Place holder
Place holder 2

Carlos Francisco Caramelo Pinto

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

Supervisor: André Passos
Supervisor: Prof. Doctor Simão Melo de Sousa

January 2024

Contents

List of Figures

Acronymms

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.

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 complex, with around 30 million lines of code, meaning that if we were to try to make any changes to it, that would present a challenging task as making any change to a codebase requires some familiarity with it. Additionally, if the change made locally was to be made part of an official Linux release, it would not simply be a matter of writing code that works, 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 to be accepted, the waiting period until it would be accessible to everyone’s machine would probably several years old, seeing that most users don’t use the Linux kernel directly, but Linux distributions that might be using versions of the kernel that are 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 148 system calls. 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, which was designed for packet filtering within the kernel Unix-like operating systems. The original BPF was created to enhance the efficiency of packet filtering in networking. eBPF was later introduced to extend on those capabilities, beyond networking, allowing for a programmable framework within the Linux kernel. It allows developers to write custom code that can be loaded into the kernel dynamically, enabling a new generation of highly performant networking, observability and security tools.

In recent years, eBPF has undergone significant advancements, particularly in the realm of security and monitoring. The programmability of eBPF has led to the development of tools and frameworks that make use of its capabilities for enhanced security measures. As eBPF allows for the creation of security probes that can be attached to specific kernel events, it provides deep insights into system activities. eBPF based tools have allowed 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 seen, as it incurs in less overhead than traditional tools, diminishing the effect it has in the performance of the system. Particularly in systems that run containers, seeing that all containers make use of the same kernel, it has proven invaluable to get relevant system metrics, troubleshooting issues and gaining insights into application behavior.

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 the 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

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 Formal Verification

Does this appear?

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.