UNIVERSIDADE
BEIRA INTERIOR

Preventing Data Exfiltration
eBPF-based approach

Carlos Francisco Caramelo Pinto

Internship report
Computer Science and Engineering
(2nd degree cycle)

Supervisor: André Passos
Counselor: Prof. Doctor Manuela Pereira
Co-Counselor Prof. Doctor Simão Melo de Sousa

June 2024

Contents

List of Figures

Acronymms

eBPF	Extended Berkeley Packet Filter
BPF	Berkeley Packet Filter
JIT	Just In Time
CO-RE	Compile Once - Run Everywhere
BTF	BPF Type Format
LSM	Linux Security Module
TOCTOU	Time of Check to Time of Use

Chapter 1

Introduction

1.1 Introduction

Data exfiltration is the theft or unauthorized transfer of data from a device or network. This can occur in several ways, and the target data can vary from user credentials, intellectual property and company secrets. The most common definition of data exfiltration is the unauthorized removal or movement of any data from a device.

The need for the prevention of data exfiltration is particularly pertinent in corporate settings, as there is the potential for the massive loss of revenue by companies which fall victim to these attacks.

Extended Berkeley Packet Filter (eBPF) presented itself as the right tool to avoid these kind of attacks, since it is a technology that allows for the programming of the Linux kernel for networking, observability, tracing and security. It can effectively monitor the entirety of the system, as it runs inside the kernel, and detect data exfiltration attempts.

In this thesis the research conducted during the internship at Scalabit will be presented, which aimed to prove that eBPF would be a feasible solution to the problem of data exfiltration. Throughout the document, the experiments conducted will be presented and discussed. The purpose of this internship was that of exploring both eBPF from a monitoring and security perspective.

1.2 Motivation

The main objective of this internship was the study and development of an eBPF application that would prevent data exfiltration across a fleet of machines. This opportunity arose when at Scalabit there was a need to study the feasibility of this solution. The problem in case is a machine that is provided to customer with root access. What was proposed in this thesis was to see if it was possible to use eBPF to avoid root access to certain files and folders. Knowing from the start that this is not the perfect solution, neither the perfect way to do this, there was also the knowledge that the real world is more complicated than it looks like. Sometimes it's not simple to update a machine that is making money at customers and change the software or even add security features that impact their way of working. Doing this with eBBF would prove incredibly handy, being that eBPF is particularly useful, since the classical approach of static configurations, although robust when configured statically during the provisioning of the machine, become brittle when deployed across a fleet of machines and in the face of changing policies.

This application if feasible, would serve the purpose of restricting services and capabilities inside the kernel, based on a per user or service approach, effectively armoring the system from unwanted accesses, either to services or files, thus preventing data exfiltration from said machine.

It presents itself as quite a pertinent problem, both from and academic and business position, since the current approaches to tackle this problem are quite limited.

1.3 Scalabit

Scalabit's objective is helping software companies to achieve outstanding levels when it comes to the software delivery process. Scalabit focuses on continuous integration/continuous delivery in order to deliver more frequently but also with less impact on the field. The lemma "fail fast recover faster" makes a lot of sense for the company. Given the current panorama worldwide, the security aspect of software is making a lot of impact. Multiple systems trust their daily operations in software. You can think on medical devices, nuclear power plants, your smartwatch, etc. We are surrounded by software. Security of these systems is a critical point. In fact the EU is already looking at the security aspect of software. That's why, in 15th of September 2022, the EU decided to release the Cyber Resilience Act. This is an EU regulation proposed for improving cybersecurity and cyberresilience through common cybersecurity standards for products with digital elements in the EU, such as required incident reports and automatic security updates. The challenge here is huge since there is software in the field older than 10 years. As you can imagine this software is difficult to update and also difficult to adapt. That's why this project appeared. When, at Scalabit, we started looking at eBPF we though about the option of using the kernel to restrict certain calls and with this increase the level of security of certain systems.

1.4 The Problem

A common data exfiltration definition is the theft or unauthorized removal or movement of any data from a device, typically involving a cyber criminal stealing data from personal or corporate devices. The definition more relevant to this dissertation is that of data exportation and extrusion, posing serious problems for organizations. Failing to control information security could mean the loss of intellectual property or cause reputational and financial damage to an organization.

The problem being faced was that of preventing data exfiltration inside machines being provisioned by a company to clients, which were then in turn trying to access intelectual property, such as software, that were to be private to the company that made said machines. The problem with the prevention of data exfiltration has seen various approaches to being solved, such as static configurations, which work best when configured statically during the provisioning of a machine, becoming brittle to operate when deployed across a fleet of machines and in the face of changing policies.

One of the many methods that are studied when preventing data exfiltration is that of data hiding, which is the one concerning this investigation. Data hiding can be used as a method for the prevention of data exfiltration by concealing sensitive information. One example of data hiding is steganography, which involves the concealment of sensitive information within another message or file to avoid detection [1].

1.5 Proposed Solution

The proposed solution to study aimed to tackle the challenges mentioned in the previous section by providing a method of easily loading configurations with the intetion of preventing data exfiltration, leakage or theft. To achieve this solution, it was chosen to leverage the ease of use and flexibility of eBPF as the pillar of the application itself. It presented many benefits, including the fact that it would not imply the change and consequent rewrite of application code, but it would work as a simple patch on the kernel.

The main objective of this internship was the study of the feasability of an eBPF application preventing data exfiltration, such that one application could be extended across various machines. As stated above, the classical approach of static configuratins, although effective when configured statically during the provisioning of the machines, become hard to operate when deployed across a fleet of said machines.

The implementation of this application serves the purpose of restricting certain system calls inside the kernel, on a per user approach, armoring certain files from unwanted accesses, thus preventing data exfiltration from said files.

As of the study and implementation of this solution the only tool on the market that was similar to it was Tetragon [2], thus making this solution both an academic and business opportunity, seeing that in some cases the application developed goes above and beyond Tetragon, but it is not without its drawbacks.

1.6 Document Organization

This document is organized as follows:

1. **Core Concepts and State of the Art** - This chapter aims to provide an overview of the Core Concepts necessary to understand the inner workings of eBPF, as well as the Linux kernel.

2. **Case Study** - In this chapter the problem to be solved will be described, as well as the proposed solution to solve it. In it there will also be an elaboration on the preliminary experiments conducted as to ascertain that eBPF was the right path in order to try to solve the problem being faced.

3. **Tool Development** - In this chapter the development of the final tool aimed to prevent data exfiltration will be present, in it the application choices will be presented, as well as an overall view of the application itself.

4. **Use Case** - This chapter serves the purpose of providing an overall view on the expected use cases of the application, as well as the tests ran on said application, as to ensure that the behaviour would be as expected.

5. **Critical Analysis** - In this chapter, a critical analysis is provided, concerning the limitations of the application and the overall work done throughout this internship.
6. **Conclusion** - This chapter serves the purpose of providing an overall conclusion to the document and the topics approached during this work.

Chapter 2

Core Concepts and State of the Art

2.1 Introduction

In order to fully understand the inner workings of the eBPF tool and its potential for security and monitoring purposes in data exfiltration scenarios it is important to have a solid understanding of the core concepts of the Linux Kernel and eBPF itself, as well as the current state of the art in security and monitoring tools. This chapter aims to provide an overview of both of these key topics.

2.2 Linux Kernel

The Linux Kernel is the core component of the Linux operating system [3]. It acts as a ”bridge” between the hardware and the software layers, it communicates between the two, managing resources as efficiently as possible.

The jobs of the Linux kernel are:

1. **Process management** The kernel determines which processes can use the CPU, and for how long.

2. **Memory Management** The kernel keeps track of how much memory is used to store what, and where.

3. **Device drivers** The kernel acts as a mediator between the hardware and the processes.

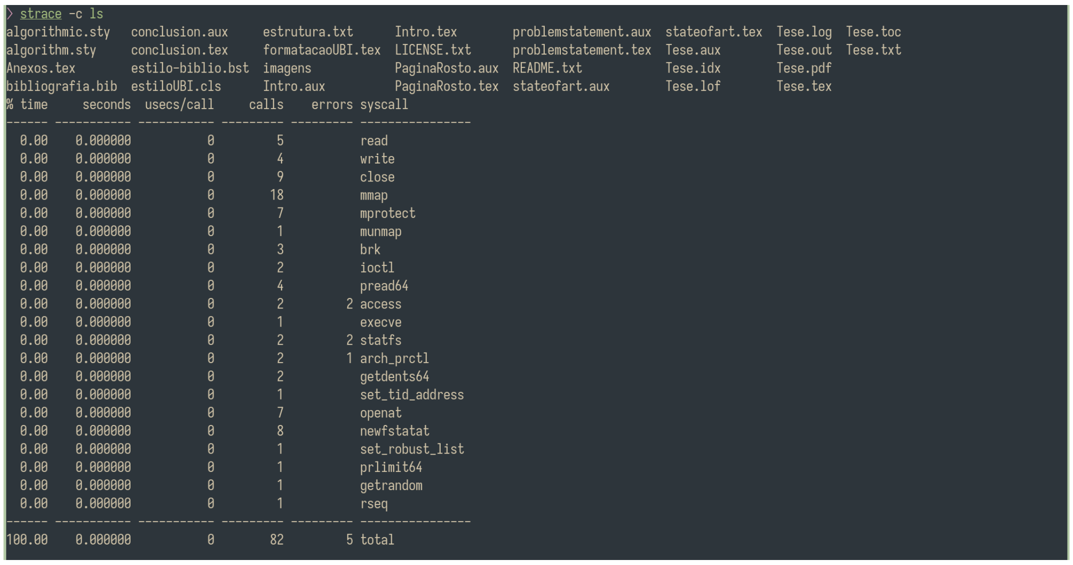
4. **System calls and Security** The kernel receives requests for service from processes.

The kernel is quite large, with 30 million lines of code, meaning that performing any change is a challenging task, as making a change to any codebase requires some familiarity with it. Additionally, if the change made locally was to be made part of an official Linux release, it would have to be accepted by the community as a change that would benefit Linux as a whole, taking into account that Linux is a general purpose operating system. Assuming that the change was indeed seen as a net benefit, there would still be a relevant waiting period until it would be accessible to everyone’s machine, since most users don’t use the Linux kernel directly, but Linux distributions, which use specific versions of the kernel, some of which might be several years old.

2.2.1 System calls

Applications run in an unprivileged layer called *user space*, which can't access hardware directly. These applications make requests using the system call interface, requesting the kernel to act on its behalf. Since we're more used to the high level abstraction that modern programming languages, we can see an example of just how many system calls are made using the `strace` utility. For example, using the `ls` command involves 82 system calls.

Figure 2.1: `strace` of `ls` command



```
> strace -c ls
algorithmic.sty   conclusion.aux     estrutura.txt      Intro.tex        problemstatement.aux  stateofart.tex  Tese.log  Tese.toc
algorithm.sty     conclusion.tex     formatacaoUBI.tex  LICENSE.txt      problemstatement.tex  Tese.aux        Tese.out  Tese.txt
Anexos.tex        estilo-biblio.bst  imagens            PaginaRosto.aux  README.txt            Tese.idx        Tese.pdf
bibliografia.bib  estiloUBI.cls      Intro.aux          PaginaRosto.tex  stateofart.aux        Tese.lof        Tese.tex
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         5           read
  0.00    0.000000           0         4           write
  0.00    0.000000           0         9           close
  0.00    0.000000           0        18           mmap
  0.00    0.000000           0         7           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           ioctl
  0.00    0.000000           0         4           pread64
  0.00    0.000000           0         2         2 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         2 statfs
  0.00    0.000000           0         2         1 arch_prctl
  0.00    0.000000           0         2           getdents64
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         7           openat
  0.00    0.000000           0         8           newfstatat
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
  0.00    0.000000           0         1           getrandom
  0.00    0.000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0        82         5 total
```

Because applications are so heavily reliant on the kernel, it means we can learn a lot by observing its interactions with the kernel. With eBPF we can add instrumentation into the kernel to get these insights, and potentially prevent system calls from being executed. Assuming we have a user who runs the `ls` command in a certain directory, eBPF tooling is able to intercept one of the several system calls involved in that command and prevent said command from being run. This makes it quite useful for security purposes, effectively modifying the kernel, running custom code whenever that system call is invoked.

2.3 eBPF

Extended Berkeley Packet Filter (eBPF) [4] originated as an extension of the original Berkeley Packet Filter (BPF), which was designed for packet filtering within the kernel Unix-like operating systems. Although evolving from it, the technology has evolved and is now considered a standalone term. eBPF is a revolutionary technology that allows for developers to write custom code to be loaded into the kernel dynamically, extending the capabilities of the kernel without requiring additional modules or modifications to the kernel source code.

In recent years, eBPF has undergone significant advancements, particularly in the realm of security and monitoring. Its programmability has naturally led to the development of tools and frameworks that leverage its capabilities. eBPF provides deep insights into system activities, allowing for the gain of real-time visibility into the inner workings of the kernel.

From a monitoring perspective, eBPF has revolutionized the way that system monitoring is developed, as it allows for the efficient and secure data collection through the Linux kernel, incurring in less overhead than traditional tools, enabling the monitoring of application processes and system resource usage through system calls, eliminating the need to use monitoring agents in user space.

Furthermore, in recent years, the eBPF ecosystem has expanded with the development of user-friendly tools and libraries, making it more accessible. As a result, researchers, developers and companies alike can harness the power of eBPF to address specific security concerns. This continuous evolution of eBPF marks it as one of the most exciting recent technologies in the Linux ecosystem.

2.3.1 eBPF Code

Writing eBPF code involves a combination of high level languages and a just-in-time (JIT) compiler, allowing for the creation of efficient and flexible programs that run within the Linux kernel.

User space programs are used to interact with eBPF from user space, usually written in high level languages and are responsible for loading eBPF programs into the kernel. This involves compiling the user-written code into eBPF bytecode, verifying its safety and then loading into the kernel using the `bpf()` system call or any abstraction provided by the language used. User space programs manage the entirety of eBPF programs, attaching or detaching them from hooks dynamically. User space programs can also respond to events triggered by eBPF kernel programs, such as the analysis of network packets, allowing syscalls to be executed, etc. This allows for the development of reactive applications that respond to real-time events in the kernel, making it useful for the detection of data exfiltration as is our objective.

Kernel space programs are attached to hooks, which are predefined locations in the kernel where eBPF programs can be run, allowing them to intercept and manipulate data at various points in the kernel's execution, as these hooks can be associated with various events, such as system calls, function calls, etc. Kernel side programs are subject to a verification process before being loaded, ensuring the safety of the code, and are then ran inside the eBPF virtual machine within the kernel.

The communication between user space and kernel space is achieved through the use of pre-defined data structures, known as eBPF maps. These maps are used to pass general information between the user and kernel. Several types of maps are supported, such as array maps, per-CPU maps, hash maps and ring buffers, each being ideal for different use cases. The access of the information held in an eBPF map, from user space, is made through two system calls `bpf_map_lookup_elem()` and `bpf_map_update_elem()`, which provide read and write operations on eBPF maps, respectively.

Since eBPF code is kernel specific, a potential limitation is the portability and compatibility across different kernel versions. This limitation is tackled by the CO-RE, (Compile Once - Run Everywhere), approach, aiming to enhance the deployment and maintainability of eBPF programs across different versions. This approach consist of a few key elements:

1. *BTF* BPF Type Format serves the purpose of expressing the layout of data structures and function signatures, detecting any differences at compilation time and runtime.
2. *Kernel headers* The Linux kernel makes use of header files, which describe the data structures it uses, which can change between versions of the kernel. `bpftool` allows for the generating of a header file containing all the data structure information that might be needed.
3. *Compiler support* The Clang compiler is used to compile eBPF programs with the `-g` flag.
4. *Library support for data structure relocations* `libbpf` allows for the compensation of any differences between the data structures present at compile time and the ones present at runtime.
5. *BPF skeleton* A skeleton file, containing helper functions to manage the lifecycle of an eBPF program can be generated using `bpftool`. This is an optional element in this approach.

Through this approach an eBPF program can run on different kernel versions, massively improving the portability of eBPF.

The verification process of an eBPF program consists in the static analysis of it. It operates in two steps: first, it performs a directed acyclic graph (DAG) check to disallow loops and validate the control flow graph. Then, it simulates the execution of every instruction and observes the state change of registers and the stack to track the range of possible values in each register and stack slot. This is done to ensure that the program is safe to run and does not violate certain safety rules, such as writing outside designated memory regions or performing restricted arithmetic operations on pointers. The eBPF verifier is not a security tool inspecting what the programs are doing, but rather a safety tool checking that programs are safe to run. [5].

2.3.2 eBPF Programs and Attachment Types

eBPF supports several program types and types. Only some are presented, since they total around 30 program types, and more than 40 attachment types.

2.3.2.1 Kfuncs

Kfuncs are functions in the Linux kernel that are exposed for use by eBPF programs. Unlike normal eBPF helpers, *kfuncs* do not have a stable interface and can change from one kernel release to another. They provide an API that eBPF applications can call, but it is less stable than the set of BPF helpers.[6]

2.3.2.2 Kprobes and Kretprobes

Kprobes enable you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address specifying a handler routine to be invoked when the breakpoint is hit. [7] The main difference between *kprobes* and *kretprobes* is that the latter provides the ability to execute a handler after a specific instruction, which can prove useful to capture the return value of a function.

2.3.2.3 Fentry/Fexit

Fentry/fexit are two types of probes that allow for the collection of information, modification of parameters or simply the observation of return values at specific stages of kernel functions execution. *Fentry* is triggered at the entry point of said functions, and *fexit* is triggered at the exit point.

2.3.2.4 Tracepoints

Tracepoints are static markers in the kernel's code that allow developers to attach code in a running kernel. These tracepoints are not exclusive to eBPF and, unlike *kprobes*, are stable between kernel releases.

2.3.2.5 User Space Attachments

User space attachments in eBPF refer to the capability of attaching eBPF programs to user space events, such as user space functions or system calls, for monitoring and tracing purposes. This functionality allows for the dynamic instrumentation of user space applications without requiring modifications to the application's source code or the need for recompilation.

There are considerations and challenges when instrumenting user space code, but despite these challenges, several useful tools leverage eBPF to instrument user space applications. Examples include tracing decrypted versions of encrypted information in the SSL library and continuous profiling of applications.

2.3.2.6 LSM

BPF LSM (Linux Security Module) is a framework in the Linux kernel that allows the dynamic loading of custom eBPF programs to implement security inspection features. It leverages the LSM framework, enabling the attachment of eBPF programs to predefined hook points, allowing for the writing of granular security policies. BPF LSM programs can be attached to LSM hooks using the `bpf` system call `BPF_RAW_TRACEPOINT_OPEN` operation. The return value of these programs influences the kernel's behaviour, allowing for the prevention of an operation to be completed. [8] [9]

2.3.2.7 BPF Attachment Types

The attachment type plays a crucial role in defining the location within the system where a program can be attached. While the attachment type can be automatically determined in some programs based on the hook to which they are attached, others can be attached to multiple points in the kernel, hence an attachment type needs to be explicitly specified. This specification influences the ability to access helper functions and restricts interaction with specific context information.

In instances where a particular attachment type is required, the kernel function `bpf_-prog_load_check_attach` serves the purpose of validating its appropriateness for specific program types.

To ascertain the valid attachment types applicable to different programs, one can refer to the comprehensive documentation provided by `libbpf`. This documentation not only outlines recognized section names for each program but also elucidates on the permissible attachment types. The correct understanding and specification of attachment types become imperative when working with eBPF programs, as this knowledge forms the foundation for ensuring the proper integration and behavior of the programs within the kernel.

2.4 State of the Art

Gathering all the information presented above, eBPF presents new capabilities in the security and observability realm. There already several projects leveraging eBPF to develop security tools. We will talk about some of these tools, as well as some classical approaches to the prevention of data exfiltration.

2.4.1 Tetragon

Tetragon [2] is a relatively new tool in the industry, leveraging eBPF's capabilities to detect and react to security relevant events, such as system calls, process execution events and I/O activity in general.

Tetragon itself is a runtime security enforcement and observability tool, applying policy and filtering directly in eBPF in the kernel. From an observability use case, as we've seen with eBPF, applying filters directly in the kernel reduces observation overhead. This tool provides rich filters in eBPF, allowing users to specify important and relevant events in their specific context.

Tetragon can also hook into any function in the Linux kernel and filter based on the information this provides. Tetragon also allows for hooking to points where data structures cannot be manipulated by user space applications inside the kernel, effectively solving several common observability and security use cases, such as system calls tracing, where data is incorrectly read, altered or missing due to user/kernel boundary errors.

Due to the topics discussed above we can conclude that Tetragon is a solid tool for the prevention of data exfiltration, allowing for users to set policies and enforcing those same policies inside the kernel space. One concrete example of this is disallowing certain user/processes to access critical files inside the system, which will then be enforced at the kernel level, preventing and presenting a trace of data exfiltration attempts. Tetragon's approach is mainly to be used in a Kubernetes environment, being that this project defines a custom Kubernetes resource typecalled *TracingPolicy*.

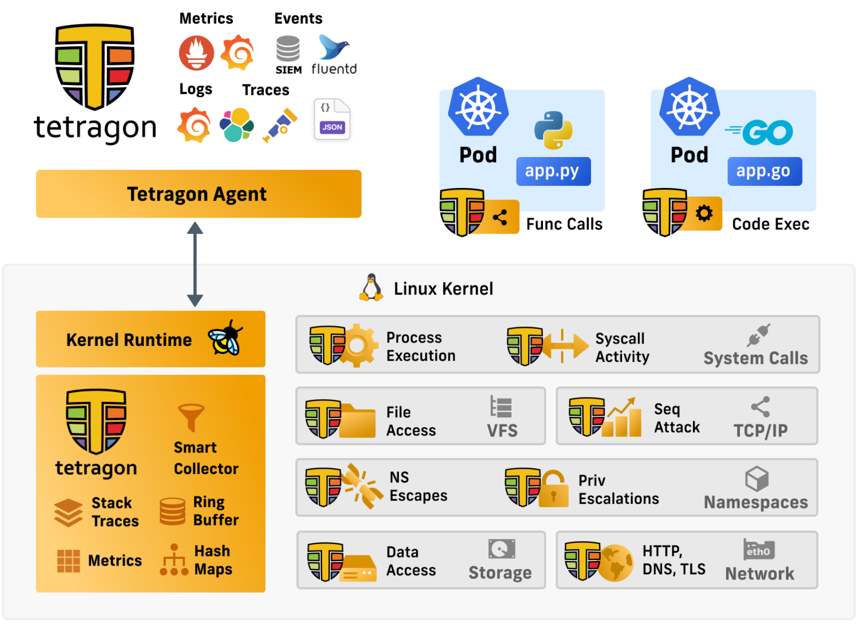


Figure 2.2: Tetragon Overview

2.4.2 *Seccomp*

Seccomp [10] [11], short for secure computing mode, is a Linux kernel feature that provides a simple and efficiet mechanism for restricting the system calls a process can make. The intention of this tool was to allow users to run untrusted code without any possibility of that code executing malicious tasks.

One iteration of this tool, known as `seccomp-bpf`, allows for the filtering of system calls using a configurable BPF program. It provides a means for a process to specify a filter for incoming system calls allowing for expressive filtering of system calls. The filter has access both to the system call and its arguments, taking one of the following actions:

1. Allow the system call.
2. Return an error code to the application that made the system call.
3. Kill the thread.
4. Notify a user space application.

However, since some of the arguments passed to system calls are pointers and the BPF code used in this tool is not able to dereference these pointers, it has limited flexibility, using only value arguments in its decision-making process. It also has to be applied to a process when it starts, making it impossible to modify the profile in real time.

2.4.3 Syscall-Tracking Security Tools

Numerous tools fall within this category; however, this methodology is susceptible to Time of Check to Time of Use (TOCTOU) issues. When an eBPF program is triggered at the entry point of a system call, it gains access to the arguments passed to that call. Typically, in kernel space, if these arguments are pointers, the kernel copies the data into its own data structures before processing it. During this process, there exists a window of vulnerability wherein an attacker can manipulate the data after inspection but before the kernel completes its copy.

This poses a challenge because the data acted upon may differ from the originally captured data. One potential solution is to attach to both the entry and exit points in system calls. While this approach can provide accurate records of security-relevant events, it cannot prevent an action from occurring, as the system call has already concluded by the time the exit check is conducted.

Therefore, an effective strategy involves attaching the program to an event that occurs after the parameters have been copied into kernel memory, with the understanding that the data is handled differently in system call-specific codes. The prevailing and considered most appropriate approach is leveraging the Linux Security Module.

2.4.4 BPF LSM

The Linux Security Module interface contains certain hookpoints that occur just before the kernel acts on a kernel data structure, triggering a function that can make a decision on whether or not to allow the action to be taken. Originally, this interface allowed for security tools to be implemented as kernel modules, being extended by BPF LSM allowing for the attachment of eBPF programs to the same hook points.

The fact that the hooks are defined in points where the kernel is about to act on certain arguments, and not the entry point of a specific system call, effectively solves the Time of Check to Time of Use issue, and as such, it is the most correct approach for security tools that aim to provide not only observability but also prevention of security relevant events.

BPF LSM was added in kernel version 5.7, and as such, it is not yet widely available accross Linux distributions.

2.5 Conclusion

This chapter provides a comprehensive overview of the core concepts of eBPF and the state of the art in eBPF based security tooling. With so many different tools available, it is essential to have a clear and concise way to compare and evaluate them. As the eBPF environment continues to see growth and interest both in the industry and in academia, one can expect for these tools to be continually improving on previous iterations. By gaining a deeper understanding of these tools and the underlying concepts, reasearchers and developers can work towards developing new tools that are more secure, identifying and solving potential problems with the approaches now in use.

Chapter 3

Case Study

3.1 Introduction

The Case Study chapter is a crucial component of the thesis as it outlines the problem, the proposed solution to it and an experiment to further sustain that solution. In this chapter, we aim to present a clear and concise description of the problem we are solving, why it is important, and why the proposed solution is a valid one. The chapter will also include a detailed description of the experiment that was conducted to validate the solution, as well as the findings from it. In this chapter, the preliminary experiment will be presented, which served as the basis to conduct the study of eBPF to tackle the problem of data exfiltration being faced by Scalabit. This experiment leverages eBPF's capabilities to access certain kernel functions and hook onto system calls.

In conclusion, the solution proposes to overcome the challenges mentioned in the previous section by providing a method of dynamically implement security policies, enforced at kernel level, to a fleet of machines, without the hassle needed with classical methods.

3.2 Preliminary Experiments

The study of eBPF and its security capabilities was made to ensure that this solution would be feasible. The experiment consisted in the development of an eBPF program that would react to the `chdir` system call, so that the contents of the folder, passed as a parameter of this kernel function, and the user responsible for said call would be flushed to `stdout`. To achieve this the CORE approach was used, to guarantee compatibility between kernel versions. The different steps of this experiment are documented in the following subsections.

3.2.1 eBPF implementation of `ls`

This experiment proved itself relevant as a means to be more familiarized with eBPF's typical struccture as well as its capabilities. To achieve that goal, it is important to understand the different files involved in such applications. Normally, there will be a file for the kernel side of the application, as well as one for the user side. We shall call these `eBPF_ls.bpf.c` and `eBPF_ls.c` for the kernel side and the user side, respectively.

3.2.1.1 `eBPF_ls.bpf.c`

`eBPF_ls.bpf.c` will contain the code meant to be run at kernel level, meaning that it can define a hookpoint to a certain system call. This is achieved by

```
SEC("ksyscall/chdir")
```

which defines that we are only interested in running this program when a system call to `chdir` is made.

Hence, when such system call is made this program is triggered. The program then acts accordingly.

```
int BPF_KPROBE_SYSCALL(hello, const char *name){
    struct data_t data = {};
    struct user_msg_t *p;
```

In the above code snippet we make use of the `BPF_KPROBE_SYSCALL` macro defined in `libbpf` that allows to acces the argument of a system call by name. The only argument the `chdir` accepts is the name of the destin directory. We can the write the data accessed to a perf buffer so that is accessible in the user side of the application.

```
data.pid = bpf_get_current_pid_tgid() >> 32;
data.uid = bpf_get_current_uid_gid() & 0xFFFFFFFF;

bpf_probe_read_user_str(&data.path, sizeof(data.path), name);
```

The first two lines are used so that we can access the process id and the user id that made the system call. The `bpf_probe_read_user_str` is an helper function that copies data from an unsafe address to the perf buffer output, in this case containing the argument to the `chdir` system call, the process id and user id from where the call was made. After the data has been written, we can then share it with user space code using the following line:

```
bpf_perf_event_output(ctx, &output, BPF_F_CURRENT_CPU, &data, sizeof(data));
```

This helper function writes the data into the perf buffer output, making it accessible from user space. Finally, there is another macro defining a license string, being a crucial requirement for eBPF programs.

```
char LICENSE[] SEC("license") = "Dual BSD/GPL";
```

This sums up the steps need in the kernel side of the application, creating a `kprobe` to the `chdir` system call, and sending the directory that made the call to the user side of the application.

3.2.1.2 `eBPF_ls.c`

`eBPF_ls.c` will contain the code to be run in user space, reacting to the data sent from the kernel side of the application. It will start by loading the BPF skeleton, containing handy functions to manage the lifecycle of the program, such as loading it into the kernel.

```
int main() {
  struct eBPF_ls_bpf *skel;
  int err;
  struct perf_buffer *pb = NULL;

  libbpf_set_print(libbpf_print_fn);
```

```
 7
 8   skel = eBPF_ls_bpf__open_and_load();
 9   if (!skel) {
10     printf("Failed to open BPF object\n");
11     return 1;
12   }
13
14   err = eBPF_ls_bpf__attach(skel);
15   if (err) {
16     fprintf(stderr, "Failed to attach BPF skeleton: %d\n", err);
17     eBPF_ls_bpf__destroy(skel);
18     return 1;
19   }
```

Line 8 in the code snippet above creates a `skel` structure representing all the maps and programs defined in the ELF bytes, loading them into the kernel. Line 14 attaches the program to the appropriate event, returning and error if it was unsuccessful. We can then create a structure to handle the perf buffer output, as presented below.

```
pb = perf_buffer__new(bpf_map__fd(skel->maps.output), 8, handle_event,
                      lost_event, NULL, NULL);
if (!pb) {
  err = -1;
  fprintf(stderr, "Failed to create ring buffer\n");
  eBPF_ls_bpf__destroy(skel);
  return 1;
}
```

The `handle_event` and `lost_event` are functions that handle the events captured when polling the perf buffer. The perf buffer will then be continuously polled:

```
while (true) {
    err = perf_buffer__poll(pb, 10000000 /* timeout, ms */);
    // Ctrl-C gives -EINTR
    if (err == -EINTR) {
      err = 0;
      break;
    }
    if (err < 0) {
      printf("Error polling perf buffer: %d\n", err);
      break;
    }
  }
```

When there is an event in the perf buffer output, we can expect to find the name of the directory that was passed to the `chdir` system call, as well as the user id that made the system call. As such, the `handle_event` function is triggered, and we can then iterate over the directory to display its files.

```
void handle_event(void *ctx, int cpu, void *data, unsigned int data_sz) {
  struct data_t *m = data;
  char *pad = "{ ";
  if (!strcmp(m->command + strlen(m->command) - 2, "sh")) {
    const char *dir_path = m->path;
    DIR *dir = opendir(dir_path);

    // Check if the directory can be opened
    if (!dir) {
      perror("opendir");
    }

    struct dirent *entry;

    printf("%s: ", getUser(m->uid));
    // Read and print the contents of the directory
    while ((entry = readdir(dir)) != NULL) {
      printf("%s%s", pad, entry->d_name);
      pad = ", ";
    }
    printf("}\n");
    closedir(dir);
    printf("\n\n\n\n");
  }
}
```

We print the user id that made the system call and then print the contents of the directory that said user is changing to.

3.2.1.3 Makefile

The Makefile for this application, being based on the CO-RE approach has several options that are worth digging into. First off, when compiling both the user side and kernel side it is needed to pass the `g` flag to the Clang compiler, so that it includes debug information, that is necessary for BTF. The `O2` optimization flag also needs to be passed, in order for Clang to produce BPF bytecode that can pass the verification process. The target architecture needs to be specified in order to use certain macros defined in `libbpf`. Joining all of these we achieve the following to build BPF code:

```
%.bpf.o: %.bpf.c vmlinux.h
        clang \
            -target bpf \
         -D __TARGET_ARCH_$(ARCH) \
              -Wall \
              -O2 -g -o $@ -c $<
         llvm-strip -g $@
```

We then need to generate BPF skeletons, which contain several useful functions to handle the lifecycle of the program, we use `bpftool` to achieve this, which uses the eBPF object in ELF file format to generate said skeletons.

```
$(USER_SKEL): $(BPF_OBJ)
         bpftool gen skeleton $< > $@
```

We then generate the header file `vmlinux.h`, containing all the data structure information about the kernel that is needed in a BPF program.

```
vmlinux.h:
         bpftool btf dump file /sys/kernel/btf/vmlinux format c > vmlinux.h
```

Lastly, we can build the user space code, using the line:

```
eBPF_ls: eBPF_ls.c
         gcc -Wall -o eBPF_ls eBPF_ls.c -L../libbpf/src -l:libbpf.a -lelf -lz
```

The typical structure for an eBPF application based on the CO-RE consists then of these three files, containing the eBPF program itself, the user side code and the Makefile specific to the application.

3.2.2 Conclusion

With this experiment, the capabilities of eBPF had been proven, from both a security and usability perspective. An application that monitors the `chdir` system call was developed, where we had access to the target directory and the user who made said system call.

3.3 Conclusion

This chapter provides a concise introduction to the problem and the proposed solution. It also discusses the experiments done to uphold the claim that the solution is a feasible one. eBPF-based security, as shown in the preliminary example, offered itself as an ingineous solution to the problem stated.

Chapter 4

Tool Development

4.1 Introduction

The tool development chapter aims to present a detailed description of how eBPF was implemented in the resulting application, the tools used, the rationale behind their selection and the challenges encountered during the implementation of this solution. This chapter focuses on the development of two essential tools to attempt to solve the problem faced by the company. The first tool is the eBPF application itself, which disallows system calls based on values stored inside BPF maps. The second tool is a shared library which replaces the `readdir` system call, effectively hiding the process ID of the program being ran. Together, these tools aim to address the complexities and challenges of data exfiltration prevention.

4.2 eBPF/limitations of the application/tests ran/tool based limitations

The development of this tool made use of two major technologies surrounding eBPF, which were the CO-RE approach, which was previously presented, and BPF-LSM. These choices stemmed from a need to have an application that was capable of being ran on different kernel versions, and the fact that it needed to be stable, using the stable hookpoints provided by BPF-LSM. Although these choices seem quite trivial now, their choice was made taking into account the state of the art in eBPF-based tooling for security.

The main goal of the application to be developed was limiting the access of users to certain sensitive files, thus using data hiding as a method to prevent data exfiltration. One of the main purposes was that the application would be capable of reading a configuration from a yaml file, and restricting users based on said configuration.

The implementation started with the preliminary example provided in the previous chapter, being that the changes made to the code consisted mainly to the eBPF side of the code. The choice was made for the application to limit access to files provided in a yaml file at loading time. This file contains a user ID that can access and alter those files, acting as a maintainer of them. Any other user that tries to access it has the attempt revoked. The hookpoints in the eBPF program that ensure that this happens are in three different system calls, being `path_chmod`, `file_open`, `path_rename`. The `chmod` system call was chosen to make use of Linux's file policies, so that the eBPF application only acts as an additional layer of security on top of this. The `file_open` system call was trivially chosen as well, in order to prevent that unauthorized users would not be able to open the file. Lastly, the `path_rename` system call was found to be useful in order to prevent users from moving or renaming the files, seeing that the files are provided to the yaml file in the string format, containing its full path, thus allowing a user to change a file named *test* to *test2* and effectively gain access to it.

4.2.1 YAML file

The yaml file provided to the application, which is then parsed from the user side part of the eBPF program, with name `config.yaml`, consists of an user ID, and a list of filesdirectories. For example:

```
uid: 1124
directory:
    - /home/user/testfile3
```

In this example the user whose ID is 1124 is the sole maintainer of the file with the path provided in `directory`. Any attempt of another user to make the system calls mentioned previously on the file is denied. This file is then parsed using `libcyaml`, an open source library to parse yaml files into C structs. The struct is:

```
struct uid_struct {
  int uid;
  char **directory;
  unsigned directory_count;
};
```

The uid and directory fields are self-explanatory, but the field directory_count exists so that when parsing the yaml file the library can store how many entries there are in the directories parameter.

The parsing of the file is then as trivial as:

```
err = cyaml_load_file(argv[ARG_PATH_IN], &config, &top_schema,
                      (cyaml_data_t **)&n, NULL);
if (err != CYAML_OK) {
  fprintf(stderr, "ERROR: %s\n", cyaml_strerror(err));
  return EXIT_FAILURE;
}
```

The name of the yaml is passed as an argument to the program itself. The `config` contains the CYAML config, which can be changed between calls to the function. The `top_schema` is the CYAML value schema for the top level mapping, where it is defined that the data to be read can be stored in a struct of type `uid_struct`.

The configuration read from the yaml file could change between runs of the eBPF program and, according to its contents, the application would then disallow the specified system calls on the files present in the yaml file to any user that was not its maintainer.

4.2.2 User side code

The user side of the code in this particular application is only responsible for loading the eBPF program and populating its maps with the correct information. This is achieved with the CO-RE approach and the use of a skeleton which configures almost everything for us.

With the skeleton generated, we simply need to open and load the file into the eBPF virtual machine, which then runs it for us. This is achieved by using `bpftool`, which is mentioned in the previous chapters. Using this tool, opening and loading an eBPF program is as simple as:

```
skel = eBPF_ls_bpf__open_and_load();
if (!skel) {
  printf("Failed to open BPF object\n");
  return 1;
}
```

We can then access the `skel` variable to make changes to the eBPF program, such as loading maps. The `skel` variable is of type `struct eBPF_ls_bpf`, which reflects the name of the program we used, being that this struct is also auto generated and it has its fields:

```
struct eBPF_ls_bpf {
        struct bpf_object_skeleton *skeleton;
        struct bpf_object *obj;
        struct {
                struct bpf_map *heaps_map;
                struct bpf_map *output;
                struct bpf_map *directories;
                struct bpf_map *my_config;
                struct bpf_map *rodata;
        } maps;
        struct {
                struct bpf_program *path_chmod;
                struct bpf_program *file_open;
                struct bpf_program *path_rename;
        } progs;
        struct {
                struct bpf_link *path_chmod;
                struct bpf_link *file_open;
                struct bpf_link *path_rename;
        } links;
};
```

As we can see from above, the struct consists of a skeletong and a bpf object, the maps used by the program, (shown by `bpf_map`), the different bpf programs that a single file may contain, (shown by `bpf_program`), and the hookpoints/links needed by each of the programs, which in this case coincide, since we gave the name of the hookpoints to the programs themselves.

This allows us then to access and populate the directories map from the user side of the code, having parsed the yaml file. It is as simple as:

```
for (i = 0; i < n->directory_count; i++) {
  strncpy(aux, n->directory[i], sizeof(aux));
  bpf_map__update_elem(skel->maps.directories, &aux,
  sizeof(aux), &n->uid, sizeof(n->uid), 0);
}
```

The yaml file is stored in struct `n`, and we update the map by iterating over the entries in said struct and calling the function `bpf_map_update_element`, which stores the `aux` variable, containing the name of the directory, and the `n->uid`, containing the `uid` to which the directory is associated in the `skel->maps.directories`, which points to the directories eBPF map.

After doing this, the maps are frozen, so that no new entries can be added from a call to `bpftool map update`. This is achieved by running the program from a script, which handles this in the elegant manner of simply calling `bpftool map freeze`, which freezes the map from the user side. The `bpftool` call is the made unavailable to the users.

We then attach the skeleton, and the program is running.

4.2.3 Kernel side code

From the kernel side of the eBPF application, the behaviour can be boiled down to attaching to the LSM hookpoints and then allowing the access to the system call or not according to the values contained in the eBPF map. We can abstract this for all three applications, so the example shown is in all three hookpoints.

```
directory_flag = bpf_map_lookup_elem(&directories, &x.path);
if (directory_flag != 0) {
  if (*directory_flag == uid) {
    bpf_printk("user %d has access to file", *directory_flag);
    return 0;
  }
  bpf_printk("Aux not empty %d\n ", uid);
  bpf_printk("Chmod not allowed to %s", file_path);
  return -EPERM;
} else {
  bpf_printk("aux is empty");
  bpf_printk("Chmod allowed to %s %d", file_path, uid);
  return 0;
  }
}
```

Essentialy we have the directory flag showing us whether this is a directory to be monitored or not, and if so we match the uid that made the call to the one contained in the bpf map, if it coincides we allow the system call to go through, returning 0, but if it doesn't we return `EPERM`, which disallows the system call from continuing. This logic is implemented in the three hookpoints in the same way.

One of the challenges, specifically to locate ourselves from where the system call was made was to get the full path from a `struct path`, as it is defined in the kernel. This proved particularly hard since eBPF defines a stack limit of 512 bytes, forcing us to make use of a per-CPU array to store the full path. We can imagine that when making a system call, from an example directory `/home/test/this`, the `struct path` only contains `this`, forcing us to go through each of its parent directories to effectively remount the full path. That is done by using the function below, which will be explained next.

```
statfunc long get_path_str_from_path(u_char **path_str,
const struct path *path, struct buffer *out_buf) {

  long ret;
  struct dentry *dentry, *dentry_parent, *dentry_mnt;
  struct vfsmount *vfsmnt;
  struct mount *mnt, *mnt_parent;
  const u_char *name;
  size_t name_len;

  dentry = BPF_CORE_READ(path, dentry);
  vfsmnt = BPF_CORE_READ(path, mnt);
  mnt = container_of(vfsmnt, struct mount, mnt);
  mnt_parent = BPF_CORE_READ(mnt, mnt_parent);

  size_t buf_off = HALF_PERCPU_ARRAY_SIZE;

#pragma unroll
  for (int i = 0; i < MAX_PATH_COMPONENTS; i++) {

    dentry_mnt = BPF_CORE_READ(vfsmnt, mnt_root);
    dentry_parent = BPF_CORE_READ(dentry, d_parent);

    if (dentry == dentry_mnt || dentry == dentry_parent) {
      if (dentry != dentry_mnt) {
        // We reached root, but not mount root - escaped?
        break;
      }
      if (mnt != mnt_parent) {
        // We reached root, but not global root
        // continue with mount point path
        dentry = BPF_CORE_READ(mnt, mnt_mountpoint);
        mnt_parent = BPF_CORE_READ(mnt, mnt_parent);
        vfsmnt = __builtin_preserve_access_index(&mnt->mnt);
        continue;
      }
      // Global root - path fully parsed
      break;
    }

    // Add this dentry name to path
    name_len = LIMIT_PATH_SIZE(BPF_CORE_READ(dentry, d_name.len));
    name = BPF_CORE_READ(dentry, d_name.name);

    name_len = name_len + 1; // add slash
    // Is string buffer big enough for dentry name?
    if (name_len > buf_off) {
      break;
    }
    // satisfy verifier
    volatile size_t new_buff_offset = buf_off - name_len;
    ret = bpf_probe_read_kernel_str(&
    // satisfy verifier
    (out_buf->data[LIMIT_HALF_PERCPU_ARRAY_SIZE(new_buff_offset)]),
      name_len, name);
    if (ret < 0) {
      return ret;
    }

    if (ret > 1) {
      // remove null byte termination with slash sign
      buf_off -= 1;
      // satisfy verifier
      buf_off = LIMIT_HALF_PERCPU_ARRAY_SIZE(buf_off);
      out_buf->data[buf_off] = '/';
      buf_off -= ret - 1;
      // satisfy verifier
      buf_off = LIMIT_HALF_PERCPU_ARRAY_SIZE(buf_off);
    } else {
      // If sz is 0 or 1 we have an error (path can't be null nor an empty
      // string)
      break;
    }
    dentry = dentry_parent;
  }

  // Is string buffer big enough for slash?
  if (buf_off != 0) {
    // Add leading slash
    buf_off -= 1;
    buf_off = LIMIT_HALF_PERCPU_ARRAY_SIZE(buf_off); // satisfy verifier
    out_buf->data[buf_off] = '/';
  }

  // Null terminate the path string
  out_buf->data[HALF_PERCPU_ARRAY_SIZE - 1] = 0;
  *path_str = &out_buf->data[buf_off];
  return HALF_PERCPU_ARRAY_SIZE - buf_off - 1;
}
```

This function starts by defining the variables needed to get the full path. It the reads the `dentry` and `vfsmount` inside the `path struct`. This representes the directory entry of the path, and the mount point inside the Linux file system. With this information we can access the root of the mount point, stored in `mnt` and the global root of the filesystem itself, stored in `mnt_parent`. With all these variables, we can then iterate over the different directory's parents, until we reach the global root of the filesystem, while adding the different directory entries to the buffer we go through in order to parse the full path. We make use of a per CPU array in order to bypass the stack limit of eBPF programs which is currently at 512 bytes of data. We then return the number of characters written, and the resulting full path is stored in `path_str`, which is accessible from the caller function since we pass it as a double pointer.

4.2.4 readdir

One of the issues faced during the development of this tool was that of the process ID being available to all users, making the program effectively useless since any user could call the `sigkill` system call on said process ID. To face this a shared library was implemented so as to intercept and modify the behaviour of the `readdir` system call, using the `LD_PRELOAD` mechanism. The program filters out directory entries for processes with a specified name, which is hardcoded into the program itself. This works since every call to list processes inside a Linux system makes use of the `proc/` directory to list the processes being ran. By doing this we effectively hide the process ID from the user. The code is shown below, working by loading the original function `readdir` using `dlsym`, and then entering a loop where it checks if the directory being accessed by the `readdir` is `proc` and if the entry corresponds with the given process name, if so, then it continues the loop, skiping over that entry, otherwise it returns the directory entry.

```
#define DECLARE_READDIR(dirent, readdir)
  static struct dirent *(*original_##readdir)(DIR *) = NULL;

  struct dirent *readdir(DIR *dirp) {
    if (original_##readdir == NULL) {
      original_##readdir = dlsym(RTLD_NEXT, #readdir);
      if (original_##readdir == NULL) {
        fprintf(stderr, "Error in dlsym: %s\n", dlerror());
      }
    }

    struct dirent *dir;
    while (1) {
      dir = original_##readdir(dirp);
      if (dir) {
        char dir_name[256];
        char process_name[256];
        if (get_dir_name(dirp, dir_name, sizeof(dir_name)) &&
            strcmp(dir_name, "/proc") == 0 &&
            get_process_name(dir->d_name, process_name) &&
            strcmp(process_name, process_to_filter) == 0) {
          continue;
        }
      }
      break;
    }
    return dir;
  }
```

4.3 Conclusion

This chapter introduced the two pivotal tools which were developed during the course of this internship. The eBPF application serves the purpose of creating maintainers of certain directories, in order to prevent any user other than the maintainer of changing or accessing any content present in said directory. The second part of the implementation, which is the shared library, acts as a replacement for the `readdir` system call, in order to prevent any calls to `ps` of showing the process ID of the eBPF app itself, thus preventing users from killing said process. Together, these tools form the solution found during this internship to prevent data exfiltration, with an eBPF approach.

Chapter 5

Use Case

5.1 Introduction

In this chapter, we build on the previous chapter's conclusion, presenting and demonstrating the pratical utility of the tools developed. The objective is to provide a comprehensive overview of the expected behaviour of the application.

To showcase this, the tests ran on said application will be presented, which serve as a case study to showcase the capabilities of the tool developed.

5.2 Expected Use Case

This application was developed with a specific flow in mind. Namely, it being ran since the start of the boot of a machine. With that in mind, it relies on a script to correctly initiate it, which will freeze the bpf maps after they are populated, and properly hide the process id from users.

```
#!/bin/bash

make

gcc -Wall -fPIC -shared -o src/processhiding/libprocesshider.so
  src/processhiding/processhider.c -ldl
sudo mv src/processhiding/libprocesshider.so /usr/local/lib/
echo "/usr/local/lib/libprocesshider.so" >> /etc/ld.so.preload

nohup sudo ./src/eBPF_app src/config.yaml >/dev/null 2>&1 &
disown $!
sudo bpftool map freeze name directories
```

This script starts by compiling the eBPF program, with the `make` command, after which it will compile and load the shared library in order to hide the PID of the program that is afterwards ran. It then runs the program, ignoring the hang up signal, which allows the command to continue running even after the user has logged out. It also redirects all output from the program to `dev/null`, which discards all data written to it, essentially silencing all output from the command. The program is then removed from the shell's job table, which means it will keep running even if the shell is closed. Afterwards, the bpf map pertaining to the directories that are tracked is fozen, which means that it will no longer be changeable from user space.

In summary, this script ensures that the program is loaded correctly, being that the only thing that is left to the user is to properly write the `config.yaml` file.

5.3 Tests

The tests ran on this application were generated using bash scripts, seeing as the application itself should change the behaviour of the system calls that it hooks onto. To achieve this the [12]bats-core testing framework was used, providing a simple way to verify that the program behaves as expected.

5.3.1 BATS-core

The bats-core framework is a `BASH AUTOMATED TESTING SYSTEM`, providing a simple way to verify the behaviour of UNIX programs. It is built on top of `Bash` and leverages its features to define test cases. Each test case is a bash function with a description. The return of a command with the prefix `run` will determine the success or failure of said function. Its return value is stored in `$status`. We can then just check if the command ran successfully or not, which is made possible since a command exits with the status code of 0 if it succeeds. The test cases will then be something along the lines of:

```
@test "Example test" {
  run command
  [ "$status" -eq 0 ]
}
```

5.3.2 User and file generation

To ensure that the program behaves as expected, a test suite was ararnged, where users and files are generted randomly and tested to confirm that their access is revoked or allowed, according to the information contained in the config.yaml file that provides the configuration to the eBPF program. To generate a random user, we use the function `create_user`, which is shown below.

```
create_user () {
    username=$1
    if id "$username" &>/dev/null; then
        echo "User $username already exists."
        return 0
    fi
    sudo useradd -m $username
    sudo passwd -d $username > /dev/null
}
```

This function relies on one parameter being passed to it, which is the username, that is generated using the `dev/urandom`. It is exemplified below.

```
username=$(cat /dev/urandom | tr -dc 'a-zA-Z' | fold -w 10 | head -n 1)
```

If the username already exists then nothing is done and the function returns, otherwise, it will call `useradd`, to create the user, after which, it will delete the password field, making the user passwordless, to prevent the need to input passwords after each command that is run in the test suite.

After the user is generated we need to randomly give it access or not, so that the tests can be ran. This is achieved using the function below:

```
give_permission (){
    flag=$1
    username=$2
    if [ $flag = 1 ]; then
        echo "uid: $(id -u $username)
directory:
    - $(pwd)/testfile" > config.yaml
else
        echo "uid: 1003
directory:
    - $(pwd)/testfile" > config.yaml
    fi
}
```

This function takes two arguments, which are the `flag`, which indicates whether or not the user should have permission to access the file, and the `username`, indicating which user we are refering to. If the `flag` is **1**, then the user should have access to the file, and as such, its ID is associated with the directory we are currently in. Otherwise, a non-conflicting `uid` is associated with the directory, in order to test if another user cannot access said file.

These show the setup steps in order for the test suite to be created.

5.3.3 Testing

To generate the tests, we abstract them into a function that is compliant with the bats framework, as such:

```
@test "Random chmod" {
    username=$(cat /dev/urandom | tr -dc 'a-zA-Z' | fold -w 10 | head -n 1)
    permission=$((RANDOM % 2))

    create_user $username
    give_permission $permission $username

    touch testfile

    sudo chown $(id -u $username) testfile
    ./eBPF_ls config.yaml &

    pid=$!

    run su -c "chmod 777 testfile" -s /bin/bash $username

    kill $pid

    sudo rm testfile

    sudo deluser --remove-home $username

    if [ $permission = 1 ];
    then
        [ "$status" -eq 0 ]
    else
        [ "$status" -ne 0 ]
    fi
}
```

This function starts by creating a user and randomly giving it permission to a testfile, as shown in the previous subsection. The ownership of said is given to the user, so we can prove that the eBPF program loaded can circumvent even Linux's permissions. After that, the program is run in the background, calling the config.yaml file in order to ensure that the program is loaded. Its PID is stored in `pid`, as to be able to kill the program afterwards. The test command is a call to `chmod` on the `testfile`, and the exit status is then **0** if the user should have permission to execute that system call on the file, or not **0** otherwise. The cleanup done at the end of this function, both removing the user and the testfile is done in order to prevent populating the `uid` group with random users.

These tests are generated for `chmod` and `cat`, which pertain to the `path_chmod` and `file_open` system calls, respectively.

5.3.4 Test generation

After writing the tests, and being that random uids and permissions are generated, then generating a test suite with a given number of cases is as simple as:

```
for i in {1..1000}
do
    sudo bats test.sh
done
```

This script will generate **1000** test cases, testing both the `chmod` and `cat` functions called on a `testfile`. At the time of writing this test suite was ran several times, and all the tests were successful.

Chapter 6

Critical Analysis

6.1 Introduction

After achieving the proposed solution, the feasibility of it was analysed. Being that the eBPF program developed correctly stopped certain system calls from users who were not allowed access to a certain directory or file, the research shifted its purpose on circumventing the program itself.

6.2 Directories

Firstly, the point in question was the way the paths were passed onto the program itself. By not being able to have a unique file ID that points to a certain file or directory the solution found was that of passing absolute paths into the program. This however, poses a vulnerability, seeing as that relies on the fact that none of the folders or sub folders in that path will change their name. This is impossible to ensure seeing that if we block all the folders from the home directory we will inevitably block the system itself of making changes to anything, being static, which is not the purpose of the solution. This vulnerability was tackled inside the directories where the files are but could not be extended due to the problems mentioned above.

To provide a clearer example of this we can imagine an eBPF program where the file that we want to protect is `file`, and its full path is `/home/user/dir/protect/file`. The eBPF program will ensure that the protect folder itself will not be changed, and all its subfiles as well. However, it would be impossible to do that on the home, user or dir folder. Circumventing the eBPF solution would then be as simple as changing `dir` to `dir2`, making the full path `/home/user/dir2/protect/file`, which would be different from the path in the config.yaml file, making the program not track this file anymore, seeing as it would have no reason to.

6.3 BPF-LSM

To understand how BPF-LSM works on the kernel level we used the `trace-cmd`, as to trace the calls being made inside functions in the kernel. To do this we used the trace-cmd [13].

This command serves as the front-end application of `ftrace`, which in itself is an internal tracer designed to enable the knowledge of the functions called inside the kernel. It can be used for debugging, latency and performance analysis. In this case the capability of function tracing was used enabling the visibility of which functions were called and when, thus leading to a better understanding of the flow of execution of the kernel.

The `ftrace` command will track a specific call and then output the result to a .dat file with the following command: `sudo trace-cmd record -p function-graph -g '**' -F sudo mv test test2`. The `-p function-graph` option tells the command to record the system calls and kernel functions in a function graph format. The `-g` option filters the functions which are to be recorded. We use the `'**'` wildcard to match all functions in this example. The `-F` specifies which command is to be traced, which in this case is `sudo mv test test2`. After this command is ran, a call to `trace-cmd report` will output the report in a human readable format. Trimming the output we can analyze the following kernel function:

```
vfs_open() {
        do_dentry_open() {
                path_get() {
                        mntget();
                }
                try_module_get();
                security_file_open() {
                        hook_file_open() {
                                get_current_fs_domain();
                        }
                        apparmor_file_open();
                        bpf_lsm_file_open() {
                                __rcu_read_lock();
                                __rcu_read_unlock();
                                __rcu_read_lock();
                                migrate_disable();
                                bpf_get_current_uid_gid() {
                                        from_kgid() {
                                        map_id_up();
                                        ............
```

We should focus on the `security_file_open()` function, which will call a number of different functions deciding whether or not to allow the `file_open` system call to be made. Inside this function we can see that there is a `bpf_lsm_file_open()` function, which is populated due to the program that was developed, starting to get the `uid` from the user making the call. It will then call a number of different kernel functions that are needed in order for the program to run. We can then consider the application that was developed as a number of kernel functions being called. The problem with this, from a security perspective is that, relying on kernel functions is not secure by design, seeing as that with the kernel being open source a skilled user could simply execute this command, and seeing the functinos used could alter the kernel itself, by patching it, and comment out any calls to `bpf_lsm` functions, thus rendering the application useless. This would imply a deep knowledge of the tools used, but it is nonetheless a vulnerability identified during the development process of this application. The resolution of this vulnerability would imply the change of the Linux kernel itself, which is unrealistic, seeing as it would go against the Linux philosophy. Therefore, this vulnerability was simply identified and not resolved.

6.4 `sudo su`

Another vulnerability noted during the development of this application was that the users of the machines where the application should be deployed have `sudo` acess. This meant that any user was capable of calling `sudo su` to change the identity to any user. This problem was tackled on an Ubuntu based system by commenting out the line `auth sufficient pam_-rootok.so` in `/etc/pam.d/su`, which disables root's ability to `su` without passwords. This is however a solution that is easily bypassed by readding the line, making it a vulnerability should any user be aware of this. It could not be prevented with eBPF, and thus it presents another vector of attack to allow data exfiltration.

6.5 Conclusion

In this chapter a critical analysis of the work that was developed is presented. As for the work developed throughout this internship, the goal of it was achieved, which was the study of eBPF as a whole, and the feasibility of an eBPF based approach to tackle the data exfiltration problem faced by the company in the machines provided to its clients. During the course of this internship and the development of this application many difficulties were faced, namely the study of the Linux kernel, the study of eBPF and its monitoring and security capabilities, its vulnerabilities, namely the ones mentioned in the previous section, and the adaptation to the overall development best practices concerning eBPF and kernel based security tools.

At the end of this investigation and development process, it should be noted that the conclusion was that eBPF is not a feasible solution to the problem at hand. Due to the nature of the Linux kernel and this tool in particular, the vulnerabilities mentioned in this chapter were intractable. Many of the vulnerabilities noted previously were resolved, by adding more hookpoints to the program and by armoring it, which were explained in the Tool Development chapter. The ones mentioned in this chapter pose the biggest challenge, seeing as they were not dependent on development choices, but rather on the choices made in the development of the tools used.

Chapter 7

Conclusion

Although the application developed during the course of this internship did not ultimately prove to be a viable solution to the problem at hand, the experience and knowledge aquired throughout this project have been invaluable. The study and research conducted on eBPF and kernel-based security provided a deeper understanding of recent technologies and their potential applications on the fields of monitoring and security.

The internship allowed for the exploration of cutting-edge technologies such as eBPF and its potential in enhancing security measures, thus expanding the understanding of these technologies as well as highlight the challenges and limitations of said technologies.

The development of this application in a company setting, proved itself to be a learning experience in of itself, providing an unique opportunity to interact with technologies that were not yet accustomed to. The cycle of development, from development, versioning and testing will surely prove to be an asset in future projects.

While the application developed may not be a bullet-proof solution, it demonstrates the potential of eBPF in enhancing security measures. The application has proven to be a functional tool that can contribute to the overall security of a system, even if in a limited way. While the end result may not serve for this particular use case the study and research will surely prove quite useful when trying to leverage eBPF's capabilities from a security perspective.

Despite the limitations and vulnerabilities noted in this application, the intership has provided an unique opportunity to engage with real world problems and leverage recent technologies to contribute to the advancement of knowledge in this field. This experience has also contributed to a greater appreciation for the importance of rigourous testing and evaluation in the development cycle.

In conclusion, while the application developed may not have achieved its inteded goals, the internship and the opportunity to conduct research on cutting-edge techonlogies proved to be an invaluable learning experience that has broadened the understanding of recent technologies and the potential applications in the field of security. The knowledge and skill will surely prove invaluable in future projects.