UNIVERSIDADE
BEIRA INTERIOR

Place holder
Place holder 2

Carlos Francisco Caramelo Pinto

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

Supervisor: Prof. Doctor Simão Melo de Sousa

February 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

No idea what to write here.

1.3 Document Organization

This document is organized as follow:

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 a 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, 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 eBPF

eBPF is a revolutionary kernel technology that allows developers to write custom code that can be loaded into the kernel dynamically, changing the way the kernel behaves. This enables a new generation of highly performant networking, observability, and security tools.

This tool has its roots in the BSD Packet Filter, which are programs that are written in the BPF instruction set, deciding wheter to accept or reject a network packet. BPF came to stand for Berkeley Packet Filter, being first introduced to Linux in 1997.

BPF evolved to what eBPF, or extended BPF, in 2014, having several significant changes, such as:

1. The BPF instruction set was overhauled to be more efficient on 64-bit machines, and the interpreter was entirely rewritten.

2. eBPF *maps* were introduced, which are data structures that can be accessed by BPF programs and by user space applications, allowing for the sharing of information between user space applications and BPF programs.

3. The `bpf()` system call was added so that user space programs can interact with eBPF programs in the kernel.

4. Several BPF helper functions were added

5. The eBPF verifier was added to ensure that eBPF programs are safe to run.

One of the biggest advancements in the eBPf toolset was made in 2020, when LSM PBF, was introduced, allowing for the attachment of eBPF programs as LSM, (Linux Security Module) kernel interface. This indicated a major use case for eBPF, bringing to light the fact that eBPF is a great platform for security tooling, in addition to networking and observability.

2.2.1 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.

2.2.2 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 Formal Verification