Place holder
Place holder 2

Carlos Francisco Caramelo Pinto

Master's thesis planning
Computer Science and Engineering
(2nd degree cycle)

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

January 2024

Contents

1 Introduction **1**

1.1 Introduction . . . 1
1.2 Motivation . . . 1
1.3 Document Organization . . . 1

2 Core Concepts and State of the Art **3**

2.1 Introduction . . . 3
2.2 Linux Kernel . . . 3
2.2.1 System calls . . . 4
2.3 eBPF . . . 4
2.3.1 eBPF Code . . . 5
2.3.2 eBPF Programs and Attachment Types . . . 7
2.4 State of the Art . . . 10
2.4.1 Tetragon . . . 10
2.4.2 *Seccomp* . . . 11
2.4.3 Syscall-Tracking Security Tools . . . 11
2.4.4 BPF LSM . . . 12
2.4.5 Formally verified approaches . . . 12
2.5 Conclusion . . . 12

3 Problem Statement, Experiments and Work Plan **13**

3.1 Introduction . . . 13
3.2 The Problem . . . 13
3.3 Proposed Solution . . . 13
3.4 Experiments . . . 14
3.4.1 **eBPF implementation of `ls`** . . . 15
3.4.2 Conclusion . . . 18
3.5 Future Tasks . . . 19
3.6 Conclusion . . . 19

List of Figures

2.1 `strace` of `ls` command 4

Acronymms

eBPF	Extended Berkeley Packet Filter
BPF	Berkeley Packet Filter
ASIC	Application-specific integrated circuit
BFT	Byzantine Fault Tolerance
DAG	Directed Acyclic Graph
DSL	Domain-Specific Language
FLP	MJ **F**ischer, NA **L**ynch, MS **P**aterson -
PoS	Proof of Stake
PoW	Proof of Work
RPC	Remote Procedure Call
SMR	State Machine Replication
TEE	Trusted Execution Environments

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) presents 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 and prevent any data exfiltration attempts.

1.2 Motivation

The main objective of this thesis is the development of an eBPF application that prevents data exfiltration from a given machine. This application will then go through a formal verification process.

The fact that we use 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 will 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 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. **Problem Statement** - In this chapter we will describe the problem we are solving and a proposed solution to solve it. In it we will also elaborate on an experiment done to further sustain that the solution is a feasible one and, to finish the chapter, the tasks for the continuation of the development of this dissertation will be uncovered.

This is the organization of the document, where firstly we describe how this tool works, and then elaborating on the problem and solution that will be worked on in this dissertation.

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. 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.

eBPF presents a quite ingenious solution to the problems mentioned above, seeing that eBPF programming does not mean direct interaction with kernel programming, and eBPF programs can be dynamically loaded and removed from the kernel. The latter presents one

the great strengths of eBPF, as it instantly gets visibility over everything happening on the machine.

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) [**?**] 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 a plethora 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. eBPF kernel code is written in a restricted C-like language, making use of libraries to provide abstractions for interacting with the eBPF subsystem. That code is then compiled into a specific type of bytecode that can be loaded into the kernel. This bytecode is subject to the eBPF verifier, which either accepts or rejects the program, making sure that it is safe and adheres to certain constraints.

User space programs are used to interact with eBPF from user space, usually written in languages like C or Rust 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 form the core of eBPF functionality, allowing for the extension and customization of Linux kernel behaviour without modifying its source code. These 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.

User space and kernel space programs communicate through the use of pre-defined data

structures, eBPF maps, serving as shared data structures. These data structures can be used to pass information, parameteres, or results between the user and kernel. eBPF supports various types of these maps, such as array maps, per-CPU maps, hash maps and ring buffers, each being suitable for different use cases. Accessing the information in an eBPF map through user space programs involves two ystem calls `bpf_map_lookup_elem()` and `bpf_map_update_elem()`, providing read and write operations on eBPF maps, respectively.

One of the potential limitations of eBPF code is the portability and compatibility of eBPF programs acorss different kernel versions. This challenge is tackled by the CO-RE, (Compile Once - Run Everywhere), concept which aims to enhance the deployment and maintanibility of eBPF programs. This approach consists of a few key elements:

1. *BTF* BTF is a format for expressing the layout of data structures and function signatures. It is used to determine any differences at compilation time and runtime. It is also used by tools like `bpftool` to dump data structures in human-readable formats.

2. *Kernel headers* The Linux kernel includes header files describing the data structures it uses, and those headers can change between versions of Linux. eBPF programmers can generate a header file using the `bpftool` containing all the data structure information about a kernel 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* When a user space program loads an eBPF program into the kernel, this approach requires the bytecode to be adjusted to compensate for any differences between the data structures present when it was compiled, and the ones present on the destination machine. This is accomplished making use of the `libbpf` library.

5. *BPF skeleton* A skeleton can be generated from an eBPF object file, containing functions that user space code can use for the management of the lifecycle of eBPF programs.

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

All eBPF programs are subject to a verification process, which involves checking every possible execution path through the program and ensuring that every instruction is safe. The verifier also updates some parts of the bytecode to ready it for execution. The verifier analyzes the program, evaluating all possible expressions, rather than actually executing them. It keeps track of the state of each register in a structure called *bpf_reg_state*. Each time the verifier comes to a branch, where a decision is made, it pushes a copy of the current state of all the registers onto a stack and explores one of the possible paths. It does this until it reaches the return at the end of the program, at which point it pops a branch off the stack to evaluate the next. If it finds an instruction that could result in an invalid operation, it fails verification, meaning that the program is unsafe to run. Verifying every single possibility is computationally unwise, therefore the verifier utilizes pruning to avoid reevaluating paths

that are essentially equivalent. When the verification of a program fails, the verifier will generate a log. It is also able to generate a control flow graph of the program in DOT format.

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 Program Context Arguments

All eBPF programs accept a context argument that is a pointer, but the structure it points to varies according to the type of event that triggered it. As such, programs need to accept the right type of context.

2.3.2.2 Kfuncs

Kfuncs in eBPF allow the registration of internal kernel functions with the BPF subsystem, permitting their invocation from eBPF programs after verification. Unlike helper functions *kfuncs* do not guarantee compatibility across kernel versions.

There exists a registration for each eBPF program type allowed to call a specific kfunc. There is a set of ”core” BPF kfuncs, including functions for obtaining and releasing kernel references to tasks and cgroups.

2.3.2.3 Kprobes and Kretprobes

Kprobe programs can be attached to almost anywhere in the kernel. They are commonly attached using kprobes to the entry to a function and kretprobes to the exit of a function, but there is the possibility of attaching kprobes to an instruction that is some specified offset after the entry to the function.

2.3.2.4 Fentry/Fexit

Fentry/fexit is, at the time of writing, the preferred method for tracing the entry to or exit from a kernel function. The same code can be written inside a kprobe or fentry type program. In contrast to kretprobes, the fexit hook provides access to the input parmeters of the function.

2.3.2.5 Tracepoints

Tracepoints are marked locations in the kernel code. They’re not exclusive to eBPF and have long been used to generate kernel trace output. Unlike kprobes, tracepoints are stable between kernel releases.

With BPF support, there will be a structure defined in `vmlinux.h` that matches the context structure passed to a tracepoint eBPF program, effectively rendering the writing of structures for context parameters obsolete. The section definition should be `SEC("tp_btf/tracepoint`

name") where the tracepoint name is one of the available events listed in /sys/kernel/tracing/available_events.

2.3.2.6 User Space Attachments

eBPF programs can also attach to events within user space code, utilizing uprobes and uretprobes for entry and exit of user space functions, and user statically defined tracepoints (USDTs) for specified tracepoints in application code or user space libraries. These user space probes use the BPF_PROG_TYPE_KPROBE program type.

There are considerations and challenges when instrumenting user space code:

1. The path to shared libraries is architecture-specific, requiring corresponding definitions.
2. Hard to predict the installed user space libraries and applications on a machine.
3. Standalone binaries may not trigger probes attached within shared libraries.
4. Containers have their own filesystem and dependencies, making the path to shared libraries different from the host machine.
5. eBPF programs may need to be aware of the language in which an application was written, considering variations in argument passing mechanisms.

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.

2.3.2.7 LSM

BPF_PROG_TYPE_LSM programs, which are attached to the Linux Security Module (LSM) API, providing a stable interface in the kernel, were initially designed for kernel modules to enforce security policies.

BPF_PROG_TYPE_LSM programs are attached using bpf(BPF_RAW_TRACEPOINT_OPEN) and are treated similarly to tracing programs. An interesting aspect is that the return value of BPF_PROG_TYPE_LSM programs influences the kernel's behavior. A nonzero return code indicates a failed security check, preventing the kernel from proceeding with the requested operation, which contrasts with perf-related program types where the return code is ignored.

2.3.2.8 Networking

Notably, these program types necessitate specific capabilities, requiring either CAP_NET_ADMIN and CAP_BPF or CAP_SYS_ADMIN capabilities to be granted.

The context provided to these programs is the network message under consideration, although the structure of this context depends on the data available at the relevant point in the network stack. At the bottom of the stack, data is represented as Layer 2 network packets—a sequence of bytes prepared or in the process of being transmitted over the network. On the

other hand, at the top of the stack where applications interact, sockets are employed, and the kernel generates socket buffers to manage data transmission to and from these sockets.

One big difference between the networking program types and the tracing-related types is that they are generally intended to allow for the customization of networking behaviors. That involves two main characteristics:

1. Using a return code from the eBPF program to tell the kernel what to do with a network packet-which could involve processing it as usual, dropping it, or redirecting it to a different destination.
2. Allowing the eBPF program to modify network packets, socket configuration parameters.

2.3.2.9 Sockets

In the upper layers of the network stack, specific eBPF program types are dedicated to socket and socket-related operations:

1. **BPF_PROG_TYPE_SOCKET_FILTER** which is primarily used for filtering a copy of socket data, being useful for sending filtered data to observability tools.
2. **BPF_PROG_TYPE_SOCK_OPS** which is applied to sockets specific to `TCP` connections, allowing for the interception of various socket operations and actions, providing the ability to set parameters like TCP timeout values for a socket.
3. **BPF_PROG_TYPE_SK_SKB** which is utilized in conjunction with a specific map type holding socket references, enabling `sockmap` operations, facilitating traffic redirection to different destinations at the socket layer.

These program types offer capabilities ranging from filtering socket data for observability to controlling parameters and actions on sockets within Layer 4 connections.

2.3.2.10 Traffic control

Further down the network stack is the TC (traffic control) subsystem in the Linux kernel, which is complex and crucial for providing deep flexibility and configuration over network packet handling. eBPF programs can be attached to the TC subsystem, allowing custom filters and classifiers for both ingress and egress traffic. This is a fundamental component of projects like Cilium. The configuration of these eBPF programs can be done programmatically or using the `tc` command.

2.3.2.11 XDP

These eBPF programs attach to specific network interfaces, enabling the use of distinct programs for different interfaces. Managing XDP (eXpress Data Path) programs is facilitated through the ip command. The effectiveness of the attachment can be verified using the `ip link show command`, which provides detailed information about the currently attached XDP program.

2.3.2.12 BPF Attachment Types

The attachment type within eBPF programs plays a pivotal role in finely controlling the locations within the system where a program can be attached. While certain program types inherently determine their attachment type based on the hook to which they are attached, others necessitate explicit specification. This specification significantly influences the permissibility of accessing helper functions and restricts the program's ability to interact 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. As an illustration, consider the `CGROUP_SOCK` program type, which has the flexibility to attach at various points within the network stack. The verification of attachment type ensures that the program can seamlessly integrate with the designated hook points, contributing to its effective functionality.

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 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*.

2.4.2 *Seccomp*

Seccomp, 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`, which uses BPF code to filter system calls. A set of BPF instructions acts as a filter, being triggered each time a system call is made. 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

Several tools fall under this category, but the best-known one is **Falco**, which provides security alerts. Falco is, by default, installed as a kernel module, allowing for users to define rules to determine what events are security relevant, generating alerts in a variety of formats. Since eBPF programs can be loaded dynamically, tools like Falco can leverage this to apply rules to applications that are already running, allowing for the modification of rules without modifying the applications or their configuration.

However, this approach is vulnerable to Time of Check to Time of Use issues. When an eBPF program is triggered at the entry point to a system call, it has access to the arguments passed to that system call. If the arguments passed, as is usual in kernel space, are pointers, the kernel will copy the data into its own data structures, before acting on that data. There exists then a window of opportunity for an attacker to change this data, after it has been inspected but before the kernel copies it. This is problematic since the data that is acted

on might not be equal to the one captured. One possible approach to solve this problem is to attach both to the entry and exit points in system calls. This approach can present accurate records of security relevant events, but it cannot prevent an action from taking place, since the system call has already completed when the point of exit check is made. Thus, the program should be attached to an event that occurs after the parameters have been copied into kernel memory, but the data is handled differently in system call specific codes. The approach currently seen as the most correct one is using the Linux Security Module (LSM) API, which requires a recent feature of eBPF: BPF LSM.

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.

2.4.5 Formally verified approaches

As of the writing of this document, no projects or tools leverage formal verification of eBPF code. The closest examples of these were found in a paper where there was the verification of the interpreter of an instruction setc closely resembling eBPF. **CITATION**, and a project where there was an attempt to create a translation validator for BPF programs, written in Coq **CITATION**.

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

Problem Statement, Experiments and Work Plan

3.1 Introduction

The Problem Statement, Proposed Solution, Experiments, and Work Plan chapter is a crucial component of the thesis as it outlines the research problem, a possible solution to it, an experiment to further sustain that solution, and the plan for executing the research. 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 experiments that will be conducted to validate the solution, as well as the work plan that outlines the tasks and the timeline for executing the research.

3.2 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 is that of preventing data exfiltration inside virtual machines. This problem 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.

3.3 Proposed Solution

The proposed solution aims to tackle the challenges mentioned in the previous section by providing a method of easily load tools with the intention of preventing data exfiltration, leakage or theft. To achieve this, the solution leverages the advantages of kernel based security, as well as the flexibility and ease of use of eBPF.

By using these approaches, the focus of this solution is the ease of deployment and verified security that eBPF provides.

The main object of this thesis is the development of an eBPF application preventing data exfiltration from virtual machines, such that one policy can be extended across various machines. As stated above, the classical approach of static configurations, although working

when configured statically during the provisioning of the machine, become hard to operate when deployed across a fleet of machines.

This solution will 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, preventing data exfiltration from said machines.

The resulting application would then go through a formal verification process on the kernel side of the application, ensuring that the rules to be applied are only that.

As of the writing of this document, there's no knowledge of projects or documents that describe a similar solution to the one proposed. There are several eBPF based tools for security purposes, mentioned in the previous chapter, but none of these leverage formal verification, and as such, this solution presents itself as both an academic and business opportunity.

The choice to formally verify this tool aims at ensuring the safety and correctness of said program. eBPF programs, although already a subject of formal methods through the verifier, do not currently ensure with absolute certainty the safety and correctness of the implementation attempted.

In the next section, an experiment will be demonstrated to showcase the feasibility and practicality of our objective. This first study is designed to validate our ideas and provide a foundation for future development. While we did not develop a full-fledged tool, this experiment serves as a crucial first step in demonstrating that there's a potential for the use of eBPF to prevent data exfiltration and leakage.

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.

3.4 Experiments

eBPF has proven itself as a flexible and secure tool for security and observability. The experiment aims to test and demonstrate the capabilities of this tool, by reacting to certain system calls, identifying the user who made it and where it came from, with the purpose of, in the future, extend it to not only present the system call made but to also act upon it, potentially preventing data exfiltration using this method. In this section we will elaborate on why eBPF is an ideal fit for this experiment and explore the inner workings of it.

The experiment at hand involves the implementation of an eBPF program, both user side and kernel side code, using the CO-RE approach, so that when a call is made to `chdir` is made the program will present the contents of said folder and identify the user who made the call, essentially behaving as an `ls` command.

We will also delve into the files that make up a typical eBPF application and explain how the program was developed based on the CO-RE approach.

In conclusion, this experiment will serve as a demonstration of eBPF's capability from a security stand point.

3.4.1 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.

3.4.1.1 `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.

3.4.1.2 `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.

```
1  int main() {
2    struct eBPF_ls_bpf *skel;
3    int err;
4    struct perf_buffer *pb = NULL;
5
6    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.

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

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:

```
1  while (true) {
2      err = perf_buffer__poll(pb, 10000000 /* timeout, ms */);
3      // Ctrl-C gives -EINTR
4      if (err == -EINTR) {
```

```
5         err = 0;
6         break;
7       }
8       if (err < 0) {
9         printf("Error polling perf buffer: %d\n", err);
10        break;
11      }
12    }
```

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.

```
1  void handle_event(void *ctx, int cpu, void *data, unsigned int data_sz) {
2    struct data_t *m = data;
3    char *pad = "{ ";
4    if (!strcmp(m->command + strlen(m->command) - 2, "sh")) {
5      const char *dir_path = m->path;
6      DIR *dir = opendir(dir_path);
7
8      // Check if the directory can be opened
9      if (!dir) {
10       perror("opendir");
11     }
12
13     struct dirent *entry;
14
15     printf("%s: ", getUser(m->uid));
16     // Read and print the contents of the directory
17     while ((entry = readdir(dir)) != NULL) {
18       printf("%s%s", pad, entry->d_name);
19       pad = ", ";
20     }
21     printf("}\n");
22     closedir(dir);
23     printf("\n\n\n\n");
24   }
25 }
```

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

3.4.1.3 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.

3.4.2 Conclusion

With this experiment, we were able to delve deeper into eBPF's capabilities, presenting a crude example of what is the goal of this dissertation, proving the usability of eBPF from a security standpoint. An application that monitors the `chdir` system call was developed, showing the target directory and the user who made the system call.

3.5 Future Tasks

The plan outlined below highlights the tasks inteded to take place in this dissertation project. It should however be noted that the direction of the project may evolve and change as we progress, leading to potential modifications to the plan. Nonetheless, this serves as a starting point for the remainder of the project.

Task 1: Implementation of an eBPF application capable of preventing data exfiltration, based on user specific policies, restraining certain files or processes from said users. The policies will be implemented in a specific format, so that they can change without having the need to hard code restricted users.

Task 2: Testing the application. In this step the application will undergo rigorous testing so as to be deployed across a fleet of machines with some certainty that it works as intended.

Task 3: Formal verification of the application. This step will involve the design of a formal verification tool to verify eBPF code. Some approaches are to be considered, but to date only the verifier has been subject to formal verification using Coq. In this step we will continue to build on the knowledge gained from the development of the application to gain a clear overview of the best approach to formal verification.

Task 4: Writing of the thesis. The thesis will present the research results and provide conclusions based on the work performed in the previous tasks. The writing of the thesis will be a final step, but will be done concurrently with the other tasks.

In addition to these tasks, additional features or improvements to existing ones might be added, depending on the results obtained. the aim is to continue advancing the state of the art in the field of eBPF security and particularly the formal verification of such tools.

3.6 Conclusion

In conclusion, this project has explored the crucial concepts of eBPF-based security. Through a review of the state of the art, it was made clear that although several tools leverage eBPF for security purposes, none of those have gone through the process of formal verification.

The proposed solution is to develop an eBPF based tool, and submit it to the process of rigorous testing and formal verification, as to guarantee the safety and correctness of said tool. The experiment presented is to be seen as a first step in achieving this goal.

The document also outlined the main contributions, goals, and objectives for future work. The next phase of the project will focus on the completion and testing of the target eBPF application, the formal verification process of it, leaving open the possibility of additional features and targets to be presented along the way. The aim is to advance the state of the art in the security field, specifically on data exfiltration prevention.