs to endeavour the detection and then determine the accuracy by evaluating the False Positives (FP) and False Negatives (FN).

5.1.1 Known Vulnerability Detection

The CVEs used for this experiment are **CVE-2021-22555** [1][29], **CVE-2022-0847** [26][29] and **CVE-2022-2588** [27][29]. Those three CVEs rely on different vulnerabilities to demonstrate the ability of **BAE** to mitigate various classes of errors. Details about the spotted conditions and how they've been detected follow.

CVE-2021-22555 : This vulnerability relies on a heap out-of-bounds error leading to privilege escalation or deny of service [29]. As explained in the write up written by Andy Nguyen in [1], in order to exploit this error, the exploit first uses a use-after-free achieved with a heap-spraying to obtain the control over a memory pointer. Thus, to monitor this exploit, we'll trace the use-after-free steps by detecting the heap spraying. The essential steps are:

1. Creating multiple sockets with `socket` and `socketpair`.
2. Initialising the message queue with calls to `msgget`.
3. Filling the queue with primary and secondary messages with `msgsnd`.
4. Creating holes with `msgrcv` leading to the **UAF**.
5. Trigger the **OOB**.

The graph of condition used to monitor these steps is the following one:

```
PRIMARY 0 SATISFIES __sys_socket CALLED == 1 DEPENDS []
PRIMARY 1 SATISFIES __sys_socketpair CALLED == 4 DEPENDS []
SECONDARY 2 SATISFIES __x64_sys_msgget CALLED == 4096 DEPENDS [0,1]
SECONDARY 3 SATISFIES do_msgsnd CALLED == 8192 DEPENDS [2]
SECONDARY 4 SATISFIES do_msgrcv CALLED == 4096 DEPENDS [3]
TRIGGER 5 SATISFIES do_msgrcv PID == 0 DEPENDS [4]
```

CVE-2022-0847 aka *DirtyPipe*: A wrong initialisation of flags for new pipe buffer structures allowed file merging for read-only files [29]. Max Kellermann, the one who discovered and exploited the bug, explains in [26] that to exploit the vulnerability, a pipe has to be prepared, that-is-to-say filled and then drained allowing file writing by splice. Here are the main steps:

1. Create the pipe with `pipe` and get its size with `fcntl`.
2. Fill it with `write`.
3. Drain it with `read`.

4. Merge files with `splice`.

So is the relevant GoC :

```
PRIMARY 0 SATISFIES do_pipe2 CALLED == 1 DEPENDS []
PRIMARY 1 SATISFIES do_fcntl ARGS ARG_1 == uint32_t(1032) DEPENDS []
SECONDARY 2 SATISFIES ksys_write CALLED >= 1 DEPENDS [0,1]
SECONDARY 3 SATISFIES ksys_read CALLED >= 1 DEPENDS [0,1]
SECONDARY 4 SATISFIES do_splice ARGS ARG_4 == uint8_t(1) \
    AND ARGS ARG_5 == uint8_t(0) DEPENDS [0,1]
TRIGGER 5 SATISFIES do_splice PID == 0 DEPENDS [2,3,4]
```

CVE-2022-2588 : An incoherent check in `route4_change` causes to add a filter only if its handler is not null but then free it even if the handler was null [29]. This causes a double-free error leading to possible privilege escalation. The steps to verify are:

1. Socket creation with `socket` and parameters `PF_NETLINK` (value 16) and `SOCK_RAW` (value 3).
2. Three calls to `route4_change` with a handler set to 0 to allocate and then double free the object.

The GoC is then :

```
PRIMARY 0 SATISFIES __sys_socket ARGS ARG_0 == uint32_t(16) \
    AND ARGS ARG_1 == uint32_t(3) DEPENDS []
SECONDARY 1 SATISFIES route4_change ARGS ARG_5-> \
    route4_filter.handle == uint32_t(0) AND CALLED >= 2 DEPENDS [0]
TRIGGER 2 SATISFIES route4_change PID == 0 DEPENDS [1]
```

These graphs have been merged in a single file to allow multiple detection. Over running the exploits up to 20 times, BAE obtained the accuracy presented in Table 5.1.

The exploits behaviour are detected in 100% of cases for the first and third CVE but the DirtyPipe one sometimes misses the detection over the conditions 2 and 3.

CVE-2021-22555	CVE-2022-0847	CVE-2022-2588
100%	90%	100%

Table 5.1: Accuracy of the detector for given CVEs

Condition	CVE-2021-22555	CVE-2022-0847	CVE-2022-2588
0	19521	62753	0
1	1698	0	0
2	0	0	-
3	0	0	-
4	0	0	-

Table 5.2: Number of processes reaching conditions for different CVEs

The **write** and **read** kernel calls are not always detected. This miss-detection may be due by some special pipes initialisation. This accuracy has also been obtained after some modifications of the **GoC** we'll talk about in the following section.

5.1.2 False Positives and False Negatives

The accuracy of the detection is extremely important. Since the goal of **BAE** is to prevent malicious processes to achieve theirs, false positives could on one hand prevent the use of a normal process and false negatives on the other hand leading to the system's exploitability. In order to evaluate this behaviour, I let **BAE** monitor the system for a total duration of 8 hours. A resume of steps reached for each of the previous vulnerabilities is detailed in Table 5.2.

This analysis demonstrates that for the given CVEs, the probability of FP is extremely low. It can almost be declared null in cases such as CVE-2022-2588 where the condition is very restrictive over the function's arguments. In other cases, one condition can be regularly achieved, but after each new reached conditions, the probability to trigger the next one with a normal behaviour decreases a lot. These results are in accordance with the project and what was stated in the section about exploit behaviours § 4.1. To trigger a vulnerability, an exploit must execute very specific calls that are very unexpected to happen in the same order and by the same process under normal conditions. Thus, for well constructed conditions, **BAE** is very reliable to avoid false positives. It should be known that depending on the system and its purpose, as an example, some functions usually never called for an average

user could be used often by some professionals.

Regarding False Negatives, the **GoC** had to be adjusted to improve the detection. Let's take the **CVE-2021-22555**. The initial graph of conditions was the following one :

GoC A

```
PRIMARY 0 SATISFIES __sys_socket CALLED == 1 DEPENDS []
SECONDARY 1 SATISFIES __sys_socketpair CALLED == 4 DEPENDS [0]
SECONDARY 2 SATISFIES __x64_sys_msgget CALLED == 4096 DEPENDS [1]
SECONDARY 3 SATISFIES do_msgsnd CALLED == 8192 DEPENDS [2]
SECONDARY 4 SATISFIES do_msgrcv CALLED == 4096 DEPENDS [3]
TRIGGER 5 SATISFIES do_msgrcv PID == 0 DEPENDS [4]
```

This graph reproduces the flow of the exploit behaviour. However, only the condition 0 was triggered and the exploit was never detected. The explanation is quite simple and is due to the tasks delegation made to UL (§ 4.4.2). The phenomenon was also explained in (§ 4.6). Due to the short lapse of time between condition 0 and 1, condition 1 was executed before UL updated condition 0 as verified. In order to bypass this issue, a solution is to declare these two events as parallels and not depending one from another. Thus, the detection is made on both at the same time. The two first conditions become PRIMARY and only condition 2 depends on both. This is represented in the new **GoC**:

GoC B

```
PRIMARY 0 SATISFIES __sys_socket CALLED == 1 DEPENDS []
PRIMARY 1 SATISFIES __sys_socketpair CALLED == 4 DEPENDS []
SECONDARY 2 SATISFIES __x64_sys_msgget CALLED == 4096 DEPENDS [0,1]
SECONDARY 3 SATISFIES do_msgsnd CALLED == 8192 DEPENDS [2]
SECONDARY 4 SATISFIES do_msgrcv CALLED == 4096 DEPENDS [3]
TRIGGER 5 SATISFIES do_msgrcv PID == 0 DEPENDS [4]
```

The logic is then modified as follow:

- An exploit detected by **GoC A** is also detected by **GoC B**
- An exploit undetected by **GoC A** can be detected by **GoC B**

To summarise, $D(\text{GoCA}) \subset D(\text{GoCB})$ with $D(G)$ the function representing the ensemble of processes detected by the Graph of Conditions G .

The expected behaviour is then an increase of FP rate. However, with specific enough conditions such as in the previous examples, the rate of FP can still be kept near a null probability.

To conclude over the detection efficiency, **BAE** and the **GoC** offer a very accurate exploit detection for well constructed conditions. We'll explain the process of generating a **GoC** in § 5.3. Due to trades off between accuracy and system performances, the **GoC** has to be put to test in order to obtain the best behaviour to monitor.

5.2 Performance Overhead

Besides from logical efficiency rely the performances of the program. As **BAE** needs to run aside all other processes and hooks each required kfunction, it is major to determine how the system is slowed down. Moreover, **BAE** must be challenged over its scalability by measuring features such as time, storage and slowness over different and increasing number of functions to monitor.

5.2.1 System Performance Impact

Since the detector is lying inside the kernel, it is important that it doesn't slow down consistently the system. The impact of the detector has been evaluated over two different variables : the *number* of hooked functions and their call *frequency*. Performances are evaluated by computing the required time to execute 1'000'000 of operations over the system using **stress-ng**. With T_A and T_B the times to execute the operations respectively just before and after the measure, T_D with the detector. The slowness is computed by

$$\text{Slowness} = \frac{T_D - T_{ref}}{T_{ref}} \quad T_{ref} = \frac{T_A + T_B}{2} \quad (5.1)$$

and the error incertitude by

$$\text{Error} = \frac{1}{2} \text{abs}\left(\frac{T_D - T_A}{T_A} - \frac{T_D - T_B}{T_B}\right) \quad (5.2)$$

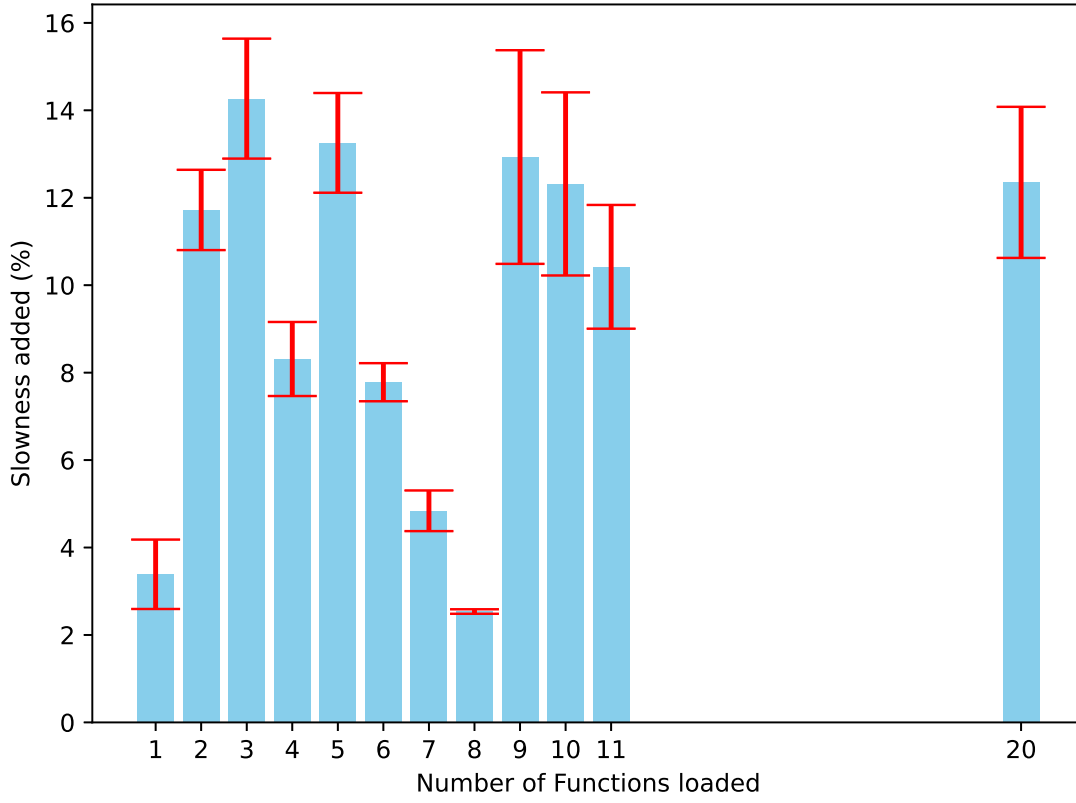


Figure 5.1: System slow down for increasing number of hooked functions

Figure 5.1 represents how the system is slow downed with an increasing number of functions to detect. Intuitively, the more the detector has to hook kernel functions, the more it reduces system performances. As performances can fluctuate overtime, a measure without the detector operating has been performed between each measure. These reference measures allow us to evaluate the accuracy of detection measures. The system has also been put in a stable state with a minimum of process running to not disturb measures.

What Figure 5.1 demonstrates is that the number of hooked functions isn't correlated with performance overhead. The detection slow down the system in average by 10%. As expected, the detector has a strong impact over performances. However, it is important to notice that this performance overhead is acceptable in real world condition. Moreover, as we'll discuss in § 5.2.2, it allows to add new GoC and so new functions to hook by keeping the same performances.

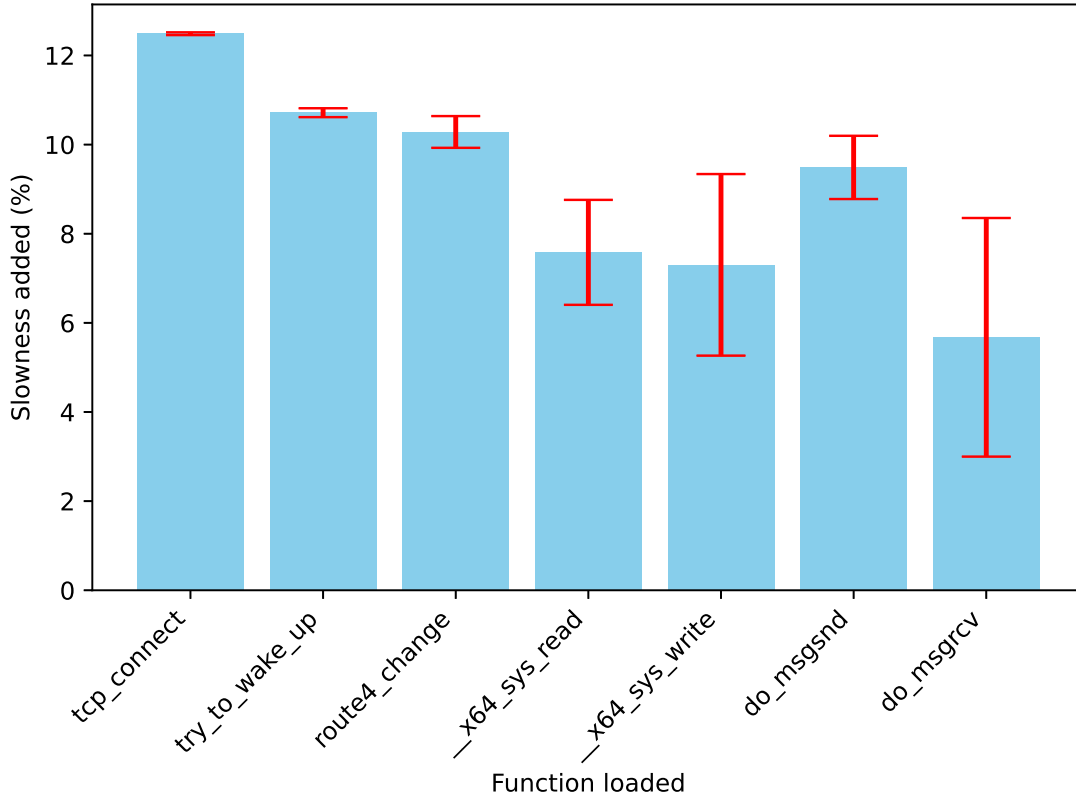


Figure 5.2: System slow down for different functions

Another variable to evaluate is the call's frequency to one hook. Again, intuitively, a function hooked 1000 times a minute should slow down the system far more than one hooked once every hour. However, what Figure 5.2 brings to light is that one more time, both are not correlated. The functions used for this experiment vary a lot, from *try_to_wake_up* called thousand of time every minutes, *tcp_connect* called once a minute or even *route4_change* never called in 5min. Again, the performance overhead remains around 10% whatever is the hooked function. This result demonstrates that the performances of the detector remain the same with functions whose frequency calls vary a lot.

5.2.2 Scalability

As the number of exploits to monitor isn't predetermined, scalability is one of the main challenges to ensure new behaviours can be added at will to the detection

list. Scalability must be offered over different sides

- **Time** before the buffer of monitored processes is fully filled
- **Storage** required to monitor more processes or more detailed behaviours
- **Performances** decrease by adding new behaviour to monitor

Time Due to compilation constraints, as explained in § 4.6, the size of the buffer storing monitored processes needs to be known at compilation time. However, the number of processes to monitor is theoretically infinite since new processes can be created at every moment. Thus, when the buffer is full and a new process arrives, one process already monitored has to be thrown. This operation can be performed with enough confidence about the maliciousness of this process after a certain time monitoring it. Indeed, a process monitored for 1 second is still more suspicious than one monitored for one hour when they have reached the same state in the **GoC**. It is then important to compute the time taken to fill the buffer so that we can have enough confidence to remove the oldest processes.

In order to evaluate the buffer size impact over the time to fill it, we monitored only one function, `try_to_wake_up`, called extremely often by the system. Figure 5.3 represents the time taken to fill it with a \log_2 scale for the X axis. The filling time looks exponential over the log scale what is confirmed by Figure 5.4. This figure scaled the Y axis with \log_2 as well and trends line are represented to evaluate the relation between buffer size and time to fill (TTF). One can observe that the growth slows down after $2^{10} = 1024$ slots available or $2^9 = 512\text{sec} \approx 8\text{min}$. Before understanding this change, let's precise the relation without log scales.

$$\log_2(y) = A \times \log_2(x) + B \quad (5.3)$$

$$y = 2^B \times x^A \quad (5.4)$$

That we can write as

$$TTF = f_{call} \times Slots^{1+C} \quad (5.5)$$

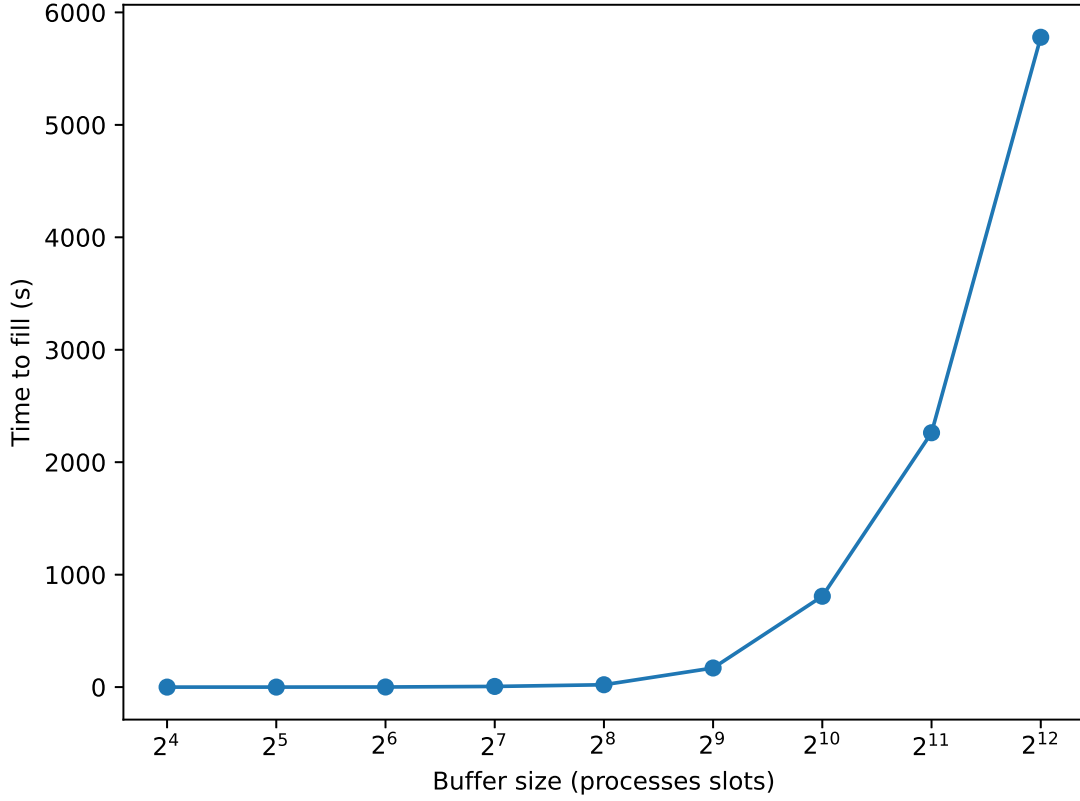


Figure 5.3: Time to fill different buffer size with function `try_to_wake_up`

Going from Equation 5.3 to Equation 5.4 is trivial using the exponential properties, and Equation 5.5 uses the current notations. The f_{call} is a constant coefficient that can be interpreted as the frequency for a given function to be called (by different processes). The C coefficient is interesting. The intuition would lead to think that C should be 0 as by multiplying the size by two, we can consider the buffer as two equals smaller buffers, and as if we realise the experience two times one after the other. The TTF is then twice the TTF for the smaller buffers. However, many processes are already running before the detector, and those processes can be caught after the detector starts. Thus the first slots will be filled by previous processes. As new processes may take time to come up, when the buffer size is too small, it is already filled by previous processes. The first phase can be interpreted as the one treating previous processes, where by increasing the size only the frequency of the function call for current processes matter, and not the process apparition, leading to a faster growth. This is in accordance with the number of processes running on the

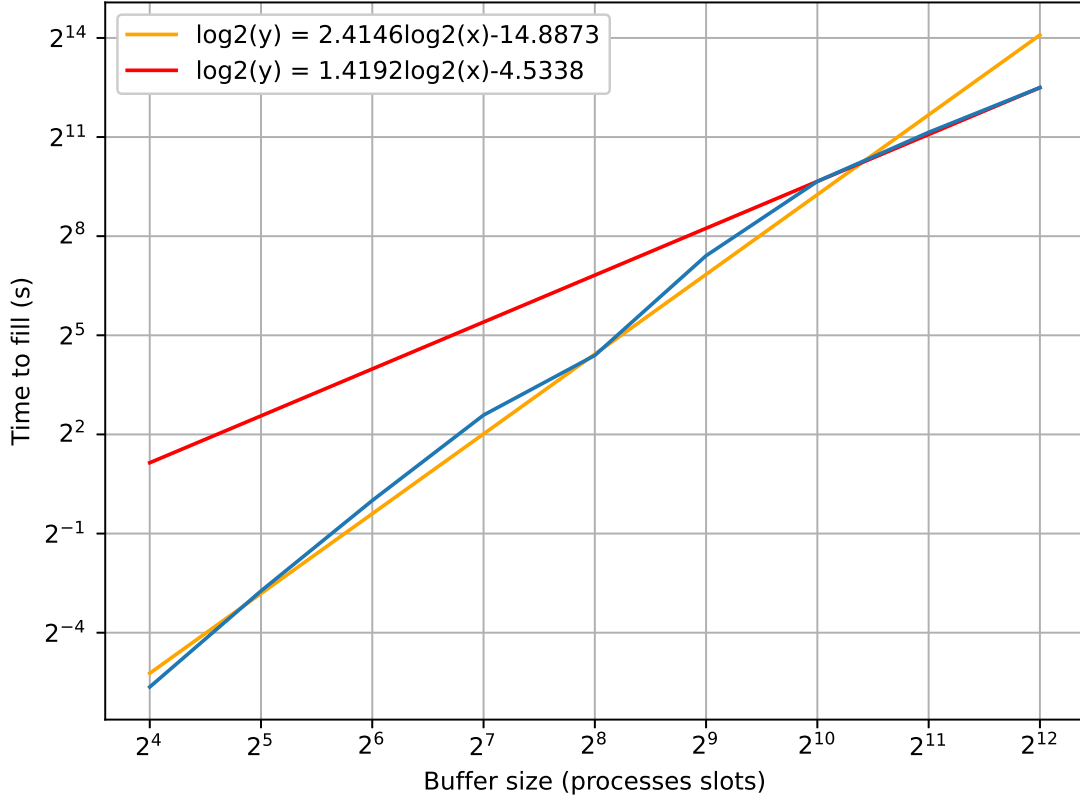


Figure 5.4: Trend line after scaling Figure 5.3 on both axes with \log_2

system before calling `BAE`. The command `ps aux | wc -l` returned 348 processes and the trend change occurred around $2^9 = 512$.

Nevertheless, these results have been obtained from one function. Figure 5.5 represents one more time the TTF for different buffer size but this time comparing three different functions. It can be noticed that for a given size, the time depends of the monitored function. Moreover some functions are almost never called in normal usage such as `do_msgsnd` which didn't fill a size 4 buffer after 20 minutes.

Overall these results demonstrate that for a given kfunction to monitor, the TTF earned is firstly quadratic before being linear. However, as explained previously, we only require the TTF being over a confidence time, in order to recycle slots safely. For functions with an very high call frequency such as `try_to_wake_up` or `__x64_sys_write`, we can obtain a TTF over 1 minute with only 512 slots. Thus, the time scalability relies over the storage capability.

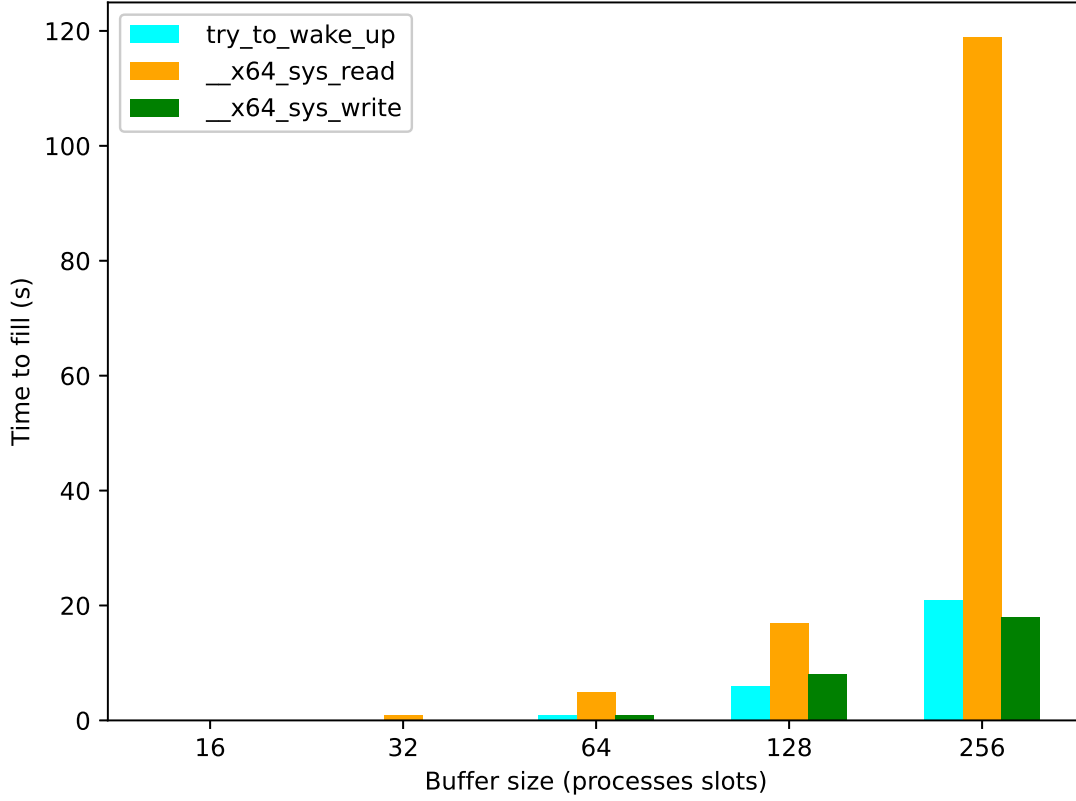


Figure 5.5: Time to fill using different functions

Storage The buffer size is:

$$\text{sizeof}(\text{Buffer}) = \lceil \log_2(|\text{States}|) \rceil \times |\text{Conditions}| \times \text{Slots}$$

with $|\text{States}|$ the number of possible states for one node in the **GoC** (here 3), $|\text{Condition}|$ the number of loaded conditions and Slots the number of slots available. If we define the slots as $N \times 1024$ and assume the State requires 1 byte, which is more than enough, the required storage is about $(|\text{Conditions}| \times N)$ KB. The storage is then $O(C \times N)$ with C the number of conditions. This is linear in the number of conditions and offered slots, what is highly scalable on modern system, as it shouldn't require more than few MegaBytes for the most complex behaviour to monitor.

Performance Figure 5.1 and Figure 5.2 demonstrated that the performances overhead don't depend over the number of loaded functions neither the hooked function. Thus the loss is a constant and about a slow down by 10%. This result is

obviously scalable and the user can be ensured that even by adding more behaviours to monitor, the system's performance will remain the same.

To conclude over scalability, **BAE** is scalable regarding performances, time before freeing a process, and behaviours to monitor. The only limitation is about the storage required in the Kernel but the current requirements are far less than what is offered by the current systems [36].

5.3 Deployment Time

Last but not least, the deployment time is a key point in developing tools aiming to fast patch vulnerabilities. As the actual deployment time is hardly measurable, the steps required for deployment will be detailed and given as an indicative time.

Installing the detector is relatively easy and only require the Rust Tool-Chain and the relevant crates installed. This step could take about 10 minutes and is not of more interest.

Identifying exploit behaviour is the very important step that must be reproduced each time a new exploit is discovered. Depending on the complexity of the exploit this step can take more or less time. If the vulnerability relies in a simple function call, this one can be directly intercepted and inspected. Nonetheless, if more steps are required to exploit the vulnerability, the following method could be followed :

1. Inspecting source code and call of interests
2. Running the exploit in a controlled environment with **strace**
3. Get the system calls from the trace and mount the **ftrace** tool
4. Get the relevant kernel functions associated to the **strace** calls by comparing the name with the **available_filter_functions**
5. Depending on the vulnerability and the exploit source code, forge the condition over the function

This analysis takes a few hours, still depending on the exploit complexity. Then the behaviours must be transcribed to **BFFL**. A good point to notice is that when described, the **GoC** can be easily shared and loaded by many users. Thus the analysis part could only be performed and reviewed by few users and then shared easily to final users.

Updating the detector is instant, having the **GoC** file ready. The file must be loaded and the detector restarted which takes a few seconds.

Overall, the real deployment time depends on the complexity of the new discovered exploit. However the analysis shouldn't take days and one can assume that a new exploitable vulnerability can be patched in a few days.

Compared to **PET** [38], **BAE** detection accuracy is higher with a similar performance overhead. However, monitoring a new behaviour requires more analysis than **PET** which only requires a sanitizer report. **BAE** is then more interesting when the behaviour has already been written down.

Chapter 6

Related Work

In this chapter, I present a brief state-of-the-art of current and past researches about topics related to **BAE** .

6.1 Kernel Patching

Because of the kernel’s sensitivity and the delay in providing official kernel updates [41], kernel patching has become a hot topic and is widely researched. Many studies focus on providing automatic patch generation, which is particularly useful for multi-kernel implementations such as in the Android ecosystem [39, 10]. The goal is to provide a patch as soon as possible; the hot patching technique offers quick remediation to vulnerabilities by replacing some of the kernel functions [22] or even generating a new kernel automatically [34]. To prevent future exploitation of a known vulnerability, most current works intercept syscalls [25, 38] and then analyse them. Additionally, as presented in [38], by performing automatic analysis of reports, probes are placed around the kernel to intercept attacks and kill the malicious process as in [35]. One feature actively researched is also to restore the kernel state after an attack has been detected [25, 34, 38].

6.2 eBPF Security

eBPF has also been extensively studied since it allows kernel interaction. Even if the eBPF is almost entirely read-only [15], some attacks achieve to bypass this restriction [21]. While eBPF can be used for enhancing kernel protection, as demonstrated in [38, 34, 25, 35], it can also be a point of access for exploitation

or escaping virtualisation [21]. Indeed, because eBPF programs run as part of the kernel, some have demonstrated in [20, 19] that eBPF could be used to build a rootkit, compromising the entire system. As explained in [14], this vulnerability is mainly due to recent functions added to eBPF features that allow writing in userland.

eBPF’s versatility in monitoring and securing the kernel is highlighted by tools like **Sysdig**, which utilises eBPF for system call and event monitoring to detect malicious activities. Some research as [23] illustrates how eBPF can be leveraged for high-performance monitoring, crucial for detecting and mitigating vulnerabilities efficiently.

6.3 Kernel Exploitation

While significant work is done to enhance kernel protection, some researchers actively seek to do the opposite and prove its exploitability. [20, 19] introduce rootkits inside the kernel, while [40] demonstrates that the kernel heap is not as secure as experts thought. New exploitation techniques have also been discovered, such as in [28], where kernel privileges can be abused by using credential shifting inside some kernel objects.

Furthermore, researches such as [16, 2] focus on protecting the kernel against control flow attacks by implementing control flow integrity mechanisms. This work is crucial for understanding how to safeguard against sophisticated exploits that target the kernel’s control flow.

6.4 Runtime Enforcement

Runtime enforcement of security policies is a critical aspect of modern kernel security. To deal with these issues, research focuses over methodologies and frameworks that utilise runtime enforcement to detect, prevent, and mitigate vulnerabilities in real time, and sometimes using an eBPF-based approach [38, 35].

Tetragon [35], an open-source project, demonstrates the use of eBPF for security observability and runtime enforcement. By monitoring system calls, process activities, and network events, Tetragon can detect suspicious activities and enforce security

CHAPTER 6. RELATED WORK

policies dynamically. It can terminate, suspend, or modify processes based on predefined security rules, demonstrating the practical application of eBPF for real-time security enforcement. This work is extremely close to my research directions.

Similarly, **KARMA**, presented in [10] uses kernel tracepoints to monitor system events and enforce security policies at runtime. **KARMA** captures a large range of kernel activities, allowing a detailed analysis and immediate response to security threats. Its design allows minimal performance overhead while maintaining strong monitoring capabilities.

System call filtering, implemented through mechanisms like **SECCOMP** [9], restricts the system calls that a process can make based on defined policies. **SECCOMP** enforces security policies at runtime by limiting the process capabilities, thereby reducing the attack surface and preventing exploitation of vulnerabilities through unapproved system calls.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis, I presented **BAE**, a new way to prevent vulnerability by recognising known exploitation behaviours. The behaviour has been described as a directed acyclic graph called *Graph of Conditions* basically representing a part of the control flow graph of the exploit. Each condition is checked using Kernel Probes from the eBPF technology. The separation between kernel and user spaces induced to share and transfer some operations with common and concurrent resources. Thus, some trade-offs are required between the detection efficiency and the behaviour accuracy.

The experiments demonstrated that **BAE** is accurate with an almost null rate of false positives. Whatever the amount and complexity of the monitored behaviours, the detection brings a relatively small performance overhead of about 10%, which is close to current research works [38, 21]. The entire detection is highly scalable regarding nowadays constraints. Finally, thanks to an easy and intuitive way to load conditions with **BFFL**, a language fitting only essential requirements, adding new behaviour to monitor is impressively easy after having analysed the exploit.

However, there is still room for many improvements and **BAE** acts as a solid basis for future research and development in this direction. The technology used is very recent, and their evolution should lead to future improvements.

7.2 Future Research Directions

For different reasons, some points have not been approached in this project and are left for further research. Here are different guidelines to follow in order to

improve the detection and pursue the behaviour detection research.

7.2.1 Acting after detecting

For now, **BAE** only detects when a vulnerability is triggered. The measures to take when an exploit has been detected haven't been described because out of the scope of this research and relatively factual. Indeed, the measure to take requires some meditation and confidence about the real dangerousness of the detected process. Because the detection is made on a given behaviour, if the decision was to kill the processes, a false positive would be killed constantly. This situation could lead to system or software malfunctions.

eBPF hooks are pieces of code inserted at function execution. When the code is executed, it returns to the real function. Thus, eBPF can't directly prevent the future function execution from happening. However, as explained by the Tetragon team [35], there exist two different ways to perform the enforcement. One is to change the actual return value of the function, and the second is to send a signal as **SIGKILL** to the detected process. Enforcement is limited due to the read-only state of the eBPF virtual machine inside the kernel.

7.2.2 The signature problem

In this master thesis we left the condition's forging to the users/analysts. This problem is a well-known problem [39, 10] and is part of active research. Detecting a malicious call only from some parameters isn't as intuitive as it is. In this research, we focused on detecting already known behaviours. However, my first idea was to detect deviant behaviours that-is-to-say behaviours that are very uncommon and that shouldn't occur in normal usage. This detection is far more powerful as it could even prevent 0-days vulnerability. However, since the deviant behaviours could involve any function, it implies probing each of them for a complete analysis. Moreover, the detection analysis would generate new detection etc...

7.2.3 Determinism

For now, the detection is deterministic and a process has to follow predetermined rules to be detected. Some randomness could be added so that modifications in the

exploit behaviour to bypass **BAE** would still be detected. Machine learning could help in this approach and is an interesting direction to follow in order to improve the detection.

7.2.4 Hooking optimization

As explained in § 7.2.2 to ensure a complete protection, all functions should be hooked and every call analysed knowing potential vulnerabilities. Moreover, for now each function involved in a condition is hooked and treated if the condition is in the **WAITING** state. However, because each **GoC** shows the current conditions to validate, functions to hook can be determined. Thus, a dynamic hooking by enabling or disabling probes could be performed to optimise hooking.

7.2.5 Language improvements

BFFL lacks some features to represent some conditions. A case in point is how the arguments are described and particularly structures. Because structures are first written in C, the conversion in another language, such as Rust, presents some hurdles, as stated in § 4.6. Thus, accessing specific fields requires knowing the data location in the structure when the conversion doesn't exist. Tools such as **aya-tool** already offer a way to do the conversion for some of them, but it could be completed by some dynamic fetching of source code [6][11].

7.2.6 Malicious usage

Since behaviours can be shared between users through **BFFL**, one has to be aware that malicious use could happen. Because only behaviours are described, nothing prevents someone from sharing the behaviour of a normal and maybe necessary process. The execution of a nonmalicious process could then be prevented leading to important weaknesses. A firewall, a web service or any software could be stopped by loading their behaviour. **BAE** is a great companion, powerful, that should be handled cautiously.

Bibliography

- [1] A. N. (theflow@), “Cve-2021-22555: Turning x00x00 into 10000\$”, 2021. [Online]. Available: <https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html>.
- [2] J. P. et al., “Taking kernel hardening to the next level”, [Online]. Available: <https://i.blackhat.com/Asia-22/Friday-Materials/AS-22-Park-Taking-Kernel-Hardening-to-the-Next-Level.pdf>.
- [3] M. J. H. et al., “Representation and analysis of software”,.
- [4] J. Arnold and M. F. Kaashoek, “Ksplice: Automatic rebootless kernel updates”, in *4th ACM European conference on Computer systems*, Nuremberg, Germany: ACM, 2009. [Online]. Available: <https://www.usenix.org/conference/lisa-02/timing-application-security-patches-optimal-uptime>.
- [5] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack, “Timing the application of security patches for optimal uptime”, in *16th Systems Administration Conference (LISA 02)*, Philadelphia, PA: USENIX Association, Nov. 2002. [Online]. Available: <https://www.usenix.org/conference/lisa-02/timing-application-security-patches-optimal-uptime>.
- [6] “Bootlin elixir cross referencer”, [Online]. Available: <https://elixir.bootlin.com>.
- [7] D. Bouman, “How the tables have turned: An analysis of two new linux vulnerabilities in *nf_tables*”, 2022. [Online]. Available: <https://blog.dbouman.nl/2022/04/02/How-The-Tables-Have-Turned-CVE-2022-1015-1016>.
- [8] “Business of apps”, [Online]. Available: <https://www.businessofapps.com>.
- [9] C. Canella, M. Werner, D. Gruss, and M. Schwarz, “Automating seccomp filter generation for linux applications”, in *Proceedings of the 2021 on Cloud Computing Security Workshop*, ser. CCSW ’21, Virtual Event, Republic of Korea:

BIBLIOGRAPHY

- Association for Computing Machinery, 2021, pp. 139–151, ISBN: 9781450386531. [Online]. Available: <https://doi.org/10.1145/3474123.3486762>.
- [10] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, “Adaptive android kernel live patching”, in *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 1253–1270, ISBN: 978-1-931971-40-9. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/chen>.
- [11] L. T. bibinitperiod cie, “Linux kernel source tree”, [Online]. Available: <https://github.com/torvalds/linux>.
- [12] T. A. Contributors, “Aya developer book”, [Online]. Available: <https://aya-rs.dev/book>.
- [13] “Cve mitre”, [Online]. Available: <https://cve.mitre.org>.
- [14] J. Dileo, “Evil ebpf: Practical abuses of in-kernel bytecode runtime”, DEF CON, <https://doi.org/10.5446/48383>, 2019. [Online]. Available: <https://doi.org/10.5446/48383>.
- [15] “Ebpf official website”, [Online]. Available: <https://ebpf.io>.
- [16] X. Ge, N. Talele, M. Payer, and T. Jaeger, “Fine-grained control-flow integrity for kernel software”, in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2016, pp. 179–194.
- [17] Google, “Android git repository”, [Online]. Available: <https://android.googlesource.com>.
- [18] “Google project zero”, [Online]. Available: <https://googleprojectzero.blogspot.com/>.
- [19] S. B. Guillaume Fournier Sylvain Afchain, “Ebpf, i thought we were friends !” DEF CON 29, 2021. [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>.

BIBLIOGRAPHY

- [20] S. B. Guillaume Fournier Sylvain Afchain, “With friends like ebpf, who needs enemies ?” Black Hat USA 2021, 2021. [Online]. Available: <http://i.blackhat.com/USA21/Wednesday-Handouts/us-21-With-Friends-Like-EBPF-Who-Needs-Enemies.pdf>.
- [21] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li, “Cross container attacks: The bewildered eBPF on clouds”, in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 5971–5988, ISBN: 978-1-939133-37-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity23/presentation/he>.
- [22] Y. He, Z. Zou, K. Sun, Z. Liu, K. Xu, Q. Wang, C. Shen, Z. Wang, and Q. Li, “RapidPatch: Firmware hotpatching for Real-Time embedded devices”, in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 2225–2242, ISBN: 978-1-939133-31-1. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/he-yi>.
- [23] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller, “The express data path: Fast programmable packet processing in the operating system kernel”, in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’18, Heraklion, Greece: Association for Computing Machinery, 2018, pp. 54–66, ISBN: 9781450360807. [Online]. Available: <https://doi.org/10.1145/3281411.3281443>.
- [24] T. IA and V. TEN, “An overview of common programming security vulnerabilities and possible solutions”,,
- [25] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, “Programmable system call security with ebpf”, *ArXiv*, vol. abs/2302.10366, 2023. [Online]. Available: <https://api.semanticscholar.org/CorpusID:257050432>.
- [26] M. Kellermann, “The dirty pipe vulnerability”, 2022. [Online]. Available: <https://dirtypipe.cm4all.com/>.

BIBLIOGRAPHY

- [27] Z. Lin, Y. Chen, X. Xing, and K. Li, “Your trash kernel bug, my precious 0-day”, BlackHat Europe, 2021. [Online]. Available: https://zplin.me/talks/BHEU21_trash_kernel_bug.pdf.
- [28] Z. Lin, Y. Wu, and X. Xing, “Dirtycred: Escalating privilege in linux kernel”, in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1963–1976, ISBN: 9781450394505. [Online]. Available: <https://doi.org/10.1145/3548606.3560585>.
- [29] “National institute of standards and technology”, [Online]. Available: <https://nvd.nist.gov/>.
- [30] “Rust community’s crate registry”, [Online]. Available: <https://crates.io>.
- [31] “Rust for linux”, [Online]. Available: <https://rust-for-linux.com>.
- [32] “Rust official website”, [Online]. Available: <https://www.rust-lang.org>.
- [33] “Statcounter global stats”, <https://gs.statcounter.com/os-market-share>.
- [34] S. M. S. Talebi, Z. Yao, A. A. Sani, Z. Qian, and D. Austin, “Undo workarounds for kernel bugs”, in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 2381–2398, ISBN: 978-1-939133-24-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/talebi>.
- [35] “Tetragon”, [Online]. Available: <https://tetragon.io>.
- [36] “The linux kernel”, [Online]. Available: <https://www.kernel.org>.
- [37] A. G. bibinitperiod G. Thomas, “Linux kernel modules in rust”, 2019. [Online]. Available: <https://ldpreload.com/p/kernel-modules-in-rust-lssna2019.pdf>.
- [38] Z. Wang, Y. Chen, and Q. Zeng, “PET: Prevent discovered errors from being triggered in the linux kernel”, in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA: USENIX Association, Aug. 2023, pp. 4193–4210, ISBN: 978-1-939133-37-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity23/presentation/wang-zicheng>.

BIBLIOGRAPHY

- [39] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, “Automatic hot patch generation for android kernels”, in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 2397–2414, ISBN: 978-1-939133-17-5. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/xu>.
- [40] K. Zeng, Y. Chen, H. Cho, X. Xing, A. Doupé, Y. Shoshitaishvili, and T. Bao, “Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability”, in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 71–88, ISBN: 978-1-939133-31-1. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/zeng>.
- [41] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, “An investigation of the android kernel patch ecosystem”, in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 3649–3666, ISBN: 978-1-939133-24-3. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-zheng>.