FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Code and Data Integrity of Modern Operating Systems

Thomas Karl-Heinz Kittel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 09.05.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 28.08.2017 angenommen.

Acknowledgements

I want to thank everybody who influenced my live. Special thanks to everybody who guided me to write this thesis. A list of names would be incomplete anyway.

Abstract

Our society is more and more based on digital information. People's lives depend on the availability of interconnected networks and the integrity of the information shared between the connected devices. Communication and sharing of information - through this network - is a cornerstone of our current way of living. Both criminals as well as nation states aim to extend their power and influence by controlling the devices and the information flowing through the network. For this, actors spend huge amount of energy into compromising the connected systems and thereby violate their integrity. For this, it is very important to be able to make assumptions about the integrity of a given device, most importantly the device's hardware and the operating system powering the device. This thesis aims to tackle this problem by proposing mechanisms to validate the integrity of modern operating system kernels during runtime.

In the first part of our work, we investigate, how runtime code integrity of a modern operating system can be achieved. While various mechanisms exist to ensure the load-time integrity of an operating system, runtime code integrity is still an open issue. Existing mechanisms agree on the assumption, that code, once loaded into memory is static. Thus, systems calculate and verify hashes of code pages in memory, or use dedicated hardware mechanisms to prohibit their modification once they are initially loaded. We show, that these approaches heavily restrict the ability of an operating system to perform benign self-optimization during runtime. We present an approach, that leverages detailed insight into the different active employed self-patching mechanisms to validate the integrity of self-modifying kernel code on a byte granularity, while also taking the current system state into account. For this, we make use of Virtual Machine Introspection (VMI), a method to monitor a system from an external view. To further highlight the importance of this part of our research, we also present a critical security issue in the Linux kernel that we discovered during our work, which allows unprivileged processes to load arbitrary code and data into the kernels code region, effectively undermining newly introduced defense mechanisms like Supervisor Mode Execution Protection (SMEP) and Supervisor Mode Access Prevention (SMAP).

After providing a way to validate the integrity of operating system (OS) kernel code, we take a look into how to improve the current state-of-the-art for kernel data integrity. Researchers have proposed various code reuse mechanisms, which allow an attacker to chain together instruction sequences already legitimately existing within a system to execute malicious behavior without introducing new code into a system. We aim to detect such code reuse malware by revealing the required control structures within data memory. We introduce Code Pointer Examination (CPE), an approach to identify and to classify code pointers in memory and provide a prototype for recent Linux kernels that is capable of detecting code-reuse malware in an efficient manner.

In addition, we also apply the methods developed in this theses to userspace applications. We present the investigations made in this direction and show to which extent the integrity of userspace applications can be validated.

Zusammenfassung

Unsere Gesellschaft basiert mehr und mehr auf digitalen Informationen. Das Leben der Menschen hängt von der Verfügbarkeit von ineinandergreifenden Netzwerken und der Integrität der Informationen die zwischen den verbundenen Geräten geteilt wird ab. Kommunikation und der Austausch von Informationen - durch diese Netzwerke - ist ein Grundstein unser derzeitigen Lebensweise. Sowohl Kriminelle als auch Regierungen versuchen ihre Macht und ihren Einfluss zu erweitern, indem sie versuchen, die Geräte und den Informationsfluss durch diese Netzwerke zu kontrollieren. Um das zu erreichen, verwenden die Akteure viel Energie um die verbundenen Geräte zu kompromittieren und verletzen dadurch ihre Integrität.

Aus diesem Grund ist es sehr wichtig in der Lage zu sein, Annahmen über die Integrität eines Gerätes machen zu können. Wichtig ist dabei sowohl die Hardware des Geräts als auch das Betriebssystem. Diese Arbeit geht das Integritätsproblem an und schlägt Mechanismen vor, mit deren Hilfe die Integrität von modernen Betriebssystemen zur Laufzeit überprüft werden kann.

Im ersten Teil der Arbeit untersuchen wir, wie die Codeintegrität eines modernen Betriebssystem zur Laufzeit überprüft werden kann. Während bereits verschiedene Mechanismen existieren, die die Integrität eines Betriebssystem zum Zeitpunkt des Ladens sicherstellen,

ist das Problem der Laufzeitintegrität noch ungelöst. Existierende Lösungsansätze basieren auf der Annahme, dass der Programmcode, sobald er in den Speicher geladen wird, statisch ist. Aus diesem Grund basieren diese Mechanismen darauf, Hashwerte von dem Code im Arbeitsspeicher zu berechnen und diese wiederum zu prüfen. Andere Mechanismen verwenden spezielle Hardware um Änderungen des Codes generell zu unterbinden. In dieser Arbeit zeigen wir, dass diese Ansätze die Fähigkeit moderner Betriebssysteme zur Laufzeit legitime Selbstoptimierungen durchzuführen stark beschränken.

Wir stellen einen Ansatz vor, der mit Hilfe von detailiertem Wissen über die verschiedenen aktiv verwendeten Selbstmodifikationsmechanismen die Integrität von selbstmodifizierendem Kernel Code auf Bytegranularität sicherstellen kann und dabei den aktuellen Systemzustand berücksichtigt. Um das zu erreichen verwenden wir eine "Virtual Machine Introspection (VMI)" genannte Technik um das entsprechende System von außen betrachten zu können. Um die Wichtigkeit dieses Teils unserer Forschung hervorzuheben, präsentieren wir zusätzlich ein kritisches Sicherheitsproblem des Linux Kernels, welches wir während dieser Arbeit entdeckt haben. Das Problem erlaubt unpriviligierten Anwendungen beliebigen Code und Daten in die Codebereiche des Kernels zu laden und damit effektiv neu eingeführte Sicherheitsmechanismen wie SMEP und SMAP zu untergraben.

Nachdem wir eine Möglichkeit gegeben haben die Codeintegrität eines Kernels sicherzustellen, legen wir unseren Fokus darauf, den derzeitigen Stand der Technik im Bereich Kerneldatenintegrität voranzubringen. Forscher haben verschiedene Code-Reuse Mechanismen vorgestellt, die es einem Angreifer erlauben, verschiedene, bereits auf dem System bestehende Instruktionssequenzen miteinander zu verketten und damit bösartiges Verhalten auszulösen ohne dazu neuen Code in das System einbringen zu müssen. Diese Arbeit beabsichtigt solche Code Reuse Malware zu erkennen, indem die von der Malware benötigten Kontrollstrukturen im Datenspeicher erkannt werden. Wir stellen Code Pointer Examination (CPE) vor, eine Technik um Code Pointer im Speicher zu erkennen und zu klassifizieren. Zusät-

zlich stellen wir eine prototypische Implementierung für aktuelle Linux Systeme zur Verfügung, welche Code Reuse Malware effizient erkennt.

Darüberhinaus adaptieren wir die in dieser Arbeit entwickelten Techniken auch auf Anwendungen im Userspace. Wir stellen die Untersuchen in dieser Richtung vor und zeigen in welchem Umfang die Integrität von Userspaceanwendungen generalisiert validiert werden kann.

Contents

List of Figures

List of Tables

List of Publications

Alexandre Bouard, Benjamin Glas, Anke Jentzsch, Alexander Kiening, **Thomas Kittel**, Franz Stadler, and Benjamin Weyl. Driving Automotive Middleware Towards a Secure IP-based Future. In *10th conference for Embedded Security in Cars (Escar'12)*, Berlin, Germany, November 2012.

Sebastian Vogl, Jonas Pfoh, **Thomas Kittel**, and Claudia Eckert. Persistent Data-only Malware: Function Hooks without Code. In *Proceedings of the 21th Annual Network & Distributed System Security Symposium (NDSS)*, February 2014.

Sebastian Vogl, Robert Gawlik, Behrad Garmany, **Thomas Kittel**, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data. In *Proceedings of the 23rd USENIX Security Symposium*. USENIX, August 2014.

Tamas K. Lengyel, **Thomas Kittel**, and Claudia Eckert. Multi-tiered Security Architecture for ARM via the Virtualization and Security Extensions. In *1st Workshop on Security in highly connected IT systems*, September 2014.

Thomas Kittel, Sebastian Vogl, Tamas K. Lengyel, Jonas Pfoh, and Claudia Eckert. Code Validation for Modern OS Kernels. In *Workshop on Malware Memory Forensics (MMF)*, December 2014.

Tamas Lengyel, **Thomas Kittel**, George Webster, and Jacob Torrey. Pitfalls of Virtual Machine Introspection on Modern Hardware. In *Workshop on Malware Memory Forensics (MMF)*, December 2014.

Fatih Kilic, **Thomas Kittel**, and Claudia Eckert. Blind Format String Attacks. In *10th International Conference on Security and Privacy in Communication Networks (SecureComm 2014)*, volume 153 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 301–314. Springer International Publishing, 2015.

Tamas K. Lengyel, **Thomas Kittel**, and Claudia Eckert. Virtual Machine Introspection with Xen on ARM. In *2st Workshop on Security in highly connected IT systems*, September 2015.

Andreas Fischer, **Thomas Kittel**, Bojan Kolosnjaji, Tamas K Lengyel, Waseem Mandarawi, Hans P Reiser, Benjamin Taubmann, Eva Weishäupl, Hermann de Meer, Tilo Müller, and Mykola Protsenko. CloudIDEA: A Malware Defense Architecture for Cloud Data Centers. In *5th International Symposium on Cloud Computing, Trusted Computing and Secure Virtual Infrastructures - Cloud and Trusted Computing (C & TC 2015)*, October 2015.

Thomas Kittel, Sebastian Vogl, Julian Kisch, and Claudia Eckert. Counteracting Data-Only Malware with Code Pointer Examination. In *18th International Symposium on Research in Attacks, Intrusions and Defenses*, November 2015.

Julian Kirsch, Clemens Jonischkeit, **Thomas Kittel**, Apostolis Zarras, and Claudia Eckert. Combating Control Flow Linearization. In *32nd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC)*, May 2017.

J. Kirsch, B. Bierbaumer, **T. Kittel**, and C. Eckert. Dynamic Loader Oriented Programming On Linux. In *1st Reversing and Offensive-oriented Trends Symposium (ROOTs)*, November 2017.

Bruno Bierbaumer, Julian Kirsch, **Thomas Kittel**, Apostolis Zarras, and Claudia Eckert. CookieCrumbl0r: Smashing the Stack Protector for Fun and Profit (unsubmitted).

Sergej Proskurin, **Thomas Kittel**, Apostolis Zarras, Sebastian Vogl, and Claudia Eckert. Follow the WhiteRabbit: Towards Consolidation of On-the-Fly Virtualization and Virtual Machine Introspection (unsubmitted).

Chapter 1

Introduction

The integrity of software is a basic requirement. In our daily lives we rely on thousands of different software programs. Not only do we expect them to function as intended, these programs are executed on operating systems, that are themselves trusted. Although the definition of trust is very hard to make in an IT security context, most people define trusted, as small and manageable in code size. That is, the code may be trusted, if both the source code is open and understandable and ideally audited. Another requirement is that the executed binary code is unchanged from what was created from the compiler during the translation process from readable source code to executable machine code. To maintain trust in an executable binary, the integrity of the executable has to be guaranteed even while executing in memory.

The integrity of code is often directly connected to its identity. One big step to maintain this trust is, for example the *reproducible-builds project*[1,2]. It aims to extend compilers and distributions so that the hash sum of a compiled binary is always reproducible and not

[1]`https://reproducible-builds.org/`
[2]`https://wiki.debian.org/ReproducibleBuilds`

dependent on for example the build time or the minor version of the compiler that was used during compilation.

Still, this approach does not solve the trust problem, as it only enhances the trust in the compiled binary. A typical assumption about code integrity is, that code is static once it is loaded to memory. For userspace applications, an executable program is usually directly copied to the applications virtual memory space by the operating systems application loader. Therefore, the code that is executed within memory is unchanged from the version that was shipped to the computer within the executable. Due to this, code integrity for userspace applications is usually achieved by simply hashing the memory pages containing code and in turn comparing the results to a hash of the code section within the binary, e. g. Executable and Linkable Format (ELF) file. So for userspace applications, the integrity of the code is usually directly bound to its identity.

Kernel Code Integrity However, the integrity of the operating system (OS) kernel is especially crucial for the security of the entire system. If the kernel gets compromised, the attacker can further disable existing protection mechanisms and take full control over the system and all applications running on it. This makes the kernel a very lucrative target for malware authors. Once the kernel is compromised, all security mechanisms that rely on its protection become useless. This is why modern hardware continues to provide additional security features that protect the kernel from the introduction of malicious code. A very prominent example of this development is trusted boot.

With trusted boot enabled, each component in the boot process validates the integrity of the next component in the boot chain before loading and executing it. This ensures that the bootloader will only load an untainted and signed OS kernel. Similarly, the kernel itself will only load modules/drivers that are signed and untainted. As a consequence, attackers can no longer simply infect the kernel by loading a malicious module or changing the kernel binary on disk, which are very popular attack vectors. Since trusted boot is part of the UEFI specification and is enabled by default since Windows 8, it has since gained wide distribution.

While trusted boot can ensure the integrity of the kernel code at load-time, it cannot provide *runtime* integrity. If the kernel contains a vulnerability, the protection mechanism can be bypassed and the kernel can be compromised. Since the security architecture of most systems is based on the integrity of the kernel, it is essential that the integrity of the kernel can also be verified during runtime. Hereby the validation of the kernel must include the data structures of the kernel as well as its code.

Unfortunately the assumption that code is static once it is loaded into memory is not true for the kernel itself. Once the operating system kernel is loaded by the boot loader, its code and data contents are copied to the main memory of the machine and an initialization routine is executed. Afterwards the program code and data get modified by the operating systems loader effectively changing the binary code. This process is similar if a new kernel module is loaded by the kernel. Previous research assumed, that after these load-time modification the kernel code is also static. To validate the code in memory, a code page in memory is cryptographically hashed, and, as long as this hash does not change, the integrity of the code is assumed. The initial hash for this type of comparison is usually taken from an already loaded reference kernel and not from the binary executable itself. This approach assumes that the binary code may be modified once during load-time but is constant during the runtime of the executable.

This assumption does not hold for modern kernels which perform dynamic code patching also at runtime (e. g. to control certain functionalities). Consequently, executable pages within kernel space may *legitimately* change during execution, which will lead to a high false positive rate if existing methods are used.

Additionally, this also allows attackers to easily circumvent the current protection mechanisms because such traditional methods are unable to distinguish between benign and malicious changes to the kernel code. To complicate things further, the runtime changes that take place are both system and architecture dependent, making it impossible to establish a single ground truth. This leads to many additional previously unconsidered challenges that must be solved to be able to validate kernel code at runtime. Being able to accomplish this task, however, is crucial, as the kernel provides the basis for virtually all protection mechanisms found on systems today.

In this theses, we take a deeper look on the question, if the assumption that kernel code is static once it is loaded into memory is valid in practice. We thereby first take a look on modern OS kernels, as these form the most important part of a running system and thus have to be most trusted. First investigations showed that the kernel code in fact changes during runtime. For this, in a first step, we investigate how and to what extent the executable code of a kernel changes during load-time and later during runtime. We show that for different reasons the executable code of an operating system kernel may legitimately change during runtime. For this, we will investigate the rationales and motivations behind for these self-modification mechanisms. With this information we answer the question how its integrity may be validated despite of these changes and build a framework that is able to validate the executable code of a state of the art Linux operating system during runtime.

Enhancing Kernel Data Integrity As code integrity mechanisms become more and more widespread, attackers are forced to find new ways to infect and control a system. A likely next step in malware evolution is thereby *data-only malware*, which solely uses instructions that already existed before its presence to perform its malicious computations [44]. To accomplish this, data-only malware employs code reuse techniques such as return-oriented programming (ROP) or jump-oriented programming (JOP) to combine existing instructions into new malicious programs. This approach enables the malware form to evade all existing code-based defense approaches and to persistently infect a system without changing its codebase [100].

Despite this capability and the substantial risk associated with it, there only exist a handful of countermeasures against data-only malware so far that can often be easily circumvented [20, 29, 41, 78, 19].

Thus, in this thesis, we continue by introducing a method to make assumptions about the integrity of data memory. As one usually is unable to make general assumptions about the contents of untyped memory, we focus on control-flow relevant data memory. An important part of control flow relevant data are pointers to executable code, as these are used to define certain callback functions during runtime. A well known example for such pointers is the list of code pointers inside the system call table. While a userspace process only requests a certain functionality to be executed in the OS, the system call table holds pointers to each concrete system call implementation. The actual system call dispatching is then done inside the kernel, where, based on the requested system call number the correct function is invoked. However, in addition to these well known places, code pointers are also used within normal data to specify the appropriate function to handle a certain object.

For this, we identify code pointers with a relatively simple heuristic. We rely on the fact that only a small amount of memory is executable within the entire address space. We identify a code pointer by the simple fact that the memory it points to is marked as executable in the page tables. With this we are able to check for each code pointer if it points to a known function or symbol or an otherwise allowed target. This technique is called Code Pointer Examination (CPE). We also implement that concept and extend our prototype framework to validate that all code pointers within kernel memory point to well known targets and thus aim to detect invalid function pointers. Thereby we are for example able to detect malware that introduces pointers to existing code into the system and that tries to reuse the existing kernel code in a ROP-style code reuse attack.

Userspace Lastly, we extend our scope to also provide integrity validation for userspace processes. We investigate to what extent the mechanisms and software components developed throughout this thesis may also be applied to validate the integrity of userspace applications. With this, we extend our framework to also validate the integrity of userspace application code as well as important control-flow relevant datastructures. We also conduct research in order to detect kernel targeting code reuse malware in the memory of userspace processes. Finally, we also investigate to which extent CPE may also be applied to detect code reuse malware that directly targets userspace applications.

1.1. Research Questions

This thesis delivers insight about the problem of assuring runtime integrity of modern operating systems. This consists of both the integrity of the kernel code as well the integrity of the kernel data. To solve this problem, we will address multiple research questions, that we lay out and describe in the following.

(Q1) Under which circumstances and to which extent does the code of a modern operating system change during runtime? With this research question we aim to gain insight into the different kernel features that an operating system employs which require dynamic self-modification of kernel code. We start to investigate how dynamic the code regions of a modern operating system are in practice. Initial tests revealed a lot of differences between the code that was contained within the kernel binary and the code that was loaded into memory. The first and obvious modifications are load-time relocations. However, additional dynamic modifications are applied to the in-memory representation of the code. These dynamic modifications within executable code exist because of various reasons. The most obvious candidates seem to be performance improvements and debug capabilities. For example, one might not want to check a certain unlikely condition all over again during code paths which are executed with a high frequency. Instead, to avoid polling, one might only want to change the executable code once, in case that condition is met and revert the code to its original state, in case the condition is not valid any more. In this research question we investigate the reason for dynamic code changes within modern operating systems. This question thus aims to thoroughly enumerate the reasons for dynamic kernel code modifications in order to be able to predict and validate code modifications, and thus the code integrity, for a specific operating system state. Previous mechanisms do not take dynamic code modification into account and allow code modifications at certain locations. Thus, allowing an attacker to take advantage of such an unvalidated location by modifying it to her specific needs. Note that the modification of a single instruction may be enough for the attacker to persistently change the control flow for malicious purposes.

(Q2) How to validate dynamic code changes within the kernel code during runtime? After the reasons for dynamic code modifications are investigated, we aim to implement a framework that is able to validate dynamic kernel code during runtime. This also includes the semantic validation of kernel internal data structures and the hardware state that corresponds to the concrete code modifications. For this purpose, a virtual machine introspection system is created which is able to predict code modifications according to a given system's hardware and software state. If a unique prediction of a modification is not possible, it should at least be possible to enumerate all valid modifications for a specific location. In addition, we evaluate the feasibility and performance of our approach.

(Q3) How can code reuse attacks be detected in kernel memory? After the integrity of kernel code is validated by giving solutions to the first research questions, we set our focus to validate parts of the integrity of kernel data. As most of the data per definition is constantly changing, it is impossible to make general integrity guarantees about kernel data. Previous approaches, further described in Section 3.2, try to validate the integrity of particular important data structures. However, due to the high amount of different objects, it is hard to validate all objects within an operating system kernel during runtime. For this reason, we restrict our focus on the detection of code reuse within the kernel. More specifically, we aim to detect control structures used for code reuse during system runtime. Thus, we aim to scrape code pointers out of the kernels data memory and try to detect clusters of code pointers that may be used as control structures for code reuse techniques such as ROP. For this, it is important to be able to not only find heuristics to detect code pointers, but to also find categories for the classification of these detected pointers. In this part of our research, we extend our framework to implement the afore mentioned pointer detection and classification and evaluate, if such an approach is applicable in practice. To be practical, the process needs to be efficient and the method should have only a very low number of false positives.

(Q4) Is it possible to also apply the developed methods to userspace processes? This question addresses how our developed technique can be applied not only to operating system kernels, but also to userspace applications. While we expect the code validation part of our work to be much simpler due to the success of other hash based approaches, we aim to find heuristics to successfully apply the results of **(Q3)** to userspace processes. This is required, as this allows to detect code reuse already, when the control structure is built up within a userspace application and before it is transferred to kernel space.

(Q5) Is there a theoretical difference between a control structure for code reuse malware and a legitimate stack? In this thesis, we developed heuristics to detect and classify code pointers in **(Q3)** and **(Q4)**. However, we finally raise our yet unanswered final question, whether it is possible, for a given program state and a given stack content, to decide if the stack contains legitimate content or a control structure for a code reuse program. While we are unable to answer this questions as part of this thesis, we think that answering this question would advance the current state of the art in malware detection.

1.2. Contributions

During this thesis, we give answers to the previously mentioned research questions. In the following we shortly summarize the contributions that we made while researching for this answers.

- We show that current code validation techniques are not suitable to validate the code integrity of modern kernels.
- We examine various load time and runtime code patching techniques employed by modern OS kernels.
- We discuss the challenges that these runtime code patching mechanisms create for code validation.
- We demonstrate the importance of correctly validating modern kernel code. We do this with a practical example that enables an unprivileged *user* to load arbitrary executable code into the Linux kernel.
- We introduce a framework that can successfully validate the integrity and identity of dynamic kernel code and enforces additional security constraints.
- We examine the state of the art of CFI mechanisms currently proposed and show that they are unable to detect all possible control-flow modifications in practice.
- We present CPE, a novel approach to identify and classify code pointers.

- We highlight important data structures that are used for control flow decisions in modern Linux kernels and thus must be considered for control flow validation.

- We provide a prototype implementation and show that it is both effective and efficient in detecting control structures of data-only malware.

- We extend the proposed kernel integrity validation framework to also validate the integrity of userspace processes.

1.3. Outline

In the following, we will shortly outline the remainder of this thesis. First, we provide the technical background required for this thesis in Chapter 2. After describing some hardware and operating system background together with current kernel protection mechanisms, we introduce the concept of Virtual Machine Introspection (VMI) and describe the semantic gap, the problem to interpret a systems memory and hardware state from an external perspective. We illustrate this problem by introducing some existing VMI frameworks and the techniques they use to solve the semantic gap. Further, we elaborate on the concept of Data-only Malware, a novel type of malware that only relies on code-reuse techniques and does not need to execute its own payloads. As this type of malware effectively mitigates the concept of Code Integrity Validation (CIV) we have to elaborate new techniques to be able to detect and defend against this type of malware.

Then we introduce related work to the research done in this theses in Chapter 3. We elaborate previous research on CIV, a technique to validate the contents of application and kernel code and the different problems that have to be solved to provide CIV for a modern OS and also also introduce related work to the problem of data integrity validation for modern OS kernels.

CFI was introduced to secure the control flow of against malicious modifications. In Chapter 4, we introduce the different concepts that where introduced and show, why, in our opinion this approach is not enough to mitigate the problems that it tries to solve. We thus do not focus on the weakness of concrete implementations but more on a general problem with the proposed approach.

Chapter 5 describes our efforts to ensure kernel code integrity on a modern OS and addresses the problem of runtime code self-patching mechanisms employed by these modern OSs. In this Chapter, we also describe a severe vulnerability within the Linux kernel that we detected during our efforts.

After proposing a solution to code integrity, we focus on data integrity and introduce a concept called CPE, in Chapter 6. CPE can be used to detect persistent code-reuse-based malware within the memory of a modern operating system. While we do not claim that CPE can mitigate code-reuse based attacks in any case, it is still able to detect its presence in memory.

Chapter 7 we apply the knowledge gained during our investigations with the Linux kernel to userspace applications and show our results in this direction.

Finally, Chapter 8 concludes this thesis including a discussion about our contributions and possible future work.

Chapter 2

Background

In this chapter, we discuss background that is relevant to the overall topic of this thesis. This also includes the current state of the art of solutions that are proposed to answer the research questions brought up in this thesis. Parts of this chapter have already been published as research papers: [54, 53].

2.1. Hardware and Operating System Background

In this section, we will cover background concepts that are important for the understanding of the remainder of this thesis. In particular, we will provide an overview of virtual address translation and existing kernel code protection mechanisms on the x86 architecture. The code verification concepts are nevertheless also applicable to other architectures like e. g. the ARM architecture.

2.1.1. Virtual Address Translation

While software on the x86 architecture usually operates on virtual addresses, the hardware operates on physical addresses. For this purpose, there exists the memory management unit (MMU), which translates the virtual address used by the software into physical addresses that can be processed by the hardware. How this translation is conducted depends on the paging mode that is used by the processor. The x86 architecture supports three different paging modes: 32-bit paging, PAE paging, and IA-32e paging [46]. In this thesis, we are only concerned with PAE paging and IA-32e paging, which are the paging modes that are commonly used nowadays. Both paging modes use multiple page table levels to translate a virtual address into a physical address. While we will not go into details on the individual pageing mechanisms, we will briefly describe the general translation mechanism in the following.

A virtual address is translated into a physical address with the help of multiple page table levels. Hereby entries in a page table either references another page table (on a lower level) or a physical frame that contains the data that a virtual address is referencing. Whether a page table points to another page table or a page frame is specified by each individual entry within a page table. The address of the top-level page table that is used during the translation process is contained in the `CR3` register. In particular, the `CR3` register contains the *physical* address of the first page table. Consequently, the hardware can directly access the table without having to translate its address first.

The entry that is used within a page table to translate a virtual address is encoded within the address. To select an entry within a page table, the hardware will take a predefined number of bits from the virtual address and use the value of those bits as an index into the page table. Hereby the hardware processes the virtual address from left (the highest bit) to right (the lowest bit). For instance, in the case of IA-32e paging the leftmost 9-bits of the virtual address determine the page table entry that is to be used in the initial page table pointed to by the `CR3` register.

To translate a virtual address, the hardware will step through the page tables following the page table entries that are encoded within a virtual address. This process will continue until a page table entry does not reference another page table, but a physical page frame. The remainder of the virtual address (i. e. the part of the virtual address that was not yet used to identify a page table entry) is used as an offset into the page frame to access the data that the virtual address is referencing.

Besides referencing a page table or a page frame, a page table entry also contains status bits. For this thesis the following bits are of interest:

P	The present bit specifies whether the entry is valid or not.
U/S	The user/supervisor bit specifies whether the page frame pointed to by the virtual address contains kernel code/data or user code/data.
R/W	The read/write bit specifies whether the page frame pointed to by this virtual address is read-only or can be written to.
XD	The execute-disable bit specifies whether the page frame pointed to by the virtual address is executable or not.

Obtaining all executable pages within kernel space can therefore be achieved by iterating through the page tables and extracting all page frames that can contain supervisor code, which are page frames that are referenced by page table entries whose present flag (P) is set while its supervisor bit (U/S) and execute-disable (XD) bits are cleared.

Modern CPUs also provide a corresponding structure within the hypervisor that translates the Guest Physical Address (GPA) to the Host Physical Address (HPA). These structures are called Extended

Page Tables (EPTs) and may be used to enforce additional policies and to track for example write operations to specific pages. The access permissions for each page can be different in the guest's page tables and in the EPTs. In that case, the most restrictive policy is enforced by the hardware.

2.1.2. Kernel Code Protection Mechanisms

In this section, we take a closer look at the code protection mechanisms that are offered by the x86 architecture and/or are leveraged by modern OS kernels.

2.1.2.1. $W \oplus X$

As mentioned in the previous section, the x86 architecture provides an execute-disable bit as well as a read/write bit. These features can be leveraged to implement $W \oplus X$. The general idea is that each page in the page tables is marked *either* as writable *or* as executable, but never as both. As a result, code pages cannot be simply modified by an attacker, because they are marked as executable and are therefore not writable. Similarly, data regions cannot be executed as they are writable and therefore not executable. Note that the code segment of the Linux kernel is not writable per default. To patch its code the kernel temporarily sets the code to being writable and executable effectively violating the $W \oplus X$ policy.

2.1.2.2. Trusted Boot & Module Signing

The goal of trusted boot is to ensure that each component in the boot chain is untainted and has not been modified by an attacker at load time. To provide this functionality a signature exists for each component that includes a hash value of the component. During the boot process each component verifies the next component in the boot chain by checking the validity of its signature, recomputing the hash of the component, and comparing the so obtained hash value with the hash value within the signature. If both the certificate as well as the hash value can be verified, the component is considered to be untainted and is loaded. Otherwise the boot process is aborted. The TPM specification [96] extends this concept so that applications can later seal their keying material depending on an untainted boot process.

While a kernel module must not necessarily be loaded as part of the boot process, the trusted boot approach can be extended to include loadable kernel modules. In this case the kernel is configured to only load modules that are signed and untainted. To allow this scheme to work each module must provide its own signature that can be verified by the kernel before it is loaded. The verification process is thereby identical to the previously mentioned process.

If trusted boot is used, an attacker can no longer simply modify the kernel binary on disk as this would change the hash value of the kernel binary and it would therefore not pass the verification process on the next boot. Similarly, if module signing is used, an attacker can no longer load arbitrary modules into kernel space as the modules are verified before they are loaded. By combining trusted boot and module signing, it is thus possible to tightly restrict the code that can be loaded into the kernel space. Notice, however, that this protection mechanism does not protect the kernel from *runtime* modifications.

2.1.2.3. Supervisor Mode Execution Protection (SMEP)

A common exploitation technique that is often used to execute attacker controlled code with kernel privileges is to place the desired instructions into userspace and then divert the control flow of the kernel to the userspace code region. Since the processor already executes at the highest privilege level when the control flow is diverted, it will execute the userspace instructions with supervisor privileges.

The advantage of this approach is that the attacker neither needs to alter kernel code nor does she have to load a kernel module. Consequently, the attacker does not have to circumvent protections such as $W \oplus X$ or trusted boot. Instead the attacker only needs to store her instructions in userspace, which only requires control of a userspace process on the system.

To protect against such attacks, Intel introduced SMEP. With SMEP enabled, loading an instruction from a userspace page (a page that is not marked as a supervisor page in the page tables) while operating at the highest privilege level will lead to a page-fault exception [46]. Therefore the attacker can no longer divert the control flow of the kernel to userspace code as all userspace pages are marked as user pages and not as supervisor pages.

2.1.2.4. Supervisor Mode Access Prevention (SMAP)

SMAP is a similar method as SMEP for data accesses. In the case of SMAP the system can be configured to cause a trap (page fault), if the system is in supervisor mode and tries to access data that is mapped as user memory. This way an attacker needs to find a way to transfer all code and data that she needs for her exploit to the kernel before she is able to use that data in her exploit.

2.1.2.5. Memory Protection Keys for Userspace (PKU)

Intel also recently introduced a new page protection mechanism into its architecture called protection keys. In addition to the classical user- and supervisor classification, a page table entry now also contains a protection key associated with it. This key allows to classify a userspace page into 16 different protection classes for which the access rights (read and write accesses) may be set differently in each execution thread. The allowed access modes for each class are specified by corresponding bits in the userspace accessible $PKRU$ register.

2.1.2.6. Address Space Layout Randomization (ASLR)

When ASLR is enabled, code regions such as the kernel code region are no longer loaded to a fixed address. Instead the loader generates a random address when a binary is loaded and places the binary at the generated address. Therefore, the memory address of a binary will be different each time it is loaded.

While this approach does not directly protect the code regions of the system from modifications, it forces the attacker to determine the address of the code regions, before she can modify them or execute them. Since the memory address of the regions cannot be predicted by the attacker, this may be a difficult task especially if the attacker does not yet have control of a process on the victim's system. There are however still two frequently exploited problems with ASLR. First, the number of random bits in the virtual address of the code page is usually small (12 bit in x86 and 28 bit in x86_64). Second, libraries and modules are still a loaded to the beginning of a page boundary. Therefore, if an attacker is able to gather information about a single symbol within a library or module, he is again able to calculate the address of the start code section and thus the location of all symbols.

ASLR is already widely adapted for userspace applications. Also, Kernel ASLR (KASLR) was implemented for the Linux kernel in 2013[1]. However, during the time of writing, KASLR is still not enabled by default. Although the mechanism is already implemented for quite some time, it is not enabled, as there exist compatibility issues with the Linux kernel's hibernation feature. This is a typical example, where usability of a system directly contradicts its security mechanisms. Currently, the developers work to enable KASLR by default[2].

2.2. Virtual Machine Introspection

VMI is a technique that can be used for unconstrained introspection and manipulation of any virtualized system that is executed inside of a hypervisor. In this work, the term Virtual Machine Monitor (VMM) is also used for hypervisor interchangeably. In this section we

[1]https://lwn.net/Articles/569635/
[2]https://lwn.net/Articles/683733/

describe basic properties of VMI and how this technique is commonly used to detect malicious software in forensic environments.

The term VMI has first been defined by Garfinkel and Rosenblum [37]. In their paper, the authors describe three properties that are required for VMI.

Isolation: The first property mandates, that the monitored guest system must not be able to access or modify anything on the target system except its own state. This ensures an attacker is unable to tamper with the security mechanism, even after she has completely subverted the monitored guest system.

Inspection: The second property requires, that the hypervisor has access to the entire (virtual) hardware and software state of the monitored virtual machine. This includes not only the current CPU state as well as all the memory and I/O device state. This also includes all elements that the guest is able to store data on.

Interposition: The last, and most important property is, that the hypervisor needs to be able to interpose on certain operations that the virtualized guest conducts. This is required, as the VMI application can use this property to instrument the monitored guest and thereby is able to infer additional information from the monitored guest.

With these properties, a VMI application is, for example, able, to extract certain information of a running guest, every time a new active process is scheduled within the virtualized guest. Due to the first property this can even be done in a completely stealth manner. Garfinkel and Rosenblum conclude that hypervisors typically fulfill these properties and are thus a perfect tool to implement VMI mechanisms.

This insight however leads to another problem of VMI that is the semantic gap.

2.2.1. The Semantic Gap

The basic problem of VMI is the problem of interpreting the current hardware and software state of a Virtual Machine (VM) from the outside without further knowledge about the current system's internals. This step of interpretation was coined as the *semantic gap* by Chen and Noble [24]. The problem in this step arises due to the large amount of binary data that has to be parsed.

In order to resolve the issue researchers have found different ways to handle this problem and to reduce the amount of information that has to be generated to generate a consistent view about the monitored system. In this process they introduced a classification of different view generation patterns [70]. This classification was made as of two reasons. First it classifies the amount of information that can be gathered by view generation pattern, second, it also classifies the amount of trust, that an external inspector should put into the generated information.

In the following we will shortly describe these view generation patterns:

2.2.1.1. Derivation

The first approach to infer information about a previously unknown system is to derive the information directly through semantic information about the underlying hardware architecture. While the amount of information that can be gathered with this approach is limited, it has two major advantages. First, the gathered information is *binding* and thus can not be changed by a malicious attacker. This is due to the fact, that the information that us used during this process is directly used by the hardware. The critical data-structures used in the derivation process are thus *rooted in hardware* [71], as one can build a chain between the data-structures and an immutable, hardware defined component. A VMI application can for example derive a sequence of system calls, as the address of the system call handler is contained in a special hardware register and any execution of that page can be trapped by the hypervisor. Also, it is for example possible to derive the memory areas that are allocated for a certain process within the guest VM, as a pointer to the page mapping for each process is contained in a special register (CR3 on x86), together with a description of the paging mode that is currently used (CR0 on x86). Using these two registers, a VMI application can extract the page tables for each process from memory and can thus extract, e. g. all executable pages. As of that, hardware-rooted data structures become *evasion-resistant*.

Second, this approach is also software agnostic, as it only relates to the underlying hardware architecture and not on certain implementation decisions made by e. g. Linux or Windows. This results in the fact that VMI applications that solely rely on a derived view are guest operating system agnostic.

A drawback of this method is, that only hardware-rooted information can be derived from a system. This still leaves a dark spot on most of the guest state. This is also, why other means of information delivery approaches are required.

2.2.1.2. Out-of-band delivery

Another important part of guest memory is defined by the concrete applications that are executed within the monitored guest. These applications usually consist of an OS kernel together with applications that are executed in userspace. To be able to make sense of the information that is processed by the kernel, a VMI application requires symbol information to connect the raw memory with its higher level representation. This symbol information is usually generated in an out-of-band approach by analyzing the source code or binary information of the executed programs. This information for example consists of all the data types that are defined within the program and the addresses of all global variables together with information about their respective types. Thus, a VMI application is enabled to navigate through the guest memory, while starting at global variables.

This approach has the advantage, that the semantic information can already be generated in an offline manner and the view generation process is independent of the monitoring process. On the other hand it has the disadvantage, that the information that was generated does not take the active system state into account. Not only is the information bound to a specific software version, malware could also change the monitored system in a way, that the generated view does not match the current system any more. These attacks are known as Direct Kernel Object Manipulation (DKOM) [17] and Direct Kernel Structure Manipulation (DKSM) [7]. Another problem with this approach is, that the type of internal data structures may also depend on the current system state and that the generated view is thus ambiguous. Therefore, this approach is also called *non-binding*, as the gathered information is not directly bound to the monitored systems state.

Due to this, current VMI mechanisms use a combination of the derivation and the out-of-band delivery approach.

2.2.1.3. In-band delivery

In contrast to the previously described view-generation approaches, one can also use an in-guest agent, to deliver information about the monitored guest system. Due to its internal position, the in-guest agent directly uses the monitored guests semantic information. Thus it is also possible to use this delivery approach to bridge the semantic gap for parts of memory, that could not be bridged using other means of view generation.

However, in-band delivery also comes with a fundamental problem, which has to be thought of, before leveraging this mechanism. As the view generation is executed within the domain of the monitored system, an attacker is able to tamper with the view generator. It can, for example, provide the agent with malicious or modified data. This means, that a VMI system must take special care and consider that the generated results might me unreliable or false.

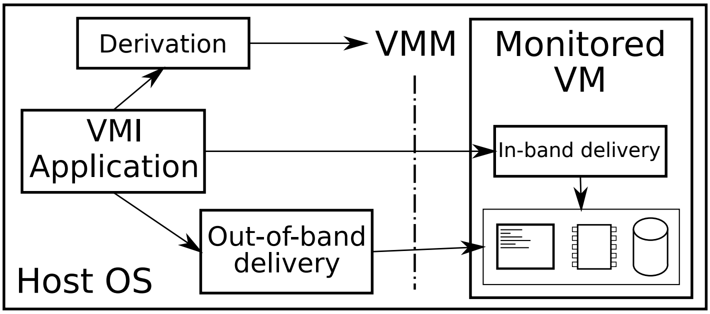

Figure 2.1.: Different view-generation approaches.

2.2.2. Combination of Approaches

As we have seen the same information may be derived or delivered from an introspected guest system using different approaches. Figure 2.1 gives an overview over the different approaches. In the following, we give an example how list of all active processes may be extracted from a monitored guest VM using the different approaches. The easiest way to extract the list of running processes, one may inject an in-guest agent into the monitored system that in turn executes the `ps` command within the monitored machine and redirects the output to the analyzer. While this is the simplest approach, malware within the monitored VM might not only recognize the in-guest agent, but also modify the information that is extracted. That is, this approach is not stealthy and the information extracted is not binding and thus does not necessarily reflect the correct information.

Alternatively, the list of executing processes may also be extracted by using out-of-bound delivery. For this, the VMI application needs external information about the location and layout of the corresponding kernel data structures. With this, it may locate the kernel structure `init_task` within the virtual memory of the monitored guest VM. In order to map this address to the corresponding physical address, the introspecting application is required to parse the page table data structures of the monitored guest, that are available at the symbol `init_level4_pgt`. After this information is extracted, the doubly linked process list can be traversed and all information about currently executing processes may be extracted. With this approach the information may be extracted from the introspected guest VM in a stealthy manner. Also, the full state of the guest system may be extracted. But still, the information that is extracted does not necessarily reflect the monitored VM's true state (is non-binding). This is because a malicious host may still have altered the information within this datastructure.

Finally, the list of processes may also be derived from the virtual hardware state. This has the advantage, that this process is both stealthy and binding. As a drawback, this mechanism does not allow to reconstruct a full state of the monitored VM. In order to extract a list of processes together with their virtual memory mappings, the VMI application needs to configure the hypervisor to trap on each change of the `CR3` register. This register holds the physical address of the currently active page table datastructure. As each process owns its own set of page tables, this address identifies a process. With access to the active page tables, the memory contents of each process may be extracted from the guest VMs physical memory. Still, metadata, such as the name of the process or its associated process ID, can not directly be derived from the virtual hardware.

While In- and Out-of-Band delivery provide a wider range of information about the monitored system, only derived information is trustworthy.

2.2.2.1. Weak vs. Strong Semantic Gap

In addition to the different view generation patterns previously discussed, Jain [48] introduces the differentiation between the *weak semantic gap* problem and the *strong semantic gap* problem.

The difference between these concepts is made in the way, VMI is used to analyze a system. In the first case, the system is started in a known and benign state. The system is then monitored throughout its entire live cycle and any malicious change is directly analyzed. In this scenario, the entire state of the system, that means the hardware architecture as well as all applications on the monitored system, is known, and any changes in the guest systems semantics are an indication of a malicious change made by malware. This is a common scenario in current (cloud) environments, where VMI is used to extend the existing monitoring environment. Jain refers to this as the weak semantic gap and postulated, that it is basically a solved engineering problem.

In contrast, the strong semantic gap is a scenario, where a system's hardware and software state is not known in advance or the system has not been under monitoring through its entire live cycle. According to Jain [48], this problem is currently still unsolved.

2.2.3. Virtual Machine Introspection Frameworks

This work is influenced and based on multiple existing VMI frameworks. In the following we will shortly introduce these frameworks and also describe how our approach leverages their previous work.

Insight [80, 79] was developed as a tool to bridge the semantic gap and build a graph of objects in kernel memory. We will later go explain the concepts of this system in more detail. In terms of definitions Insight is based on the out-of-band derivation pattern as it uses compiler generated symbol and debug information to detect these objects. The framework was specifically implemented for the Linux kernel. The framework's initial concept was to parse the kernel‘s source code and compiler generated debug information of the Linux kernel, specifically the Debugging With Attributed Record Formats (DWARF) debug information. The goal was to map the kernel memory by starting with global objects, as the type and location of these global objects is well known. From this, these objects reference all other objects within kernel memory through a chain of pointers. By traversing the generated tree, one could find and identify all objects.

While the initial versions of this thesis have been implemented as modules for the insight framework, some limitations emerged. The most important have been the lacking support for live introspection of a running virtual machine and the complex setup phase for new kernel versions. For this reason we decided to generate a separate framework as part of this thesis. One big advantage, that we take from the concepts already implemented in Insight is the possibility to bridge the semantic gap and extract information from the analyzed kernel.

Another project that this thesis builds upon is *LibVMI* [64]. LibVMI is generated as a generic library to interact with a Virtual Machine in a generic way. While its initial name was *XENaccess* and it was initially only built for use with XEN, it is now also compatible with other hypervisors such as, for example, KVM.

While LibVMI is a generic wrapper to control a hypervisor, like pausing and resuming a virtual machine, in the context of our thesis, the library is mainly used to transparently access the memory of the target virtual machine. This is possible, as LibVMI is able to parse the page tables of an introspected virtual machine and reads or writes to the memory of the target virtual machine. Recent versions of LibVMI have also been updated to support the use of Intels EPT.

LibVMI is also frequently combined with the VMI frameworks Volatility [102] and Rekall [74]. Both of these frameworks aim to allow a user to gather information from the memory of running virtual machine. This information, besides others, contains a list of running processes, a list of loaded kernel modules. As we are highly depending on low level information of the introspected VM, we base our work direktly on *LibVMI*.

Next, we will introduce the reader to the basic underlying concepts of code reuse attacks and data-only malware, a novel type of malware, which we aim to protect against as part of this thesis.

2.3. Code Reuse Attacks

The concept of data-only malware is both a foundation and a motivation for this thesis. To describe data-only malware we first need to take a quick look at the history of malware and exploitation. In the past decades, security mechanisms and secure programming have not been to wide spread in the software and hardware development industry. Code and data memory have not been separated. It was typically enough for an attacker to search for a piece of code that takes some input from the user and does not correctly filter the length of the input. This way, an attacker was able to achieve two goals. First, he was able to overwrite some memory location, that the program uses to manage its control flow, namely the current functions return address on the stack, and in addition to also introduce his own executable machine code into the attacked system. For a long time, this was enough to alter the control flow of the attacked program.

To mitigate this problem, operating system developers introduced means to prevent the execution of data pages, as the stack. First, software workarounds such as PaX [42] for Linux or Data Execution Prevention (DEP) from Microsoft[3] (Windows XP Service Pack 2) were introduced. Later, Intel introduced a dedicated hardware flag to disable the execution of a page within the page tables data structure with the release of the Pentium 4 (Prescott) processor.

2.3.1. Simple Code Reuse Attacks

With this separation attackers were not easily able to introduce new executable code into a system any more. Thus, attackers started to develop new techniques to reuse executable code already present on a target system [89]. They started to recycle existing functions present within a library that was already loaded in the attacked system, like, for example, the C library. While only a single function could be

[3] `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366553.aspx`

executed, attackers quickly understood to chain multiple functions together [104] by carefully crafting the contents of the attacked stack.

Later Shacham *et al.* [85] extended on this concept by introducing a variant of return-to-libc, that does not aim to reuse entire functions, but instead only uses short instruction sequences which themselves end in a return instruction, called *gadgets*. The important feature of this return instruction is that it pops the value at the current top of the stack to the instruction pointer and thus directs the control flow to the gadget at that next address. To achieve the intended functionality, an attacker is required to create a control structure, that contains a list of addresses of gadgets, that need to be executed. To conduct the attack, the control chain, is loaded into the memory of the attacked program, in the easiest case, onto the stack. Every time, during the attack, a return instruction is executed, the address of the next gadget is read from the control chain and the control flow is redirected to the selected instruction. For this, this technique was quickly known as return-oriented programming (ROP). Figure 2.2 illustrates a ROP-chain on the stack. Note, that during such an attack, the control structure might also contain gadgets, that, for example, pop intermediate values from the stack into a selected register. Thus an attacker may actively influence the control flow by modifying register contents during the attack. It was shown that ROP can achieve Turing complete computation [22]. Still one problem of ROP attacks is that the content of the stack is modified and destroyed during its execution. Similar techniques where also presented that use indirect jumps or signals instead of return instructions [22, 12, 14]. For this, this category of attacks is now known as Code-Reuse Attacks (CRAs), as only existing instructions are used for the attack and no new code is introduced.

From this time on, CRAs were commonly used as the first part of an exploit. It used to infiltrate a system and to disable certain protection mechanisms and to download a second stage. The second stage, in turn, contains the intended malicious functionality and is itself not based on code reuse, but typically still consists of classical binary code. Thus, the second stage could then be directly executed

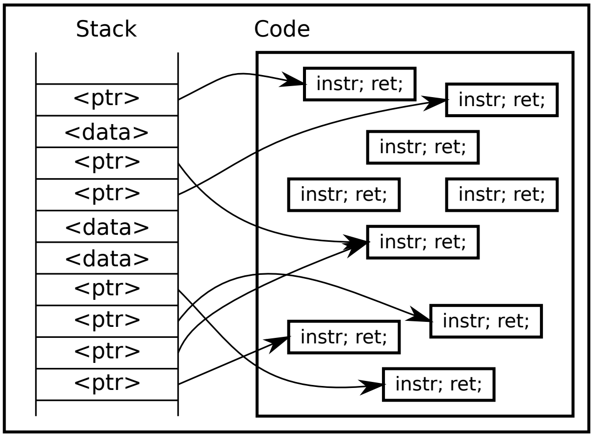

Figure 2.2.: A ROP chain on the stack. Different gadgets are chained together. After a `ret` instruction is executed, the next gadget is activated.

on the attacked system, as the security mechanisms were already disabled by the first stage. Note, that even current `glibc` versions contain the string `/bin/sh`. Thus a simple exploit is only to find this string in memory and to direct the control flow of the attacked system to the `system()` function, while providing the address of the string as a parameter to the function. With this, the attacker may open a shell on the attacked device.

2.3.2. Persistent Data-only Malware

Over the years, defense mechanisms improved again and were again able to detect such second stage malware. As the second stage introduces code into the system this code might also be detected by a defensive security system. Therefore malware authors are more and more required to implement not only the first exploits, but instead their entire malware leveraging code reuse techniques. This leaves two possibilities for an attacker: Either the malware lacks of persistence and the attacked system has to be exploited over and over again to execute the malicious functionality. In this case, of course, the malware is not able to react on the behavior of the attacked system (e. g. system call hooking to implement, for example, a key-logger). This also means that the malware is easily removable by simply fixing vulnerability which was exploited by the malware. An example for such a data-only rootkit was first published by Hund *et al.* [44].

Another possibility is to implement all rootkit functionality, including memory persistence, entirely with code reuse techniques. With persistent malware, in this context, we mean a rootkit that is part of the attacked system and may be activated during selected events within the attacked host, such as for example function hooking in the simplest case or on system call invocation or during interrupt handling. In addition, persistent data-only malware has the requirement, to not only intercept certain events, but also to always hand control back to the original control flow. Persistent malware by its definition is not required to survive, for example, a system reboot. Vogl *et al.* [100] have shown, that it is possible to achieve a persistent data-only rootkit without modifying existing code or introducing new code into a system. Vogl *et al.*'s work also contains *Chuck*, the, to our knowledge, first publicly available implementation of a persistent data-only rootkit. Although it was written as pure data-only rootkit, it provides key-logger functionality and is capable of hiding processes and files within the attacked system.

While code reuse *exploits* usually only make use of a very small control data structure that simply allocates a writable and executable memory region which is then used to execute traditional shellcode, control data structures of data-only malware are in general quite large. The reason for this is that data-only *malware* solely relies on code reuse to function. Each functionality that the malware provides must be implemented by using code reuse. The result are huge chains that contain hundreds of reused instruction sequences [100].

As the author of this thesis was also involved in Vogl *et al.*'s work, and his work serves as a motivation for this thesis, we, in the following, describe persistent data-only malware and the challenges involved to create such malware in more detail. First, we shortly discuss the general problems and requirements of persistent data-only malware and then introduce the architecture of the example implementation.

In a classical CRA scenario, an attack can typically be divided in to a first code reuse based exploit that in turn downloads a payload and the execution of that payload as a second chain. In an abstract view, this is also true for persistent data-only malware. While the first stage already contains the payload, it prepares the attacked system to *trigger* the second stage payload on certain hooks. For this, the first stage ROP chain is also called the *initialization chain*. The initialization chain needs to accomplish multiple steps. First it loads the second stage payload into a suitable memory location of the attacked process, that is not overwritten by the attacked process itself. Then, it alters the attacked system to regularly execute the second stage ROP chain. To remind the reader, this requires to load the virtual address of the payload into the stack pointer to load the chain (the stack pivot) and to execute a return instruction to redirect the control flow to the chains first gadget. In contrast to the first exploitation, where the attacker has a detailed view over the target and may have even control over multiple registers, there is only little influence to the current CPU state, once the second stage chain is triggered.

Up to Vogl *et al.*'s work the problem of executing callbacks in a return oriented rootkit was still an open problem [23]. Vogl *et al.* propose to leverage unused hardware mechanisms as a stack pivot. Amongst others, they propose to use the `sysenter` mechanism, a mechanism to trigger context switches from userspace to kernel space. For this, also the stack pointer and the instruction pointer need to be replaced in an atomic manner. On x86_64 systems, the `sysenter` mechanism is superseded by the `syscall` interface and is thus unused.

To use the `sysenter` mechanism to create a callback to the persistent part of the data-only rootkit, the initialization chain modifies the appropriate model-specific register (MSR) registers. These are namely the registers `IA32_SYSENTER_ESP` and `IA32_SYSENTER_EIP`.

The first register is set up to hold the address of the start of the persistent stage. The second register is set up to hold the virtual address of a `return` instruction. This way, every time a `sysenter` instruction is executed, the persistent stage of the data-only rootkit is executed. Thus to enable the callback during an interesting event, the control flow must be redirected to execute a sysenter instruction. For this, a pointer may be modified to point to the correct location. Note that this is orthogonal to the use of the system call mechanism, as this typically uses the `syscall` instruction.

After we have described the initialization of the malware, we will now discuss the requirements to the persistent portion of the malware. There are multiple problems, that an attacker needs to face in order to successfully implement such malware. The first problem, is that the control structure of the malware is on the stack. This means, that the executed instructions may also modify the stack and thus destroy the chain. This is especially a problem if parts of the chain execute external functions. A second problem is, that the malware may be triggered by multiple events at the same time. Thus the payload needs to be reentrant. A third major problem is, that the malware needs to restore the systems state after it finishes its execution. That is, not only must the malware store the current state of all registers, the registers need to be restored once the chain ends.

To comply with these requirements, Vogl *et al.* propose to separate the persistent chain into three different sub-chains. The *copy chain*, the *dispatcher chain*, and the *payload chain*. The copy chain is held to a minimum. Its only purpose is to save the current CPU state to a dedicated memory location that is already allocated by the initialization chain and to create a unique version of the dispatcher chain for each invocation. Thus each invocation of the dispatcher and the payload chain may destroy their chain while executing. In addition, the *copy chain* uses the fact, that the `sysenter` mechanism temporarily disables interrupts on the system. For this, it guarantees, different concurrent invocations of the callback do not interfere with each other. Lastly, the copy chain enables interrupts again and hands over control to the dispatcher chain. This next chain is

Figure 2.3.: This figure illustrates the tasks of the different ROP-chains required to create data-only malware. It is is originally taken from Figure 4 of [100].

then responsible to create a custom version of the payload chain by copying it to a new location in memory and modifying process relevant information. Finally, the task of the payload chain is to provide the malicious functionality and then to restore the original register state of the system at the time of the hook to solve the third problem described above. For this, it needs to access the information stored by the copy chain. A well-arranged summary of this process is given in Figure 2.3, originally taken from Figure 4 of the original work [100].

Note that the content of the original stack pointer is lost when leveraging the stack pivot mechanism described above. To solve this problem, the reference implementation makes use of the old frame pointer. As the callback used for the example rootkit is within the system call handler function, the implementation knows the offset between the stack pointer and the frame pointer at the time of the hook.

This has a big impact on defensive security mechanisms. Defenders not only need to implement code integrity mechanisms to ensure, that the code within a system is not malicious and can not be modified by an attacker. Also defenders need to be able to detect control structures for data only malware in memory by ensuring the integrity of the data memory that is contained within a system. This data can typically be separated into control flow relevant data like stacks and jump tables, as well as control flow irrelevant data. Vogl *et al.* [99] also showed, that it is possible to attack a system by only changing non-control data, as this data is also considered for control flow decisions.

Chapter 3

Related Work

As we have discussed the background relevant for this thesis in the last chapter, we will now introduce related work and discuss its influence on this work.

3.1. Code Integrity Validation

One goal of this thesis is to efficiently and securely validate the integrity of kernel code of modern operating systems which perform different forms of self-patching during both loadtime and more importantly during runtime. In the following we introduce different existing solutions to code validation and motivate our research effort in this direction.

Lots of approaches where presented to handle CIV. Many of these use dedicated hardware extensions, like Trusted Platform Modules (TPMs) or leverage smart-cards to execute code sections which deserve particular protection. In contrast, in this work we focus on mechanisms that can be applied to modern operating systems using off-the-shelf hardware. Another avenue to protect code integrity is the application of program obfuscation. However it was shown, that program obfuscation is not applicable in practice [9, 10]. In addition, obfuscation based program integrity mechanisms aim to defend the code against malicious modification and hope, that unauthorized modification leads to a crash of the program. In our work we aim to validate the integrity of program code, even if some level of self modification is applied.

3.1.1. Early Hash-based Approaches

One of the first available tools to check the integrity of software was Tripwire in 1994, which at its base, checks hashes of all files in the filesystem [52]. For this, Tripwire maintains a database of known files. For each file in the filesystem, Tripwire stores a hash of the file in the database. During runtime, all files in the system are hashed by Tripwire in a regular interval. The resulting hash is compared to the known hash in the database. While Tripwire also includes several non cryptographic CRC functions as supported hash mechanisms, the authors proposed to prefer secure cryptographic hash functions such as $MD5$ and $SHA1$. Although this approach is comparably simple, it is one of the first approaches, which tries to validate the integrity of executing code.

Later in 2001 a mechanism was introduced to check the integrity of code during runtime [21]. Chang *et al.* introduce guard code into a program which checks the integrity of specific code sections (defined by start and end address) within memory by calculating a checksum of the code region. In case of a checksum mismatch, the system modifies the stack pointer of the program. This should in turn lead to a crash of the protected program. To mitigate attacks targeted against the guard code, the authors propose to introduce a network of guards within the program, which in addition to the production code, also check the integrity of the guards themselves. However, not surprisingly, it was later shown that checking the integrity of an entity from within the entity itself is generally flawed, as the attackers code might also be executed with the same or higher privileges than the defense mechanism [105]. Note, that the malicious modification of the translation lookaside buffer (TLB) described by Wurster is not only invisible to the application that tries to protect itself, but is even invisible to the operating system where maximal privileges are assumed.

Copilot [69] was one of the first systems that calculated a hash of all Linux kernel and module text regions to detect rootkit modifications to code areas. To achieve this, Copilot makes use of trusted hardware that is capable of calculating and comparing the hashes at runtime. The "good" hashes, which are required as a basis for the detection of modifications, are thereby obtained by calculating the hashes in a system state that is considered to be non-compromised.

3.1.2. Hypervisor-based Hash-based Approaches

Garfinkel *et al.* [36] first propose to check the integrity of software through a secure and trusted hypervisor. *Terra* is designed to enable attestation for remote parties. For this, Terra calculates hashes of all files that are read from the virtual disk. The files are divided into smaller blocks (for example $4kb$, the size of a page in memory) and for each block a separate hash is calculated. To optimize the handling of large amounts of hashes, the authors propose to use a Merkle hash tree. A whitelist of hashes (or the root node of the Merkle hash tree) is then stored in the VM descriptor and thus secure from external access. As *Terra* validates the integrity during load-time, runtime modifications are not considered in this work.

Similar to Terra , *Pioneer* [84] aims in providing runtime remote attestation. In contrast to Terra, Pioneer does not depend on a trusted hypervisor. Instead, Pioneer calculates hashes of all kernel code pages and all static data pages. This way, the system is able to detect malware that modifies the kernel's code to include itself into the Control Flow Graph (CFG) of the kernel. Unfortunately, it also does not take runtime self-modification of an OS kernel into account, but still compares the calculated hash values for each page against a whitelist of hashes.

Petroni and Hicks [68] also compute a hash to validate kernel and module code regions, but move the validation component out of the guest system with the help of a hypervisor to increase the isolation of the validation component. To provide a base for the comparison, the authors make use of a trusted store that contains all trusted executables. To make this scheme work, binaries within the trusted store are relocated before the hash is compared based on their current location within the guest system. One problem with such an approach is that the set of possible hashes can grow arbitrary large during runtime. Other approaches that make use of hashes to detect kernel code modifications include Manitou [58], Blacksheep [11], and OScK [43].

SecVisor [83] further elaborates this idea, by leveraging the memory management functionalities of modern CPUs and entirely prevents modification of kernel code, once it is loaded to memory. This way, SecVisor is able to handle load-time modifications of the binary and ensures the code integrity of kernel code, but it prevents the kernel's optimization functionalities, by also disabling its ability of runtime self-modification. This policy also forbids to load custom modules into the kernel, as the code of a module in memory is different from the code of the kernel module within the object file on disk due to relocation. To allow loading of custom kernel modules, the authors decided, to implement the relocation functionality inside the trusted hypervisor. While this seems to be a good idea in the first place, it is even worse in practice, as it allows an attacker to trick the hypervisor to do arbitrary computations [86] based on artificially crafted relocation metadata. Note, that the problem of handling self-modifying kernel code was already mentioned by the authors of SecVisor, but not considered for their implementation, as *supporting self-modifying code is likely to complicate the approval policy of SecVisor* [83].

NICKLE [76] on the other hand redirects the instruction fetches to kernel code to a secure shadow version kept within the VMM. The most recent approach, MoRE [95], splits the TLB so that data accesses, such as write attempts, point to a different memory location then the actual code pages. While the specifics on how they enforce static OS code pages differs, these approaches all come with the same penalty: All these approaches consider kernel code to be static, once it has been relocated. While this assumption was true for older kernel versions, modern kernels make use of runtime patching. These legitimate kernel patching mechanisms, which are often there to improve performance, are disabled by these approaches. For this reason Ianus [61] advocates for only a partial enforcement against kernel patching where kernel modules are restricted from modifying code that does not belong to the module itself. Also Srivastava *et al.* [90] introduced a system to restrict untrusted modules to modify the code pages of the core kernel. However, without understanding the specific changes that happen to the main kernel code, such restrictions can be easily circumvented.

3.1.3. Handling code modifications

The first work that claims to handle self-modifying kernel code is *Patagonix* [57]. Equal to previous work, Patagonix leverages the isolation of a hypervisor to reason about the integrity of executable code within a monitored machine. Patagonix does not solely depend on a hash-based mechanism to identify code pages in memory. The authors argument, that due to different load-time modifications, such as relocation, many different hashes need to be managed for each binary file. For this, Patagonix extracts all locations from the binary, that are modified during load-time and reverts these locations during hash generation. This way, only one set of hashes (one hash for each $4kb$ page) needs to be generated and compared for each binary that is loaded to memory. Note, that changed locations that are reverted during hash generation need to be validated manually, to prohibit an attacker to use this unnoticed space to alter the control flow of the code in memory. In Patagonix two different types of Linux kernel self-modification mechanisms are handled. The first mechanism is the application of deterministic hardware dependent code modifications during load-time (called alternative instructions in this thesis). This is, the kernel may substitute instruction sequences with other, more efficient sequences, based on the current CPU, for example to support newer CPU features. While the authors state that alternative instructions are patched by the Linux kernel during runtime, they are patched during the kernel's loading process (*occur early during boot* [57]) before any userspace processes or external network communication is executed and are thus considered as load-time modifications in this work. The second runtime self-modification mechanism, the authors present is loading of new kernel modules. Also, as this does not change the already present the code of the Linux kernel but only adds new code, this modification is also not considered as runtime self-patching in this work. A more detailed discussion about the differences between our work and Patagonix is given in Section 5.7. Patagonix, however, shows the limitations of hash-based integrity validation and the necessity to be able to detect and validate kernel self-modification that is conducted after the kernel is loaded into the system.

3.1.4. Recent Approaches

Finally, Sprobes [39] is a recent system that aims to provide kernel code integrity by the use of the ARM TrustZone. This work again makes the assumption that kernel code is static, once the boot process hash finished.

Parallax [5] introduces a completely different CIV approach. Instead of comparing hashes of expected versions of code pages, Parallax validates the code integrity of a program by implementing the code validation component as a ROP program that uses overlapping gadgets within the instructions of the code to be validated. This way, once the code of the program is changed, the generated ROP code stops to function. Parallax converts some functions from the source code into semantically equal ROP code, so called *function chains*. By executing those function chains, the program automatically checks its code integrity and crashes, once the code is changed. During compilation the system thus does not only need to convert source code to an appropriate function chain, but also take care, that the rest of the code (1) contains enough gadgets to be used by the function chain and (2) contains usable gadgets in all parts of the code that need to be validated. Due to the nature of the approach, the protection can not cover the entire code integrity, but only about an average of 75% of the contained instructions [5]. Therefore, this approach seems very promising to protect specific important functionality within a program, such as anti-debugging features or license key validation. Still, it is unable to validate the code integrity of an entire program.

3.1.5. Summary

Unfortunately all of the existing solutions assume, that the codebase does not change during execution. The only dynamic modification that was also targeted are load-time changes. As these changes are conducted before the code is executed, these changes are also seen as static. For this, we analyze the importance to handle self modifying code within modern operating systems and introduce a framework to validate the code integrity of a modern operating system by leveraging the isolation and interposition properties of VMI. Thus we are able to validate 100% of the code, which also includes the runtime modifications conducted by the OS itself.

After the first publication of our work, we became aware of related work that also investigate the runtime self-patching capabilities of the Linux kernel [92]. Although that work discovers similar mechanisms, their approach and the results of their work differ from the results provided in this thesis. We describe the differences of that work compared to our work in detail in Section 5.7 on page 104.

3.2. Kernel Data Integrity Validation

Our second major research question was how to validate the integrity of kernel data memory. Our first approach was to extend on existing work that aims to build a map of objects within kernel memory.

Thus, in the following, we shortly describe previous efforts in this direction.

3.2.1. Mapping Kernel Objects

One of the first proposals to build a map of objects within kernel memory was *KOP* [18]. The basic idea of this concept is to start at a list of global symbols. With information about the layout of all datastructures at hand, each known object is scanned for pointers to other objects. Recursively, all objects within the kernel may be found. In their work, the authors propose a solution for two major problems during the map generation process. First, many objects contain pointers to other objects that do not name the correct type of the referenced object, but are instead of the type `(void *)`. Thus at this point the type of the next object is unknown and the mapping process must be stopped. Second, for union types it is unknown, which representation is currently active. This again permits to interpret the data correctly. To solve these issues, Carbone *et al.* propose to use a *points-to* analysis [4]. They statically analyze the kernel's source code and extract an extended type graph that contains all uses of a certain object. They extract a list of types to which an object is casted. The referenced object is then assumed to be of the extracted type. In addition, similar to this thesis Carbone *et al.* propose to check the integrity of all function pointers contained in the object tree found by KOP. In contrast to our approach, however, they only check if the pointer points to unmodified kernel code. They do not check if it points to the beginning of a function or an otherwise allowed symbol.

A revised object mapping approach called *InSight* was later also implemented by Schneider *et al.* [81]. In addition to the points-to analysis introduced by Carbone, InSight also makes use of *used-as* relations between objects. Together with a static rule set for frequently used types, Insight is able to find and identify a large part of the objects located in Linux kernel memory.

These solutions, however, still have problems to distinguish ambiguous types. In this thesis, we initially tried to extend on this concept by also taking constants of objects into account. The general idea was, that, for example, a union type may contain a constant member that indicates the current *used-as* type of the union. For this, we parse the kernel‘s source code to find constants. If a certain data structure, for example a member of a struct or a union, is set up in the kernel and only one or a small number of different constant values are assigned to that member we save both the type of the data structure, as well as the constant value. This information is then later used to sort out invalid subtrees from the generated kernel object graph.

During the mapping of a specific kernel instance, starting from global variables, we check for each identified object, if the identified object is an instance of a datatype that was identified to contain a constant. If this is the case, we check if the object actually contains this constant in memory. If this is also the case we assume to have found a valid object with high probability. On the other hand, if the object does not contain the constant, the identified object is not of the assumed type. Although we have found quite some structures within the Linux kernel that contained constant members, initial experiments showed, that the additional information could not significantly improve the quality of the resulting kernel object graph significantly.

Finally note, that the Linux kernel contains an object caching mechanisms, the SLAB allocator [13], that is used to allocate memory for frequently (re-)used objects. For example for process management, this allocator is used to allocate multiple process management objects (*struct task_struct*) at the same time. Whenever a new object is required by the kernel, the cache returns an unused object from the cache. Once the object is not required by the kernel any more, the corresponding object is marked as unused in the cache. Thus all allocated objects are then located in memory side by side. This holds an advantage for object validation, as important objects in kernel memory may be found by analyzing the management data of the

SLAB allocator. Thus building a complete type graph of the kernel data might not always be necessary.

While this approach helps to bridge the semantic gap, it did not help us to make valid assumptions about data integrity. For this, we set ourselves more in the direction of data integrity validation.

3.2.2. Semantic Data Integrity Validation

Rhee *et al.* [75] proposed an architecture that monitors each memory modification using a hypervisor. For each kernel object in memory, a set of corresponding functions is defined which is is allowed to make modifications, while all other code is not allowed to modify the object. During execution, for each memory access the corresponding kernel function is matched based on the current value of the instruction pointer. This concept still has multiple drawbacks. First, like the previous approach it requires to manually build a specification about the object—function mappings, which is cumbersome in practice. In addition, the work assumes that a malicious attacker would use dedicated debug functionalities like `/dev/kmem` or `/dev/mem` to modify kernel data. They admit, that their mechanism does not protect from modifications using code reuse techniques as they might use the correct instructions to modify an object. Lastly, this system requires to intercept each executed instruction. While this was not a major issue, as virtualization was mainly based on instruction emulation at that time anyway, today this introduces significant performance impacts.

The Linux Kernel Integrity Measurer (LKIM) [59] was another approach to validate the integrity of a Linux kernel. In their work they combine classical hash-based techniques (checking hashes of kernel code pages and important data structures as the System Call Table, the Interrupt Descriptor Table (IDT) and Global Descriptor Table (GDT)) with contextual inspection. For the latter, LKIM verifies that the validated data structures are actually referenced by the monitored system and that no malicious modified copy of such a data structure is used by the system. Note, that this work already proposed to simulate the loading process of kernel modules to create a valid baseline of allowed modifications to executable code sections introduced due to relocation. The proposed prototype was later, based on Stanley's work [92] on kernel code integrity, enhanced in cooperation with the Research Directorate of the National Security Agency to use a code integrity validation approach similar to the approach suggested in this thesis Chapter 5) [66]. In their conclusion the authors claim the approach is working as expected and is now used in several important governmental institutions: *Because LKIM does not rely on signatures of known malware, it is able to detect zero-day infections, making it ideal for countering the "advanced persistent threats" of concern to many of APL's [Johns Hopkins University Applied Physics Laboratory] sponsors. Together with the Research Directorate of the NSA, APL has developed LKIM from a concept to a prototype solution and is now working toward deploying LKIM in high-impact environments for our broader sponsor base* [66].

Petroni *et al.* [67] were the first to propose an architecture to detect semantic integrity violations in dynamic kernel data. In their work, they argue that the content of kernel data structures may be maliciously modified by attackers. An analyst thus needs to analyze and validate the invariant relationships among kernel objects, effectively proposing what we now call *Lie Detection*. They propose to build a formal model that describes the relationship between the different objects within the kernel. With this, the data could be automatically validated. While their idea is valuable, they do not create a sufficient model for the entire kernel. Still many integrity validation frameworks, as ours, manually validate the integrity of important kernel data structures as the ones described in their work.

Later Gibraltar [8] was introduced to automatically generate such specification invariants by monitoring a trusted uncompromised kernel and detecting values in kernel data memory that are static over the entire execution. Gibraltar in turn makes use of the Daikon System to generate these invariants [31]. Carbone *et al.* show that this specification based system misses up to 72% of the dynamic kernel data [18]. Still, this approach is orthogonal to the approach proposed in this thesis. We propose to combine our technique with a system like Gibraltar.

Another approach to data integrity is to detect and validate objects of a certain type in memory. Dolan-Gavitt *et al.* [30] propose to generate robust signatures for specific important data structures, in order to detect these data structures in memory even if they have been unlinked from management datastructures. In their example they created a signature for the Windows `EPROCESS` data structure, the Windows data structure that represents a process in the kernel.

OSck [43], a hypervisor-based integrity protection system, targets to *verify safety properties for large portions of the kernel heap*. For this, in-memory data structures are validated using out-of-band generated type information extracted from the kernel source code. Instead of building a map of kernel objects, OSck only focuses on selected, SLAB allocated, types of objects contained on the kernel heap and use a linear scan through the SLAB allocated memory for efficiency reasons. In addition OSck prevents the OS to modify hardware registers that contain pointers to kernel code and marks specific important data structures like the system call table as read-only. This is done in an effort, that an attacker is unable to leverage, for example the x86 debug registers or modify the pointer to the system call dispatcher within the corresponding MSR or modify the corresponding system call handler itself. Unfortunately, unlike Gibraltar, OSck does not automatically generate specification invariants about the contents of different data structures, but leaves the generation of these invariants to skilled kernel developers.

In addition, there also exists work like HookFinder [56] that keep track of all memory modifications made by a system. In case a modification is caused by a monitored process and the modified memory is later used to set the instruction pointer (e. g. in a call instruction), a hook is detected. Nevertheless, while more accurate in practice, this class of systems is meant for forensic analysis instead of continuous inspection and are thus seen as orthogonal research.

Finally, Szekeras *et al.* [93] introduced the concept of Code-Pointer Integrity (CPI), the requirement to enforce the integrity of code pointers in memory. An implementation of CPI that is based on memory splitting was then proposed by Kuznetsov *et al.* [55]. In their work they introduce a binary instrumentation framework that protects control flow relevant pointers. The basic idea thereby is to separate control flow relevant pointers into a separated space in memory and to limit the access to that area. Thus they split process memory into a safe region and a regular region, where the safe region is secured by the kernel and can only be accessed via memory operations that are *autogenerated and proven at compile time* [55]. However, Evans *et al.* [32] showed that restricting access to pointers in memory is not enough, because this separation can still be broken with the help of side channel attacks.

3.2.3. Summary

Many of the existing approaches go into the direction of building a map of all objects within the kernel and validate properties of the resulting graph. Other approaches try to build a set of static invariants about the content of these data structures and validate these invariants during runtime. To summarize, we find, that a general approach to validate kernel data is very hard to find. For this, we restrict ourselves to a subset of kernel memory and try to validate the part of data memory that is relevant for the kernels control flow. To increase the performance of continuous monitoring, we restrict ourselves to detect and validate only code pointers within data memory in order to detect malicious hooks and possible code reuse attacks. Later in this thesis, we introduce CPE, a technique to find and extract memory locations that may contain pointers from memory and check, if they only target allowed destinations.

Chapter 4

Control Flow Integrity and its Limitations

In addition to code integrity, securing the integrity of an operating system requires securing the integrity of the control flow of the monitored system. Researchers proposed different solutions to validate the integrity of a programs control flow. As CFI is related to the topic of this thesis, we discuss the different variants of CFI recently discussed in academia in the following. While the concept of CFI is sound in theory, we will show the practical weaknesses of CFI and discuss that CFI alone is not enough to secure the integrity of a modern operating system.

Note, that there is also published work in breaking the proposed CFI mechanisms in practice. As this thesis is more interested on the conceptional limitations of this technique, papers only showing novel attack methods against specific implementation weaknesses have been skipped in this section.

4.1. Coarse grained CFI

CFI was first formalized by Abadi *et al.* [2] in 2005 and an updated version of his work was published in 2009 [3]. The general idea of CFI is to build a complete Control Flow Graph (CFG) for the executing program in question ahead of time. In the CFG every instruction or basic block of the program is represented as a node, while edges exist for every possible transition, e. g. every valid branch, between different nodes. During execution, the CFI mechanism checks for each branch (direct call, indirect call, direct jump and indirect jump) if (1) both the source and the destination of the branch are valid nodes within the CFG and if (2) these nodes are also directly connected through an edge in the CFG.

This simple concept of CFI is sufficient to prevent a large number of typical attack patterns in practice. In this scenario, an attacker might still be able to overwrite a function pointer that is used within the program or a return address by using a simple stack-based buffer overflow. However, there does typically not exist a corresponding edge in the CFG of the program. Thus, coarse-grained CFI mitigates most of the classical *return-to-libc*-style attacks.

Note, however, that attacks that stay in the bounds of the allowed CFG are not prevented using this simple CFI implementation. There are multiple examples where this is possible. First, a function might legitimately call a function which is required by an attacker. In this case an attacker might be able to chain legitimate function calls within the program until he reaches the desired function. Second, for a specific node in the CFG multiple destinations may be allowed. This is especially a problem for return addresses, as CFI does not take the current system context into account. If there exists an edge in the CFG, the control flow change is allowed, even if it is not valid in the current system state. So a function may legitimately return to every function that calls the function in question, as the CFG contains a valid return edge for each of the calling functions. This still leaves an attack surface for an attacker. For this reason, Abadi refer to their proposed CFI mechanism as *coarse-grained* [2].

To solve this problem, Abadi *et al.* already propose to combine coarse-grained CFI with a shadow stack. A shadow stack is a redundant memory location, in which trusted copies of the return addresses are stored. This is used to validate the contents of the stack and hinder an attacker that is only able to modify the return address on one stack. A typical problem, however, is to also secure the location of the shadow stack, as an attacker is sometimes also able to modify this second location. Researchers have thus provided different techniques to implement a shadow stack and hinder an attacker to modify the trusted return values. These implementations either depend on the inability of an attacker to guess the memory location of the new shadow stack or to separate the shadow stack by leveraging dedicated hardware features. An example for the latter is segmentation [87] in the simple case or more advanced features like virtualization and Intel's performance counters [98]. It is also possible to leverage the security of dedicated security peripherals like ARMs TrustZone or Intels SGX. Alternative approaches, that encrypt the return address with a random cookie have also been proposed [35].

Over time many coarse-grained CFI solutions [62, 63, 109, 108, 25, 26] have been proposed in academia. In addition to approaches that aim to secure userspace binaries, CFI solutions emerged that focus in securing the control flow of operating system kernels [68, 51, 26] and hypervisors [103, 38]. A common problem of CFI is that all branches need to be validated during runtime. While direct branches can be validated statically, indirect branches need to be validated dynamically. This is typically achieved either by compiler-based modifications to the binary or by binary-instrumentation. Thus, CFI solutions suffer from a significant performance overhead (as high as 21% in its first implementations [3]). As a result, solutions were proposed, that limit the number of checks to increase the performance without a significant loss in the precision of the approaches. *kBouncer* [62] for example only checks the control flow integrity of a program when a system call is executed. At this point the last 16 indirect branches are checked for integrity. The constant 16 thereby is given by the use of Intels Last Branch Recording feature [46], which when enabled stores the source and destination addresses of the last 16 taken indirect branches.

Davi *et al.* [29] have analyzed the current state of the art of coarse-grained CFI solutions in 2015. They showed, that while reducing the performance overhead to a usable range, all of the proposed systems are relaxing the originally proposed CFI policies and thus are not sufficient to defend against code reuse attacks. In fact they even provide less security to gain better performance. In addition, systems made new assumptions about different classes of indirect branches, for example, that a certain number of instructions has to be executed between different branch instructions. It was shown, that even using the strongest assumptions made, it is still possible to create data-only programs, which only consist of gadgets, that are valid destinations in the control flow graph. This subset of gadgets still allows Turing-complete computation and real-world attacks can still be launched. Even with strict enforcement policies in place. The authors claim that all examined systems were broken “having access to only a *single* — and common used system library” [29]. A key observation for this problem is the fact, that the soundness of the initial CFI proposal is directly related to the quality and granularity of the generated CFG. While this CFG does not take the current state of the program execution into account, successive solutions even relaxed its granularity to gain performance benefits. Similar research was conducted by Goktas *et al.* [40].

Another problem of CFI is the generation of the CFG for a given program. As this is usually done for performance reasons prior to the execution of a program using binary analysis, it is not possible to take the current software state into account (e. g. a called function has to return to the instruction following the branch instruction that initially called the function). In addition, there are much more fundamental problems in generating a CFG. The Linux kernel for example provides a core infrastructure for its developers, that is similar to the functional range of the standard library for userspace programs. This core infrastructure also contains functions, that take a function pointer as argument and then during execution call this function pointer as a callback. We will describe these issues in more detail later in this section. Due to these conventions, it is sometimes impossible to make a priori assumptions about the destination of a specific call instruction in the CFG. Thus an attacker might use such a function to hide the usually forbidden control flow transfer.

In an attempt to solve the previously discussed issues with coarse-grained CFI, research started to focus on two distinct sub problems: (1) forward-edge CFI and (2) backward-edge CFI. While the latter problem is solvable by securing the stack and implementing a shadow stack, the first problem is harder to deal with.

4.2. Fine Grained CFI

Fine-grained CFI aims to minimize the amount of valid targets for the forward edges in the CFG. The proposed techniques thereby focus on restricting indirect jumps and indirect calls.

One major part of forward-edges in the control flow of a program are indirect calls that use function pointers that are stored in memory. An example of such functions pointers for programs written in *C++* are *vtables*. Vtables list the methods that can be called for a certain object. For this, each instance of an object contains a pointer to its corresponding vtable. Once a method of an object is called, the program loads the corresponding vtable and uses the function pointer at the specific offset to call the intended functionality.

An attacker can leverage this fact by either changing the pointers for existing vtable entries or by creating fake vtables in memory. In the latter case, the attacker also has to create fake objects in memory which, instead of the pointer to the valid vtable, contain a pointer to the fake vtable object.

4.2.1. Forward edge validation

To mitigate these attacks, researchers proposed solutions that aim to secure and validate vtables of C++ objects using the available source code. Since there was a lot of research in this specific area, we will only introduce some of the latest solutions and highlight the core ideas that where proposed.

4.2.1.1. Forward edge validation for C++ vtables

Jang *et al.* [49] first proposed *SafeDispatch.* SafeDispatch is an enhanced C++ compiler built on top of the LLVM compiler infrastructure. During compilation, SafeDispatch analyses the source code to generate what the authors call a *class hierarchy analysis.* In this analysis, SafeDispatch generates *the set of valid method implementations, that may be invoked by an object* of a specific type. This set contains all methods of the specific class, as well as all functions from classes in its class hierarchy. It then inserts validation code into the resulting binary, which, during runtime, checks, if a method invocation of an object calls a valid function according to the previously generated set. This is similar to a whitelist of functions that may be called. This way, an attacker is unable to arbitrarily change a pointer in an objects vtable, but is restricted to a previously defined set of targets.

Later in the same year, Tice *et al.* [94] published their work on compiler extensions for both the GCC, as well as the LLVM compiler. Due to the different architectures of both compiler frameworks, Tice *et al.* implemented two different approaches (*Virtual-Table Verification (VTV)* for GCC and *Indirect Function-Call Checks (IFCC)* for LLVM. The first approach (VTV) is equal to the method that was implemented by SafeDispatch. During compilation VTV creates *vtable-map variables*, which contain all valid vtable pointers for each polymorphic class. These variables are stored within a special read-only section of the resulting binary. Therefore, it is possible to merge different sets, when the source code is compiled incrementally.

During runtime VTV checks for each indirect call, if the executed function pointer is contained in the previously generated set. In contrast, IFCC *operates on the LLVM IR during link-time optimization*. It generates *jump tables* for each indirect call target and adds code for each indirect call to use these jump tables. Also IFCC stores the jump tables in read-only memory, making it impossible for an attacker to maliciously change the call target. The authors managed to upstream their work into the source code of both of the compilers. Thus their work is widely available and usable for real-world applications. The usability of their work was evaluated by compiling the entire ChromeOS project. Still these approaches incorporate a high performance penalty.

In the mean time, two solutions for this performance issue have been proposed: Bounov *et al.* [15] propose to increase the performance by replacing the costly set membership tests conducted by both SafeDispatch and VTV with a simple range check. Therefore, they propose to restructure the layout of the generated vtables, to not only contain all function pointers for a single class, but to combine all vtables for an entire class hierarchy into one larger vtable. This is achieved by both *VTable ordering (preorder traversal of the class hierarchy)* and *VTable Interleaving*. Depending on the type of the object in question the valid range within the vtable can be adjusted by the compiler.

Zhang *et al.* [107] proposed *VTrust*. VTrust's solution to minimize the runtime overhead is not to check the set membership during the invocation of an indirect call, but instead to check the RunTime Type Information (RTTI) of the target function against a value given at compile time. During compile time, the RTTI is merged into a unique 4-byte value, such that the comparison is reduced to a simple integer comparison. In addition VTrust applies *VTable Pointer Sanitization* to ensure the validity of vtable pointers. For this, VTrust maintains a list of all function pointers used by the program. This list is generated during compile time. In addition the vtables do not contain function pointers, but instead only contain an index to the corresponding function pointer within the previously generated whitelist. VTrust adds code to the binary, which decodes the function pointer directly before the indirect call. This way VTrust tries to hinder an attacker to introduce custom handcrafted vtables into a program.

An alternative approach, Opaque CFI was implemented by Mohan *et al.* [60]. In their approach, the authors combine CFI with automated software diversity. Automated software diversity aims to regularly re-randomize the memory layout of an application. This re-randomization may either be conducted by the compiler, by the loader during loading of the application or even during program execution. Opaque CFI relies on load-time re-randomization to secure the control flow of an application. In addition, similar to IFCC, Opaque CFI introduces jump tables into the code that point to valid call targets for each indirect branch target. During execution, for each indirect branch, only a specific range within the jump tables are allowed for each branch. Due to the re-randomization, both the content of the jump table as well as the allowed ranges change for each program invocation.

To summarize, most of the effort in securing the forward edge of the control flow was conducted to secure the vtables of object oriented languages. To restrict branches to a valid forward edge, call targets are classified according to their type (e. g. RTTI) or their class hierarchy. In the worst case however only a small number of classes is generated, which again allows a large number of call targets.

4.2.1.2. Limitations of Forward edge validation

Carlini *et al.* [19] and Evans *et al.* [33] independently showed, that fine-grained CFI still does not provide enough security to completely hinder control flow attacks. Carlini *et al.* argue, that fine-grained CFI is breakable by *using just calls to the standard library [19]* and the deployment of shadow stacks is necessary in practice. Evans *et al.* go a step further and argument, that fine-grained CFI *is ineffective in protecting against malicious attacks* [33].

This is the case as large software projects (as Apache and Nginx in their experiments) intentionally *use coding practices that create* *flexibility in their intended control flow graph (CFG)* [33]. Examples for such constructs are higher-order functions. Higher-order functions are used to increase the modularity and flexibility of source code. In *C* this concept is used to create structures (structs) that behave like objects in *C++*. In *C++* techniques like vtable validation have been proposed, as described before. While such defense techniques are hard to implement in *C++*, they can not be implemented in programming languages that lack the concept of object orientation, like *C*. This is because, the relation and type of the involved structures and functions is not known by the compiler. For example, imagine a construct where two pointers are passed to a function. The function uses the first pointer as a function pointer, which is then called and the second pointer is passed to the called function as an argument. Another example are structures that contain functions to operate on a specific object. During this thesis such constructs were also found in the Linux kernel and will be introduced in detail later in this section.

4.2.1.3. Binary-only Forward edge validation

To tackle this problem, Veen and Göktas *et al.* [97] proposed *TypeArmor*. *TypeArmor* tries improve on the current research activities by proposing a more fine-grained CFG generation technique for binary-only CFI. The authors propose a method which they call Control Flow Containment. This approach classifies indirect function calls as well as every function concerning to two different heuristics. For indirect function calls, *TypeArmor* checks how many parameters are prepared and if the caller expects a return value from the callee. Equally, for every function, *TypeArmor* analyses, how many parameters are used by the function and if the functions provides a return value. With this information, *TypeArmor* instruments the code in such a way, that indirect calls are only allowed to target functions, which use at most the same number of parameters than the number of parameters prepared by the calling function. In addition a call site expecting a return value may not call a function that has an inferred return type of *void*. For callees, that expect less parameters, than set by the caller, *TypeArmor* scrambles the unused parameters by overwriting the corresponding CPU registers with random data. For function calls that do not expect a return value, the return value, if set by the callee, is also scrambled. Unfortunately, the evaluation shows that only about 20% of the functions returning *void* can be identified as such.

Notice, that this approach does not hinder a successful CRAs, but is *likely to crash* as the registers are not valid any more. While this work improves on current research on fine-grained CFI, it still has one major drawback. It assumes, that the protected program does not employ self modifying code. Therefore, in its current form, it can not be applied to kernel software. In addition, while talking about variadic functions, the authors did not talk about intended callback functions, which take a pointer to an array of arguments as their input parameter. While *TypeArmor* is able to defend against a large class of attacks called *COOP* attacks [82], it does not defend against pure data-only attacks as the one proposed by Carlini *et al.* [19].

4.2.2. Backward edge validation

In contrast to the previous approaches, Davi *et al.* [27] propose a hardware architecture to secure backward edges of the control flow. Their solution does not require a dedicated shadow stack and promises increased performance. The idea of their solution is to introduce two new CPU instructions. The first instruction `CFIBR label` is inserted at the start of a function, while the second new instruction `CFIRET label` is inserted after each call instruction in the binary. With these instructions in place their approach works as follows: When ever the program executes a call instruction, the next instruction is required to be an `CFIBR` instruction. With this, the system checks, that a call instruction can only target the beginning of a valid function. Note that additional measures should be applied to restrict the number of allowed target functions. More important, the `CFIBR` instruction also contains a label, which on invocation of the instruction is pushed to a dedicated *label state*. Now once the function hits a return instruction, the return instruction pops the current label from the stack (and eventually checks it against a label encoded into the return instruction). After a return instruction, the next executed instruction is required to be a `CFIRET` instruction, which checks if the current active label on the *label stack* is correct. This way, a function is only able to return to the correct caller. Unfortunately, the system was only described, but not implemented in practice.

4.3. CFI for Kernel Software

As we have now introduced important related work in the direction of CFI, we will in the following introduce research that specifically targets CFI for kernel software.

KCoFI [26] is an example for a coarse-grained CFI framework, that provides CFI for a commodity operating system. KCoFI was implemented for the FreeBSD kernel. The core implementation of CFI is based on previous work from Zeng [106]. The authors describe *KCoFI* as follows: *KCoFI does not attempt to compute a call graph of the kernel. Instead, it simply labels all targets of indirect control transfers with a single label.* As a result, their design uses a *very conservative call graph* [26]. With this KCoFI only restricts indirect call instructions to call entire functions. It does not restrict the set of allowed functions. In addition they also claim to protect the kernel from *ret2user* style attacks, which can also be achieved by the consequent use of SMEP. More importantly, the authors focus on handling *low-level state manipulations performed by the OS*, like trap handlers, context switching and signal delivery. So they hinder malware to modify the saved CPU state for execution contexts that are currently not scheduled. For this, the authors include a layer between the OS and the processor that handles these state manipulations. Instead of trusting the OS internal datastructures, KCoFI stores the entire CPU context of each executing thread into a special memory region on each context switch. This region can only be modified by their security layer and can not be modified by the operating system. Thus, the control flow can not be changed in a malicious way during context switches.

The general problem of hardening context switches within the kernel is a problem for CFI in the context of operating system kernels. KCoFI uses a heavy-weight mechanism to ensure the integrity of saved CPU state, which makes its general applicability unlikely due to its high performance overhead. The overhead could be reduced, if the integrity of the corresponding kernel internal data structures can be ensured. Still the problem to restrict indirect calls during normal execution was not discussed by KCoFI.

Another attempt to apply CFI to kernel software was proposed by Ge *et al.* [38]. In their work they try to apply fine-grained CFI to the FreeBSD kernel as well as the MINIX microkernel. In addition they partly apply their approach to the BitVisor hypervisor. This work has some major drawbacks. The first problem, that is partly addressed in the paper, is that this work does not support preemptive kernels. They do not support kernels, that interrupt normal execution in favor of an exception with higher priority. The reason for this choice is, that (1) it can not be predicted, when such preemption takes place and (2) it is not possible to verify the return address from such an exception, as it may point to any kernel address.

The second, more fundamental problem, is that this work is based on two assumptions, which we will shortly discuss in the following:

(A1) *The only allowed operation on a function pointer is assignment.*

(A2) *There exists no data pointer to a function pointer.*

The first assumption is valid, but as we will show, the second assumption does not hold in practice. While this assumption might be true for the kernels that where analyzed by the authors, this is not the case for the Linux kernel, that was analyzed within this thesis. The authors claim, that function pointers might be contained within existing structures, they claim no direct data pointer to a function

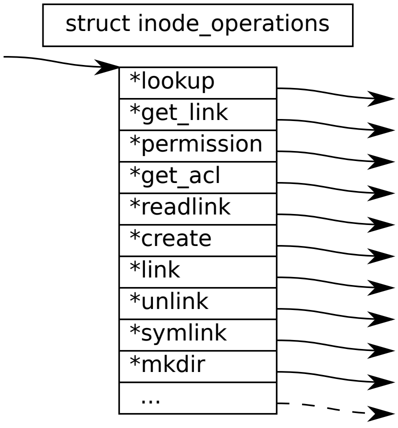

Figure 4.1.: Beginning of the contents of *struct inode_operations* from Linux 4.5. A pointer to this struct is also a data pointer to a function pointer, thus violating assumption *(A2)*.

pointer exists. For the Linux kernel, this is the case for function pointers that are the first member of a structure. We will now show examples that violate assumption (A2) extracted from the Linux kernel. One type of examples for violations of (A2) can be found in the file system related structures like *struct inode_operations* and block device related datastructurs like *struct block_device_operations* in the Linux kernel. These structures contain a set of functions pointers that can operate on inode objects or block_device objects. This way, the kernel code emulates object oriented programming. The beginning of *struct inode_operations* is depicted in Figure 4.1.

Table 4.1 provides an overview of such object operation structures taken from the Vanilla Linux 4.5 kernel. The table shows the name of 35 vtable like objects, as well as the source file where each structure is defined. Note that this table only lists such structures which name contains *_operations*. Our tests show, that there exist 290 structures within the Linux 4.5 kernel that are called **_ops* and only contain function pointers. A table that lists all these structures can found in Table A.1. We did not spend more time on to research if there are more structures like this, which do not adhere to such a systematic naming convention. While not all of these structures have a function pointer as first member, this example clearly shows that assumption *(A2)* does not hold for the Linux kernel.

Note that these objects are all similar to vtables that where already described and handled in different approaches and described earlier in this chapter. They contain pointers to functions that handle certain functionality for a given object. However, in contrast to the vtables that were also discussed earlier in this section, there does not exist a reference (pointer) between the object, on which the functions in the vtable are used and the vtable object itself. Thus it is not possible to handle these datastructures as it is possible for C++ vtables. For this we assume that CFI can not easily be applied to kernel software. In addition, there is currently not even a generic way to extract all vtable like objects and relate them to their corresponding objects.

Even more, the Linux kernel contains functions, which explicit purpose is to call lists of function pointers that are passed to them as arguments. One example for such a construct is the function *notifier_call_chain()* in `kernel/notifier.c`. The *notifier* mechanism is also used for the kernel's *tracepoint* mechanism. The tracepoints mechanism allows kernel developers to register arbitrary functions during runtime, that are executed whenever the kernel execution hits a certain tracepoint. Constructs like these make it very hard, if not impossible, to implement efficient fine-grained CFI for modern operating system kernels such as Linux or Windows.

For this reason, in this thesis we assume, that practical CFI can not be achieved in an modern OS kernel. Therefore, we try to provide mechanisms for both code integrity as well as code pointer integrity for modern operating systems. We provide a mechanism to detect return oriented programs and other means of code reuse by detecting the gadget chain within the memory of the already compromised system.

Finally, as already proposed by Abadi *et al.* [2], Intel has recently taken up the discussion about CFI and announced the introduction of its own solution to the problem[1]. In their implementation, the processor expects a certain instruction (*ENDBRANCH*) at any location that is allowed to be the target of a branch [47]. In contrast to the proposal made by Davi *et al.* [27], Intel does not differentiate between call, return or jump branches and also does not contain a label. This might have two reasons. First, including a *label state* is effectively introducing a new type of stack, which also requires management and second, differentiating between different types of branches might introduce compatibility problems, with features like tail recursion or tail call optimization. Although this is only a very coarse-grained implementation of CFI, with only one label for all target instructions, it is expected to reduce the attack surface in practice.

4.4. Summary

In summary, a lot of research effort was put into validating and enforcing the control flow integrity of both userspace applications and kernel software. While the initial approach suffers from a high performance penalty, over time many ideas have been proposed to solve the performance problem without creating incisive security problems. But despite the different solutions, researchers showed that even by using the strictest, fine-grained CFI policies it is not always possible to detect all control flow violations. In addition, current software design and kernel implementation also hinders the application of proposed CFI mechanisms in practice. For this reason, this work does not focus on CFI and on mitigating attacks but instead, we focus on analyzing and detecting the integrity of an operating system even after the system was compromised. We think that by

[1] `https://software.intel.com/en-us/blogs/2016/06/09/intel-release-new-technology-specifications-protect-rop-attacks/`

combining our approach with CFI, an attack may be detected, even if it could not be mitigated in advance.

Recent work in the direction of CFI also starts to look into this problem [110]. Note that our work on userspace code and data integrity goes in a similar direction, but instead of relying on a trusted preshared library that is stored within the protected OS, we base our security model on VMI. We think that it is very important to be able to detect advanced code reuse attacks on an already compromised machine. The techniques proposed in this thesis build a fundamental requirement for such detection.

struct name	**defined in**
address_space_operations	include/linux/fs.h
ata_port_operations	include/linux/libata.h
block_device_operations	include/linux/blkdev.h
ceph_connection_operations	include/linux/ceph/messenger.h
configfs_group_operations	include/linux/configfs.h
configfs_item_operations	include/linux/configfs.h
dentry_operations	include/linux/dcache.h
dquot_operations	include/linux/quota.h
efivar_operations	include/linux/efi.h
export_operations	include/linux/exportfs.h
fc_rport_operations	include/scsi/libfc.h
file_lock_operations	include/linux/fs.h
file_operations	include/linux/fs.h
fmc_operations	include/linux/fmc.h
inode_operations	include/linux/fs.h
kobj_ns_type_operations	include/linux/kobject_ns.h
lock_manager_operations	include/linux/fs.h
media_entity_operations	include/media/media-entity.h
media_file_operations	include/media/media-devnode.h
oprofile_operations	include/linux/oprofile.h
page_ext_operations	include/linux/page_ext.h
parport_operations	include/linux/parport.h
pccard_operations	include/pcmcia/ss.h
pernet_operations	include/net/net_namespace.h
pipe_buf_operations	include/linux/pipe_fs_i.h
posix_clock_operations	include/linux/posix-clock.h
proc_ns_operations	include/linux/proc_ns.h
psci_operations	include/linux/psci.h
qtree_fmt_operations	include/linux/dqblk_qtree.h
seq_operations	include/linux/seq_file.h
super_operations	include/linux/fs.h
tty_operations	include/linux/tty_driver.h
tty_port_operations	include/linux/tty.h
usb_mon_operations	include/linux/usb/hcd.h
vm_operations_struct	include/linux/mm.h

Table 4.1.: Example for object oriented constructs in the Linux kernel. These structures are equivalent to vtables in C++. Note that this is only the subset of structures that adheres to the naming convention **_operations*. There are 290 structures that are called **_ops*. See Table A.1 for details.

Chapter 5

Runtime Kernel Code Integrity

In this chapter we present an approach for syntactically validating the integrity of kernel code with the use of semantic (binding) information. With this research we give answers to the research questions **Q1** and **Q2** raised in the beginning of this thesis. By leveraging virtual machine introspection, we examine all kernel code pages at runtime to validate their contents and to reconstruct the active system state. By emulating the OS's patching mechanisms, our system is able to successfully differentiate between malicious and benign code changes. We demonstrate the ability to detect malicious kernel code with a set of rootkit samples. Our method does not restrict modern OS kernels from using otherwise benign patching routines. To further highlight the importance of practical kernel code validation, we also present a critical security issue in the Linux kernel that we discovered in our research which thus far remained unnoticed. Parts of this chapter have already been published in the paper "Code Validation for Modern OS Kernels" [54].

5.1. Problem statement

As we have shown in Chapter 3, detecting changes to the kernel code base up until now generally comes down to validating the integrity of the kernel's code region with a whitelist of known good states. In previous research ([58, 76, 84, 68, 83, 11, 43]), this is typically achieved by calculating the hash of each executable page and comparing it to a whitelist of hashes that were calculated beforehand in a secure environment. If a hash is calculated that is not contained in the whitelist, one can deduce that either existing code was changed or new code was introduced into the system.

While approaches that do not depend on hash-comparison, such as MoRE [95], have better performance and can ensure that code-pages remain static during the execution of the OS, they prevent the kernel from applying otherwise benign optimizations at runtime. Considering that the Linux kernel has recently introduced new features such as JIT code and dynamic security patching, the short-comings of these approaches will become even more limiting in the future.

In this chapter we describe the different self-patching mechanisms employed in the Linux kernel and show that they could be abused by an attacker to modify kernel code pages in a manner that are particularly difficult to detect. We will show that simple hash-based approaches or approaches that only make use of non-binding information and are not taking the current system state into account are no longer sufficient to perform code validation for modern kernels.

Solving this challenge requires a deeper understanding of the kernel's various load and runtime self-patching mechanisms. In fact, there are several mechanisms for which the integrity and consistency of a change may be validated but the resulting state still may be malicious. We also present a novel method to perform runtime kernel-code integrity checking that addresses the short-comings of existing systems. Based on this knowledge we will present a proof of concept (POC) implementation that is capable of not only validating the integrity of kernel code at runtime, but also its identity. That is, the prototype is able to attribute kernel code pages found within memory to the kernel binary itself or to a loaded module. Since our prototype makes use of VMI, it cannot be evaded and remains functional even if the attacker compromises the monitored system.

As our experiments show the proposed prototype is not only very effective against kernel-level malware, but also very efficient (0.54% overhead in the worst case), which makes it well-suited for real-world applications. Although we focus on Linux within this chapter, we argue that many of the challenges that we discovered are also adaptive to other OSs such as Windows or MacOS.

In summary, in this chapter, we make the following contributions:

- We show that current code validation techniques are not suitable to validate the code integrity of modern kernels.
- We examine various load time and runtime code patching techniques employed by modern OS kernels.
- We discuss the challenges that these runtime code patching mechanisms create for code validation.
- We demonstrate the importance of correctly validating modern kernel code. We do this with a practical example that enables an unprivileged *user* to load arbitrary executable code into the Linux kernel.
- We introduce a framework that can successfully validate the integrity and identity of dynamic kernel code and enforces additional security constraints.

5.2. Kernel Runtime Patching

In this section, we present constructs and mechanisms within modern OS kernels that make simple hashing techniques ineffectual. Our investigation is primarily based on the Linux kernel (Version 3.8, 64-bit and Version 3.16, 64-bit), however, some constructs also apply to the Windows kernel as well. We begin by taking an in-depth look at the load time patching mechanisms that the kernel uses and then discuss commonly used runtime patching mechanisms in more detail.

5.2.1. Position Independent Code

The first mechanism that makes plain hashing difficult is position independent code. While the virtual start address for the kernel itself is still fixed at compile time, the load address for each module is dynamic. This is due to the fact, that the kernel cannot guarantee a fixed address region for each module. Thus, the compiler can not predict the addresses of external symbols at compile time and the addresses are unknown until load time. Therefore, the necessary addresses are patched during the module's loading process. This is known as relocation. The relocation information is contained within each module binary. For each segment within the module a relocation table lists all references to both internal and external symbols. As the Windows kernel has a similar issue in that it cannot reserve unique space for every driver that may ever load, Windows drivers and the driver loader employ similar techniques.

It is clear that such a mechanism would hinder simple page hashing techniques, unless the initial hash is taken after the driver is loaded and running. In this case, however, we cannot be sure that driver is not already infected or altered. The common solution to this is to take the initial hash in a "secure environment", however such an approach will lead to other complications as we will describe in the following section.

5.2.2. Configuration-specific Patching

In addition to patching the modules for external symbols as described above, the Linux (module) loader may also replace code with architecture and configuration-specific opcodes or code blocks that are only present in certain configurations and are replaced with no operation (NOP) instructions otherwise. In fact, such patching takes place in the kernel code as well. This is done to improve performance, leverage special features such as *symmetric multiprocessing (SMP)* or (para-)virtualization on architectures that support them, or to accommodate debugging or tracing to provide extensibility. With these mechanisms the kernel can even replace entire functions. Such patching not only takes place in kernel modules, but in the kernel code as well.

In the following, we present the four cases in which such patching takes place in more detail. All of them have in common, that they patch an instruction and overwrite the rest of the available space with *NOP* instructions. The kernel unfortunately does not only use the straight forward 0x90 instruction as a *NOP* instruction. For performance reasons it also uses multi-byte instructions like: `xchg ax, ax` (0x60, 0x90) or `nop dword [rax+rsi*4-0x40]` (0xf, 0x1f, 0x44, 0x0, 0x0). Unfortunately, the exact multi-byte instructions used are also specific to the processor architecture and type currently used. Also, to patch code during runtime, the kernel does not need a special synchronization method. Instead, it first replaces the first byte of buffer with an *int3* instruction, thus every CPU trying to execute this instruction will be trapped. Then the rest of the space is filled with the new content. As a last step the kernel replaces the first byte with the right instruction and notifies all waiting CPUs. To even further improve the performance, the kernel sets the interrupt handler function to the address directly after the patched buffer. Thus the *int3* instruction is effectively changed into a NOP instruction.

5.2.2.1. Alternative Instructions

For certain functionalities, the specific opcodes used by the kernel vary depending on the available CPU's feature set. Mostly, this approach is used when later CPU models support more efficient instructions. For every instruction sequence that should be replaced, a list of alternative instructions is provided with each kernel binary, ordered by the most preferred as last in the list. The Linux module loader replaces each instruction if the requested feature is supported by the CPU. This feature is mostly used, when a new CPU feature is introduced and the kernel implements support for the new feature. In this case, the kernel adds `nop` instructions into the compiled binary code, that are then replaced by the required instructions during load-time (e. g. to enable/disable SMAP or to synchronize memory accesses). In other cases, the operand of a `call` or `jmp` instruction is changed in order to execute different code during runtime. However, also some cases exist where entire kernel functions are replaced entirely with this feature. Notice that this is the only configuration-specific patching mechanism that is considered by Patagonix [57]. To be able to validate alternative instructions, we have to extract that specific information from the monitored guests memory or generate it based on the knowledge we have about the used hardware.

5.2.2.2. Hypercalls

Another situation in which code is commonly patched within the kernel is related to virtualization, in the form of hypercalls. Hypercalls are analogous to system calls, where a call is passed from a userland process to the kernel, in a virtualized environment which enable the guest system to interact with the hypervisor. Examples of such hypercalls include enabling and disabling interrupts, writing to specific processor registers, and MMU related functions.

As with relocation, hypercalls can not be inserted during compile time, as the kernel or its modules do not know if they are executed on real or virtualized hardware. Furthermore it is unknown which virtualization technology is used (e. g. XEN, KVM, etc.). Thus, the compiler provides hints in the compiled binary that a specific function (e. g. writing to the CR3 register, which holds the page tables for the currently executing process) is requested at a specific location in the kernel code. During load time the kernel is then able to decide how the requested function is provided and inserts the appropriate functionality. This can be done by inserting the corresponding opcodes directly, calling a function provided by the hypervisor, or even by jumping to a predefined location.

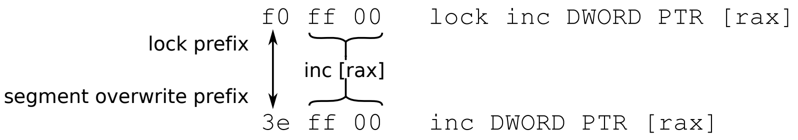
f0 ff 00 lock inc DWORD PTR [rax]
lock prefix
inc [rax]
segment overwrite prefix
3e ff 00 inc DWORD PTR [rax]

Figure 5.1.: Implementation of SMP in practice.

To facilitate this mechanism, the kernel maintains a table of alternatives introduced by a hypervisor-specific driver that can be used as a replacement for a given instruction. This table must adhere to the kernel's `paravirt_patch_template` interface. This feature is even employed in a non-virtualized or full-virtualized environment. For full-virtualized environments, the kernel provides the native implementations for the requested functionality.

The kernel also suffers from another virtualization related problem. To enable userspace applications to use the correct system call mechanism the kernel maps a common page into every userspace process called the `vdso` page, containing functions that wrap the kernel's current system call interface. Equal to this mechanism, a hypervisor can also map a custom page into the memory of a guest VM.

The patching mechanisms described so far are all conducted at load time. A common solution to these load time patching mechanisms is to take the initial hash after all modules have been loaded such that relocation is no longer an issue. However, this approach has a fundamental flaw: we cannot be sure that kernel code has not already been altered when the hash is calculated. Next we will describe patching mechanisms that are applied during *runtime*.

5.2.2.3. Symmetric Multiprocessing Locks

SMP describes an architecture with multiple CPUs that share memory. This is very common in modern PCs. There are portions of the kernel code that become critical sections (i. e. they share data that should only be accessed in an asynchronous, mutually exclusive fashion) and only when SMP and multiple CPU cores are enabled. In this case, the critical section must be protected with locks. However, in the interest of performance the kernel chooses to patch in these locks only if it recognizes it is operating in an environment in which more than one CPU is present. An example for such a modification is shown in Figure 5.1. This makes sense as the locking and unlocking operations are computationally expensive tasks and become unnecessary if only one CPU is active.

Furthermore, the Linux kernel supports enabling and disabling CPUs at runtime. This results in such patching also taking place at runtime. Note that adding and removing CPUs in a virtualized environment is frequently used for scalability. In addition to SMP locks it is conceptually also possible to patch arbitrary other instructions, in a fashion similar to alternative instructions, during runtime, once the number of active CPUs changes.

5.2.2.4. Jump Labels

The Jump Labels mechanism is used within the kernel to optimize “highly unlikely” code branches to the point that their normal overhead is close to zero. Instead of checking whether a branch should be taken or not every time the control flow reaches a specific point, the kernel either replaces the conditional jump with a NOP instruction, and thereby omits the unlikely code, or with an unconditional jump to the unlikely code. Thus, the enabling/disabling of the function is an expensive task, but the runtime cost is completely avoided. An illustration for a jump label is shown in Figure 5.2 on page 76. Figure 5.2a shows the source code for a jump label in the kernel. During normal execution, the jump label is deactivated. Thus the code in memory looks like represented by Figure 5.2b. Once the unlikely condition is fulfilled, the code in memory is replaced to the state shown in Figure 5.2c. Although this feature was mainly intended for debugging and tracing features, this mechanism is now frequently used both in the Linux scheduler and in the networking subsystem. For example, the latter uses it to activate and deactivate certain netfilter hooks. The current state of each jump instruction is also maintained within a kernel data structure.

5.2.2.5. Function Tracing

Another mechanism that requires runtime code patching is the Ftrace function tracer. This tracer is mainly used to debug the kernel or measure performance. It is commonly called at the beginning of each function within the kernel or its modules. For performance reasons, each tracer call is replaced by a NOP slide when the feature is currently disabled. Although the feature is similar to the Jump Labels mechanism, its implementation is different and it depends on other kernel data structures. Consequently, both mechanisms must be considered separately.

5.2.2.6. Function Patching

In the meantime the kernel even supports live patching of entire kernel functions. This is to support security updates without the need to reboot the entire machine. In that case, the kernel either replaces an entire function with its new content, if the new version of that function is shorter then the original version, or the kernel uses the space that was reserved for the ftrace function tracer to insert an unconditional jump to the address of the new function.

5.2.3. Summary

In this section we provided the reader with an overview of the different patching mechanism that the Linux kernel uses. The shown mechanisms can be divided in two groups. The first group contains patches that only are applied during the loading of the kernel or its modules. The second category includes the mechanisms that are also used during runtime and thus are not precomputable. Table 5.1 contains a list of patching mechanisms for both categories.

```
1  int function() {
2    [...]
3    if (unlikely(...)) {
4      doSomething();
5    }
6    [...]
7  }
```

(a) Jump Label in Source

```
 1  function:
 2    <function prologue>
 3    [...]
 4    nop
 5  label1:
 6    [...]
 7    <function epilogue>
 8  jump_label:
 9    <prepare parameters>
10    call doSomething
11    jmp label1
```

(b) Jump Label deactivated

```
 1  function:
 2    <function prologue>
 3    [...]
 4    jmp jump_label
 5  label1:
 6    [...]
 7    <function epilogue>
 8  jump_label:
 9    <prepare parameters>
10    call doSomething
11    jmp label1
```

(c) Jump Label activated

Figure 5.2.: Example of Jump Label implementation in the Linux kernel. In case the *unlikely* condition is fulfilled the `nop` in line 4 is replaced with an unconditional `jump`.

Load-Time	Runtime
Relocation	SMP Instructions
External Symbols	Jump Labels
Alternative Instructions	Function Tracing
Paravirtualization Instructions	Function Patching

Table 5.1.: Classification of self-patching mechanisms in the Linux kernel

The introduced features are examples in which dynamic runtime patching may take place at any time during execution and not simply at load time, making out-of-band hashing-based troublesome. Most of the mechanisms described in this section are also relevant for architectures other then x86. All of the runtime patching mechanisms mentioned above can be used at any time within the operation of the kernel. Consequently, the kernel's code pages are not static, meaning that simple hashing of code pages is not enough to validate its integrity. In addition, as the patching of the kernel's code depends on the current state of the running system, we must consider semantic information taken from within the guest OS to validate code integrity. To the best of our knowledge, we are the first to apply semantic knowledge for runtime validation of the kernel.

As we have pointed out in the last section, the Linux kernel applies various code validation of modern kernels is a challenge that requires a deeper understanding of the kernel's various patching and relocation mechanisms. This means that simple hash-based approaches or approaches that simply make use of the kernel binary are simply not sufficient. In fact, there are several mechanisms for which the integrity and consistency of a change is verifiable but the resulting state still could be used by an attacker. In the following section we will introduce the architecture of our proposed solution to validate the kernel code during runtime.

In the simple case, with alternative instructions, an invalid change might only be an instruction which is normally not used on the specific architecture. However, in the case of jump labels, an attacker might enable a hook within the system, while the systems internal state is not aware of the hook.

5.3. System Design

In the following we describe a new architecture for kernel code integrity validation that handles dynamic changes within kernel code. We first give an abstract overview over our approach before we describe each of its components in more detail. We then introduce our concrete implementation in the next section, where we will also discuss the implementation related issues we had to solve in our framework.

5.3.1. Requirements & Goals

The main goal of the proposed architecture is to validate the identity and integrity of kernel code pages reliably, even in cases where dynamic changes are applied to kernel code during runtime. To achieve this goal we need to solve several subproblems. First of all, to validate the integrity of a code page, we depend on a trusted secure version of the code page that we can use for comparison. That is for each executable page in the target system we need to be able to identify the corresponding executable file which was used to originally load the page. We then need to extract all information from a trusted reference binary that is relevant for the code page in question. This includes all information that is relevant for the dynamic self-patching. Finally, we need to validate each executable code page and dynamically decide for each custom code patch if the modification is valid in the current system state. Secondly, to be able to validate all executable kernel code pages, the architecture requires both access to the target's page tables, since they provide a reliable source of information, and to the target's memory to validate the consistency of the current state. In addition, the security critical components of the architecture must be executed in an isolated manner to guarantee that the validation component cannot be modified by an attacker.

The three main components of our system that solve the first set of issues are discussed more in depth in the following: (1) the *Preselector (PS)* that identifies all code pages that need to be validated and maps each page to a corresponding trusted binary, (2) the *Runtime Verifier (RV)* that conducts the actual validation, and (3) the *Lazy Loader (LL)* that is responsible to load all required metadata required by the RV from the trusted executables identified by the PS. The overview of our architecture is shown in Figure 5.3. To solve the second issues, our architecture makes use of VMI to provide both isolation and tamper resistance for our system.

The decision to base our architecture on VMI does not limit our approach to the world of servers and cloud environments, as modern smartphones also begin to employ virtualization techniques through the use of the ARM Virtualization Extensions [1]. Finally, it is important to mention that data-only attacks such as return-to-libc [89], ROP [85], or data-only malware [101] are beyond the scope of this chapter and will be discussed in the next chapter. The reason for this is that a code integrity validation mechanism can not protect against such attacks, since they do not require code modifications to function.

5.3.2. Preselector (PS)

The task of the *PS* is to obtain the executable pages of the kernel, to divide code pages from data pages, and to associate code pages with a specific module or the kernel. To accomplish the first, the PS walks over the target system's page tables and extracts all supervisor pages that are present and executable. Since the virtual memory mechanism and the position and structure of page tables are specified by the hardware architecture, this information is *binding.* That is, it reflects the true state of the system at the time of the validation.

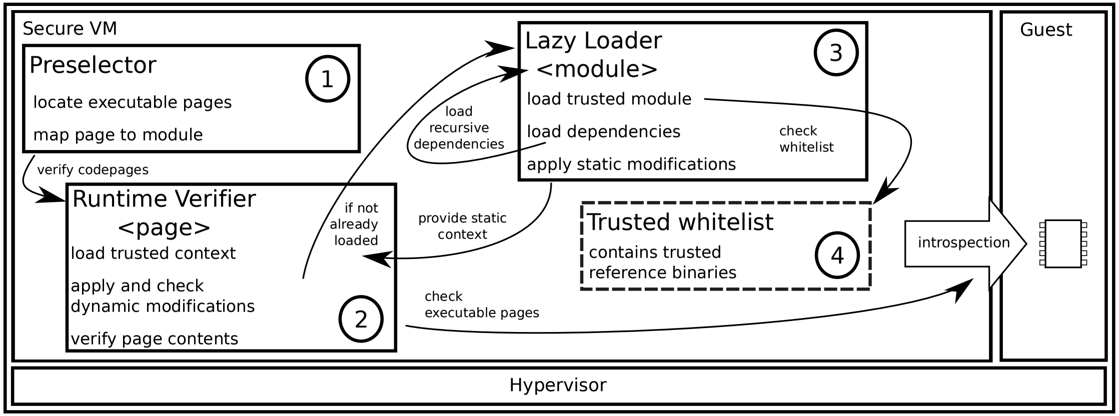

Figure 5.3.: Architecture of the proposed code page validation framework.

After having obtained all kernel code and data pages within the system, the PS assigns pages to a specific module or the kernel's code region on the system. For this purpose it obtains the list of currently loaded modules from the monitored system and extracts the virtual address of the code and data regions of each module. Based on the extracted data, it assigns the physical page frames to the modules by converting the obtained virtual addresses into physical addresses. The kernel binary is similarly processed. In this step the PS also separates code pages from executable data pages, by checking to which section the physical page belongs to.

After the code and data pages have been mapped, the PS performs an initial integrity check. It identifies any pages that could not be mapped to either the kernel or module code pages and have the supervisor flag set. If such a page exists it is considered malicious. Consequently, if a rootkit introduces code into the kernel and removes itself from the list of loaded modules, which is a commonly used technique, it will be detected at this point, since there is no module that can be associated within the code page of the rootkit. Note that this integrity check is based on *non-binding* information. We will defer a more detailed discussion of the security of the mechanism to Section 5.4.

Finally, the PS processes executable kernel data pages. Although data pages should be marked as non-executable, this is not always the case in practice. In the case of Linux, for example, all allocated data segments and especially pages that are allocated with `kmalloc` are marked as both writable and executable by default. To solve this issue, the PS marks all data pages as non-executable, a simple and effective solution, since data pages should actually never be executed. To space out that this design decision has a negative impact, we verified that setting all data pages in the Linux kernel to non-executable does not affect the system's stability.

5.3.3. Runtime Verifier (RV)

The *RV* is the heart of our system, because it processes the code pages that it obtained from the PS. It must be aware of all dynamic changes that can be conducted by the guest's kernel. For this purpose it first extracts all information from the monitored system that influences dynamic changes. For example, in the case of Linux tracing functionality can be enabled and disabled at runtime. Thus the runtime verifier must extract the status of the tracing component from the monitored system before it can validate the code pages reliably. When the RV processes a page, it first checks whether the module the page belongs to has already been processed by the *LL*. If this is not the case it will invoke the LL, which is responsible for loading a *trusted* version of the respective module into memory. After a module has been processed by the LL, the RV applies all predictable dynamic changes to the trusted reference binary and then compares each extracted kernel code page with the corresponding trusted reference. This process consists of three steps. First, the RV calculates the offset of the code page to be validated within a module's (or the kernel's) executable code segment. Second, it will use this offset to access the corresponding code page within the trusted representation that it obtained from the LL. Third, it will compare both pages byte-by-byte. In case, an inconsistency between the obtained code version and the trusted code version is detected, the RV checks whether this change is due to legitimate dynamic patching. After the changes can be reconstructed and validated, the identity of the code page is confirmed.

To check whether a change was conducted due to a dynamic patch, the RV makes use of a list of dynamic patch information that it obtains together with the module from the LL. This list is unique for each module (and the kernel) and contains information about each dynamic patch symbol within the binary. For each dynamic patch symbol, the list states the location of the symbol and the reason of the patch. Based on this information the RV can check whether a change was conducted at a location that belongs to a dynamic patch symbol and if the currently applied change is valid for the given patch symbol.

The latter can be achieved based on the information that the RV extracted from the running guest system. To make this scheme work, the RV additionally extracts all information from the monitored system that influences dynamic changes. This includes both information about the current architecture as well as information about the current system state. For example, a jump label can be enabled and disabled at runtime. Thus the RV must extract the current status of each jump label from the monitored system before it can validate the code pages reliably. In this step the RV also checks the internal state of the running guest that is related to dynamic patching. This is both the kernel's information about the patching symbol as well as the current state. Only if the changes can be reconstructed and validated, the identity of the code page is validated. Notice that both information sources - the dynamic symbol list as well as the runtime information - are crucial, since an attacker is otherwise able to launch mimicry attacks, which we further discuss in Section 5.4.

5.3.4. Lazy Loader (LL)

In order to validate the integrity of a kernel code page in memory, the RV requires a trusted copy of that page from a trusted reference binary. The task of the LL is to load this trusted version of each loaded module and the kernel as a basis for this comparison. To achieve this, the LL loads each module's (and the kernel's) binary from a trusted location, performs all load time modifications on it, and generates a list of dynamic patch symbols. In the following, each of these steps are described in more detail.

When the LL is invoked by the RV, it first attempts to find the requested module using the modules name. For this, it requires access to a secure location containing all trusted kernel binaries. The contents of this location essentially functions as a whitelist and it is therefore crucial that its location is protected against attackers. We achieve this isolation through virtualization and store the trusted reference binaries outside of the monitored guest. In case where the requested binary is not contained within the secure location, the module (or the kernel) is considered to be malicious. Otherwise the LL will load the binary into memory and applies all predictable modifications to it. Notice that the specific modifications that are applied heavily depend on the hardware and the software of the monitored system.

Since code pages can usually be loaded at an arbitrary address, the lazy loader will first update all addresses within the trusted binary such that they match the binary within the monitored system. To do this it makes use of the information that is provided to it by the RV. In particular, as every code page is associated with a specific module and the virtual address of a module's code and data regions are known, the LL can make use of these addresses to conduct the relocation. This is achieved by calculating the base address for each module and then applying the relocations according to this base address.

Aside from relocation, symbol resolution is another important step that is conducted by the LL. Since a module may make use of kernel functions and the address of these functions is unknown at the time of compilation, the loader must resolve all external symbols when a module is loaded. Similarly, the LL performs this step for the trusted modules. However, in contrast to the loader within the target machine, the LL must resolve all external symbols without relying on information within the machine because this information is untrusted. To solve this problem, the LL must determine the dependencies of each module (i. e. the list of other modules that the current module is dependent on as it makes use of symbols exported by them) based on the symbols that the module imports. Once this information is obtained, the LL can recursively load the dependencies and finally resolve all external references based on the trusted version of the modules. Note that the kernel itself is an implicit dependency of every module.

In the final step, the LL extracts all patchable locations for each module. It iterates over all patchable locations within the module and extract all of the symbols that are relevant for patching. In the process, it can already apply all patches that can be predicted (i. e. that are not dynamic). The list of the dynamic patch symbols as well as the relocated binary is then returned to the RV. Notice that all dynamic patch symbols are also contained in data structures within the monitored kernel's memory, since the kernel would otherwise not be able to locate and patch the symbol during execution. These in guest data structures are also checked for integrity during this process.

5.4. Implementation

After giving a general overview over our proposed framework in the last section, we now discuss the implementation and the associated challenges in more detail in this section. In the next section we will then evaluate our approach in terms of effectiveness and discuss the security implications that result from specific decisions made during the implementation.

In the following we present our implementation of a framework to enable integrity validation of Linux kernel code pages on the x86 architecture. To handle dynamic patching, it is essential that our system is able to access the hardware and high-level software state of the guest system at runtime. While the former is easily possible from the hypervisor, accessing the high-level software state of the guest kernel from the VMM requires that we bridge the semantic gap [24]. That is, we must be able to identify data structures and functions as the guest kernel sees them from the hypervisor. To solve this problem, we navigate to the desired data structures by following pointers through the object graph starting at global variables. The location of the global variables is obtained from the trusted kernel reference binaries. The layout of the corresponding structures is derived from the binaries *ELF* and *DWARF* debug information.

5.4.1. Identifying Executable Pages

The first component of our architecture is the PS. To be able to validate kernel code pages and to detect hidden code pages, our system initially requires a list of all executable kernel pages contained within the guest's memory. It is essential that this list is trustworthy as it is critical for the security of our architecture. This is why the preprocessor obtains this list directly from the underlying hardware. In particular, it creates the list of all executable kernel code pages by iterating through the page tables that are currently used by the system. The physical address of these page tables is contained within the `CR3` register.

Once the list of executable kernel pages is generated, the PS maps each page in the list to the kernel's or a module's code section. For this purpose, it extracts the addresses of important kernel structures such as the location of the `.text` or `.data` segment of every kernel module directly from the guest systems memory by reading the corresponding data structures. The addresses of the kernel's `.text` and `.data` section are extracted from its binary representation. An example of a typical page mapping is given in Table 5.2.

Type of pages	Absolute #	Size (in kb)
Kernel Code pages	4	8192
Module Code pages	137	548
Kernel Data pages (.data)	240	960
Kernel Data pages (.vvar)	1	4
Kernel Data pages (.bss)	17	2112
Total	399	11816
Total validated	141	8740

Table 5.2.: Types of executable pages in the Linux 3.8 kernel

In practice there are two types of executable pages: dedicated code pages and executable data pages. In our test environment all of the kernel's data pages were actually mapped as executable, which is consistent with the fact that newly allocated pages also are mapped as executable per default. As previously mentioned we set all pages non executable.

5.4.2. Handling Load Time Patching

Similar to the kernel, our implementation also uses a multi-staged process to reconstruct the contents of each executable page. In the first phase all load time code modifications are precomputed in the LL. In addition to the relocation and external symbol resolution, this phase also includes the patching of hypercalls and the processor dependent improvements. For a specific target system these steps are only reproduced once. The dynamic runtime patching mechanisms are considered in the second phase, which is conducted by the RV.

After the executable pages are identified as code pages of the kernel or one of its modules, the PS component calls the RV to validate the contents of each page. To do this, the runtime verifier first invokes the LL for each page in order to obtain the validation context for each module. If the module's context was already initialized, the LL loader returns the module's trusted context. Otherwise the loader initializes the context as follows:

The loader component first loads the binary ELF representation of the requested module from a trustworthy location. This is equivalent to providing a whitelist of binaries. Due to this whitelist an attacker is no longer able to load arbitrary modules. Should an attacker load an unknown module into the guest system, the LL returns an empty context and notifies our framework of the malicious module. This approach implies that the trusted repository of kernel binaries must be updated if the kernel of the monitored system is updated intentionally. In addition, it is essential that the administrator ensures that only trusted binaries are contained in the whitelist. Note that this essentially replicates the kernel's loading process, as the LL also loads all of the dependencies of the requested modules.

After loading the trustworthy reference and all of its dependencies, our framework takes care of the relocation of internal symbols. Therefore, the binary representation of each module contains a list of locations and their corresponding symbols that need to be relocated. For each location and each internal symbol we calculate its virtual address in the memory of the inspected VM. We then replace each reference to a symbol with its absolute virtual address or a relative offset to this address depending on the type of relocation (absolute or relative).

After the relocation we resolve external symbols. As with relocation, we replace all references to external symbols with the absolute address of the external symbol or a relative offset to its location. As to avoid relying on potentially compromised data-sources, our system doesn't rely on the monitored kernel's resources (e. g. its `System.map` or its internal list of exported symbols). Instead, we follow the kernel's dependency mechanism, by recursively loading all dependencies of the current module and initializing their full context for later usage. Thereby we also create our own list of (exported) symbols for each of the kernel binaries. When resolving an external symbol we consult our internal list of symbols.

Next, we process alternative instructions that are provided with each binary. As a reminder, this feature allows one to substitute specific instructions within the code with other, more efficient instructions based on the current hardware. To decide whether the substitutions should be conducted, we obtain the necessary information from the virtual hardware. With a list of features at hand we now walk through the list of alternative instructions and substitute the referenced instructions, whenever the required feature is available. If the compensating instructions requires less space than the original instruction, the rest of the reserved space is filled up with NOP instructions that correspond to the CPU model in use.

Finally, the LL updates the instructions within the binary which depend on the host's hypervisor. For this purpose, each binary contains a segment with a list of locations together with the type of patch that is to be applied. In contrast to the alternative instructions, the

1. Return context for the binary from cache if already loaded.
2. Load the binary from trusted location.
3. Return empty context if no trusted binary is found.
4. Extract and recursively load dependencies.
5. Extract code from trusted reference binary.
6. Relocation of internal symbols.
7. Relocation of external symbols.
8. Extract and process list of alternative instructions.
9. Extract and process list of virtualization related modifications.
10. Extract metadata about runtime patching mechanisms.
11. Internally save all known symbols of the binary.
12. Add the generated context to the cache and return it.

Figure 5.4.: Summary of the binary loading process.

possible patch *values* for the locations are contained within a kernel data structure provided by the virtualization solution (e. g. KVM, Xen), not by the kernel binary itself. Whether the patch that is applied is a simple instruction patch or a jump/call to another function is determined by the specific virtualization driver that is in use.

While replacing an instruction with a jump instruction is not used within KVM, it is used in a paravirtualized Xen environment. In this environment, the following functions are jumped to by using this mechanism: `xen_iret`, `xen_sysexit`, `xen_sysret32` and `xen_sysret64`.

As we know exactly which hypervisor is being used, we are able to validate these patches through a whitelist. For this purpose our framework provides a plugin system to be easily extensible. This enables the framework to support different hypervisors (e. g. KVM, Xen) or additional patching mechanisms that may be introduced in the future as well. Since we, in this case, cannot trust the kernel's data structure, the whitelists for each hypervisor are generated from a trusted copy of this kernel data structure. In contrast to the original structure, the copy does not contain addresses, but only the name of the symbol that is used. This is because the address of a function or symbol may vary due to relocation, while the symbol name is unique. The LL is then able to reliably resolve these symbols as described above. In addition, we also validate that the kernel's data structure that contains the hypervisor dependent replacements contains the correct instructions. A summary of the tasks of the LL is given in Figure 5.4.

5.4.3. Handling Runtime Patching

After load time modifications are applied, the LL hands the initialized context to the RV. The task of the RV is to validate all dynamic modifications that cannot be predicted, e. g. changes that are not related to load time patching and are thus not already managed by the LL. Therefore, the RV analyzes the current system state and updates the initialized context concerning changes conducted by the runtime patching mechanisms. For this the RV needs to analyze and evaluate the state of the monitored system that is related to the corresponding runtime self-modification mechanisms. Afterwards the RV iterates byte-by-byte over all executable pages of the monitored system and compares their contents with the appropriate pages of the internal reference binaries within the trusted context. If a difference is detected, the RV matches the difference against one of the known runtime patching mechanisms. We implement this as a two step process, as for some of the mechanisms (e. g. the SMP locks) it is easier to predict and apply the current state of the code section just before the validation is conducted, while for other mechanisms (e. g. the Jump labels), there are multiple variants possible and it is easier to do the validation when the current state of the code section within the monitored system is known in the live validation. As we have described these mechanisms in Section 5.2 we will now describe how our POC implementation handles them in more detail.

The first mechanism checked is the SMP related patching. We obtain the number of currently active CPUs and adapt the locks within the trusted reference accordingly. Note that the number of CPUs is not necessarily static on a modern system. For example, in virtualization-based cloud environments it is common that vCPU cores are added and removed at runtime.

The next mechanism handled is the Ftrace function tracer. To validate an Ftrace function call, the RV first checks if the corresponding tracing functionality is enabled within the guest. Based on the current state of the tracing mechanism, it ensures that the Ftrace function call is either replaced with NOPs (tracing disabled) or that the call points to the `__fentry__` symbol (tracing enabled). The latter can be validated using the internal symbol list of our framework.

Jump Labels are another feature of modern Linux kernels that requires patching at runtime. To reiterate, with this feature a kernel developer is able to mask unlikely branches during normal execution. The mechanism provides a list of offsets inside the executable code together with a jump destination for each offset. At runtime the branch can be enabled or disabled by calling a specific function. The kernel therefore handles all jump targets within an internal data structure. Each entry in that structure contains a key, that indicates the current status of the jump label. To validate jump targets, our framework compares the value of this key with the current state of the jump target on the executable page. If the states match, the current target on the executable page is compared with the original target specified in the trusted reference binary. The change is only considered benign, if all information is consistent.

Although this is not directly a dynamic patch, the final check that is performed by the RV is to ensure that pages that only partially contain code are valid. This situation arises when a code segment of a module or the kernel is smaller than an entire page of memory. In this case the code segment will only occupy a part of the page, while the remainder is unused. Since the page is executable, an attacker could try to inject code by modifying the unused code areas. In practice such an attack should be easily detectable as all unused kernel code regions are by default set to zero on Linux. Thus, our framework protects against such attacks by ensuring that the unused space on the last page of a code segment does not contain any non-zero bytes after the end of the code. We will further discuss in Section 5.6 how and why some of this space is currently also dual mapped into userspace.

5.5. Evaluation

After having introduced our framework for dynamic code integrity validation, we now present the evaluation of our framework. First, we evaluated the effectiveness of our approach in the case of kernel code integrity violations. For this purpose we evaluated the detection capabilities of our system using multiple rootkits. In a second set of experiments, we measured the performance impact introduced by our system. In the following we describe the experiments and their results in more detail. We conducted all tests on an AMD Phenom II X4 945 Processor with 16 GB of RAM. As monitored guest we chose Ubuntu 13.04 with 512 MB of RAM on the KVM hypervisor. The validation component of our framework as described in Section 5.4 was executed within the host OS.

5.5.1. Effectiveness

The primary goal of our proposed framework is to reliably detect kernel code integrity violations. To test our framework in this regard, we conducted multiple experiments with kernel rootkits. As one integral purpose of rootkits is to provide stealth, this enabled us to validate the effectiveness of our framework in a real-world scenario. The rootkits we tested with were four well-known kernel rootkits: adore-ng, enyelkm.en.v1.1, intoxonia-ng2 and override. All of these rootkits change the code of the victims kernel and make use of DKOM to hide themselves inside the victim kernel's memory. Thus we expect our system to be effective if it is able to detect all of these rootkits.

With our experiments we verified that we are successfully able to detect all of these rootkits. In our tests we could distinguish three detection cases. In the case that the rootkit removed itself from the module list within monitored guest, our framework was unable to match its executable code pages to a kernel component. It marked the rootkits executable pages as malicious. When a rootkit did not hide itself from the list of loaded modules, our framework's LL didn't find a corresponding trusted representation of the module and thus marked the module as malicious. To further test our system we also renamed one of the rootkits to reflect a module contained in our whitelist. More precisely, we renamed the *override* rootkit to *decnet*, disabled its hiding features and reran our validation framework. In this experiment the LL loaded the trusted version of the *decnet* module and the RV detected the mismatch. In case, the rootkit did not only load itself as a new module but modified existing kernel code, we also detected its changes.

Additionally, in all cases our framework detected all changes to the kernel code that were introduced by the rootkits. As our framework successfully identified all valid dynamic patches within the code pages, we had no false positives related to the kernel's dynamic patching mechanisms. Furthermore, the framework correctly informed us about the code pages that contained content that originated from userspace and automatically set all executable data pages to not executable.

5.5.2. Performance

To validate the applicability of our approach we also measured the performance imposed by our framework in a second step. We used KVM/QEMU in combination with `hugetablefs` to create a file representation of the guest's physical memory within the host. With this setup, our framework can directly access the guest's physical memory. Note that this file representation enables us to memory map the guest's memory as shared memory within the validation framework. Shared memory is an efficient technique as it automatically syncs between the processes and has very little overhead.

As the kernel code is usually only patched infrequently, we provide a worse case evaluation scenario. We decided to monitor and validate the guest system live in a continuous manner which generates as much stress to the system as possible. With this decision, we effectively use one dedicated CPU of the system for the integrity validation. Another similar approach was recently proposed by Brookes and Taylor [16].

To measure the overhead of our approach we executed a memory and I/O intensive task within the virtual machine. We chose this test case, as due to the architecture of our framework and its frequent memory accesses, we expect to gain a worst case performance penalty when also using an I/O intensive task as the benchmark, while we expect no performance penalty for CPU intensive workloads. That is because the validation framework is executed on a different CPU than the monitored guest VM. As our test case we compiled the Ubuntu version of apache2 repeatedly. The test was conducted in the VM both without external validation and with our validation framework checking the kernel's pages from within the host system. In the case with external validation enabled we also did not only validate the executable code pages once in a predefined interval, as one would do that in a real-world use case. Instead we verified the monitored guest's code pages in a continuous manner to provide as much stress to the system as possible. The evaluation results of our tests can be seen in Table 5.3.

First we measured the our test case without external code validation. The mean real time for a single test run took $222.59s$ ($176.951s$ sys + usr). We then enabled external validation and reran the test case. In this case, the mean real time of one test run was $224.474s$ ($179.409s$ sys + usr).

We also measured the average performance overhead that was introduced by our code validation framework. This experiment is separated into two cases. In the first test, we only consider the

Test	time result
Compile Apache2 without code validation	222.590 s
Compile Apache2 with code validation	224.474 s
Code validation without initialization	4.051 s
Code validation with initialization	0.279 s

Table 5.3.: Results of performance evaluation (mean)

case in which our framework is uninitialized and the LL component of our framework needs to load all trusted reference binaries. On average it took $4.051s$ to validate all kernel code pages without prior initialization. As this loading step is only required once, the case in which the framework is already initialized is a more accurate measurement of runtime overhead. In this case, the validation of all kernel code pages took on average only $0.279s$. During that time all 141 executable kernel code pages of our test environment where identified and validated (2185 pages if we count the $2MB$ pages with a page size of $4kB$). The check of a single page thus only took $0.00012s$ ($0.1ms$), which is a negligible overhead.

It is also of interest that our architecture makes use of shared memory when accessing the monitored guest's memory. We therefore do not have to interact with the monitored guest itself. On a multicore architecture the validation process can thus be done in parallel to the VM's execution. Therefore the introduced performance degradation is very small.

The performance could even be further increased. By leveraging an event based mechanism, e. g. the hypervisor's EPT mechanism, only one individual page validation is required after a write to a code page was identified. The performance degradation comes down to ca. $0.1ms$ every time the kernel has patched one of its code pages. In addition it is also possible to validate the contents of a page in subpage granularity, which further decreases our overhead. In such a case malicious changes could even be reverted automatically.

5.5.3. (Not) Trusting the Guest State

After evaluating the effectiveness and the performance of our system we now talk about the security properties of our approach.

To validate the integrity of the kernel code pages our framework considers untrusted semantic information provided by the guest. Since this information is *non-binding* and could be tampered with, careful consideration has to be given to this potential serious security issue. However, as we will discuss within this section, any information that we use from the guest system is actually *not security relevant*. That is, even if an attacker changes the information that our framework extracts, it will not undermine the security of our system. In the following, we will show that all information that is used by our system is either trustworthy or not security relevant. In the following we will discuss the binding and non-binding information our framework uses. In our analysis, we reiterate the order of steps that are conducted by our framework. For this, we will discuss all information that is extracted from the monitored guest.

Our system first extracts information about all executable supervisor pages within the monitored system. As this information is directly obtained from the virtual hardware it reflects the true state of the system and is therefore trustworthy. Furthermore, we also extract a list of all currently loaded kernel modules. We use this list to relate the pages in the system to specific modules. If an attacker manipulates this list (e. g. by removing a module from it) our system will not relate the pages of the hidden module to a known module and considers them as malicious. Since hidden code pages are considered malicious, the attack is detected. Also, if the addresses of the text or data segment of a module is changed we detect this inconsistency. Thus using information contained from this data structure does not affect the security of our system, as any malicious alteration will be detected.

To perform external symbol resolution, the next piece of information that we process is the list of exported kernel and module symbols. The kernel's list of loaded symbols (which is used for locating exported symbols within the kernel's module loading process) is a data structure that is commonly used for this purpose. As we reconstruct the loading process of the kernel and its modules and thus regenerate this information within our framework, we do not depend on the kernel's possibly compromised data structure. In fact, we are even able to detect such modifications and generate an alarm if such an inconsistency is detected.

In the next step, our framework generates the contents for SMP locks and alternative instructions that depend on the current state of the monitored system. This step is once more based on the virtual hardware, and thus can also be considered trustworthy. (Para-)virtualization related patching on the other hand is more difficult to validate. This is because the modifications conducted by the kernel directly depend on a kernel data structure provided by the corresponding hypervisor's driver. Our framework also validates the integrity of this data structure. Without this data validation, we could only validate that a call is targeting a valid function within verified code. This leaves an attacker an opening in which she could modify control flow from one verified function to another. To remedy this, we include a small data validation step to establish the integrity of any changes. As we use KVM as a hypervisor for this part of our work, this step becomes quite straightforward due to the fact that we can assume the native implementation of the paravirtualization interface is used. We simply validate that the corresponding native instructions and functions are patched in. However, we also consider the possibility of a paravirtualized Xen environment to show the flexibility of our system. In this case, for example, jump destinations are: `xen_iret`, `xen_sysexit`, `xen_sysret32` and `xen_sysret64`. As there is no generic way to validate jump destinations, we provide a trusted copy of the entire kernel data structure as a whitelist.

To extract information as to whether a hook for the Ftrace or Jump Label mechanism is enabled, we again inspect the current state of the guest kernel. The current state of the jump/call is contained both within the kernel code and inside a kernel internal management data structure. To validate the integrity of the current system state we check if the state of the code is consistent with the kernels control data structure. If the current state is not reflected in the trusted data structure, we consider the change malicious. We furthermore validate that the control data structure is consistent with the information that is contained within the trusted reference binary.

Having outlined the security considerations in our system, we see that handling code changes due to the paravirtualization feature requires a whitelist. We added this functionality to increase the security of the guest kernel, though it is possible to validate the integrity without this whitelist. It is however, very important to understand that this mechanism can be leveraged to subvert the security of the kernel without directly affecting its integrity as this mechanism is intended to allow for arbitrary patching and hooking. The fact that this mechanism can be abused is a matter of the design of the mechanism itself. Virtualization therefore represents a perfect example of the edge cases that are not considered by existing systems. We introduce the ability to leverage whitelists in this situation to reduce the attack surface.

In addition to the code segments mentioned above, the kernel clearly maintains data as well. Generally, these segments are the `.data`, `.vvar` and `.bss` segments. While these segments are to contain data, we found that many of the pages inside of these segments are in fact marked as executable. Specifically, in our experiment we came across 258 data pages taking up around 3076 kbytes of memory that were marked executable. This creates a problem as we must consider these pages during our validation as well if they are executable. Simply hashing these pages creates issues as these pages generally contain some data that will change.

Currently our system's main focus is on detection of malicious changes to the kernel's code pages, regardless of the schedule at which it is run. The simplest schedule is to simply run the validation process at regular intervals, however our system could be used in conjunction with event based mechanisms as well. For example, one could leverage the EPT mechanism. This enables us to set the monitored hosts executable kernel pages to read-only in the hypervisors page-table representation. Whenever the monitored system writes to its code pages to patch a dynamic hook this generates an EPT violation in the hypervisor [65]. The hypervisor can then allow the write to the corresponding page and notify our integrity detection framework to re-validate the integrity. In this case our framework can be configured to automatically revert malicious changes. This way our system also prevents malicious integrity violations.

5.6. User Code in the Linux Kernel

To underline the importance of research in the field of kernel code integrity, we want to describe a particularly alarming behavior of the Linux kernel that we discovered during our research. The sheer fact that this problem remained unnoticed so far provides further proof that existing methods are unable to validate the kernel's code regions reliable. On a high level, the issue we encountered allows an unprivileged user to load arbitrary code into kernel space. While the code is not executed by default, this makes exploitation in many cases trivial, as the attacker only needs to find a way to point the instruction pointer into her code. This represents a critical issue, as there are dozens of security mechanisms such as secure boot, signed driver loading, $W \oplus X$, SMEP, and SMAP that solely exist to hinder an attacker from loading code into kernel space or executing userland code from kernel space.

The root of the problem is related to performance optimizations, by which the x86 version of the Linux kernel uses 2MB pages to store its code segments. This results in a hole at the end of the kernel code segments, but before the end of the last page which is effectively allocated to the kernel, but not used. This hole is filled with what we initially mistook for uninitialized memory. After some further investigation we found that this hole contains code and data belonging to userspace libraries and processes.

In a single test we found more that 15 different libraries that had a part of their pages also mapped inside the kernel. Besides common libraries as `libc` and `ld` there were also libraries such as `libdbus` that are potentially more vulnerable. But even more concerning is that stack and heap pages of some userspace libraries were mapped to this region. We made sure that we could reproduce this on a native system as to exclude the possibility of a bug in the hypervisor and, indeed, we noticed the same behavior.

In fact, we found that instead of letting this memory go to waste the kernel also maps it as 4KB pages for userland processes despite the fact that it is already mapped into the kernel code segment. We

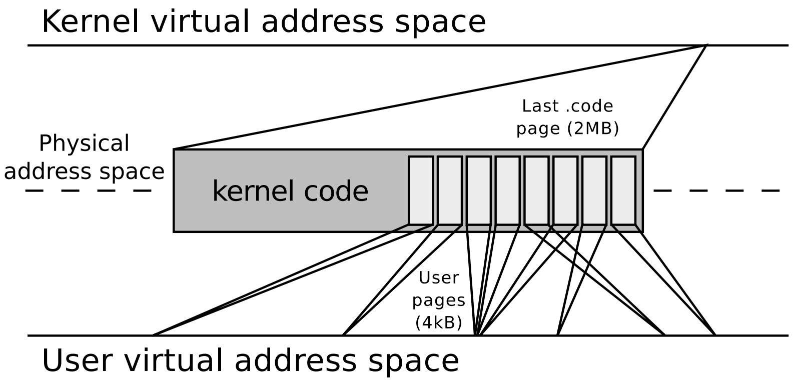

Figure 5.5.: Different mappings of the same physical memory.

presume that this is done in an effort to not waste physical memory. This is illustrated in Figure 5.5. As we cannot predict what pages will be mapped into this space it is impossible to perform hashing on a page granularity.

Table 5.4 shows the userspace libraries that had some of their content in the region of physical memory that was also marked as executable kernel code for a single test.

In addition to creating a challenge for code validation, this has other security implications. Effectively, this allows one to load arbitrary code into the kernel. There are two arguments that one may have in defense of this design decision. First, the user pages that are mapped are mapped out of order and are not guaranteed to be mapped to these physical frames. The other argument may be that the code could be loaded, however it is never executed, so there is no problem. We consider these two arguments separately.

We begin by addressing the argument that malware cannot influence where its pages are mapped from userspace. This is technically true, but is a problem that is easily overcome. The first problem is that the code is constrained to a single page as it is *very* unlikely that two pages will be mapped in order and in adjacent frames. This simply means that the malware must confine itself to a single 4KB page in the simplest case. On the other hand, we are not even guaranteed that any single page will be mapped into the kernel segment. While this is also true, there is little stopping the process from deallocating and allocating memory until it finds it is mapped into kernel space. To find out if the process has some pages mapped to the interesting parts of physical memory, the process may read the `maps` file inside the processes directory on the `/proc` filesystem. There it finds the information which virtual addresses are mapped to the process. With this information it can extract the physical mapping of each virtual page by reading the `pagemaps` file in the same directory. Note, that this is also allowed with only user privileges.

To address the second argument that there is no problem as long as the code is not executed, we argue that this stance is fundamentally flawed. This argument assumes that there will never be a vulnerability in the kernel that allows one to control the instruction pointer. Mechanisms such as signed driver loading and SMEP work to explicitly prevent unsigned code from being loaded into the kernel or to prevent the CPU from jumping into user space code while in supervisor mode. Allowing code to simply be mapped into kernel space undermines these mechanisms completely even if it is never executed.

To show the concern of this architectural decision, we implemented an unprivileged userspace application capable of inserting code into the kernel. In particular, we use the already mentioned `pagemap` feature within the `/proc` file system to read the current mapping of virtual to physical pages. To ensure that our process has a page mapped into the kernel's text pages, we allocate multiple page-aligned memory areas, checking `/proc/self/maps` and `/proc/self/pagemap` each time to determine whether the most recent allocation is also mapped in the kernel's code segment. In practice, this was surprisingly effective. We found that it only takes a couple of tries until a page is mapped into the kernel.

Once we control a physical page that is also mapped inside the kernel text segment, we need to obtain its virtual address in order to be able to invoke it. As it turns out, calculating this address is as simple as: `<.text> + (pagenr * 0x1000)`.

This is the case as the first 8 MB of physical memory are mapped to that address in the so called *identity mapping* on the x86_64 architecture and the current versions of Linux do not implement ASLR in kernel space. If we assume a vulnerability inside the kernel which lets us redirect the control flow to a specific address, we are able to use that vulnerability to jump to the code we loaded into the kernel. Note that the vulnerability can be relatively simple as we only need to be able to control the instruction pointer. Traditionally, a vulnerability would require the ability to upload a payload, control the instruction pointer, and in some cases control the stack pointer in order to exploit it.

Of course, we cannot be sure that our page will be indefinitely mapped to this area in the kernel text segment. It is possible that we are evicted by the page replacement algorithm of the kernel at some point in the future. However, we simply need to execute once. Since we are executing in the context of the kernel, we can simply allocate memory within kernel and copy a persistent payload to that area.

Kemerlis *et al.* [50] recently found a similar problem in the Linux kernel. They describe that the Linux kernel employs an identity mapping of the entire physical memory for performance reasons and that this area is not required to be executable by the kernel. This is a different mapping than the kernel identity mapping we described in this section. They then go on to describe how the physical identity mapping can be exploited and propose a solution for mitigation. Their mitigation method unmaps a page from the physical identity mapping once it is allocated to userspace. After the page is no longer allocated in userspace, the page contents are wiped before it will be mapped to the identity mapping again. In contrast, the vulnerability we discovered shows that a section of the kernel identity mapping is not occupied by kernel code and is instead allocated to unprivileged userspace applications. While the attack surface described by Kemerlis does overlap with our vulnerability, the mitigation method does not apply. Thus, we propose that the physical memory, that is part of the kernel identity mapping, must not be allocated to userspace applications.

Of course, we also inspected the Windows 7 kernel for similar conduct and found that there is no similar behavior. To inspect Windows, we iterated over the pagetables of all Windows processes using the KPCR[111] debugging mechanism. We compared all 2 MB and 1 GB kernel page mappings[1] with all user mode mappings to look for overlaps. We verified that none of the pages mapped into user space occupies a physical page that is also mapped as executable page in the kernel.

5.7. Delimitation from previous work

The only previous work that slightly considers configuration-specific patching is Patagonix [57]. To recapitulate, at its core, Patagonix is also a hash-based validation system in which the hypervisor stores a hash of all valid code pages. Patagonix makes the assumption that patching generally only occurs during load and early boot time. Load time patches are handled with the help of a list of all possible memory locations that can be updated, and a list of all possible values for each memory location. This information is extracted from each binary before the validation process. Furthermore, the code validation relies only on *binding information*, that is, they do not rely on any “semantics implied by source and symbol information”. While we agree that it is important to handle non-binding information carefully, as malicious code is not bound to this information, we argue that one *must* consider non-binding information to be able to properly validate the dynamic runtime changes conducted by modern OS kernels. For example, many of the runtime patching mechanisms used by the Linux kernel depend on the current software state of the running system. This information is non-binding by its nature.

[1] In our tests there was no 1 GB mapping used.

As Patagonix does not make use of non-binding information, it is unable to validate the changes conducted by these mechanisms.

The main difference between our work and Patagonix is that Patagonix tries to validate all changes conducted to a code page by the guest solely on information contained within a binary. That is, Patagonix does not include information about the current system state in its validation process. This leads to several problems on modern kernels. First and foremost, there are memory updates conducted by the kernel that cannot be predicted beforehand. For example, if a module imports functions from an other module the symbol resolution of the module obviously depends on the memory location of the other module and thus on the state of the system. Second, Patagonix will consider all potential modifications that can be applied to a specific memory location as valid independent of whether the modification is actually legitimate for the current software and hardware state. Third, the information whether a memory location is patched during runtime can in the case of modern kernels no longer be directly extracted from the binary. For example, the kernel may substitute specific instruction sequences if it is running in a paravirtualized environment. However, this information is not contained within the binary.

The common problem with all these approaches is that they consider kernel code to be static once it has been loaded into memory. Modern kernels, however, make use of many optimizations that require runtime patching, which renders these approaches obsolete. To validate kernel code it is thus essential to understand the individual runtime patching mechanisms that the kernel uses. Once these mechanisms are understood, we can implement code integrity validation mechanisms that reliably detect malicious modifications at *runtime*. In this thesis, we make the first step in this direction and investigate the runtime patching mechanisms of modern Linux kernels.

After the first publication of our previous work [54] we encountered similar research conducted at the same time by Stanley *et al.* [92, 91]. We consider our work as more comprehensive for various reasons. Similar to our work, the authors introduce and shortly describe different kernel self-patching mechanisms within the Linux kernel and improve the NICKLE hypervisor [76] to support code integrity validation under consideration of runtime kernel self-patching. In order to validate the patches that are conducted by the guest OS during runtime, a whitelist of possible change sets is generated in an offline phase in advance. This whitelist consists of tuples containing the offset of the start address of a change within the kernels code segment and the content of the patch. With this, the hypervisor-based system intercepts every self-modification attempt and only permits the modification, if the corresponding patch is contained in the previously generated whitelist. While this is a first important step, we argue that it is also important to check the current state of the guest operating system and validate whether the change is valid according to the current system state. For this, our system does not only detect and validate kernel self-modifications, but also validates the corresponding kernel state related to the modification. Our handling of different types of multibyte NOP instructions on different architectures is another example. To additionally support module loading, our system generates the patching related information on demand by extracting the information from trusted versions of the loaded binary files. Another difference is the handling of some of the mechanisms. For example, Stanley arguments, that patches related to (para-)virtualization are not handled by their system due to the fact, that they did not use paravirtualized kernels. In contrast, we show that it is in fact important to handle virtualization related kernel code modifications as they are used even in full virtualized environments.

Finally, the most important difference is in the evaluation of the approach. Stanley describes the existence of the self-patching mechanisms and shows that hypervisor-based kernel code validation is possible with low overhead. During our evaluation, we were not only able to show that kernel code validation is possible with minor overhead, but we also found problems in the management of physical pages in the Linux kernel. We show that the Linux kernel benevolently maps userspace code into the kernels code sections, opening potential for an attacker to easily load malicious code into the kernel without tampering with the kernel code itself, effectively rendering protection mechanisms like SMEP as useless. We further elaborate on this problem in Section 5.6.

5.8. Summary

Validation of kernel code integrity is an important aspect of runtime integrity checking. Previous approaches assumed the kernel code to be static at runtime or only depend on non-binding information while checking the integrity of kernel code. In this chapter, we have shown that modern kernels also employ dynamic runtime code modification.

We have examined various load time and runtime code patching techniques employed by modern OS kernels in detail and discussed the challenges that these runtime code patching mechanisms create for code validation.

Due to complicated loading, debugging, and optimization processes, kernel code must be considered highly dynamic. Thus current state-of-the-art code validation techniques are not suitable to validate the code integrity of modern kernels. To address this, we designed and implemented a dynamic code validation framework using both binding and non-binding state information from the monitored guest. We reconstruct trusted copies of kernel code pages and differentiate between valid and invalid changes inside these pages. We also validate the integrity of the kernel's internal state related to the dynamic patching.

To show the viability of our system, we present several experiments to test both the effectiveness and performance overhead of our system. We were able to show that our framework is able to detect all of the modifications that where introduced by the rootkits we tested. Additionally, we were able to show that the performance overhead is as low as $0.1ms$ for every change that is made to the kernel code. And due to the nature of our approach we are able to validate the entire content of the kernel code section.

Finally, we discussed several security concerns we had as a result of our investigation. We identified a yet undiscovered a double mapping of executable supervisor code pages that are also mapped to userspace. This enables an unprivileged attacker to place arbitrary executable code within the kernel without violating protection mechanisms such as signed driver loading, $W \oplus X$ or SMEP. We discussed these issues and also successfully implemented a POC to exploit the issue.

[heap]
- init
- upstart-udev-br
[stack]
- console-kit-daemon
/lib/x86_64-Linux-gnu/ld-2.17.so
/lib/x86_64-Linux-gnu/libc-2.17.so
/lib/x86_64-Linux-gnu/libdbus-1.so.3.7.2
/lib/x86_64-Linux-gnu/libnsl-2.17.so
/lib/x86_64-Linux-gnu/libnss_compat-2.17.so
/lib/x86_64-Linux-gnu/libnss_files-2.17.so
/lib/x86_64-Linux-gnu/libnss_nis-2.17.so
/lib/x86_64-Linux-gnu/libpthread-2.17.so
/lib/x86_64-Linux-gnu/libresolv-2.17.so
/sbin/dhclient
/sbin/getty
/usr/lib/sudo/sudoers.so
/usr/lib/x86_64-Linux-gnu/libgssapi_krb5.so.2.2
/usr/lib/x86_64-Linux-gnu/libpolkit-backend-1.so.0.0.0
/usr/sbin/console-kit-daemon

Table 5.4.: List of userspace libraries which had some of their data memory on phsical pages that are also mapped to executable kernel memory.

Chapter 6

Code Pointer Examination

In the last chapter our focus was set on the integrity of the code region of the kernel. In this chapter we set our focus to the integrity of parts of the data areas of the kernel. In this chapter, we introduce CPE, a novel approach which aims to detect data-only malware by identifying and classifying code pointers. Instead of targeting control flow changes, our approach targets the control structure of data-only malware, which mainly consists of pointers to the instruction sequences that the malware reuses. Since the control structure is comparable to the code region of traditional malware, this results in an effective detection approach that is difficult to evade. We implemented a prototype for recent Linux kernels that is capable of identifying and classifying all code pointers within the kernel. With this we provide an answer for our research question **Q3** set in the beginning of this thesis. As our experiments show, our prototype is able to detect data-only malware in an efficient manner (less than 1% overhead). The main results of this chapter have already been published in the academic paper “Counteracting Data-Only Malware with Code Pointer Examination” [53].

6.1. Problem statement

In this chapter, we explore a new approach to the detection of data-only malware. The key idea behind this approach is to detect data-only malware based on "malicious" pointers to code regions (from here on simply referred to as code pointers). Similar to traditional malware, data-only malware has to control which reused instruction sequence should be executed at which time (e. g. event). To achieve this, data-only malware makes use of a control structure that contains pointers to the instructions that should be (re)used. This control structure can essentially be seen as the "code region" of the data-only program that the malware introduces. By identifying malicious code pointers in memory, we in essence aim to apply the idea of code integrity checking to the field of data-only malware by detecting malicious control data within the system. For this purpose, we introduce the concept of Code Pointer Examination (CPE).

The idea behind CPE is to identify and examine each possible code pointer in memory in order to classify it as benign or malicious. This is essentially a two-step process. In the first step, we will iterate through the entire memory of the monitored machine on a byte by byte granularity in order to identify all code pointers. In the second step, we will then classify the identified code pointers based on heuristics. As our experiments show, this approach results in an effective and high-performance (less than 1% overhead) detection mechanism that can detect data-only malware in many cases. It is thus well-suited for both live monitoring as well as forensic investigations where only a single memory snapshot is analyzed.

Since the OS is the integral part of the security model that is used on most systems today, we focus our work primarily on the Linux *kernel*. We chose this OS, since it is open and well documented, which makes it easier to understand and reproduce our work. However, the concepts and ideas that we present are equally applicable to userspace applications and other OSs such as Windows. As a groundwork for the approach described in this chapter, we require information about the monitored operating system, specifically about the memory regions occupied as well as the locations of the code within memory. This information is extracted from trusted ELF images of the monitored kernel using the kernel code integrity framework already described in the previous chapter.

In summary, in this chapter, we have made the following contributions:

- We present CPE, a novel approach to identify and classify code pointers.
- We highlight important data structures that are used for control flow decisions in modern Linux kernels and thus must be considered for control flow validation.
- We provide a prototype implementation and show that it is both effective and efficient in detecting control structures of data-only malware.

6.2. Attacker Model & Assumptions

In this chapter we assume that the monitored system is protected by a virtualization-based runtime code integrity validation framework. In addition, we assume that an attacker has gained full access to the monitored system, which she wants to leverage to install kernel malware. While the attacker can in principle modify any part of the system, the code integrity validation framework will detect some parts of the changes that the attacker may conduct. Most importantly, it will detect any changes to executable kernel code and will in addition enforce SMEP and SMAP from the hypervisor-level. As a result, the attacker is forced to use data-only malware to infect the kernel. In this process, the control structure that is used by the attacker must reside within kernel's memory space since SMAP is in place. We also assume that the kernel's identity mapping that maps the entire physical memory into kernel space is marked as usermode in the page tables, as was previously proposed by Kemerlis et al. [50]. Finally, we assume that the data-only malware that will be introduced into the system by the attacker is persistent, i. e. will permanently reside within the memory of the target system. Notice that this is usually the case for malware as Petroni and Hicks [68] showed.

6.3. Proposed Approach

In this chapter we aim to detect the control data structures of persistent data-only malware. In this process, we want to achieve three main properties:

Isolation Since the main goal of our framework is to *detect* rather than to *prevent* kernel data-only malware infections, it is crucial that the detection framework is strongly isolated from the system it monitors. This is why we will leverage virtualization as a building block for our framework.

Performance The overhead that our detection framework incurs on the monitored system should be as small as possible. Since we use virtualization as a foundation for our framework, it is thereby of particular importance that we keep the number of VM exists as small as possible as they will heavily impact the performance of the overall approach.

Forensic Due to the ever increasing number of malware attacks, the investigation of incidents becomes more and more important in order to understand the approach of an successful attacker and to avoid future breaches. This is why another crucial goal of our framework is to support forensic investigations in addition to live monitoring. In this regard, its particular important that an human investigator can easily assess and analyze the situation once an anomaly is detected by our framework.

The key idea behind our approach is to detect data-only malware based on its control structure. As described in Section 2.3, the control structure is the most important component of data-only malware that essentially defines which reused instruction sequence should be executed when. Due to this property it is comparable to the code section of traditional malware, which makes it predestined as a basis for a detection mechanism.

To detect the control structure in memory, we use a three-step process. In the first step, we start by checking the integrity of important control flow related kernel objects. This is done for multiple reasons. First, we can use additional contextual information about these kernel objects and secondly these objects contain a lot of code pointers by their design. By validating these objects at the beginning, we can thus increase the performance of our approach, as the code pointers within these known objects do not need to be validated in the steps that follow. We refer to this step as *Kernel Object Validation*. In the second step, we *identify* all code pointers within the kernel's memory space. Based on this information, we will in the third step *classify* the identified code pointers into benign and malicious code pointers by applying multiple heuristics. The combination of these latter two steps is the *Pointer Examination* phase. Figure 6.1 provides an overview of this process. In the following, we will describe these steps in more detail. For the sake of simplicity, we will thereby focus on the Intel x64 bit architecture and the Linux OS. However, most of what we present is equally applicable to other OSs such as Windows and other architectures, such as ARM. While this section will provide an overview of our approach, we will defer a discussion of the implementation details to Section 6.4.

6.3.1. Control Flow Related Data Structures

We first describe control flow relevant kernel objects that we check using special semantic knowledge in the first step of our process.

Kernel Dispatcher Tables and Control Flow Registers The most traditional control flow related data structures are the system call table and the interrupt descriptor tables. As control flow related

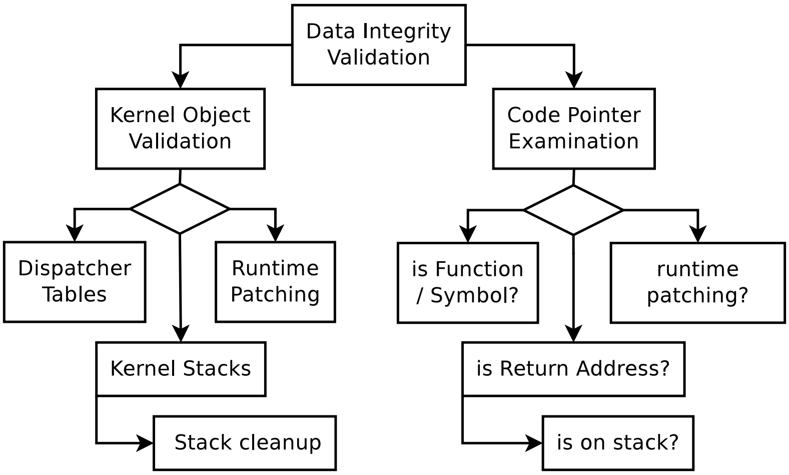

Figure 6.1.: Pointer classification within the proposed framework.

data structures have already seen a lot of attention, we only mention this type of data structures here for sake of completeness. Our system checks every entry within these tables and ensures that it points to the correct function. This can be done by comparing the entire object to the corresponding version inside a trusted reference binary. In this step, we also validate the values of all control flow relevant registers such the MSRs and the *Debug* registers.

Tracepoints Tracepoints are another type of data structure that is control flow relevant. An administrator can use the tracepoints feature to insert arbitrary hooks into the kernel's control flow that are executed always when a certain point in the kernel's control flow is hit and the corresponding tracepoint is enabled. The addresses of the callback functions are stored in a list and are iteratively called by the kernel once the tracepoint is activated. Tracepoints impose a big problem for control flow integrity validation as arbitrary function addresses can be inserted into all tracepoint locations at runtime. To counter this threat, we check that every hook that is installed with this mechanism calls a valid function within the Linux kernel.

Control Structures For Kernel Runtime Patching To manage different runtime-patching mechanisms, the kernel maintains different data structures. These data structures in turn contain pointers to the kernel code, as they need to store the locations where kernel code should be patched at runtime. In our approach we check the integrity of the related data structures.

Kernel Stacks Another examined type of data structure is the kernel stack of each thread in the system. We separate each kernel stack into three parts: At the very beginning of the stack, the active part of the stack is located. This part is empty if the corresponding process is currently executing in userspace. Next to the active part of the stack old stack content is residing, which is not used any more. On the very top of the stack, after all usable space, resides a structure called `thread_info`. It contains management information about the thread, to which this stack belongs to like a `task_struct` pointer and the address limit of the stack.

While it is possible to validate the active part of the stack and its management structure, an attacker could use the old stack content to hide persistent data-only malware. Therefore, this space is reset to zero by our framework when used in live monitoring mode. As the examination of the stack is very complicated, we move that discussion into Section 6.4.

6.3.2. Pointer Identification

After we have validated control flow relevant data structures, we start to identify all other code pointers in memory in the second step. To identify code pointers, we first of all need to obtain a list of all executable memory regions within kernel space. For this purpose, we make use of the page tables that the hardware uses. We also generate a list of all readable pages that do not contain code, as these pages contain the kernel's data. The physical address of the initial level of page tables can thereby be obtained from the `CR3` register.

Starting with the initial page table, we iterate through all referenced pages and extract the virtual addresses of all pages that are marked as both supervisor pages and executable. In this process, we also save the address of all kernel data pages (pages that are marked as non-executable, but have the supervisor bit set) as these pages contain the actual pointers.

Equipped with a list of all kernel code and data pages, we identify all kernel code pointers by iterating through each data page byte by byte and interpreting each 64-bit value as a potential pointer. If the potential pointer should point to a code region (i. e. the 64-bit value represents an address lying within one of the code pages), we consider it to be a code pointer. While this seems like a very simple approach that might produce many false positives, we like to stress that we did not observe any false positives during our experiments with various Linux kernels. In our opinion the primary reason for this is that the 64-bit address space is much larger than the former 32-bit address space and makes it thus much more unlikely that values looking like pointers appear within memory. We will provide a more detailed discussion on this topic in Section 6.5.2.

6.3.3. Pointer Classification

After we have found a pointer, we classify it based on its destination address in order to decide whether it is malicious or benign. In a legitimate kernel there are multiple targets where a pointer is allowed to point to. In the following, we will list those valid targets and describe how we are able to determine to which category the pointer belongs to.

Function Pointer One important type of kernel code pointers are *function pointers*, which are frequently used within the kernel. To determine whether a code pointer is a function pointer, we make use of symbol information that is extracted from a trusted reference binary of the monitored kernel. In particular, the underlying code integrity validation framework implements its own binary loader. In the process, the loader obtains the name and the address of all symbols in order to perform symbol resolution. Amongst these symbols are all functions that the kernel provides. We leverage the symbol list to verify whether a code pointer points to a function or not. In the former case, we consider the pointer to be benign. Otherwise, we continue with the classification process in order to determine whether the code pointer belongs to one of the other categories that we will discuss below. Note that this implies that our approach might still be vulnerable to data-only malware that solely makes use of ret2libc. We discuss this issue in more detail in Section 6.5.2

Return Addresses Another important type of code pointers are *return addresses.* In contrast to a function pointer, which must point to the beginning of a function, a return address can point to an instruction within a function that is preceded by a call instruction. To identify whether a code pointer is a return address, we leverage multiple heuristics. We will further discuss this in the next section. Note that most of the return addresses are located on a stack which is already checked during the Kernel Object Validation phase.

Pointer Related to Runtime Patching A third type of pointer destinations are addresses that are stored by the kernel and point to a location where dynamic code patching is performed. While most of these pointers are contained within special objects that are checked in the Kernel Object Validation step as previously described, there are still some exceptions that must be considered separately. We will describe this issue in more detail in the next section.

Unknown Pointer Destinations Any code pointer that points into executable code and that can not be classified into one of the above categories is considered as being malicious. Therefore a human administrator is notified to further investigate this issue. If multiple malicious pointers are residing on the same page the probability that these pointers are part of a ROP chain is rather high. Whenever a malicious pointer is found it is presented to the user together with the name of the function that the pointer is pointing to. This should help the human to easily judge the situation and to further investigation of the issue.

6.4. Implementation

After describing the general idea of our approach, we cover the details of our implementation in this section. The code pointer examination framework presented in this work is based on the kernel code integrity framework presented in the previous chapter. This framework provides multiple advantages for our implementation:

First, it keeps track of all kernel and module code sections and ensures their integrity during runtime. In addition, it also keeps track of all functions and symbols that are available inside the monitored kernel, as it already resembles the Linux loading process. This ensures that the information about the monitored kernel is binding by its nature, that is it reflects the actual state of the monitored system. In our implementation we can use this database as a ground truth to classify kernel code pointers.

Secondly, the underlying framework keeps track of all dynamic runtime code patching that is conducted by the Linux kernel. We use this information to identify and validate data structures that are related to kernel runtime patching.

Third, our approach is usable for multiple hypervisors, while most of the features also work to analyze memory dumps in a forensic scenario. Currently tests have been conducted with both KVM as well as XEN, while LibVMI [64] was used as the VMI middleware.

6.4.1. Kernel Object Validation

Before we scan the kernel's memory for pointers, we check the integrity of important kernel data structures. This allows us to minimize the parts of kernel data that may contain arbitrary function pointers or other pointers into executable kernel code. The validation of those structures leverages semantic information about the kernel that was generated by the underlying code integrity validation framework or that was manually collected while analyzing the kernel. We here only list a couple of examples to illustrate the requirement of this step.

First, we validate various dispatcher tables and the kernel's read-only data segments. These locations usually contain a lot of kernel code pointers, whereas the target of each pointer is well defined. The validation is performed by comparing these objects to the trusted reference versions of the binaries that are loaded by the underlying validation framework and that have previously been extracted from the trusted reference binaries.

Next, we validate kernel data structures that are used for runtime kernel patching. These are for example:

- Jump Labels (`__start___jump_table`),
- SMP Locks (`__smp_locks`),
- Mcount Locations (`__start_mcount_loc`),
- Ftrace Events (`__start_ftrace_events`).

To validate these structures we semantically compare them to the data that was extracted from trusted reference binaries by the underlying framework. That is, we use the knowledge, we extracted about the structure of these datastructures to ensure, that the same information is encoded in both representations. In addition to these runtime patching control data structures, there also exist data structures in the kernel that are used to actually conduct the runtime patch. For clarification, we will discuss one example for legitimate pointers to kernel code related to self patching here: the kernel variables `bp_int3_handler` and `bp_int3_addr`.

To understand why these pointers are required, we explain how runtime patching takes place in the Linux kernel. If the kernel patches a multibyte instruction in the kernel, it can not simply change the

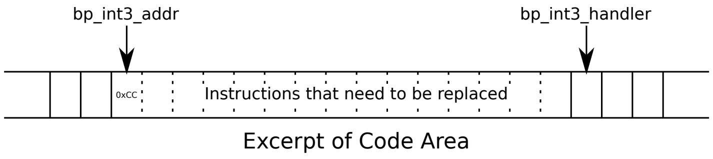

Figure 6.2.: The kernel makes sure to not execute code while it is replaced during self-patching. It replaces the first replaced instruction with an interrupt and handles the interrupt by jumping to the instruction after the modified area. After the rest of the instructions are replaced, the kernel replaces the first byte.

code in question, as otherwise the kernel's code would be in an inconsistent state for a short period of time, which might lead to a kernel crash. Thus, the kernel implements a special synchronization method. It first replaces the first byte of the change with an *int3* instruction. As a result, every CPU trying to execute this instruction will be trapped. Then the rest of the space is filled with the new content. As a last step the kernel replaces the first byte with the right instruction and notifies all waiting CPUs. During this process the address that contains the *int3* instruction is saved in the variable `bp_int3_addr`. This will enable the *int3* interrupt handler upon invocation to determine whether the interrupt originates from the patched memory location or not. While the interrupt handler will simply process the interrupt normally in the latter case, it will in the former case invoke a specific handler, whose address is stored within the variable `bp_int3_handler`. This is depicted in Figure 6.2. In the case of a patched jump label, for example, the *handler variable* will point to the instruction directly after the patched byte sequence, which turns this sequence into a *NOP* sequence during the patching process. Since both of the `bp_int3` variables are never reset once patching is complete, they always point to the last patched location and the last handler respectively. To solve this issue, our framework checks whether the current value of the `bp_int3_addr` points to a self patching location and if the handler address matches the type of patching conducted.

Finally, we iterate through all pages that contain a stack. Each process running in a system owns its own kernel stack that is used once the application issues a system call. To gather the address of all stacks from the monitored host, we iterate through the list of running threads (`init_task.tasks`) and extract their corresponding stacks (`task->thread->sp0`). In case the process is not currently executing within the kernel, the current stack pointer is also saved within that structure. Ideally the process is currently not executing in kernel space in which case its stack must be *empty*.

The unused parts of the kernel stack could potentially be used to hide data-only malware. For this reason, our framework can be configured to reset these parts of the kernel stacks to zero in regular intervals to hinder persistent data-only malware to store the persistent payload in this part of the kernel.

In order to minimize the problem of malware that hides within the active part of a stack we also validate the contents of each stack. Note that while this is not a comprehensive solution to the problem, it still increases the frameworks ability to detect data-only malware that is hiding within a processes kernel stack. That is even though we classify the sequence of function calls on the stack as valid, the executed functionality might eventually still be malicious, although the probability for such a case seems rather low.

In order to validate a stack we use the following approach: For each return address that we find on the stack, we save the addresses of two functions. First, we save the address of the function that the return address is pointing to (`retFunc`). In addition, we also extract the address of the call target, that the call instruction preceding the return address is pointing to (`callAddr`), if possible. This is possible, as in most cases, the destination of the call is directly encoded in the instruction, or a memory address is referenced in the instruction that can in turn be read from the introspected guest systems memory.

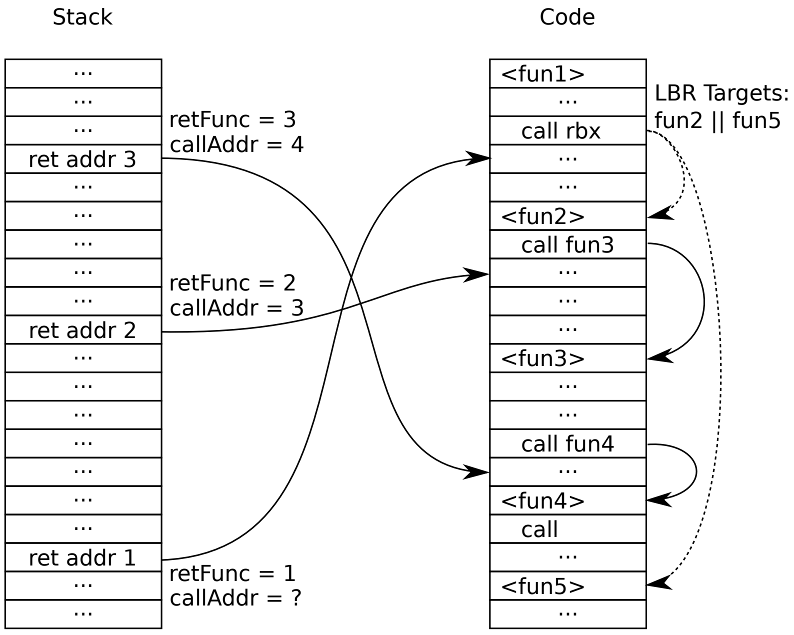

Figure 6.3.: Stack frame validation.

This information is then used to validate the next return address that is found. In particular, the `callAddr` of the next frame needs to match the `retFunc` of the previous stack frame, as the previous function must have called the function, that the return address is pointing to. This process is also illustrated in Figure 6.3.

Since it is not possible to extract all call targets using the mechanism described above, we used an additional mechanism to extract all possible targets of indirect calls. In particular, we monitored the execution of the test systems (see next section) in a secure environment and activated the processors Last Branch Register (LBR) mechanism in order to extract the call and the target address of every indirect branch instruction that was executed by the systems

CPU. Using this mechanism we generated a whitelist of targets for each call for which the target address is generated during runtime. This list is then also used by our stack validation component, as can be seen during the validation of `ret addr 1` in Figure 6.3.

Using this approach we were in our experiments able to validate most of the kernel stacks within our test system. Sometimes the stack validation required further manual investigation, for example in cases where the size of a stack frame was not known and an additional return address was stored in a local variable in a stack frame or in cases where an interrupt is handled during the execution, as the interrupt reuses the current stack for its execution. While our stack validation, in its current implementation is not perfect, it certainly reduces the attack surface further.

The entire problem arises as the stack is currently not designed to be verifiable even under normal circumstances. However, the kernel developers currently discuss an enhancement to the code that would make stack validation more reliable. While this change is intended to simplify the generation of stack traces in case a bug or a segmentation fault is detected in the kernel, this could, once implemented, be used to improve our current return address validation approach and would allow us to remove the external whitelist[1].

Note, after our work on this topic, the propsed patchset to allow for validation of the kernel stack[2] was in the mean time included into the mainline Linux kernel with version 4.6.

6.4.2. Code Pointer Examination

After we have checked important data structures, we scan through the rest of kernel data memory to find pointers to executable kernel code. This is achieved in the following steps: We first extract the memory regions of executable kernel code sections in the monitored virtual machine using the page tables structure. As a second step,

[1] `https://lkml.org/lkml/2015/5/18/545`
[2] `https://lwn.net/Articles/677109/`

we extract the data pages of the monitored guest system. For this purpose, we obtain all pages that are marked as supervisor and not executable in the page tables. These pages contain the data memory of the kernel and therefore all pointers that are accessible from within the Linux kernel. Note that the information we use for our analysis is binding, since it is derived from either the hardware or the trusted kernel reference binaries.

Having obtained the code and data pages, we iterate through the extracted pages in a byte by byte manner. We interpret each eight byte value (independently of its alignment) as a pointer and check if it points into one of the memory locations that was identified as containing kernel code.

If we found a pointer that points to executable kernel memory we first check if its destination is contained in the list of valid functions. We will discuss this decision later in Section 6.5.2.

In case the pointer does not point to a valid function, we in turn check if the pointer is a return address. There are currently multiple approaches used in our framework to identify a return address. First and foremost, a return address must point to an instruction within a function that is preceded by `call` instruction. Consequently, our initial check consists of validating whether the instruction it points to is actually contained within the function.

For this purpose, we will disassemble the function the pointer allegedly points to from the beginning using the Capstone Disassembly Framework [73] and verify that the value of the pointer points to a disassembled instruction and not somewhere in between instructions. If this should be the case, we additionally validate a `call` instruction resides before the instruction that the pointer points to. If any of these conditions should fail, we consider the code pointer not to be a valid return address and continue to check for the next category.

While most of the return addresses that are used within the kernel are stored within one of the kernel stacks, there exist a few functions within the kernel that save the return address of the current function to the kernel heap in order to be able to later identify the current caller of that function. This was first introduced as a debug feature to print the address of the calling function to the user in case of an error. However, in the meantime this feature is also used for other purposes such as timers. For example, the struct `hrtimer` contains a pointer `start_site` that points to the instruction after the call instruction that started the timer.

With such a feature in place and used by the kernel it is hard to differentiate between legitimate return addresses and specially crafted control structures for code reuse techniques. To limit this problem we created a whitelist of all calls to functions that contain the problematic instruction and only allow return addresses in the kernel's data segment if they point to one of the functions in question.

If the pointer does not point to valid function or a return address, the pointer is considered as malicious and a human investigator is notified. At this point the system also enriches the error message with the name of the function or symbol, that the pointer is pointing into. With this additional information the analyst may decide if the chain is malicious or not.

6.4.3. Detection of Dispatcher Calls

In the previous section we have discussed code pointers that are accepted as valid by our framework, however, to mitigate possible attacks we further restrict the list of allowed pointers.

A wrapper function that basically calls one of its arguments with a certain offset, would be very helpful for an attacker. The Linux kernel currently contains some functions with exactly that property. Examples for such a function are: `do_one_initcall` or `ops_init`. Therefore our system uses a blacklist approach and checks that no pointer to these functions is contained within memory. To generate a comprehensive blacklist we conducted an experiment where we extracted all indirect branch instructions from a running Linux kernel together with the corresponding target for each branch using the CPUs LBR feature. Therefore we gather a list of all call instructions with their corresponding targets. If a branch instruction is targeting multiple other functions, we assume that the function containing the branch is usable by an attacker and therefore no pointer to that function is allowed in memory. Another type of indirect branch instructions are used in the Linux kernel is for example directly at the beginning of a function. The called function then calls a function whose address is provided in as an argument to the function. However, this still requires the function pointer of the transitively called function in memory. As such the call target is detectable with our system.

6.5. Evaluation

In this section, we evaluate our approach using the prototype implementation described in the last section. In order to determine whether our framework is able to achieve the goals that we set in Section 6.3, we will first determine its performance characteristics, before we evaluate its effectiveness against data-only malware in both live monitoring as well as forensic applications. We follow this with an in-depth discussion of the security aspects of our system.

6.5.1. Experiments

Our host system consisted of an AMD Phenom II X4 945 CPU with 16 GB of RAM running Linux kernel version 3.16 (Debian Jessie). As guest systems we used two different VMs running Linux 3.8 as well as Linux 3.16. Each VM had access to two virtual CPUs and 1 GB of RAM. In these experiments, we used XEN as the underlying hypervisor.

Performance and False Positives First of all, we evaluated the performance of our system as well as its susceptibility to false positives. For this purpose, we used the *Phoronix-Test-Suite* to run a set of Benchmarks on our system. In detail, we ran the *pts/kernel* test suite. This suite consists of different tests programs that look at different aspects of the system's kernel performance such as file I/O and CPU intensive tasks. We conducted the benchmark three times on each test kernel. During the first set of tests, we disabled all external monitoring to obtain a baseline of the normal system performance. In the second test set, we then enabled the code integrity validation component to be able to differentiate between the overhead of our framework and the code integrity validation system. Finally, we enabled both the code integrity validation component as well as our new pointer validation module in order to identify the additional overhead that our system incurs. During the tests, the integrity validation component was executed in a loop, if enabled, to stress the guest system as much as possible. The results of the benchmarks of each set of experiments as well as the overall performance degradation are shown in Table 6.1 for Linux 3.8 and in Table 6.2 for Linux 3.16.

While evaluating the Linux 3.8 kernel, the kernel contained 80 code pages and 426 data pages. One complete *Code Integrity Validation* was completed in 255.8 *ms*, while in the experiment with Code Integrity Validation *and* Pointer Examination enabled, one iteration took 567.58 *ms* (341.78 *ms* for CPE). The Linux 3.16 kernel that was used during our evaluation contained 408 code pages and 986 data pages. The Code Integrity Validation alone took 639.8 *ms* per iteration, while the combined CIV and Pointer Examination took 962.0 *ms* per iteration (322.2 *ms* for CPE). Note that these values are calculated mean values. This shows that it requires less than 1 *ms* in the mean to check the integrity of one page.

As one can see the performance overhead that our framework incurs is very small. In fact, the use of the underlying code integrity validation component incurs a larger overhead than our CPE framework. The performance impact of our system is for the most benchmarks well under one percent. The main reason for this is that our framework, in contrast to many other VMI-based approaches, uses passive monitoring of the guest system wherever applicable. As a result, the guest system can execute through most of the validation process without being interrupted by the hypervisor, which drastically reduces the performance overhead of the monitoring. Only for the FSMark benchmark a performance degradation of about 2.65 percent is noticed on Linux 3.8. This degradation can not be seen in

Test (Unit)	w/o	CIV (%)	CIV & CPE (%)
FS-Mark (Files/s)	32.57	30.10 (8.21%)	31.73 (2.65%)
Dbench (MB/s)	69.84	66.53 (4.98%)	71.54 (−2.38%)
Timed MAFFT Alignment (s)	20.63	20.70 (0.34%)	20.63 (0.00%)
Gcrypt Library (ms)	2857	2853 (−0.14%)	2837 (−0.70%)
John The Ripper (Real C/S)	1689	1689 (0.00%)	1688 (0.06%)
H.264 Video Encoding (FPS)	35.38	35.23 (0.43%)	35.31 (0.20%)
GraphicsMagick 1 (Iter/min)	95	95 (0.00%)	95 (0.00%)
GraphicsMagick 2 (Iter/min)	58	58 (0.00%)	58 (0.00%)
Himeno Benchmark (MFLOPS)	593.59	585.73 (1.34%)	586.24 (1.25%)
7-Zip Compression (MIPS)	4715	4702 (0.28%)	4706 (0.19%)
C-Ray - Total Time (s)	130.96	131.00 (0.03%)	130.99 (0.02%)
Parallel BZIP2 Compression (s)	36.35	36.58 (0.63%)	36.47 (0.33%)
Smallpt (s)	445	445 (0.00%)	446 (0.22%)
LZMA Compression (s)	234.50	236.39 (0.81%)	236.12 (0.69%)
dcraw (s)	124.24	124.38 (0.11%)	124.35 (0.09%)
LAME MP3 Encoding (s)	25.20	25.19 (−0.04%)	25.19 (−0.04%)
Ffmpeg (s)	27.00	27.02 (0.07%)	26.82 (−0.67%)
GnuPG (s)	15.34	14.98 (−2.35%)	14.94 (−2.61%)
Open FMM Nero2D (s)	1137.17	1148.95 (1.04%)	1144.94 (0.68%)
OpenSSL (Signs/s)	173.70	173.73 (−0.02%)	173.80 (−0.06%)
PostgreSQL pgbench (Trans/s)	115.11	114.69 (0.37%)	115.21 (−0.09%)
Apache Benchmark (Requests/s)	10585.45	10481.21 (0.99%)	10506.23 (0.75%)

Table 6.1.: Results of the Phoronix Test Suite for Linux 3.8.

Test (Unit)	w/o	CIV (%)	CIV & CPE (%)
FS-Mark (Files/s)	30.90	31.37 (−1.50%)	31.67 (−2.43%)
Dbench (MB/s)	61.42	60.76 (1.09%)	61.04 (0.62%)
Timed MAFFT Alignment (s)	20.74	20.79 (0.24%)	20.75 (0.05%)
Gcrypt Library (ms)	3747.00	3740 (−0.19%)	3733 (−0.37%)
John The Ripper (Real C/S)	1693.00	1693 (0.00%)	1692 (0.06%)
H.264 Video Encoding (FPS)	34.60	34.32 (0.82%)	34.35 (0.73%)
Himeno Benchmark (MFLOPS)	598.71	582.78 (2.73%)	585.78 (2.21%)
7-Zip Compression (MIPS)	4850.00	4805 (0.94%)	4730 (2.54%)
C-Ray - Total Time (s)	89.80	89.81 (0.01%)	89.80 (0.00%)
Parallel BZIP2 Compression (s)	31.25	31.41 (0.51%)	31.37 (0.38%)
Smallpt (s)	407.00	407 (0.00%)	407 (0.00%)
LZMA Compression (s)	236.62	241.49 (2.06%)	242.17 (2.35%)
dcraw (s)	117.54	117.47 (−0.06%)	117.29 (−0.21%)
LAME MP3 Encoding (s)	23.39	23.41 (0.09%)	23.40 (0.04%)
GnuPG (s)	13.72	13.65 (−0.51%)	13.98 (1.90%)
OpenSSL (Signs/s)	173.63	173.37 (0.15%)	173.57 (0.03%)
Apache Benchmark (Requests/s)	9504.78	9156.01 (3.81%)	9383.66 (1.29%)

Table 6.2.: Results of the Phoronix Test Suite for Linux 3.16.

the results of the benchmark on Linux 3.16. While using the guest system with monitoring enabled we did not observe any noticeable overhead from within the guest system. This clearly shows that our framework can achieve its performance goal that we set in Section 6.3 and is from a performance point of view well suited for real world applications.

Sometimes the results even showed, that the tests where better with our pointer examination framework enabled than without our framework. We argue, that this may be due to the fact, that the performance impact of our system is much smaller than the impact of other standard software within the tested Debian system that also influenced the result.

At the same time we did not observe any false positives during our experiments. That is, when enabled, our system could classify all of the pointers it encountered during the validation process using the heuristics we described in Section 6.3. However, note that we can, due to the design of our system, not rule out false positives entirely. This is why we will perform a more detailed discussion about the possibility of encountering false positives in Section 6.5.2.

Malware Detection After having evaluated the performance of our system and touched upon its susceptibility to false positives, we continued to evaluate the effectiveness of our framework against data-only malware. For this purpose, we infected our test VMs with the persistent data-only rootkit presented by Vogl *et al.* [100]. We chose this rootkit, since it is, to the best of our knowledge, the only persistent data-only malware available to date.

While our framework did not detect any malicious code pointers during the performance experiments, our system immediately identified the various malicious control structures used by the rootkit. In particular, our system identified the modified `sysenter` MSR and the modified system call table entries for the `read` and the `getdents` system call during the prevalidation step and thus classified the system as malicious. As these hooks are also found by other systems, we then removed these obvious manipulations manually and once more validated the system state. While the prevalidation step yielded no results in this case, the pointer validation found all of the malicious code pointers in memory. This proves that our framework can be very effective against data-only malware even if the malware avoids the manipulation of key data structures such as the system call table.

Finally, to evaluate the usefulness of our framework in forensic applications, we conducted an experiment where we randomly installed the rootkit on the test VMs while we periodically took snapshots of the guest systems. Our system detected all of the infected snapshots reliably. As before, we did not observe any false positives in this test.

6.5.2. Discussion

In this section, we will provide a detailed discussion of the security relevant properties of our system.

False Positives Although, we did not encounter false positives throughout our experiments, we cannot rule out false positives entirely, since our system relies on heuristics to identify code pointers. However, we like to stress that we consider the probability of encountering false positives in our system to be quite small on a 64-bit architecture. To encounter a false positive with our system, we essentially would need to find a value in kernel space that contains the address of a kernel code section even though it is not a pointer. Since the virtual address space on a 64-bit system has a size of $1.8 * 10^{19}$ bytes and the kernel code section typically only has a size of 15 megabytes at maximum, the chance of encountering such a rare case is merely $8.5 * 10^{-11}\%$. And that is only the case if the kernel is not optimized as the kernel code section even becomes smaller in this case. In other words, we consider a 64-bit address space to be sufficiently large that the chance of random data looking like a pointer by chance are small at best. Consequently, we assume that false positives are not a big issue in most scenarios.

Detection vs. Prevention As we are examining data structures on a regular basis and not using an event-based notification mechanism, we are trading soundness of our approach with performance. In the worst case, we may not prevent an external intrusion using data-only malware. We are nevertheless able to detect the control structure that is used for data-only malware in memory, after the system was compromised, as pure data-only malware is usually very large in size, up to 2 megabytes according to [100]. To prevent the execution of data-only malware entirely we would need to examine the current stack, whenever the stack pointer is switched during legitimate execution. Not only does no such hardware notification mechanism exist, it would also significantly decrease the performance of our system. Also there are many legitimate reasons for stack switching, such as process scheduling or context switches for interrupts. Therefore it is not possible to detect and identify the pivot sequence of data-only malware in all cases.

ret2libc When searching for malicious pointers in memory, we currently do not penalize pointers that point to function entry points. As a consequence, our system is at the moment unable to detect data-only malware that solely makes use of entire kernel functions to perform its malicious computations. While this is certainly a weakness of our approach, its important to know that this is a very common limitation that almost all existing defense mechanisms against code reuse attacks face [28, 82]. In fact, to the best of our knowledge, the detection of ret2libc attacks still remains an open research problem, which we will further discuss in the following.

While ret2libc is a powerful technique that is very difficult to detect, we argue that it is actually quite difficult to design pure data-only malware that solely relies on entire functions to run on a 64-bit architecture. The main reason for this is that in contrast to 32-bit systems, function arguments in Linux and Windows are no longer passed on the stack on a 64-bit architecture, but are provided in registers instead. As a consequence, to create 64-bit ret2libc data-only malware, an attacker must actually have access to “loader” functions that allow her to load arbitrary function arguments into the registers that the calling conventions dictate. Otherwise, without access to loader functions, the attacker would be unable to pass arguments to any of the functions she wants to invoke, which would significantly restrict her capability to perform attacks.

It goes without saying that such loader functions are probably rare if they exist at all. A possible approach to further reduce the attack surface could thus be to analyze the kernel code for such loader functions. If they should exist, one can then monitor the identified functions during execution to detect their use in ret2libc-style attacks.

Return Addresses If an attacker requires gadgets in addition to entire functions to execute her persistent data-only malware (e. g. to load function arguments into registers), the only location that she can place the required control structure to without being detected is the kernel stack of a process. Should a code pointer that points inside a function appear anywhere else within the kernel memory, it will be classified and identified as malicious by our system. In addition, due to the fact that our system enforces SMAP from the hypervisor, the control structure cannot be placed in userspace if it should be executable from kernelspace. This only leaves a kernel stack for kernel data-only malware. But even here the attacker faces various constraints. First of all, she can only make use of gadgets that appear legitimately in the code and that are preceded by a call instruction, since all other pointers into a function would be classified as malicious. Secondly, as the kernel stack where the control structure resides may also be used by the process it belongs to, the attacker must ensure that her persistent control structure is not overwritten by accident. While this is not necessarily an issue for data-only exploits, this is crucial in the case of persistent data-only malware as the persistent control structure of the malware may never be changed uncontrollably. Otherwise, if the control structure would be modified in an unforeseen way, it is very likely that the malware will fail to execute the next time it is invoked. This is comparable to changing the code region of traditional malware. This is also why our system zeroes all data that belongs to a memory page that is part of the kernel stack, but currently resides at a lower address than the stack pointer points to as a final defense layer. Since this data should be unused, zeroing it will not affect the normal system behavior. However, in the case of persistent data-only malware, this approach may destroy the persistent control structure of the malware, which will thwart any future execution. This will be the case if the malware is currently executing while our system performs the validation. Since an attacker cannot predict when validations occur as our system resides on the hypervisor-level, this makes it difficult for her to stay unnoticed in the long run.

As a further enhancement one could also set the kernel stacks of processes that are currently not executing to not readable within the page tables. This could for example be done during the process switch. As a result, the attacker would only be able to use her control structure when the process on whose kernel stack the structure resides is currently executing. This raises the bar if the attacker wants to hook the execution of all processes instead of just one, which is generally the case.

Taking all this into account we argue that while our system cannot eliminate the threads of persistent data-only malware entirely, it significantly increases an attackers effort to evade detection and thus reduces the attack surface.

6.6. Summary

In this chapter, we have proposed *Code Pointer Examination*, an approach that aims to detect data-only malware by identifying and classifying pointers to executable memory. To prove the validity and practicability of our approach, we employed it to examine all pointers to executable kernel memory in recent Linux kernels. In the process, we discussed important control flow relevant data structures and mechanisms within the Linux kernel and highlighted the problems that must be solved to be able to validate kernel control data reliably. Our experiments show that the prototype, which we implemented based on the discussed ideas, is effective in detecting data-only malware, while only incurring a very small performance overhead (less than 1% in most of the benchmarks). In combination, with code integrity validation, we can thus provide the first overall approach to kernel integrity validation. While our framework still provides a small attack surface, we argue that it considerably raises the bar for attackers and thus provides a new pillar in the defense against data-only malware.

Chapter 7

Dynamic Integrity Validation for Userspace Applications

Up until now, we have presented how our kernel integrity framework is able to (1) handle code integrity validation for dynamically modifying kernel code and is able to (2) detect control structures of code reuse malware within kernel data. In this chapter we will investigate to what extent our work is also applicable to userspace applications. With this we give an answer to the research question **Q4** that we raised in the beginning of this thesis.

7.1. Problem statement

In this section, we describe our efforts to adapt the approach developed in our previous research to userspace applications. With this we extend the scope of our research and provide integrity mechanisms for general application software. For this, we first set our focus on code integrity validation. While the codebase of userspace applications is static during runtime, we take an in depth look on the application loading process. We examine if there are mechanisms within the loading process, that complicate the validation code integrity in practice due to differences in the loading process, that make a generation of the ground truth cumbersome.

In a second step, we apply our code pointer examination technique to userspace memory. We do this, as the concept showed encouraging results when applied to kernel memory. The basic idea is to not only scan for kernel code pointers within kernel data regions, but to extend this validation to the entire guest VM memory. With this, we aim to detect data-only kernel rootkits before an attack against the victims kernel is conducted. We intend to detect control structures, once they are loaded into the memory of a userspace program. This is especially interesting for malware, which is generated on the target machine on-the-fly, such as, for example Just-In-Time (JIT)-ROP [88].

Lastly, we aim to investigate, to what extent CPE may be applied to detect code reuse attacks that directly target userspace processes. In this last investigation, we discuss if and to what extent it is possible to apply CPE also to userspace processes. As an attacker model for this investigation, we assume, that an attacker introduces a ROP-chain into a process, that leverages gadgets provided by the application code, for example the underlying C standard library. We also discuss the problems we encountered during our experiments and propose improvements that are required to enhance the current situation.

In the following, we also provide a technical description of our kernel integrity framework and how it was extended to also support information generation and integrity validation for userspace applications.

In summary, we propose the following contributions in this chapter:

- We extend our VMI-based kernel integrity framework to also validate the integrity of userspace processes.
- We show, that CPE is able to detect control structures of code reuse malware that targets the OS kernel within userspace memory.

- We discuss the applicability of CPE to detect control structures of code reuse malware that directly targets userspace applications.

7.2. Code and State Integrity Validation

As previously introduced we now first focus on the dynamic validation the parts of userspace applications that should be static during runtime. As this first seems to be an easy task, we hereby do not only focus on the program code and static data sections, but we also focus on the current state which the kernel holds about the executing process.

7.2.1. Process State Validation

In order to validate userspace related memory within the VM, we first extract and validate the information, the kernel stores about the process from the monitored kernel. As a first step, our system checks the correctness and consistency of the information the kernel holds about a process. This effort is done in order to detect malware that modifies that state on its behalf, for example in order to loads itself into the virtual memory of a process. One important part of that information are the virtual memory mappings of each loaded process. The kernel maintains a list of all memory regions that are mapped into a process's virtual address space, called Virtual Memory Areas (VMAs).

To validate the information about the memory mappings, we extract both the VMA information that the kernel maintains as well as the root of the page table directory for a given process. Note, that the address of the page table directory may be derived from the hypervisor, while the process in question is executing. In addition this information is also stored within kernel internal datastructures. We then validate, that for each page that is mapped within the page tables of the process, a corresponding mapping within the kernels VMA information exists. Through that step we ensure that the kernels memory management information is consistent with the contents of the page tables and no additional, possibly malicious memory is mapped within the process. The integrity of this information as well as the contents of the corresponding pages are then validated in a later step.

In a next step, we validate important parts of the applications environment. Our system extracts and first checks for the existence of suspicious settings. This is important, as an attacker might be able to alter the code of the program by leveraging special environment variables, as for example `LD_LIBRARY_PATH` or `LD_PRELOAD`. While this technique is already well known, its detection still increases the integrity guarantees we are able to make about the programs executed within the monitored VM. If an unexpected entry is detected, our framework issues a warning to the administrator. This is done, as our framework is, by design, unable to decide if there is a legitimate reason for that dangerous setting in the current situation.

Next, our framework extracts information about the executable binary which was used to load and start the process from the monitored Linux kernel. This includes the name of the executable and its path within the file system of the monitored VM. Note, that the operating system stores this information in multiple places. Once inside its process management datastructures and once within the environment information of the process (i. e. in `argv[0]`). We check that this information is consistent, as a malicious process may overwrite the information within its environment block. This is a problem, as this information is used when an administrator gathers information about the system, by listing all active processes with the `top` command.

Our framework then searches for the executed binary within a whitelist of all allowed executables and libraries stored within our trusted environment. With this we ensure, that only legitimate applications are allowed to execute on the monitored target VM. This is similar to a feature called User Mode Code Integrity (UMCI) which is part of Device Guard, a security mechanism recently introduced by Microsoft[1]. If the corresponding binary is not contained within the trusted store, the administrator is informed about the malicious process.

7.2.2. Process and Library loading

The next step in the userspace code integrity validation process is the actual loading of the executable binary file in order to generate a ground truth for the final validation of the memory contents. If the executable is found in the trusted store, the framework starts to emulate the application loading process.

During this process, in the relocation step, the binaries `.got` (Global Object Table) and `.plt` (Program Linker Table) sections are initialized by our framework. These sections typically contain links to all external symbols that are referenced by the library. With this step we (1) aim to be able to validate the contents of all VMAs that are mapped into the process as executable and (2) are able to compare the contents of both the `.got` and `.plt` sections within the guest VMs memory against a trusted reference version. The second point is also important, as instead of modifying the read-only mapped code segments directly, malware typically changes these datastructures to be able to access functions within other libraries, that are not originally referenced by the binary. Again, this technique is not new in our thesis, but increases the integrity guarantees provided by our framework. Thus we include this check within our framework.

During the replication of the application loading process, we noticed two limitations, which make it hard to fully validate the integrity of a process that is loaded into memory. Both of these issues are related to the way how relocations are implemented within Linux userspace applications. To reiterate, the idea of relocations is, that a library may reference an external function or symbol, independent of the memory addresses that the different libraries are loaded to. For this, the userspace binary loader inserts a pointer to the referenced

[1]https://technet.microsoft.com/de-de/library/dn986865(v=vs.85).aspx

external symbol into a location in memory that is known by the loaded library during load time. However, a feature exists (*lazy binding*), which does not require the relocation to happen directly during load time of the library, but symbols may also be resolved on demand during runtime. This is done to optimize the load time of large applications that contain a lot of relocation entries. Lazy loading was the default option for relocations within Linux for a long time. Due to lazy loading, the Global Object Table (`.got`) may have a different state depending on which external symbols have already been used during program execution. This, however, is just a consistency problem, when checking the integrity of the `.got` section in memory. However, lazy loading is not supported by or framework. The decision to not support lazy loading is not critical, as lazy loading nowadays only provides minor impact on performance and may easily be disabled in a security aware context. In addition, there was also a new loader feature introduced, which aims to prevent applications to maliciously modify the contents of the `.got` section. This feature also demands to fully execute the relocation at load time and then maps the corresponding data structure as read-only within the monitored process. This feature is called *relro*. Thus the first limitation is not critical, as further security mechanisms already exist, that solve the problem.

In contrast, the second limitation we encountered during our investigations is more critical. In 2011 `gcc` introduced a new type of relocation[2] by implementing support for a new symbol type extension, called *IFUNC*[3]. In effect, this mechanism is similar to the alternative instructions mechanism present in the Linux kernel, which was presented in Section 5.2 of this thesis. It allows the application developer to select a specific implementation of a function during the load time of the binary. The decision may depend on external conditions like the current hardware and software state. The difference between the *IFUNC* mechanism and the kernels alternative instruc-

[2]Version 4.6, March 2011
[3]`STT_GNU_IFUNC`

tions mechanism is that instead of selecting a specific implementation for the function depending on concrete criteria, the loader executes an external function that is provided by the application developer during the relocation step. Instead of the address to the requested symbol, the relocation entry contains the address of the selection function. During relocation of the corresponding symbol, the loader simply executes the given function, which sole purpose is to select which concrete implementation is selected for the current relocation. The global object table is then updated with the functions return value.

Our framework is currently not able to reproduce a full list of possible relocation entries and thus is unable to entirely validate the contents of the global object table of an executable in memory in case the binary uses the *IFUNC* mechanism. This is, as our validation framework currently does not support symbolic execution and we do not want to execute external code within our security framework. We did not implement this as part of this thesis, as the `.got` data structure is mapped as read only by the loader anyway for all systems that enable the nowadays common `relro` security mechanism. Note, that this problem also applies to other state of the art CFI tools as there is, to the best of our knowledge, no related work that discusses this issue.

To support this feature, our framework would need to be extended to support symbolic execution to automatically extract the address of the intended function from the binary code in a secure way. It, however, should not only be able to extract the address of the function that should be executed in the current environment, but in addition also extract a list of all possible functions together with information about the decision criteria. With this information also a ruleset for each possible implementation could be created, which could be used during the `.got` validation step within our framework.

7.2.3. Detection of additional code pages

After the loading of a trusted reference version of the process memory is conducted by our framework, we validate, that only libraries are loaded to the process space, that are contained within the binaries dependency graph. If a library is found, that is not contained as an explicit dependency, the administrator is notified. This might either be a executable code page, that has no corresponding VMA entry within the kernel, but might also be a library, that has a corresponding VMA entry in the kernel but is not listed as a dependency of the trusted version of the loaded program. Note, that there are legitimate reasons, why additional libraries might be loaded into a process (i. e. due to the use of dynamic library loading (`dlopen`). However, we think that an administrator needs to be aware of this fact. During this investigation, we detected, that sometime legitimate libraries are loaded into the address space of every process even if they are not listed as an explicit dependencies. In our case we found authentication related libraries like, for example `libnss`. Therefore, a whitelist of additional libraries needs to be configured by an administrator.

7.2.4. Userspace Code Validation

In a final step, we validate the code integrity and code identity of every code page loaded within the process. We iterate through all executable pages and check if the kernels VMA information contains a related library. If that is the case, we compare the trusted version of that libraries code section which was generated in the previous loading steps with the memory page in question. In our case, userspace code pages in memory have never been legitimately altered from their trusted versions that where loaded from the binaries. The result of this comparison shows, that code sections within Linux userspace processes are in fact static. Thus comparison of hashes and signatures is an adequate mechanism to validate code pages in Linux userspace processes. Still it is not a valid approach for kernel code, as we could show in our previous research.

7.3. Code Pointer Examination

After we have described the static userspace code and state validation conducted by our framework, we now present how our concept of CPE is applied to userspace applications executed within the monitored VM.

7.3.1. Kernel Code Pointers within Userspace Applications

Our first goal within this line of research is to detect persistent data-only malware that is targeting the operating system kernel while it is still contained inside a userspace application on the attacked machine. This is both interesting if the data-only malware is uploaded to the victims system through a userspace service as well as in cases, where the code reuse malware is directly created on the target machine on-the-fly. For this, we scan the entire systems memory for potential pointers to kernel code. We extract all physical pages from the test machine which are mapped into any userspace application as both readable and writable. The restriction to only scan on pages that are both readable and writable was made due to the assumption, that an attacker who has already gained userspace access to the machine may use these pages to prepare the control structure for code reuse which is then in turn used to exploit the systems kernel.

To efficiently identify all memory locations that contain pointers to kernel code, we combine all physical pages that are mapped into any executing process. That is, we do not conduct this scan on a per process basis. For this, we iterate through the page tables of all processes currently executed on the target guest OS and collect a list of all physical pages that are mapped to these processes. This is done in in an effort to optimize the performance of the pointer detection, as we can combine memory mappings in multiple processes that map to the same physical memory. After this list is generated, we extract all pages from that list that are both read and writable, and scan through these pages on a byte-by-byte basis.

This process is implemented in two different modes of operation. In the first setting, we scan for pointers to kernel code as described above. We then also created an additional test, that does not check the first two bytes of an address, when a pointer is suspected. This is to represent an optimization done in the Linux kernel. Modern processors have an address length of 64 bit, but usually only 48 physical address lanes are available. For this, the topmost 16 bits of an address are currently still unused during an access to the physical memory hardware. As enforced by the hardware, in userspace applications, these bits are checked for consistency ($= 0$) before an access to the corresponding memory is performed. In kernel space, however, this integrity validation step is typically omitted. The first two bytes of a kernel address may be in a random state although the pointer is technically still valid. While we expect only a very small number of false positives in our first test case, our intuition is, that we might suffer from a lot of false positives in the later case. We will give an overview over the actual results of these tests in Section 7.4.

7.3.2. Classification of Userspace Code Pointers

In a final set of experiments, we investigate, whether CPE may also be used as a simple heuristic to reliably detect control structures of data-only malware which is directly targeted against a userspace application. For this, we apply our method to benign userspace memory and expect the number of unidentifiable code pointers to be low in practice.

7.3.2.1. Proposed Approach

In this step, we iterate through the memory of every process that is currently executed on the target guest OS in order to detect code pointers to their respective code sections. Thereby, we intend to detect CRAs that leverage the codebase of large binaries or libraries. In practice, for example gadgets in the `libc` library are commonly used to build return-oriented programs. For this, we extract a list of all writable pages of each process and scan through these pages (again in a byte-by-byte manner). This time, we check if the current value is in the range of an executable code segment that is loaded into the current process. As in the kernel code pointer examination case (Chapter 6), we expect to find legitimate valid pointers in this case. Therefore, we further classify the detected pointers into multiple classes.

7.3.2.2. Pointer classification

For classification, we first check, if the detected pointer, does not only point into an executable page, but also points to a memory location that is occupied by an executable section of the loaded library. This is done, as executable sections do not always occupy entire pages, but may only use parts of a page. In theory, the rest of the page should be unused and the content of the page at that specific location should be zero. In fact, this is not always the case in practice, as a physical page may contain both code and data and may be mapped into virtual memory twice, while each mapping has different page permissions. To mitigate this issue, the binary loader could be enhanced to create two separate physical pages and delete the contents on the page that are not intended to be part of the corresponding mapping. Pointers that to not point into a code section of the binary or library are currently ignored by our framework.

Next, we further classify the pointers that point into an executable section of a binary. Within an executable mapping, only the `.text` section of a binary should contain valid instructions. The rest of the sections mapped into the executable segment contain information that is relevant for the loader or the control flow of the application and that should not be changed by the application. For this reason our framework contains a list of sections that are also typically part of an executable mapping but are not intended to be executed. These additional sections are placed inside this mapping as it is mapped as read-only and thus can not be modified during runtime.

Examples for whitelisted sections are the string section, that contains among other things the names of the symbols that are referenced in the relocation phase, the `.rodata` section or hash and note sections that are created by the compiler. We propose to strictly separate the different sections within virtual memory and introduce a separate VMA that only contains read-only contents of the binary. With this, an attacker is unable to use unintended instruction sequences that are part of sections that do not need to be executed. While the previously introduced *relro* security mechanism introduces such a read-only memory mapping, currently not all read-only sections are moved into this mapping. We count the number of pointers that point to a whitelisted executable section, but still, our framework currently does not further analyze these pointers. It is part of future work to evaluate, to what extent the content of these pages may be used as gadgets for code reuse malware.

If a pointer is not sorted out in the previous categories it points into a section of the binary that contains executable code. Thus it should belong to one of the following categories: (1) a known symbol or (2) the start of a function within the code or to (3) a return address. To identify pointers of the first and second type, we use the information that was generated during the binary loading process of our framework, which we consult, if a symbol or function is known at the specified location. In this step we must differentiate between two types of symbols. On the one hand, we may find an exported symbol. The address of that symbol is known due to the relocation information gathered during the reproduction of the loading process. On the other hand, the symbol may be an internal symbol. The information about internal symbols can currently only be gathered, if the debug information of the specific library is not stripped from the library within our trusted binary store. Note, that this is not a limitation of our approach, as the addresses of internal functions may also be reverse engineered from the binary with, for example, an approach introduced by Andriesse *et al.* [6]. To identify return addresses, our framework disassembles the function, the pointer points to using the capstone library [73] and checks, if the targeted instruction is directly preceded by a call instruction. Note, that we only expect to encounter pointers that are classified as return addresses inside an applications stack segment. In our experiments we will summarize these pointers into the following categories:

1. **Overall Ptrs:**
   Number of pointers that was detected in data memory.

2. **Unique Ptrs:**
   Number of unique pointers that was detected.

3. **Ptr to `.text`**
   Number of pointers that point to code and where further analyzed.

In case the pointer can not be classified into one of the previous categories, we assume the pointer to be malicious. In this case, we further classify the pointer into multiple categories:

4. **Unknown Ptrs:**
   A superclass, that contains all unidentifiable pointers.

5. **Invalid Instruction:**
   Pointers that point to an Invalid Instruction.

6. **Unintended Instruction:**
   Pointers that point to an unintended, but yet valid instruction.

7. **Unintended Return Address:**
   Pointers that point to a unintended instruction after an unintended call instruction.

For further classification, we check, if the pointer points to a valid instruction. If this is the case, we disassemble the entire function the pointer points into and check, if the pointer points to an *intended instruction*. If the instruction was an *unintended but valid instruction*, we check if the instruction preceding the target instruction is a branch instruction (e. g. a call instruction). In this case, we classify the unknown pointer as *Unintended Return Address*.

For pointers that point to an unknown but valid instruction, we check, if the instruction sequence may be used as a gadget for a return oriented program. For this we check, if it ends with a return instruction and does not contain any illegal or invalid instructions. We then output information about the length of the gadget. Note that we currently do not further check, if gadget is actually usable in practice. That is, we do not further extract higher level semantic information about the potential gadget. This could be enhanced by incorporating symbolic execution in future work. That is, if our approach shows to be valuable in our experiments. Our system may be extended to symbolically execute the detected gadget in order to further classify its usability in an actual CRA. With this the number of false positives can be further reduced in order to limit the workload of the human analyst that needs to interpret the output of our system.

We expect our method to be applicable to userspace applications if the number of false positives is small (ideally zero) when applied on a benign environment. We nevertheless expect a small number of false positives and also expect that our general mechanism needs to be adapted for every analyzed application in order to assess the generated results.

7.3.2.3. (Un-)Legitimate Code Pointers

During our initial experiments we found two major problem classes, which make the application of CPE to userspace applications cumbersome in practice: First, programs or libraries may be loaded into a non randomized address range at the beginning of virtual memory. In this case, we might misinterpret legitimate counter values as malicious code pointers. Second, memory write accesses are not always conducted in an aligned way and parts of old pointers may be overwritten by new content. In the following we will shortly discuss these problems and show how these problems may be alleviated in practice.

The first problem lies in the virtual address range, where the program or library code of a program is loaded to within the applications virtual memory address space. If this virtual address is relatively near to the beginning of virtual memory and thus code addresses are small, arbitrary counters may look like malicious code pointers in our investigation. In practice, this problem arises, when an executable application is loaded to a fixed address in memory during the loading phase ($0x400000$ on amd64-based Linux systems).

This, however, is a well known issue, as it also allows an attacker to guess the address of specific instructions in memory. Due to this static loading address applications are also currently not able to take the advantage of ASLR. In order to randomize the address space (ASLR) two compiler extensions have been introduced, that allow to load the binary code to randomized addresses in memory: Position Independent Code (PIC) for position independent *library* code and Position Independent Executable (PIE) for position independent *application* code. The load address of position independent code is generated during load time of the application.

Both PIC and PIE have already been introduced some time ago. While PIC is already widely adopted, Linux distributions such as Debian are only currently in the transition to enable PIE for all applications that are shipped with the distribution. PIE was enabled per default in Debian since October 2016 starting from `gcc-6 6.2.0-7`[4].

We expect the number of legitimate counters that are misinterpreted as pointers to be much lower in case both PIC and PIE are enabled for an application and the code sections are loaded to randomized addresses. So, in addition to the previous classification, we classify each detected code pointer according to whether the address of the text section was randomized during loading or not.

The second problem that is causing a lot of unidentifiable pointers, when applying CPE to userspace processes in practice, are partial

[4]`https://tracker.debian.org/news/806845`

overwrites. The reason for this problem is, that (data) memory is not always modified in entire blocks of memory and old content of memory is not always discarded, once it is not valid any more. Thus old unused pointers may be partially overwritten. Typically write operations are conducted in an aligned way (for example eight bytes are written in a continuous manner) and pointers start at addresses that are a multiple of eight for better access performance. Thus when data is written to memory, a pointer would normally be overwritten by new data entirely.

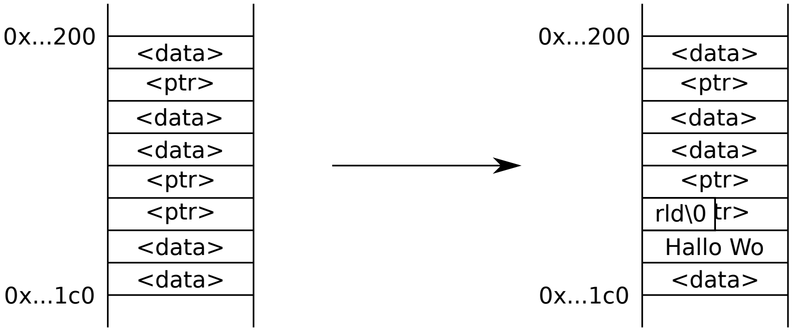

Figure 7.1.: Example of a string overwriting parts of a pointer.

However this is not always the case in practice. If, for example, a null terminated string is written to memory, the data is written sequentially. Thus only the bytes that later hold the string are modified and the rest of the old content of memory is left as is. With this, if the old memory contained a valid pointer, writing a string to an address in front of the pointer may overwrite only part of that pointer. Figure 7.1 gives an example for such an overwrite, when the string `"Hallo World"` is written to the address $0x1c8$. Note that in the picture the beginning of the pointer is overwritten. Pointers, however, are stored in little endian format on the Intel platform. Thus the bytes of the pointer are stored in reverse order. With this a legitimate pointer may be overwritten by the application resulting in a unidentifiable, but yet benign code pointer. Unfortunately there is no way to distinguish between a malicious and a partially overwritten benign pointer.

To soften the effects of this issue we add an additional classification: *printable pointers*. Our framework classifies detected pointers that do not point to a known location as printable, if the pointer contains a nullbyte followed by only printable characters. With this we are able to give a hint if a valid pointer may have been overwritten with a string.

Note, that this problem did not occur within the kernel context, as the kernel usually does not require the handling of dynamic strings, but instead mostly uses fixed buffers of static size to store strings. Another reason, why we assume that this problem did not show up during our investigation on kernel level CPE is the way how memory management is implemented in the Linux kernel. Instead of allocating new memory for every new object and deallocating the memory afterwards, the kernel makes use of a dedicated object cache (the SLAB-Allocator [13]). The objects within this cache are always reused for the same type of object. For this, memory that holds a pointer will be overwritten by a pointer again. Thus the probability for a partial overwrite of a pointer is much smaller within the Linux kernel.

Currently these identified pointers still require for manual investigation. One might apply additional heuristics such as allowing a small number of unidentifiable pointers as ROP chains usually consist of multiple pointers. Also, our framework could be enhanced to further filter out pointers that end with printable content. On the other hand, a single pointer to executable code may be enough for an attacker to conduct his malicious intention. For this, we chose to evaluate if the number of falsely detected benign pointers in a benign scenario is small and give a notification to an administrator for every detected unknown pointer.

7.4. Experimental results

We have now described our concepts to also apply the results gained in the previous chapters to userspace applications. In this section, we present our experiments together with the results of our investigations. A more general discussion of these results is provided in the next section.

For the following experiments, the host system in use was a AMD Phenom II X4 945 CPU with 16 GB of physical RAM running Debian Stretch (at the time of writing currently still distributed as Testing) with Linux kernel version 4.8 and the XEN hypervisor. As guest we used a VM running a custom Linux 3.16 kernel on Debian Stretch with 1 virtual CPU and 1 GB of guest physical RAM. To evaluate userspace processes, we equipped our test VM with common server applications like among others apache2 (Version 2.4), mysql-server (Version 5.7), isc-dhcp-server (Version 4.3), bind9 (Version 9.10), exim4 (Version 4.88) and openssh-server (Version 7.5). These constitute common server applications and present a realistic scenario for our evaluation. For all these applications, we also added the corresponding debug information into our trusted binary store. This is required as our framework requires the information about internal symbols of each binary file as explained in the last section. In all test cases our framework directly scanned the memory of the monitored virtual machine from within the host operating system. The virtualized guest was not paused during our experiments.

7.4.1. Userspace Code Integrity Validation

In a first set of experiments, we tested the userspace code validation capabilities of our framework and validated the code integrity of all executable code of all processes executed in the monitored VM as explained in the previous section. As expected, we did not detect any inconsistencies between the in memory representation of the executable code and the trusted representation created by our framework. Performance wise, this experiment is currently still rather slow, as the binary loading mechanism within our framework currently does not take advantage of internal caching mechanisms. Each binary file is loaded and processed for each process where it is included, instead of reusing the information for the next process validation. The entire process of loading and code validation currently took around 60 seconds for 83 different processes. Note that the majority of time (around $38.7s$) is spent in our framework for process loading and information gathering, which is only required once in a scenario with regular memory testing. The actual code validation step requires around $20s$ for all 83 processes (751 different executable code mappings, around 221.46 MB in total). Note, that applications, that are executed on the system multiple times are also checked multiple times in this experiment. This could also be optimized further, if a physical page that is mapped into multiple virtual address spaces is only validated once per iteration.

With this, we assume, that our mechanism to validate the code base of userspace applications from an external point of view by leveraging VMI techniques may be practically used as an efficient mechanism to check the code integrity of userspace processes once our frameworks internal binary loader is able to take advantage of caching. As this is not a conceptual issue, this feature was not implemented during the time of writing but will be implemented as part of future work.

7.4.2. Kernel Code Pointers in Userspace

In a next set of experiments we evaluate the effectiveness of CPE to detect kernel code pointers within userspace applications. We have already shown, that our framework is able to detect data-only malware in the previous chapter. Thus, in this experiment, we measure if our framework suffers from false positives in a benign scenario. In the following, we do not discuss the performance impact of our experiments to the guest VM. Due to the architecture of our framework, we assume this impact to be of similar magnitude as during our kernel only CPE experiments (see Section 6.5.1). The amount of guest physical RAM scanned during these experiments exceeds the amount of RAM scanned during kernel only validation. For this, we assume, that an application of CPE to all applications of the monitored target system in a live manner, like proposed in Chapter 5 is only able to check each individual application in the order of seconds. Thus a transient attack might not be detected by this approach. Still, as we assume persistent data-only malware as an attacker model, the malware needs to stay inside the system as long it intends to take control over the attacked system and in that case, our system is able to detect the malware.

First, we start to scan all userspace memory for kernel pointers. As described earlier, we first extract all writable memory mappings from our monitored VM. This is done in order to scan each single page only once. In this experiment, our framework inspected $1,297$ different memory mappings from 97 different processes resulting in $30,356$ different physical pages (approx. 118.5 MB). During the experiment (and also during every other execution of this test case) no memory location was found that contained data that could be (mis-)interpreted as kernel pointer. For this, CPE seems to be a suitable instrument to detect the creation of kernel level code reuse malware within a userspace application like for example proposed by Snow *et al.* [88].

In the next experiment, we lifted our restriction to only scan for pointers on writable pages. Thus, we also investigated pages that are mapped as not writable from userspace. In this experiment $35,828$ pages where scanned ($2,823$ mappings in 97 processes, approx. 140 MB). In this test 28 different locations have been found to contain data that might also be interpreted as kernel pointers (15 unique values). While most of these would only point to kernel data, only 2 values have been found to point to kernel code. Both of these were found by misinterpreting legitimate instructions inside the executable mapping of `libc-2-24.so` as pointers. In this experiment we detected a small number of possible kernel pointers within executable userspace code sections. This is not critical to our approach, as the code sections of libraries are separately checked for their integrity by our framework.

In a next experiment, we adapt our search to scan for kernel pointers that are accepted by the memory management unite despite the fact that they are not valid in practice. We to lift the constraint that the first 16 bits of a kernel address need to be set on a 64 bit system with only 48 physical address lines. For the first case, when only both readable *and* writable pages are checked, still no false positives are detected by our framework. Thus, CPE is a practical technique to scan for kernel targeting data-only malware in userspace.

In contrast, in the second case, when also scanning on pages that are only readable from userspace and ignoring the 16 highest bits of an inspected pointer, 177 different pointers were detected (136 unique kernel pointers, 77 unique kernel code pointers). This is still a low number, when we keep in mind, that we only need to find a collision for 48 bits. Still, this number is to high for a manual analysis. For this, we propose to enable the consistency check for the correctness of the first 16 bits of the address in kernel space. Note, that Intel recently announced to increase the number of physical address lines from 48 to 57 and to announce an additional level in the page tables [45]. Thus this problem will likely be also minimized with future hardware versions.

With this set of experiments we show, that our approach may be used detect kernel targeting CRA-based malware in userspace applications as it does not show any false positives in a benign scenario.

7.4.3. Userspace Code Pointers

Up until now, we conducted our experiments to validate the code integrity and detect kernel targeting CRA-based attacks in userspace data memory. In the following we present the results of our final set of experiments. With this set of experiments we investigate, if CPE may also be practically applied to detect return oriented malware that leverages gadgets within userspace applications. We assume the mechanism to be applicable, if the number of false positives detected by our framework is small (ideally zero).

For this we executed multiple experiments to scrutinize our assumption in practice. During the first execution of this experiment, all binary packages used on the monitored system were directly taken from the, at that time current, Debian repository. Thus the software tested during this experiment represents typical server software as widely deployed in the internet. Also, as previously discussed, PIE was not enabled for all applications. A summary of our results for all processes executed on our test system during the experiment can be found in Table A.2 in the Appendix of this document (page 188). The numbers in the table summarize the number of code pointers that where detected within the memory of each inspected application according to our classification (described in Section 7.3.2). In the numbers specify the total number of pointers for that category. The numbers for `bash` and `apache` in marked with an asterisk represents the share of pointers that point to a non randomized code segment (no PIE). Applications that are listed in the table multiple times also had multiple instances running while our test was executed. Still the numbers for these processes are typically similar, as these processes mostly share the same address space (code and data pages) due to the way the processes where initialized (`fork()`).

Our experiment shows, that our framework detected possible code pointers that could not be tagged as benign using our classification. Still the results of this experiment look promising. Compared to the overall number of pointers detected within the memory of a process, a large number of pointers could be identified by our framework. Still, the processes *apache2*, *bash* and *mysql* show a large number of not identifiable pointers. For the rest of applications, for which full debug information was available, the number of unknown pointers is smaller than 10. Thus, from this experiment, it seems that CPE is not usable to detect malicious pointers in the data memory of userspace applications. However, we will show, that if important boundary conditions are considered our approach is in fact usable.

In the following, we focus our discussion of the results on selected applications that serve as example to discuss the practical problems that need to be solved in order for an efficient application of our approach in practice. Table 7.1 gives a summary of the results for only these processes that we will further discuss in the following. In addition, detailed results for this run of our experiment with the *apache2* process can be found in Table A.3, results for the *mysqld* process can be found in Table A.4. Due to their respective lengths both tables have been moved to the appendix. Note that, the second column in these tables contains a ✓, in case our framework had access to the internal debug information for a binary during our test. As described earlier, the number of unidentifiable pointers is also larger in our system when no debug information is available. That is due to the fact that internal symbols are unknown to our framework.

Processname	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to `.text`	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
apache2	✓	3180	1769	1244	86	82	4	63	1
bash	-	1492	1195	338	226	66	6	0	0
bash*	-	733	500	281	225	66	6	0	0
mysqld	✓	165115	4280	1700	1323	666	91	473	67
mysqld*	✓	98834	2140	1192	1041	485	72	301	38

Table 7.1.: Summary of interesting processes during the first run of userspace CPE experiments. The numbers for processes in marked with an asterisk represents the share of pointers that point to a non randomized code segment (no PIE). The entire results of this experiment can be found in Table A.2 on page 188.

We further analyze and discuss these results in the following and show that our mechanism still may be applied in practice.

Both the `bash` and the `mysqld` processes serve as examples for applications that are loaded to a memory area near the beginning of the virtual address space. While *bash* is the de facto standard shell program on Linux systems, *mysql* is a popular database server implementation that is widely deployed in the internet. At the time of writing, both of these network facing applications had not been compiled to support the security feature PIE by the Debian project. We will now discuss the detailed results of our scan of the *mysqld* process. The detailed results of our experiment with the MySql server are presented in Table A.4 in the Appendix section. In this scan, we identified $165,115$ different pointers ($4,280$ unique pointers) in the data memory of the process.

Among these, we could not identify around $1,323$ different unique pointers that were detected during our experiment. Note that the majority of the pointers that could not be identified (1041 unique pointers) point to the program code segment of the loaded executable and not to any other library within that process. The reason for this is, that the code segment of the `mysqld` binary is mapped to the memory area between $0x00400000$ and $0x019fe000$ of virtual memory as this process is not compiled with as PIE. The code segment of this application is relatively large (21 MB), and is mapped to the relative beginning of the virtual address space. For this, internal data like counters have been falsely identified as pointers. To test our assumption, we recompiled all applications executed on the monitored test system to also support PIE. The result of this test case is shown in Table A.5 on page 197. The results for the experiment with the *mysqld* process are detailed in Table A.6. We summarize the results for the processes `bash` and `mysqld` in Table 7.2.

As expected, the number of unidentifiable pointers has dropped multiple orders of magnitudes. For the *bash* process, it dropped from 226 to 1. For *mysqld* we only identified 32.211 different memory locations as pointers (2.272 unique pointers). The number of unknown pointer values in this case only is 23. While this number is still significant, it is now in a range that may be manually analyzed by a human expert. 8 of these unidentifiable pointers locations are detected on the program stack and all unidentifiable pointers still point into the program code segment of the *mysqld* executable.

Processname	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to `.text`	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
bash	✓	1393	1212	375	1	0	0	0	0
mysqld	✓	32211	2272	678	23	16	5	11	0

Table 7.2.: Summary of the results of important processes in the evaluation on a system with PIE enabled. Detail results in Table A.5 (page 197).

Currently no further manual research has been conducted to further classify and identify these pointers.

Our experiment shows that by consequently enabling the existing security mechanism PIE, the number of false positives that are detected by or system dropped significantly. Thus bringing the number of false positives into a maintainable level. Still legitimate data is encountered, that is identified as code pointers when PIE is enforced on the monitored system. We further analyzed this issue and found that the reason for this problem are partial overwrites.

The amount of such invalid pointers was low in our experiments (typically below 10) in applications for which the debug information with information about internal symbols was available. The numbers in Table 7.1, however show one test run of our framework, in which the number of false positives was specifically high. The detailed test results for the *apache2* process are listed in Table A.3 on page 193. In that test run we detected 86 unknown pointers of which 77 alone point from memory that belongs to `libnss_resolve.so.2` to the program code of `libc-2.24.so`. From these assumed pointers 73 are also flagged as printable by our framework. The other 3 pointers contained two sequential nullbytes followed by only printable characters and thus seem to be overwritten by printable characters twice.

For this we further analyzed the source of this issue. These pointers where not continuous in memory. However, all the 77 pointers that we detected point into the function `strcpy_ssse3`. During our test, this function was mapped to the virtual address `0x7f1ab10055e0` in the virtual address space of the *apache2* application. That is, in all the cases when our framework detected an unknown pointer inside the virtual memory of the *apache2* process, the first three bytes of a valid pointer have been overwritten by a string. Thus in this testcase, all false positives of our framework stem from the fact, that legitimate pointers have been partly overwritten by null terminated character strings.

Finally note, that for the processes that are not marked with a checkmark in the Tables A.2 and A.5, not all symbol information was available during the experiments. Thus we expect that most of the unknown pointers are caused by pointers to internal symbols.

7.5. Discussion and Limitations

After we evaluated our approach in the last section, we shorty summarize our results in this section. Our prototype framework was successfully able able to validate the integrity of all userspace application code executed in a state-of-the art Linux server distribution. By reiterating the loading process, we are able to not only validate the code itself, but also important control flow relevant datastructures and also process management related kernel state. With this our hypervisor-based system shows to be applicable to provide code integrity in practice.

In addition we could show, that the detection of kernel pointers in userspace only holds small amount of false positives in a benign scenario in practice. In case, where only pages where inspected that are writable no only pointer was identified that could be misinterpreted as a kernel code pointer. Interestingly, the false positives we have found during our analysis were only contained within executable only mapped pages. For this, we assume that scanning userspace memory for kernel pointers is an interesting and efficient approach to detect control structures of data only malware before the guest kernel itself is attacked.

In contrast, we showed that CPE can not be easily applied to userspace applications as is. The two major problems we encountered are partial memory overwrites of old legitimate pointers in memory and code mappings at the beginning of virtual memory. Due to this problem the application of CPE to detect code reuse malware against userspace applications suffers from a lot of legitimate data that is misinterpreted as malicious pointers by our approach. We could, however, show that this situation is improved when we applied CPE to userspace binaries that have both PIC and PIE enabled. We found that the number of false positives of our system in this case is manageable. Still, we found that partially overwritten legitimate pointers are an issue for our system.

Unfortunately, we are unable to distinguish between partially overwritten pointers and malicious pointers. In our experiments, we conducted further manual investigation which is very time consuming and can not always be applied on a production system. To cope with this problem we introduced the classification of printable pointers. Our experiments show, that a large number of partial overwrites are caused by string writes. However, this is not the only reason for partial overwrites. As such we think that this approach may not be efficiently applied to userspace memory. Future research may however find further differentiation factors, which make such an approach feasible. Nevertheless, we have learned about some aspects in the handling of userspace applications in modern OSs, that make the validation of data integrity in general and code pointer integrity in concrete a interesting research problem.

To summarize, the research in this part of our thesis, has lead to two results. First, it shows, that ensuring code integrity for both kernel and userspace code is possible from an external viewpoint using VMI. Also the brute-force search for kernel code pointers in physical memory seems to be a promising and efficient technique to detect kernel code reuse gadgets in memory. However, we also discovered, that deficiencies in memory management of userspace applications leads to a high number of false positives when CPE is utilized to detect and identify invalid userspace code pointers.

With this, our framework could be employed in a cloud environment and not only validate the integrity of the guests kernel state, but also the integrity of important userspace applications. In such a scenario a dedicated CPU of the host system could be applied to validate the integrity of all virtualized guests on the system.

During the development of this work, we have also enhanced our framework to an, as we think, helpful tool for the development of further VMI applications. Thus in the next section, we introduce important features that we implemented.

7.6. Application of the Kernel Integrity Framework

After we have have described our approach to apply our research to userspace applications, in this section we give a technical overview over the kernel (and userspace) integrity validation framework, which we have implemented during this thesis. As the kernel integrity portion of the framework was already described in detail in previous chapters of this thesis, we now focus on the technical aspects of the framework as well as the parts of the framework, which have been extended in order to conduct integrity validation for userspace memory presented in this chapter.

To implement our system, we were required to understand and implement large parts of the operating systems binary loader and the file formats of executable files. For the binary loader, this includes both the kernel loader as well as the userspace loading mechanism. In terms of file formats, we based our efforts on the ELF file format and the handling of DWARF debug symbol format. In addition to the file format handling, we also need extract information from kernel memory which is related to the OSs userspace management. This is required to reiterate the userspace binary loading process in a way that is consistent to the current OS state. This for example includes information about the current memory mappings of a process, as this is required to conduct the library relocations. In order to achieve this, we implemented an generic interface to bridge the semantic gap and allow easy extraction of semantic information from the introspected guest operating system and its internal state.

In terms of nomenclature, our framework is based on both derivative, as well as out-of-bound generated information. The derivative information is gathered from the hypervisor, in our case through the help of LibVMI [64]. For this, our framework is agnostic about the underlying virtualization technology. Most important, this information consists of access to the current register contents and access to the guests physical memory.

In contrast, the information about the loaded binary executables is extracted in an out-of-band manner from trusted executables. It is mostly generated during the reiteration of the binary loading and the relocation phase. For this, as previously stated, the information is extracted from the ELF headers as well as from the DWARF debug information. The latter is only processed if available. The ELF headers contain information about the in-memory layout of the binary. Special sections exist that contain information about the offsets within the binary that need to be adapted due to relocation or about the offsets and names of (internal) symbols. The DWARF debug information does not only contain the name and memory location of a symbol, but also additional type and layout information. This is especially important if further information needs to be extracted from a specific library.

In its current version, our approach only depends on the availability for DWARF debug information from the kernel itself. This conceptually allows to easily extend our work to further analyze complex userspace applications like for example browsers or the Android ecosystem as part of future work.

We have already implemented our own ELF parsing and loading mechanism for the previous work. While its functionality was first limited to the functionality required to handle the OS kernel, it was extended to also handle userspace programs as well as libraries.

The binary loading process for the kernel is relatively straight forward. The ELF header contains a list of memory sections associated with their corresponding addresses in virtual memory. Also the kernel contains an initial set of page tables within its data section that is consistent with the information given in the ELF file. Thus in order to initially load the kernel, the boot loader only needs to load the kernel executable to a given offset within physical memory and execute a short initialization routine that is already part of the kernel. This gives the advantage, that the loaded sections are already in the correct order and our replicated kernel loading mechanism only is required to extract the correct information from the ELF header, take care of the load-time relocations and the extraction of the information that is relevant for the kernel runtime self-modifications. This process was already described in Chapter 5 of this thesis.

In contrast, the loading mechanism for kernel modules and userspace applications is more complicated, as the addresses for the different memory segments that are contained within the binary file need to be extracted from the executing kernel. In addition, unlike the kernel itself they contain a list of external dependencies, which need to be loaded into the virtual address space in advance in order to be able to resolve external symbols. For this, our framework recursively extracts a dependency tree for the executable and for all its dependencies. The loading process is then performs as a post-order traversal of the dependency tree. Note, that for userspace applications the virtual Dynamically linked Shared Object (vDSO) page that is provided by the kernel is an additional implicit dependency of every loaded executable. This page is part of the operating system and is used as a mechanism to increase the performance of system calls.

For the loading of userspace executables, two additional implementation problems need to be solved. First, the kernel is only organized in sections, while the memory layout of userspace applications is managed in different segments. This makes rebuilding the loading process more complicated. Second, the physical memory of one specific library may also be mapped within multiple different virtual addresses within multiple processes. Thus the relocation phase needs to take care of the different virtual offsets, on which the library is loaded in the different processes. In other words, while the code segment of a library is the same for all processes the library is loaded to, the data has to be relocated for every process.

During the loading of an individual userspace binary, also a shadow copy of all generated VMAs is generated. This is required in order to validate the information that the kernel holds about the corresponding process. The memory content of each VMA consists of a list of different segments which are defined within the binary file. In a next step, the in memory representation of each segment is relocated according to its actual position within the processes virtual memory. This requires our framework to resolve all required external symbols which are already loaded into the process due our frameworks dependency handling.

During the emulation of the loading process, we create a database of all symbols referenced within each accessed binary. For each loaded executable, the names, absolute virtual addresses and available type information of all known symbols are extracted by our framework after the binary is loaded. This is required for multiple reasons. First, it is required to resolve external symbols for binaries that depend on a specific symbol. Second, it can later be used to extract additional information from the executing process. This information is now maintained within a separate *SymbolManager*.

A dedicated *SymbolManager* instance is created for the kernel as well as for every userspace process. This is in order to represent the scope of a specific symbol. When feed with additional type information information, our framework is able to transparently extract information from the applications executed on the monitored machine, as it is able to extract information from the kernel in the current version. Figure 7.2 shows the class hierarchy provided by the SymbolManager subsystem of our framework to access the guests memory. This hierarchy is directly adapted from the DWARF specification. For each abstract symbol an instance may be created

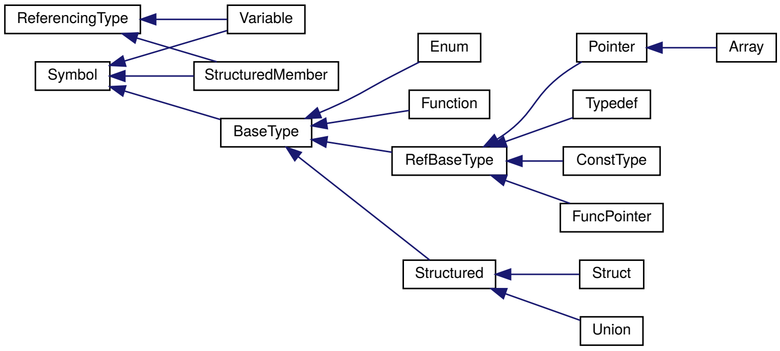

Figure 7.2.: Class Hierarchy of to access Guest datastructures.

by associating a virtual address to the type. In combination with our *LibVMI* wrapper it is thus directly possible to extract the contents of a symbol from the virtual machine and navigate within the datastructures.

With this infrastructure in place we have built an easy to use mechanism to access both kernel and userspace datastructures from the monitored guest virtual machine. We are thus able to extract arbitrary information from the monitored kernel. This includes access to all the executing userspace processes including their respective virtual address spaces.

To ease the development of custom memory introspection applications, we implemented a kernel specific custom helper classes, to extract commonly required information from the monitored guest VM. We currently implemented two example classes: (1) the *Kernel* class and (2) the *TaskManager*. The first allows to extract high level state information about the kernel. This includes for example the list of loaded kernel modules. The *TaskManager* allows to extract userspace management related information from the kernel, such information about currently executing userspace applications as well as information about the current memory mappings of each process.

```cpp
ElfKernelLoader kl = KernelValidator::loadKernel();
[...]
auto tasks = kl->getTaskManager()->getTasks();

for (auto && task : tasks) {
  pid_t pid = task.memberByName("pid").getValue<int64_t>();
  std::string comm = task.memberByName("comm").getValue<std::string>();
  std::cout << pid << "\t" << comm << std::endl;
}
```

Figure 7.3.: Example code to extract all running processes from an introspected VM.

Note, that this information is orthogonal to the information that is stored within the page tables of a specific process.

A developer is is thus easily able to extend our framework to build custom introspection or validation plugins. An example to extract a list of all active processes of an introspected VM from within our framework with only a little bit of C/C++ code[5] is shown in Figure 7.3. The code shows, how the different layers of accessor functions work together in our framework. First, an object of class `ElfKernelLoader` is used, to extract information about the relevant kernel datastructures. Based on the gathered information the `TaskManager` is able to extract specific information from the kernels userspace management datastructures. In this example about the struct `task_struct`. This list can then be further be processed through our `SymbolManager` API which provides functions to directly access the in guest data structures (as shown in Figure 7.2). Note, that this API uses *LibVMI* to access the underlying guest VMs physical memory.

With this framework we are able to transparently extract information about the target system on a per process level. With this infrastructure in place, future versions of our framework may be able to handle larger userspace applications, which themselves employ

[5]A python wrapper with basic functionality is also provided.

their own mechanisms of managing applications. Such as for example Browsers that employ a lot of Extensions or the Android Framework with its own Kernel like Runtime Environment and its Apps.

7.7. Summary

In this chapter, we introduced the userspace validation features that were implemented in our kernel integrity validation framework during this thesis. Userspace code validation was implemented emulating the userspace loader within our framework and solely comparing the in memory representation of the code with the trusted representation. This is possible due to the valuable property of userspace code to be static in practice. In addition we successfully applied CPE which was already introduced in Chapter 6 to the userspace part of the monitored virtual machine. We could show, that this mechanism, even though handling a lot more memory, is still a promising approach to detect kernel code reuse in practice.

Finally, we also showed the practical problems that hinder the application of CPE to detect userspace targeting CRAs.

Chapter 8

Conclusion and Future Work

As already discussed in this thesis, the detection and prevention of malware is a major problem in the IT industry. Different stakeholders provide different anti virus products that are marketed as a panacea to the problem of malicious software. Unfortunately most of these commercially available products have in common, that they depend on the integrity of the underlying operating system to successfully perform their main task: protect the integrity of the underlying operating system. In this thesis, we propose mechanisms to protect the integrity of the operating system without depending on its integrity. With this section, we want to conclude this thesis. First, we shortly iterate the contributions of this thesis, then discuss possible applications and finally give some ideas about possible future work.

8.1. Contributions

When our research effort started, the research in the direction of code integrity validation became stale. The dominant opinion in academia on that topic was, that code integrity is easy to validate. This was due to the fact, that it was believed that the binary code in memory is only a copy of the code within the application binary. Thus, to validate the integrity of code, the memory of the OS only needs to be compared to a trusted reference version. While it was acknowledged that load-time modifications like relocations are conducted, the codebase was assumed to be static during runtime.

Kernel Code Integrity In Chapter 5 we scrutinized this well established assumption and showed, that this assumption, while being wide spread, is not correct. Over the years of development, modern operating systems have included a multitude of different features and have thus become more and more complex. In contrast, to support interactive usage, an operating system requires to be as fast and as efficient as possible. For this reason, OS developers included different mechanisms into the codebase of the OS kernel that make use of runtime code self-modification in order to adapt the kernel code to the current situation and requirements. This is done to optimize the kernel code to the point that the overhead for context related decisions during runtime is close to zero. As the first major contribution of this thesis, we conducted a detailed analysis on the different self-modification mechanisms employed by modern operating systems. This includes a detailed discussion about the different self-patching mechanisms that are employed as well as a study on how the codebase and the runtime changes can still be validated. This also includes a validation of the kernels internal state related to these runtime self-patching mechanisms. To support our theoretical analysis a VMI-based prototype framework has been implemented and evaluated, which is able to validate the different runtime changes made by the OS kernel and that is thus able to validate the code integrity of a monitored OS during runtime. With this contribution we give an answer to the first two of our research questions (**Q1** and **Q2**) that we raised during the introduction of this thesis. During this work, we also detected and described, what we think is a major problem of the Linux kernel. Userspace applications are able to load arbitrary data into portions of physical memory that are marked as executable within the OS kernel. The existence of this problem shows, that we were the first ones to make a detailed investigation on operating system code integrity validation.

Kernel Data Integrity In the second part of this theses, we directed our attention to the rest of the contents of OS memory, more specifically the kernel's data memory. While it is clear, that there is no way to make general integrity assumptions on data memory, we tried to still make integrity assumptions about important parts of kernel memory. For this reason, we turned our attention in the direction of code pointers within memory. While code pointers are part of data memory they are also directly related to the control flow of an operating system. Thus if we are able to make sense about all code pointers in memory, we are able to make assumptions about the integrity of the control flow of the OS. To aid this research, we theoretically analyzed existing CFI approaches. These approaches require to closely monitor the execution of a target system in order to detect a control flow violation directly at the point it happens. Thus these mechanisms suffer from significant performance overheads. While researchers tried to increase the performance of these systems they where required to trade performance against the security of the system. In Chapter 4 we argue that current CFI concepts may not be sufficient to be applied to operating system kernels. Together with our knowledge about the internals of modern operating systems, we concluded that CFI mechanisms alone can not completely mitigate control-flow violations in practice.

In contrast, while our system is unable to actively detect and mitigate a control flow violation, we are able to make integrity assumptions based only on the contents of data memory, as we are able to detect the control structures of data-only malware. Thus, in Chapter 6 we elaborate on CPE, our approach to detect and classify code pointers in kernel data memory. We have implemented a practical prototype of our concept and in an evaluation, we show that our mechanism is both efficient and has a low number of false positives in practice. With this, our framework is able to detect control structures of data-only malware in memory, thus giving a hint about an infection, in case the infection could not be prevented by using CFI mechanisms in the first place. Note, that we did not encounter a single false positive in our experiments. With this part of our research we give an answer to the third research question (**Q3**) that we raised within this thesis. As our system is unable to actively detect control flow violations we propose to combine our approach with other research in the field of CFI.

Userspace Applicability Finally, we also investigated if the results of this thesis may also be applied to userspace applications. For this, we extended the framework developed in this thesis to also support the replication of the Linux userspace loading process. In Chapter 7 we present our results of this work. We successfully implemented both userspace code validation as well as the detection of kernel targeting data-only malware in userspace memory. In short, we think that it is possible to apply CPE to userspace memory. This way, one may even be able to detect malware that makes use of code reuse such as return oriented programming already before it is loaded into the kernel by an attacker. This could be used as defense mechanism, similar as the search for executable instructions or shellcode is used in network based intrusion detection systems today.

In a last investigation, we made experiments to detect and classify userspace code pointers in order to detect code reuse malware targeting userspace applications. We showed that under certain circumstances CPE can also be applied in this use case. However, we found that this application does not seem to be successfully applicable in practice due to limitations that are set by the current system architecture. With this we give an answer to the forth research question (**Q4**) that was raised in the beginning of this thesis.

Open Framework for Integrity Validation To conclude, while we are not able to give an binary answer to the question if a current system state can be trusted, we are able successfully make integrity assumptions and detect code reuse malware within an operating system without continuous control flow monitoring by only analyzing the memory contents from an external position. During this research, we also extended our prototype framework in a way, that it is successfully able to bridge the semantic gap and allows developers to easily extract information from the memory of a monitored virtual machine. With this we support the development of further memory introspection tools that aim to validate further parts of the state of important systems. To promote further development of integrity validation tools, the prototype framework created within this thesis was published under an open-source license:

- `https://github.com/kittel/kernel_integrity`
- `https://github.com/kittel/libdwarfparser`

8.2. Practical Application

In this section, we shortly want to discuss the practical application of the proposed framework. Due to its architectural nature, VMI-based protection mechanisms are generally seen as inefficient. To extract information about its current execution, the monitored target system is typically paused in regular intervals (e. g. each instruction, each branch instruction, or every time the instruction pointer exceeds a page boundary in memory). To minimize the overhead, the amount of information that is monitored and extracted in a synchronous manner through VMI should be kept to a minimum in a production environment. For this reason, a differentiation between lightweight and heavyweight detection mechanisms should be made [34]. In such a scenario, the approach presented in this thesis could be used as a lightweight detection mechanism and, in case an indication for a malicious modification is found, a more heavyweight detection mechanism may be activated to scan for further evidence. So, high overhead is prevented in normal system execution.

Another idea is to virtualize the target system on-the-fly, an idea first proposed by Rutkowska in 2006 [77]. This is effectively an extension of the scenario proposed above. Instead of executing the target system in a virtualized environment during normal execution, the system may be executed without active virtualization. Once any evidence for malicious behavior is detected, the system may be virtualized and lightweight VMI-based malware detection mechanisms may be applied. If the evidence can not be confirmed through the VMI-based detection mechanisms the hypervisor may unload itself leaving the system in an unvirtualized state again. A version of this mechanism targeting the Intel architecture was implemented as a master thesis [72], during this thesis. At the time of writing this work however was not yet published as academic paper.

8.3. Future Work

After we have wrapped up our conclusions and shortly provided some ideas for practical application, in the following we shortly introduce some future work and research directions for each of the research direction that we have presented throughout this thesis.

Kernel Code Integrity Future work related to kernel code integrity validation is mostly implementation work. Our current prototype implementation of our kernel integrity framework is based on the Linux kernel. Currently for each supported kernel version some modifications are necessary. In order to extend the applicability of our system a kernel version agnostic abstraction layer needs to be implemented. This layer should contain abstract information about the different self-patching mechanisms that need to be validated and information about the corresponding in memory representation of important data structures. That is different kernel versions rename or change the layout of required kernel objects. In order to support future kernel versions, an abstract database needs to be created that contains information on where to find specific information within kernel memory.

In addition, since our work on this topic another runtime self-modification mechanism has been included into the main line kernel. That is, the kernel now supports *KPatch*[1]. With this mechanism security patches can be applied during runtime, eliminating the need for a kernel reboot after updates. With KPatch entire functions can be replaced during runtime. If the new version of the function is shorter than the old version, the new function replaces the old function. If the new function requires more space, the new function is placed in another part of executable memory and an unconditional branch to the new function is inserted at the beginning of the old function. Our framework currently does not support the validation of this new feature. Thus support for this feature should be included as part of future work.

Next, the ideas and the prototype implementation created throughout this thesis currently are only thoroughly tested on the Intel architecture. The prototype could be extended to further architectures such as for example the ARM architecture. With this, its scope could be extended from validating the integrity of servers in a cloud environment, to for example validate the OS code integrity in mobile devices. This seems even more important, as these devices are more commonly used and are thus also a profitable attack target. Finally, the scope of our kernel code integrity prototype was restricted to the Linux platform. Thus it would be interesting to also study the wide spread Windows platform for similar runtime code patching mechanisms.

Code Pointer Integrity During our work on CPE, we implemented a simple algorithm to validate the contents of kernel stacks. As we explained, this algorithm is currently not able to validate the contents of a stack reliably. Thus future work should be dedicated to this topic.

[1]http://rhelblog.redhat.com/2014/02/26/kpatch/

A technique should be proposed, that is able to reliably validate the content of a given stack frame in memory. While this problem seems to be easily solvable in theory, it is not trivial, as the stack might contain additional unrelated pointers and old uninitialized contents. An extension of this work could also give hints on whether the given stack is a legitimate stack that was created during program execution or whether the execution of the sequence of return instructions on the stack results in malicious behavior. We already proposed this problem as, what we think is an important research question (**Q5**) in the beginning of this thesis. Another topic for future work is the inclusion of symbolic execution. Once a malicious pointer is identified by our approach, more information about the potential gadget could be extracted. With this a further classification of the pointer could be made. For example, concerning information about the different registers or memory addresses that are manipulated by the gadget.

Another interesting topic would be to extend existing research in the area of CFI to also be able to handle how, object orientation is emulated within C program such as the Linux kernel. As already introduced in Chapter 4 existing CFI mechanisms already aim to handle C++ vtable structures. The Linux kernel uses a similar mechanism to simulate object oriented functionality. It contains dedicated datastructures (`*_ops`), similar to vtables, that contain pointers to functions that are directly related to a certain datastructure. In contrast to C++ vtables, the C structures to not contain a reference to their corresponding `*_ops` data structures. First such a relation should be automatically inferred, for example from the source code, or by analyzing the function signatures of the corresponding functions. Then, mechanisms should be developed, to efficiently check the integrity of such data structures in memory.

Application to Userspace Additional future work should be conducted in the validation of userspace processes during runtime. One important part is the generation of ground truth for the `ifunc` relocation mechanism introduced in Section 7.2. This could be implemented by integrating an additional symbolic execution plugin into our framework. As we do not expect any algorithmic mechanisms to handle partly overwritten legitimate pointers, it would be interesting to find methods to be able to differentiate between partly overwritten legitimate pointers. For this, first the question should be answered, if it is possible to implement return oriented malware that solely makes use of gadgets that are stored at such ambiguous addresses. If such malware exists, its detection becomes rather complicated in practice. We also suggest to spend efforts in the question of how to generally separate a malicious control flow from a benign one. Thus, for example, given the contents of the current stack of a process, decide, whether the application is malicious or not.

Finally, as part of our thesis large efforts have been made to provide a transparent interface for a developer to bridge the semantic gap and enable the easy development of introspection tools. As part of future work, the created libraries could be enhanced to support not only OS kernels, but also common middleware application such as the Android Runtime (ART) or modern browsers, which themselves are as complicated as an operating system. With this effort, our framework could be extended to also validate the integrity of for example single Android Apps.

8.4. Final Words

In this work we made efforts to make integrity assumptions about modern operating systems. We hope, that the investigations and results made in this thesis, especially the outlined problems, that we exposed within the OS design help to improve the design and architecture of future versions of operating systems.

Appendix A

Appendix

A.1. Tables

The following table shows a list of vtable like structures in the Linux kernel adhering to the naming convention **_ops*. The list was extracted from the source code of Linux 4.5.

Table A.1

struct name	defined in
acpi_debugger_ops	include/linux/acpi.h
acpi_device_ops	include/acpi/acpi_bus.h
acpi_pci_root_ops	include/linux/pci-acpi.h
assoc_array_ops	include/linux/assoc_array.h
atmdev_ops	include/linux/atmdev.h
atmphy_ops	include/linux/atmdev.h
atm_tcp_ops	include/linux/atm_tcp.h
auth_ops	include/linux/sunrpc/svcauth.h
backlight_ops	include/linux/backlight.h
bcma_host_ops	include/linux/bcma/bcma.h
blk_mq_ops	include/linux/blk-mq.h
bpf_map_ops	include/linux/bpf.h
bpf_verifier_ops	include/linux/bpf.h
c2port_ops	include/linux/c2port.h
cdrom_device_ops	include/linux/cdrom.h
ceph_auth_client_ops	include/linux/ceph/auth.h
cfhsi_cb_ops	include/net/caif/caif_hsi.h
cfhsi_ops	include/net/caif/caif_hsi.h
cleancache_ops	include/linux/cleancache.h
clk_hw_omap_ops	include/linux/clk/ti.h
clk_ops	include/linux/clk-provider.h

Table A.1 – continued from previous page

struct name	defined in
component_master_ops	include/linux/component.h
component_ops	include/linux/component.h
coresight_ops	include/linux/coresight.h
coresight_ops_link	include/linux/coresight.h
coresight_ops_sink	include/linux/coresight.h
coresight_ops_source	include/linux/coresight.h
cx2341x_handler_ops	include/media/drv-intf/cx2341x.h
dca_ops	include/linux/dca.h
dcbnl_rtnl_ops	include/net/dcbnl.h
devfreq_event_ops	include/linux/devfreq-event.h
dev_pm_ops	include/linux/pm.h
dlm_lockspace_ops	include/linux/dlm.h
dma_buf_ops	include/linux/dma-buf.h
dma_map_ops	include/linux/dma-mapping.h
drbg_state_ops	include/crypto/drbg.h
dss_mgr_ops	include/video/omapdss.h
dst_ops	include/net/dst_ops.h
dw_mci_dma_ops	include/linux/mmc/dw_mmc.h
ep93xx_spi_chip_ops	include/linux/platform_data/spi-ep93xx.h
ethtool_ops	include/linux/ethtool.h
exynos_media_pipeline_ops	include/media/drv-intf/exynos-fimc.h
fb_ops	include/linux/fb.h
fb_tile_ops	include/linux/fb.h
fence_ops	include/linux/fence.h
fib_rules_ops	include/net/fib_rules.h
flow_cache_ops	include/net/flow.h
fpga_manager_ops	include/linux/fpga/fpga-mgr.h
frontswap_ops	include/linux/frontswap.h
fscache_cache_ops	include/linux/fscache-cache.h
fsnotify_ops	include/linux/fsnotify_backend.h
ftrace_ops	include/linux/ftrace.h
ftrace_ops_hash	include/linux/ftrace.h
ftrace_probe_ops	include/linux/ftrace.h
genl_ops	include/net/genetlink.h
gpd_dev_ops	include/linux/pm_domain.h
gss_api_ops	include/linux/sunrpc/gss_api.h
hdac_bus_ops	include/sound/hdaudio.h
hdac_ext_codec_ops	include/sound/hdaudio_ext.h
hdac_io_ops	include/sound/hdaudio.h
hdlcdrv_ops	include/linux/hdlcdrv.h
header_ops	include/linux/netdevice.h
host1x_bo_ops	include/linux/host1x.h
host1x_client_ops	include/linux/host1x.h
hotplug_slot_ops	include/linux/pci_hotplug.h
i915_audio_component_audio_ops	include/drm/i915_component.h
i915_audio_component_ops	include/drm/i915_component.h
ib_dma_mapping_ops	include/rdma/ib_verbs.h
ide_disk_ops	include/linux/ide.h
ide_dma_ops	include/linux/ide.h
ide_port_ops	include/linux/ide.h
ide_tp_ops	include/linux/ide.h
iio_buffer_setup_ops	include/linux/iio/iio.h
iio_dma_buffer_ops	include/linux/iio/buffer-dma.h
iio_sw_trigger_ops	include/linux/iio/sw_trigger.h
iio_trigger_ops	include/linux/iio/trigger.h
inet_connection_sock_af_ops	include/net/inet_connection_sock.h
iommu_ops	include/linux/iommu.h
iommu_ops	include/linux/iommu.h
ipack_bus_ops	include/linux/ipack.h
ipack_driver_ops	include/linux/ipack.h
ip_tunnel_encap_ops	include/net/ip_tunnels.h

Table A.1 – continued from previous page

struct name	defined in
irq_domain_ops	include/linux/irqdomain.h
iscsi_conn_ops	include/target/iscsi/iscsi_target_core.h
iscsi_sess_ops	include/target/iscsi/iscsi_target_core.h
kernel_param_ops	include/linux/moduleparam.h
kernfs_ops	include/linux/kernfs.h
kernfs_syscall_ops	include/linux/kernfs.h
kexec_file_ops	include/linux/kexec.h
kset_uevent_ops	include/linux/kobject.h
kvm_device_ops	include/linux/kvm_host.h
kvm_io_device_ops	include/kvm/iodev.h
l2cap_ops	include/net/bluetooth/l2cap.h
l3mdev_ops	include/net/l3mdev.h
latch_tree_ops	include/linux/rbtree_latch.h
lcd_ops	include/linux/lcd.h
led_flash_ops	include/linux/led-class-flash.h
lwtunnel_encap_ops	include/net/lwtunnel.h
mbox_chan_ops	include/linux/mailbox_controller.h
mbus_hw_ops	include/linux/mic_bus.h
mcp_ops	include/linux/mfd/mcp.h
mdiobb_ops	include/linux/mdio-bitbang.h
mipi_dsi_host_ops	include/drm/drm_mipi_dsi.h
mipi_dsim_master_ops	include/video/exynos_mipi_dsim.h
mmc_host_ops	include/linux/mmc/host.h
mmp_overlay_ops	include/video/mmp_disp.h
mmp_path_ops	include/video/mmp_disp.h
mmu_notifier_ops	include/linux/mmu_notifier.h
mpc8xx_pcmcia_ops	include/linux/fsl_devices.h
msi_domain_ops	include/linux/msi.h
mtd_blktrans_ops	include/linux/mtd/blktrans.h
mtd_oob_ops	include/linux/mtd/mtd.h
nci_driver_ops	include/net/nfc/nci_core.h
nci_ops	include/net/nfc/nci_core.h
nci_uart_ops	include/net/nfc/nci_core.h
neigh_ops	include/net/neighbour.h
net_device_ops	include/linux/netdevice.h
nfc_digital_ops	include/net/nfc/digital.h
nfc_hci_ops	include/net/nfc/hci.h
nfc_ops	include/net/nfc/nfc.h
nfc_phy_ops	include/net/nfc/nfc.h
nf_hook_ops	include/linux/netfilter.h
nf_ipv6_ops	include/linux/netfilter_ipv6.h
nfs_commit_completion_ops	include/linux/nfs_xdr.h
nf_sockopt_ops	include/linux/netfilter.h
nfs_pageio_ops	include/linux/nfs_page.h
nfs_pgio_completion_ops	include/linux/nfs_xdr.h
nfs_rpc_ops	include/linux/nfs_xdr.h
nfs_rw_ops	include/linux/nfs_page.h
nft_expr_ops	include/net/netfilter/nf_tables.h
nft_set_ops	include/net/netfilter/nf_tables.h
nsc_gpio_ops	include/linux/nsc_gpio.h
ntb_client_ops	include/linux/ntb.h
ntb_ctx_ops	include/linux/ntb.h
ntb_dev_ops	include/linux/ntb.h
nvm_dev_ops	include/linux/lightnvm.h
of_pdt_ops	include/linux/of_pdt.h
omapdss_atv_ops	include/video/omapdss.h
omapdss_dpi_ops	include/video/omapdss.h
omapdss_dsi_ops	include/video/omapdss.h
omapdss_dvi_ops	include/video/omapdss.h
omapdss_hdmi_ops	include/video/omapdss.h
omapdss_sdi_ops	include/video/omapdss.h

Table A.1 – continued from previous page

struct name	defined in
omap_hdmi_audio_ops	include/sound/omap-hdmi-audio.h
omap_mcbsp_ops	include/linux/platform_data/asoc-ti-mcbsp.h
otg_fsm_ops	include/linux/usb/otg-fsm.h
pci_ops	include/linux/pci.h
pcr_ops	include/linux/mfd/rtsx_pci.h
phy_ops	include/linux/phy/phy.h
pinconf_ops	include/linux/pinctrl/pinconf.h
pinctrl_ops	include/linux/pinctrl/pinctrl.h
pingv6_ops	include/net/ping.h
pinmux_ops	include/linux/pinctrl/pinmux.h
platform_freeze_ops	include/linux/suspend.h
platform_hibernation_ops	include/linux/suspend.h
platform_suspend_ops	include/linux/suspend.h
plat_sci_port_ops	include/linux/serial_sci.h
plat_vlynq_ops	include/linux/vlynq.h
powercap_control_type_ops	include/linux/powercap.h
powercap_zone_constraint_ops	include/linux/powercap.h
powercap_zone_ops	include/linux/powercap.h
ppi_ops	include/media/blackfin/ppi.h
ppp_channel_ops	include/linux/ppp_channel.h
preempt_ops	include/linux/preempt.h
pr_ops	include/linux/pr.h
proto_ops	include/linux/net.h
pwm_ops	include/linux/pwm.h
qcom_smem_state_ops	include/linux/soc/qcom/smem_state.h
Qdisc_class_ops	include/net/sch_generic.h
Qdisc_ops	include/net/sch_generic.h
qed_common_cb_ops	include/linux/qed/qed_if.h
qed_common_ops	include/linux/qed/qed_if.h
qed_eth_cb_ops	include/linux/qed/qed_eth_if.h
qed_eth_ops	include/linux/qed/qed_eth_if.h
quotactl_ops	include/linux/quota.h
quota_format_ops	include/linux/quota.h
rate_control_ops	include/net/mac80211.h
regulator_ops	include/linux/regulator/driver.h
request_sock_ops	include/net/request_sock.h
reserved_mem_ops	include/linux/of_reserved_mem.h
reset_control_ops	include/linux/reset-controller.h
rfkill_ops	include/linux/rfkill.h
rio_ops	include/linux/rio.h
rio_switch_ops	include/linux/rio.h
rpc_call_ops	include/linux/sunrpc/sched.h
rpc_pipe_dir_object_ops	include/linux/sunrpc/rpc_pipe_fs.h
rpc_pipe_ops	include/linux/sunrpc/rpc_pipe_fs.h
rpc_xprt_ops	include/linux/sunrpc/xprt.h
rproc_ops	include/linux/remoteproc.h
rtc_class_ops	include/linux/rtc.h
rtnl_af_ops	include/net/rtnetlink.h
rtnl_link_ops	include/net/rtnetlink.h
sbc_ops	include/target/target_core_backend.h
scpi_ops	include/linux/scpi_protocol.h
sh_clk_ops	include/linux/sh_clk.h
shdma_ops	include/linux/shdma-base.h
sh_mobile_lcdc_sys_bus_ops	include/video/sh_mobile_lcdc.h
skb_checksum_ops	include/linux/skbuff.h
snd_ac97_build_ops	include/sound/ac97_codec.h
snd_ac97_bus_ops	include/sound/ac97_codec.h
snd_ak4xxx_ops	include/sound/ak4xxx-adda.h
snd_compr_ops	include/sound/compress_driver.h
snd_device_ops	include/sound/core.h
snd_hwdep_ops	include/sound/hwdep.h

Table A.1 – continued from previous page

struct name	defined in
snd_i2c_bit_ops	include/sound/i2c.h
snd_i2c_ops	include/sound/i2c.h
snd_info_entry_ops	include/sound/info.h
snd_pcm_ops	include/sound/pcm.h
snd_rawmidi_global_ops	include/sound/rawmidi.h
snd_rawmidi_ops	include/sound/rawmidi.h
snd_sb_csp_ops	include/sound/sb16_csp.h
snd_soc_compr_ops	include/sound/soc.h
snd_soc_dai_ops	include/sound/soc-dai.h
snd_soc_ops	include/sound/soc.h
snd_soc_tplg_bytes_ext_ops	include/sound/soc-topology.h
snd_soc_tplg_io_ops	include/uapi/sound/asoc.h
snd_soc_tplg_kcontrol_ops	include/sound/soc-topology.h
snd_soc_tplg_ops	include/sound/soc-topology.h
snd_tea575x_ops	include/media/drv-intf/tea575x.h
snd_vx_ops	include/sound/vx_core.h
soc_camera_host_ops	include/media/soc_camera.h
ssb_bus_ops	include/linux/ssb/ssb.h
ste_modem_dev_ops	include/linux/ste_modem_shm.h
superhyway_ops	include/linux/superhyway.h
svc_serv_ops	include/linux/sunrpc/svc.h
svc_xprt_ops	include/linux/sunrpc/svc_xprt.h
svm_dev_ops	include/linux/intel-svm.h
switchdev_ops	include/net/switchdev.h
syscore_ops	include/linux/syscore_ops.h
sysfs_ops	include/linux/sysfs.h
target_backend_ops	include/target/target_core_backend.h
target_core_fabric_ops	include/target/target_core_fabric.h
tc_action_ops	include/net/act_api.h
tcf_ematch_ops	include/net/pkt_cls.h
tcf_proto_ops	include/net/sch_generic.h
tcp_congestion_ops	include/net/tcp.h
tcp_request_sock_ops	include/net/tcp.h
tcp_sock_af_ops	include/net/tcp.h
team_mode_ops	include/linux/if_team.h
tegra_cpu_car_ops	include/linux/clk/tegra.h
thermal_cooling_device_ops	include/linux/thermal.h
thermal_zone_device_ops	include/linux/thermal.h
thermal_zone_of_device_ops	include/linux/thermal.h
ti_clk_ll_ops	include/linux/clk/ti.h
timewait_sock_ops	include/net/timewait_sock.h
tpm_class_ops	include/linux/tpm.h
tty_ldisc_ops	include/linux/tty_ldisc.h
uart_ops	include/linux/serial_core.h
ulpi_ops	include/linux/ulpi/interface.h
usb_ep_ops	include/linux/usb/gadget.h
usb_gadget_ops	include/linux/usb/gadget.h
usb_phy_io_ops	include/linux/usb/phy.h
v4l2_clk_ops	include/media/v4l2-clk.h
v4l2_ctrl_ops	include/media/v4l2-ctrls.h
v4l2_ctrl_type_ops	include/media/v4l2-ctrls.h
v4l2_flash_ops	include/media/v4l2-flash-led-class.h
v4l2_ioctl_ops	include/media/v4l2-ioctl.h
v4l2_m2m_ops	include/media/v4l2-mem2mem.h
v4l2_subdev_audio_ops	include/media/v4l2-subdev.h
v4l2_subdev_core_ops	include/media/v4l2-subdev.h
v4l2_subdev_internal_ops	include/media/v4l2-subdev.h
v4l2_subdev_ir_ops	include/media/v4l2-subdev.h
v4l2_subdev_ops	include/media/v4l2-subdev.h
v4l2_subdev_pad_ops	include/media/v4l2-subdev.h
v4l2_subdev_sensor_ops	include/media/v4l2-subdev.h

Table A.1 – continued from previous page

struct name	defined in
v4l2_subdev_tuner_ops	include/media/v4l2-subdev.h
v4l2_subdev_vbi_ops	include/media/v4l2-subdev.h
v4l2_subdev_video_ops	include/media/v4l2-subdev.h
v4l2_subscribed_event_ops	include/media/v4l2-event.h
vb2_buf_ops	include/media/videobuf2-core.h
vb2_mem_ops	include/media/videobuf2-core.h
vb2_ops	include/media/videobuf2-core.h
vexpress_config_bridge_ops	include/linux/vexpress.h
vfio_device_ops	include/linux/vfio.h
vfio_iommu_driver_ops	include/linux/vfio.h
vga_switcheroo_client_ops	include/linux/vga_switcheroo.h
vgic_ops	include/kvm/arm_vgic.h
vgic_vm_ops	include/kvm/arm_vgic.h
videobuf_qtype_ops	include/media/videobuf-core.h
videobuf_queue_ops	include/media/videobuf-core.h
virtio_config_ops	include/linux/virtio_config.h
vpbe_device_ops	include/media/davinci/vpbe.h
vpbe_osd_ops	include/media/davinci/vpbe_osd.h
vringh_config_ops	include/linux/vringh.h
watchdog_ops	include/linux/watchdog.h
wkup_m3_ipc_ops	include/linux/wkup_m3_ipc.h
wm97xx_mach_ops	include/linux/wm97xx.h
wpan_dev_header_ops	include/net/cfg802154.h
zbud_ops	include/linux/zbud.h
zpool_ops	include/linux/zpool.h

Table A.1.: Vtable like structures in the Linux kernel adhering to the naming convention **_ops* extracted from Linux 4.5.

Table A.2

Processname	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
agetty	-	399 (0)	274 (0)	130 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)
agetty	-	419 (0)	289 (0)	140 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)
apache2	✓	3180 (0)	1769 (0)	1244 (0)	86 (0)	82 (0)	4 (0)	63 (0)	1 (0)
apache2	✓	3682 (0)	1868 (0)	1289 (0)	86 (0)	82 (0)	4 (0)	63 (0)	1 (0)
apache2	✓	3682 (0)	1868 (0)	1289 (0)	86 (0)	82 (0)	4 (0)	63 (0)	1 (0)
atd	-	438 (0)	324 (0)	115 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)
bash	-	1492 (733)	1195 (500)	338 (281)	226 (225)	66 (66)	6 (6)	0 (0)	0 (0)
bash	-	1507 (737)	1195 (497)	331 (274)	225 (225)	66 (66)	6 (6)	0 (0)	0 (0)
bash	-	1515 (740)	1193 (496)	330 (272)	226 (225)	66 (66)	6 (6)	1 (0)	0 (0)
bash	-	1522 (753)	1202 (502)	335 (275)	224 (224)	65 (65)	6 (6)	0 (0)	0 (0)
bash	-	1764 (843)	1309 (520)	378 (305)	233 (231)	70 (70)	8 (8)	0 (0)	0 (0)
cron	-	571 (91)	372 (54)	140 (20)	6 (3)	2 (1)	0 (0)	2 (0)	0 (0)
dbus-daemon	-	819 (0)	546 (0)	217 (0)	44 (0)	5 (0)	0 (0)	0 (0)	0 (0)
dhclient	-	721 (0)	531 (0)	187 (0)	98 (0)	9 (0)	0 (0)	0 (0)	0 (0)
exim4	-	1046 (0)	653 (0)	201 (0)	9 (0)	0 (0)	0 (0)	3 (0)	2 (0)
irqbalance	-	489 (121)	322 (76)	132 (27)	5 (5)	2 (2)	0 (0)	0 (0)	0 (0)
mysqld	✓	165115 (98834)	4280 (2140)	1700 (1192)	1323 (1041)	666 (485)	91 (72)	473 (301)	67 (38)
named	-	2774 (0)	1353 (0)	371 (0)	151 (0)	12 (0)	0 (0)	0 (0)	0 (0)
rpcbind	-	802 (84)	561 (39)	159 (14)	8 (7)	1 (1)	0 (0)	0 (0)	0 (0)
rsyslogd	-	1917 (0)	870 (0)	553 (0)	249 (0)	16 (0)	0 (0)	0 (0)	0 (0)
sftp-server	-	367 (0)	236 (0)	109 (0)	4 (0)	3 (0)	0 (0)	3 (0)	0 (0)
sshd	-	1339 (0)	988 (0)	219 (0)	22 (0)	1 (0)	1 (0)	1 (0)	1 (0)
sshd	-	1731 (0)	1270 (0)	338 (0)	23 (0)	1 (0)	0 (0)	0 (0)	0 (0)
sshd	-	1855 (0)	1342 (0)	372 (0)	23 (0)	1 (0)	0 (0)	0 (0)	0 (0)
sshd	-	1855 (0)	1342 (0)	372 (0)	23 (0)	5 (0)	0 (0)	0 (0)	0 (0)
sshd	-	1972 (0)	1280 (0)	337 (0)	44 (0)	3 (0)	0 (0)	0 (0)	0 (0)
sshd	-	2051 (0)	1332 (0)	374 (0)	45 (0)	8 (0)	0 (0)	1 (0)	0 (0)
sshd	-	2066 (0)	1339 (0)	378 (0)	45 (0)	3 (0)	0 (0)	1 (0)	0 (0)
systemd	✓	1096 (0)	769 (0)	220 (0)	1 (0)	1 (0)	0 (0)	1 (0)	0 (0)
systemd	✓	3141 (0)	1587 (0)	374 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)
systemd	✓	3473 (0)	1351 (0)	290 (0)	1 (0)	1 (0)	0 (0)	1 (0)	0 (0)
systemd-journald	✓	644 (0)	454 (0)	163 (0)	1 (0)	1 (0)	0 (0)	1 (0)	0 (0)
systemd-logind	✓	837 (0)	629 (0)	171 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
systemd-resolved	✓	877 (0)	590 (0)	191 (0)	3 (0)	2 (0)	1 (0)	1 (0)	0 (0)
systemd-timesyncd	✓	882 (0)	610 (0)	178 (0)	1 (0)	1 (0)	0 (0)	1 (0)	0 (0)
systemd-udevd	✓	800 (0)	530 (0)	212 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
test	✓	836 (0)	568 (0)	128 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)

Table A.2 – continued from previous page

Processname	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
tmux	-	1203 (0)	944 (0)	205 (0)	35 (0)	0 (0)	0 (0)	0 (0)	0 (0)
tmux	-	381 (0)	266 (0)	92 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (0)
vim.basic	-	1137 (0)	930 (0)	171 (0)	25 (0)	17 (0)	3 (0)	16 (0)	0 (0)

Table A.2.: Statistic of number of identified userspace pointers during one run through the entire system. The system solely consists of applications packages taken from the current Debian repository. During the experiment all libraries have been compiled to be position independent (fPIC). However, not all applications have been compiled as position independent (fPIE). Processes that are executed multiple times concurrently are listed multiple times. The number in brackets shows the number of pointers that point to non randomized code segments.

This table shows a detailed report about an executing instance of the `apache2` process.

Table A.3

From Mapping	To Mapping	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
<unknown>	mod_mpm_event.so	✓	1	1	1	0	0	0	0	0
<unknown>	libc-2.24.so	✓	12	8	4	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	7	6	2	0	0	0	0	0
<unknown>	ld-2.24.so	✓	6	6	6	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0

Table A.3 – continued from previous page

From Mapping	To Mapping	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0

Table A.3 – continued from previous page

From Mapping	To Mapping	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	2	2	2	0	0	0	0	0
<unknown>	apache2	✓	1	1	1	0	0	0	0	0
<unknown>	mod_mpm_event.so	✓	2	2	2	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	9	5	4	0	0	0	0	0
<unknown>	libapr-1.so.0.5.2	✓	5	4	2	0	0	0	0	0
<unknown>	ld-2.24.so	✓	6	6	6	0	0	0	0	0
<unknown>	libgcc_s.so.1	-	22	21	3	0	0	0	0	0
<unknown>	libc-2.24.so	✓	1	1	0	0	0	0	0	0
libnss_compat-2.24.so	libnss_nis-2.24.so	✓	16	11	11	0	0	0	0	0
libpthread-2.24.so	libdl-2.24.so	✓	1	1	1	0	0	0	0	0
libpthread-2.24.so	libc-2.24.so	✓	14	14	13	0	0	0	0	0
libapr-1.so.0.5.2	libdl-2.24.so	✓	3	3	3	0	0	0	0	0
libapr-1.so.0.5.2	libc-2.24.so	✓	43	43	43	0	0	0	0	0
libapr-1.so.0.5.2	libpthread-2.24.so	✓	21	21	21	0	0	0	0	0
libaprutil-1.so.0.5.4	libc-2.24.so	✓	5	5	5	0	0	0	0	0
libaprutil-1.so.0.5.4	libapr-1.so.0.5.2	✓	8	8	8	0	0	0	0	0
libpcre.so.3.13.1	libc-2.24.so	✓	4	2	2	0	0	0	0	0
<unknown>	apache2	✓	16	1	1	0	0	0	0	0
zero	libc-2.24.so	✓	7	6	6	6	6	0	4	0
zero	libapr-1.so.0.5.2	✓	3	3	2	0	0	0	0	0
zero	libnss_resolve.so.2	✓	2	2	0	0	0	0	0	0
zero	apache2	✓	16	7	3	0	0	0	0	0
libnss_resolve.so.2	mod_status.so	✓	45	6	5	0	0	0	0	0
libnss_resolve.so.2	mod_setenvif.so	✓	90	6	1	0	0	0	0	0
libnss_resolve.so.2	mod_negotiation.so	✓	37	6	3	0	0	0	0	0
libnss_resolve.so.2	mod_mpm_event.so	✓	185	27	19	1	1	0	1	0
libnss_resolve.so.2	mod_mime.so	✓	260	10	2	0	0	0	0	0
libnss_resolve.so.2	mod_filter.so	✓	17	5	4	0	0	0	0	0
libnss_resolve.so.2	mod_env.so	✓	12	2	2	0	0	0	0	0
libnss_resolve.so.2	mod_dir.so	✓	13	3	1	0	0	0	0	0
libnss_resolve.so.2	mod_deflate.so	✓	16	5	4	0	0	0	0	0
libnss_resolve.so.2	mod_autoindex.so	✓	78	12	2	1	1	0	1	0
libnss_resolve.so.2	mod_authz_user.so	✓	16	3	1	0	0	0	0	0
libnss_resolve.so.2	mod_authz_host.so	✓	42	10	3	0	0	0	0	0
libnss_resolve.so.2	mod_authz_core.so	✓	94	15	6	0	0	0	0	0
libnss_resolve.so.2	mod_authn_file.so	✓	25	5	2	0	0	0	0	0
libnss_resolve.so.2	mod_authn_core.so	✓	14	6	4	0	0	0	0	0
libnss_resolve.so.2	mod_auth_basic.so	✓	36	4	3	0	0	0	0	0
libnss_resolve.so.2	mod_alias.so	✓	28	4	3	0	0	0	0	0
libnss_resolve.so.2	mod_access_compat.so	✓	12	4	3	0	0	0	0	0
libnss_resolve.so.2	libexpat.so.1.6.2	-	13	11	3	0	0	0	0	0
libnss_resolve.so.2	libdl-2.24.so	✓	17	13	3	0	0	0	0	0
libnss_resolve.so.2	libcrypt-2.24.so	✓	14	11	3	0	0	0	0	0
libnss_resolve.so.2	librt-2.24.so	✓	24	16	3	0	0	0	0	0

Table A.3 – continued from previous page

From Mapping	To Mapping	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
libnss_resolve.so.2	libuuid.so.1.3.0	-	20	16	4	0	0	0	0	0
libnss_resolve.so.2	libc-2.24.so	✓	178	109	78	77	73	4	56	1
libnss_resolve.so.2	libpthread-2.24.so	✓	22	13	0	0	0	0	0	0
libnss_resolve.so.2	libapr-1.so.0.5.2	✓	202	29	13	1	1	0	1	0
libnss_resolve.so.2	libaprutil-1.so.0.5.4	✓	26	10	2	0	0	0	0	0
libnss_resolve.so.2	libpcre.so.3.13.1	-	12	7	0	0	0	0	0	0
libnss_resolve.so.2	ld-2.24.so	✓	4	4	0	0	0	0	0	0
libnss_resolve.so.2	libc-2.24.so	✓	10	9	2	0	0	0	0	0
libnss_resolve.so.2	libpthread-2.24.so	✓	7	7	3	0	0	0	0	0
libnss_resolve.so.2	libapr-1.so.0.5.2	✓	25	15	9	0	0	0	0	0
libnss_resolve.so.2	libaprutil-1.so.0.5.4	✓	8	7	3	0	0	0	0	0
libnss_resolve.so.2	libpcre.so.3.13.1	-	7	7	3	0	0	0	0	0
libnss_resolve.so.2	ld-2.24.so	✓	8	5	0	0	0	0	0	0
ld-2.24.so	libc-2.24.so	✓	6	6	5	0	0	0	0	0
ld-2.24.so	libpthread-2.24.so	✓	5	5	5	0	0	0	0	0
ld-2.24.so	libapr-1.so.0.5.2	✓	1	1	0	0	0	0	0	0
apache2	mod_status.so	✓	1	1	0	0	0	0	0	0
apache2	mod_authn_core.so	✓	2	2	2	0	0	0	0	0
apache2	mod_access_compat.so	✓	1	1	1	0	0	0	0	0
apache2	libc-2.24.so	✓	1	1	1	0	0	0	0	0
apache2	libapr-1.so.0.5.2	✓	479	479	479	0	0	0	0	0
apache2	libaprutil-1.so.0.5.4	✓	291	281	279	0	0	0	0	0
apache2	mod_authz_core.so	✓	1	1	1	0	0	0	0	0
heap	libselinux.so.1	-	25	17	3	0	0	0	0	0
heap	libnss_files-2.24.so	✓	16	12	3	0	0	0	0	0
heap	libnss_nis-2.24.so	✓	22	14	3	0	0	0	0	0
heap	libnsl-2.24.so	✓	17	12	3	0	0	0	0	0
heap	libnss_compat-2.24.so	✓	34	24	3	0	0	0	0	0
heap	mod_status.so	✓	11	10	4	0	0	0	0	0
heap	mod_setenvif.so	✓	12	11	5	0	0	0	0	0
heap	mod_negotiation.so	✓	14	12	5	0	0	0	0	0
heap	mod_mpm_event.so	✓	15	11	3	0	0	0	0	0
heap	mod_mime.so	✓	17	13	5	0	0	0	0	0
heap	mod_filter.so	✓	14	12	5	0	0	0	0	0
heap	mod_env.so	✓	10	10	5	0	0	0	0	0
heap	mod_dir.so	✓	14	12	5	0	0	0	0	0
heap	libz.so.1.2.8	-	28	24	3	0	0	0	0	0
heap	mod_deflate.so	✓	15	12	4	0	0	0	0	0
heap	mod_autoindex.so	✓	19	14	5	0	0	0	0	0
heap	mod_authz_user.so	✓	11	10	3	0	0	0	0	0
heap	mod_authz_host.so	✓	11	10	4	0	0	0	0	0
heap	mod_authz_core.so	✓	12	11	5	0	0	0	0	0
heap	mod_authn_file.so	✓	11	10	4	0	0	0	0	0
heap	mod_authn_core.so	✓	13	11	5	0	0	0	0	0
heap	mod_auth_basic.so	✓	15	12	5	0	0	0	0	0
heap	mod_alias.so	✓	16	12	5	0	0	0	0	0
heap	mod_access_compat.so	✓	22	10	4	0	0	0	0	0
heap	libc-2.24.so	✓	31	17	5	5	5	0	2	0
heap	libapr-1.so.0.5.2	✓	3	2	0	0	0	0	0	0
heap	libnss_systemd.so.2	✓	34	21	1	0	0	0	0	0
heap	libnss_resolve.so.2	✓	37	22	1	0	0	0	0	0
heap	apache2	✓	7	3	1	0	0	0	0	0
stack	libselinux.so.1	-	1	1	0	0	0	0	0	0
stack	libnss_files-2.24.so	✓	12	8	5	0	0	0	0	0
stack	mod_status.so	✓	2	2	0	0	0	0	0	0
stack	mod_setenvif.so	✓	4	2	0	0	0	0	0	0

Table A.3 – continued from previous page

From Mapping	To Mapping	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
stack	mod__mpm__event.so	✓	8	6	5	0	0	0	0	0
stack	mod__mime.so	✓	1	1	1	0	0	0	0	0
stack	mod__deflate.so	✓	2	1	0	0	0	0	0	0
stack	libdl-2.24.so	✓	6	3	2	0	0	0	0	0
stack	libc-2.24.so	✓	60	39	32	0	0	0	0	0
stack	libpthread-2.24.so	✓	15	9	4	0	0	0	0	0
stack	libapr-1.so.0.5.2	✓	35	24	17	0	0	0	0	0
stack	libpcre.so.3.13.1	-	8	8	4	0	0	0	0	0
stack	ld-2.24.so	✓	70	24	21	0	0	0	0	0
stack	libnss__systemd.so.2	✓	6	6	1	0	0	0	0	0
stack	libnss__resolve.so.2	✓	7	7	5	0	0	0	0	0
stack	apache2	✓	46	37	16	0	0	0	0	0

Table A.3.: Detailed results of pointer destinations in userspace CPE experiment with *apache2* process without PIE enabled

This table shows a detailed report about an executing instance of the `mysql` process.

Table A.4

From Mapping	To Mapping	Symbols available	Overall Ptrs	Unie Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
heap	mysqld	✓	101635 (64343)	2702 (1058)	613 (440)	585 (423)	360 (247)	32 (20)	194 (97)	3 (1)
heap	libnss_files-2.24.so	✓	16 (0)	12 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
heap	libc-2.24.so	✓	10 (0)	8 (0)	6 (0)	5 (0)	4 (0)	1 (0)	3 (0)	0 (0)
<unknown>	mysqld	✓	6 (5)	3 (2)	3 (2)	3 (2)	3 (2)	0 (0)	1 (0)	0 (0)
<unknown>	mysqld	✓	6 (5)	3 (2)	3 (2)	3 (2)	3 (2)	0 (0)	1 (0)	0 (0)
<unknown>	mysqld	✓	36 (20)	20 (11)	7 (6)	7 (6)	5 (4)	0 (0)	3 (2)	0 (0)
<unknown>	mysqld	✓	308 (234)	81 (53)	49 (28)	46 (27)	18 (12)	3 (2)	11 (5)	0 (0)
<unknown>	mysqld	✓	11 (6)	7 (3)	6 (2)	6 (2)	5 (2)	0 (0)	1 (0)	0 (0)
<unknown>	mysqld	✓	38 (24)	25 (16)	22 (15)	14 (10)	5 (2)	1 (1)	4 (1)	1 (0)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	2 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	21 (18)	16 (13)	14 (11)	13 (10)	4 (2)	1 (1)	4 (1)	1 (0)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	2 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	43 (31)	29 (20)	24 (16)	16 (12)	6 (4)	1 (1)	5 (1)	1 (0)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	3 (0)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	38 (29)	23 (16)	21 (14)	15 (11)	4 (2)	2 (2)	4 (1)	1 (0)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	3 (0)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	373 (305)	67 (51)	40 (37)	39 (36)	15 (12)	2 (2)	11 (8)	0 (0)
<unknown>	mysqld	✓	365 (299)	59 (45)	32 (28)	32 (28)	14 (10)	1 (1)	10 (6)	0 (0)
<unknown>	mysqld	✓	911 (748)	132 (103)	80 (66)	78 (66)	30 (23)	7 (4)	26 (19)	0 (0)
<unknown>	mysqld	✓	466 (378)	85 (71)	51 (46)	50 (45)	25 (21)	4 (3)	14 (10)	0 (0)
<unknown>	mysqld	✓	345 (275)	120 (87)	88 (70)	34 (30)	11 (8)	2 (2)	9 (6)	6 (5)
<unknown>	libc-2.24.so	✓	15 (0)	15 (0)	12 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	6 (0)	5 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	206 (134)	97 (48)	65 (33)	33 (26)	16 (11)	1 (1)	16 (10)	9 (6)
<unknown>	libc-2.24.so	✓	27 (0)	22 (0)	17 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libstdc++.so.6.0.22	-	5 (0)	5 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	5 (0)	5 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	170 (109)	80 (58)	62 (46)	34 (30)	14 (12)	2 (2)	9 (5)	5 (3)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	3 (0)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)

Table A.4 – continued from previous page

From Mapping	To Mapping	Symbols available	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
<unknown>	mysqld	✓	484 (269)	149 (93)	118 (74)	64 (49)	26 (20)	4 (4)	31 (16)	21 (11)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	7 (0)	6 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	159 (98)	79 (55)	62 (43)	34 (29)	12 (10)	3 (3)	9 (4)	6 (3)
<unknown>	libc-2.24.so	✓	5 (0)	5 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	7 (0)	4 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	538 (291)	154 (87)	118 (68)	62 (44)	25 (17)	5 (5)	30 (12)	19 (8)
<unknown>	libc-2.24.so	✓	5 (0)	5 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	7 (0)	7 (0)	7 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	80 (52)	47 (30)	40 (25)	24 (18)	6 (4)	1 (1)	10 (5)	7 (4)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	4 (0)	4 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	40 (31)	25 (19)	20 (15)	15 (12)	4 (2)	1 (1)	6 (3)	2 (1)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	3 (0)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	mysqld	✓	7 (6)	5 (4)	5 (4)	5 (4)	3 (2)	2 (2)	1 (0)	0 (0)
<unknown>	mysqld	✓	29 (21)	21 (15)	19 (13)	13 (9)	5 (2)	1 (1)	4 (1)	1 (0)
<unknown>	libc-2.24.so	✓	4 (0)	4 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
<unknown>	libpthread-2.24.so	✓	4 (0)	4 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
libnss_files-2.24.so	mysqld	✓	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)	0 (0)	0 (0)	0 (0)
libnss_files-2.24.so	mysqld	✓	36 (25)	25 (17)	22 (15)	15 (11)	5 (2)	1 (1)	5 (1)	1 (0)
libnss_files-2.24.so	mysqld	✓	1779 (1611)	514 (496)	373 (356)	355 (342)	154 (143)	28 (28)	153 (141)	2 (0)
libnss_files-2.24.so	mysqld	✓	139 (87)	66 (36)	54 (32)	33 (25)	14 (10)	4 (4)	8 (4)	4 (2)
libnss_files-2.24.so	mysqld	✓	39 (29)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
libnss_files-2.24.so	mysqld	✓	39 (29)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
libnss_files-2.24.so	mysqld	✓	40 (29)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
libnss_files-2.24.so	mysqld	✓	63 (38)	41 (23)	36 (19)	20 (14)	8 (4)	1 (1)	9 (3)	5 (2)
libnss_files-2.24.so	mysqld	✓	39 (29)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
libnss_files-2.24.so	mysqld	✓	39 (29)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
libnss_files-2.24.so	mysqld	✓	40 (29)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
libnss_files-2.24.so	mysqld	✓	39 (29)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
libnss_files-2.24.so	mysqld	✓	56 (37)	34 (24)	31 (22)	19 (15)	6 (3)	1 (1)	7 (4)	3 (3)
libnss_files-2.24.so	mysqld	✓	42 (32)	26 (19)	24 (17)	17 (13)	6 (3)	1 (1)	7 (3)	3 (2)
<unknown>	mysqld	✓	53343 (27529)	1033 (817)	639 (550)	609 (531)	330 (264)	53 (45)	231 (179)	4 (2)
libnsl-2.24.so	mysqld	✓	1 (1)	1 (1)	1 (1)	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)
libc-2.24.so	mysqld	✓	57 (48)	14 (11)	11 (9)	11 (9)	5 (3)	1 (1)	4 (2)	0 (0)
libc-2.24.so	mysqld	✓	18 (12)	15 (12)	10 (8)	10 (8)	7 (5)	1 (1)	4 (2)	0 (0)

Table A.4 – continued from previous page

From Mapping	To Mapping	Symbols available	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
libgcc_s.so.1	mysqld	✓	34 (10)	23 (6)	10 (5)	7 (3)	5 (1)	2 (2)	3 (1)	0 (0)
libgcc_s.so.1	libc-2.24.so	✓	2 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
libm-2.24.so	mysqld	✓	32 (25)	20 (16)	4 (2)	4 (2)	3 (1)	1 (1)	2 (0)	0 (0)
libstdc++.so.6.0.22	mysqld	✓	18 (10)	5 (2)	5 (2)	5 (2)	3 (0)	1 (1)	3 (1)	0 (0)
libstdc++.so.6.0.22	libc-2.24.so	✓	14 (0)	14 (0)	14 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
libstdc++.so.6.0.22	mysqld	✓	145 (92)	11 (8)	7 (4)	6 (4)	3 (1)	1 (1)	4 (2)	0 (0)
libstdc++.so.6.0.22	libc-2.24.so	✓	13 (0)	13 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
libatomic.so.1.2.0	mysqld	✓	1 (1)	1 (1)	1 (1)	1 (1)	0 (0)	1 (1)	0 (0)	0 (0)
librt-2.24.so	mysqld	✓	8 (4)	5 (3)	3 (2)	3 (2)	1 (0)	1 (1)	2 (1)	0 (0)
libz.so.1.2.8	mysqld	✓	38 (11)	27 (9)	8 (5)	8 (5)	6 (3)	1 (1)	4 (2)	0 (0)
libdl-2.24.so	mysqld	✓	20 (14)	15 (12)	10 (7)	10 (7)	9 (6)	0 (0)	5 (2)	0 (0)
libcrypt-2.24.so	mysqld	✓	1 (1)	1 (1)	1 (1)	1 (1)	0 (0)	0 (0)	0 (0)	0 (0)
libwrap.so.0.7.6	mysqld	✓	15 (14)	12 (11)	10 (9)	10 (9)	9 (8)	0 (0)	2 (1)	0 (0)
liblz4.so.1.7.1	mysqld	✓	32 (15)	23 (11)	20 (9)	19 (9)	17 (8)	1 (0)	13 (7)	0 (0)
libnuma.so.1.0.0	mysqld	✓	39 (32)	24 (22)	19 (17)	19 (17)	18 (16)	0 (0)	3 (1)	0 (0)
libnuma.so.1.0.0	libc-2.24.so	✓	19 (0)	17 (0)	17 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
libpthread-2.24.so	mysqld	✓	17 (0)	2 (0)	2 (0)	2 (0)	1 (0)	0 (0)	1 (0)	0 (0)
libpthread-2.24.so	libc-2.24.so	✓	16 (0)	16 (0)	15 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
libpthread-2.24.so	mysqld	✓	5 (4)	4 (3)	4 (3)	4 (3)	2 (1)	0 (0)	1 (0)	0 (0)
ld-2.24.so	mysqld	✓	32 (23)	9 (7)	8 (6)	7 (5)	3 (1)	1 (1)	2 (0)	0 (0)
ld-2.24.so	libc-2.24.so	✓	6 (0)	6 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
ld-2.24.so	libpthread-2.24.so	✓	5 (0)	5 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
ld-2.24.so	mysqld	✓	65 (57)	17 (14)	13 (10)	10 (8)	4 (2)	1 (1)	4 (2)	0 (0)
ld-2.24.so	libstdc++.so.6.0.22	-	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
stack	mysqld	✓	2139 (1166)	357 (216)	244 (147)	138 (97)	69 (47)	10 (8)	66 (37)	36 (23)
stack	libc-2.24.so	✓	75 (0)	26 (0)	23 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
stack	libstdc++.so.6.0.22	-	4 (0)	4 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
stack	libpthread-2.24.so	✓	6 (0)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
stack	ld-2.24.so	✓	13 (0)	6 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)

Table A.4.: Detailed results of pointer destinations in userspace CPE experiment with *mysqld* process without PIE enabled

Table A.5

Processname	Dbg. Symbols	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
agetty	-	399	274	130	2	0	0	0	0
agetty	-	419	289	140	2	0	0	0	0
apache2	✓	3696	1839	1251	5	4	1	3	0
apache2	✓	4129	1938	1295	5	4	1	3	0
apache2	✓	4144	1941	1295	5	4	1	3	0
atd	-	438	324	115	3	0	0	0	0
bash	✓	1393	1212	375	1	0	0	0	0
bash	✓	1516	1247	403	3	0	2	0	0
bash	✓	1534	1256	406	4	0	2	0	0
bash	✓	1812	1387	466	4	0	3	0	0
cron	-	618	427	198	12	5	2	6	0
dbus-daemon	-	819	553	224	51	5	0	0	0
dhclient	-	720	532	188	98	9	0	0	0
exim4	-	1047	653	201	9	0	0	3	2
irqbalance	-	734	503	185	4	1	0	0	0
mysqld	✓	32211	2272	678	23	16	5	11	0
named	-	2776	1353	371	151	12	0	0	0
rpcbind	-	822	584	187	9	0	0	7	3
rsyslogd	-	1919	863	546	246	15	0	0	0
sftp-server	-	372	237	110	5	4	0	2	0
sshd	-	1340	989	220	22	1	1	1	1
sshd	-	1730	1267	335	23	4	0	0	0
sshd	-	1855	1342	372	23	1	0	0	0
sshd	-	1855	1342	372	23	1	0	0	0
sshd	-	1855	1342	372	23	1	0	0	0
sshd	-	1971	1281	338	44	9	0	0	0
sshd	-	2055	1334	376	45	3	0	1	0
sshd	-	2056	1334	376	45	3	0	1	0
sshd	-	2071	1341	380	46	3	0	2	0
su	-	933	685	275	5	0	0	0	0
systemd	✓	1095	768	219	1	1	0	1	0
systemd	✓	3092	1560	374	3	1	0	0	0
systemd	✓	3304	1325	292	3	1	1	1	0
systemd-journald	✓	652	450	166	0	0	0	0	0
systemd-logind	✓	841	627	169	0	0	0	0	0
systemd-resolved	✓	901	579	183	4	2	2	1	0
systemd-timesyncd	✓	882	610	178	1	1	0	1	0
systemd-udevd	✓	813	531	213	1	1	0	1	0

Table A.5.: Statistic of number of identified userspace pointers during one run through the entire system. The system solely consists of applications packages taken from the current Debian repository. During this experiment all libraries have been compiled to be position independent (fPIC) and all applications have been compiled as position independent (fPIE). Processes that are executed multiple times concurrently are listed multiple times.

This table shows a detailed report about an executing instance of the `mysql` process.

Table A.6

From Mapping	To Mapping	Symbols available	Overall Ptrs	Unie Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
<unknown>	mysqld	✓	899	216	36	3	1	1	0	0
<unknown>	mysqld	✓	3	3	3	0	0	0	0	0
<unknown>	mysqld	✓	4	1	0	0	0	0	0	0
<unknown>	libc-2.24.so	✓	5	5	2	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	mysqld	✓	181	116	100	2	1	1	1	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	mysqld	✓	3	3	3	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	3	3	3	0	0	0	0	0
<unknown>	mysqld	✓	10	10	10	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	3	3	3	0	0	0	0	0
<unknown>	mysqld	✓	8	8	8	0	0	0	0	0
<unknown>	libc-2.24.so	✓	27	22	17	0	0	0	0	0
<unknown>	libstdc++.so.6.0.22	-	5	5	5	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	5	5	5	0	0	0	0	0
<unknown>	mysqld	✓	89	62	48	0	0	0	0	0
<unknown>	mysqld	✓	30	15	10	0	0	0	0	0
<unknown>	mysqld	✓	30	15	10	0	0	0	0	0
<unknown>	mysqld	✓	30	15	10	0	0	0	0	0
<unknown>	mysqld	✓	30	15	10	0	0	0	0	0
<unknown>	libc-2.24.so	✓	16	16	13	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	6	5	5	0	0	0	0	0
<unknown>	mysqld	✓	106	83	65	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	4	4	4	0	0	0	0	0
<unknown>	mysqld	✓	11	9	9	0	0	0	0	0
<unknown>	libc-2.24.so	✓	17	11	5	0	0	0	0	0
<unknown>	libgcc_s.so.1	-	18	16	7	1	0	0	0	0
<unknown>	libpthread-2.24.so	✓	21	15	12	1	1	0	1	0
<unknown>	ld-2.24.so	✓	15	9	9	0	0	0	0	0
<unknown>	mysqld	✓	10	10	10	0	0	0	0	0
<unknown>	libc-2.24.so	✓	13	13	10	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	5	5	5	0	0	0	0	0
<unknown>	mysqld	✓	42	33	27	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	libc-2.24.so	✓	6	6	3	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	mysqld	✓	25	23	22	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0

Table A.6 – continued from previous page

From Mapping	To Mapping	Symbols available	Overall Ptrs	Uniue Ptrs	Ptr to .text	Unknown Ptrs	Printable	Invalid Instr.	Unintended Instr.	Unint. Ret
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	2	2	2	0	0	0	0	0
<unknown>	mysqld	✓	17	17	17	0	0	0	0	0
<unknown>	libc-2.24.so	✓	4	4	1	0	0	0	0	0
<unknown>	libpthread-2.24.so	✓	1	1	1	0	0	0	0	0
<unknown>	mysqld	✓	11	11	11	0	0	0	0	0
<unknown>	mysqld	✓	25030	30	16	5	4	1	2	0
libgcc_s.so.1	libc-2.24.so	✓	2	2	2	0	0	0	0	0
libstdc++.so.6.0.22	libc-2.24.so	✓	14	14	14	0	0	0	0	0
libstdc++.so.6.0.22	libc-2.24.so	✓	13	13	0	0	0	0	0	0
libnuma.so.1.0.0	libc-2.24.so	✓	19	17	17	0	0	0	0	0
libpthread-2.24.so	libc-2.24.so	✓	16	16	15	0	0	0	0	0
ld-2.24.so	libc-2.24.so	✓	6	6	5	0	0	0	0	0
ld-2.24.so	libpthread-2.24.so	✓	5	5	5	0	0	0	0	0
ld-2.24.so	libstdc++.so.6.0.22	-	1	1	0	0	0	0	0	0
mysqld	libstdc++.so.6.0.22	-	1	1	1	0	0	0	0	0
heap	libnss_files-2.24.so	✓	16	12	3	0	0	0	0	0
heap	libc-2.24.so	✓	5	3	1	0	0	0	0	0
heap	mysqld	✓	4400	1515	191	5	3	2	1	0
stack	libc-2.24.so	✓	74	26	23	0	0	0	0	0
stack	libstdc++.so.6.0.22	-	4	4	4	0	0	0	0	0
stack	libpthread-2.24.so	✓	5	3	3	0	0	0	0	0
stack	ld-2.24.so	✓	13	6	6	0	0	0	0	0
stack	mysqld	✓	734	189	160	8	8	0	7	0

Table A.6.: Detailed results of pointer destinations in userspace CPE experiment with PIE enabled *mysqld* process

Bibliography

[1] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, July 2012.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 2009.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[5] D. Andriesse, H. Bos, and A. Slowinska. Parallax: Implicit Code Integrity Verification Using Return-Oriented Programming. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 125–135. IEEE, 2015.

[6] D. Andriesse, A. Slowinska, and H. Bos. Compiler-Agnostic Function Detection in Binaries. In *EuroS&P*, Apr. 2017.

[7] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu. DKSM: Subverting Virtual Machine Introspection for Fun and Profit. In *29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010)*, New Delhi, India, October 2010.

[8] A. Baliga, V. Ganapathy, and L. Iftode. Detecting Kernel-Level Rootkits using Data Structure Invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5):670–684, 2011.

[9] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im) possibility of obfuscating programs. In *Annual International Cryptology Conference*, pages 1–18. Springer, 2001.

[10] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (Im)Possibility of Obfuscating Programs. *Journal of the ACM*, 59(2):6:1–6:48, May 2012.

[11] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Blacksheep: Detecting Compromised Hosts in Homogeneous Crowds. In *Conference on Computer and Communications Security (CCS)*. ACM, 2012.

[12] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[13] J. Bonwick. The slab allocator: An object-caching kernel memory allocator. In *USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 6–6, Berkeley, CA, USA, 1994. USENIX Association.

[14] E. Bosman and H. Bos. Framing Signals - A Return to Portable Shellcode. In *2014 IEEE Symposium on Security and Privacy*, SP '14, pages 243–258, Washington, DC, USA, 2014. IEEE Computer Society.

[15] D. Bounov, R. Kici, and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[16] S. Brookes and S. Taylor. Rethinking operating system design: Asymmetric multiprocessing for security and performance. In *2016 Workshop on New Security Paradigms*, 2016.

[17] J. Butler. Dkom (direct kernel object manipulation). *Black Hat Windows Security*, 2004.

[18] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping Kernel Objects to Enable Systematic Integrity Checking.

In *16th ACM conference on Computer and Communications Security (CCS'09)*, pages 555–565. ACM, 2009.

[19] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, Washington, D.C., 2015. USENIX Association.

[20] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, 2014. USENIX Association.

[21] H. Chang and M. J. Atallah. Protecting software code by guards. In *ACM Workshop on Digital Rights Management*, pages 160–175. Springer, 2001.

[22] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[23] P. Chen, X. Xing, B. Mao, and L. Xie. Return-Oriented Rootkit without Returns (on the x86). In M. Soriano, S. Qing, and J. López, editors, *Information and Communications Security*, volume 6476 of *Lecture Notes in Computer Science*, pages 340–354. Springer Berlin Heidelberg, 2010.

[24] P. M. Chen and B. D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001.

[25] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *NDSS*. The Internet Society, 2014.

[26] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *IEEE Symposium on Security and Privacy*, 2014.

[27] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *51st Annual Design Automation Conference*, pages 133:1–133:6, New York, NY, USA, 2014. ACM.

[28] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Proc. 22nd Network and Distributed Systems Security Sym.(NDSS)*, 2015.

[29] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, 2014. USENIX Association.

[30] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *16th ACM conference on Computer and communications security*. ACM, 2009.

[31] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of likely Invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[32] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *IEEE Symposium on Security and Privacy (Oakland'15)*, May 2015.

[33] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 901–913, New York, NY, USA, 2015. ACM.

[34] A. Fischer, T. Kittel, B. Kolosnjaji, T. K. Lengyel, W. Mandarawi, H. P. Reiser, B. Taubmann, E. Weishäupl, H. de Meer, T. Müller, and M. Protsenko. CloudIDEA: A Malware Defense Architecture for Cloud Data Centers. In *5th International Symposium on Cloud Computing, Trusted Computing and Secure Virtual Infrastructures - Cloud and Trusted Computing (C&TC 2015)*, 2015.

[35] M. Frantzen and M. Shuey. StackGhost: Hardware Facilitated Stack Protection. In *USENIX Security Symposium*, 2001.

[36] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM*

SIGOPS Operating Systems Review, volume 37, pages 193–206. ACM, 2003.

[37] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[38] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symposium on Security and Privacy*, pages 179–194, Mar. 2016.

[39] X. Ge, H. Vijayakumar, and T. Jaeger. SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture. 2014.

[40] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming Control-Flow Integrity. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2014.

[41] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, 2014. USENIX Association.

[42] GRSecurity. PAGEEXEC. `https://pax.grsecurity.net/docs/pageexec.txt`, December 30 2006.

[43] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring Operating System Kernel Integrity with OSck. In R. Gupta and T. C. Mowry, editors, *ASPLOS*, pages 279–290. ACM, 2011.

[44] R. Hund, T. Holz, and F. C. Freiling. Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *Proceedings of 18th USENIX Security Symposium*, 2009.

[45] Intel Corporation. 5-Level Paging and 5-Level EPT. Technical report, December 2016.

[46] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, June 2016.

[47] Intel Corporation. *Control-flow Enforcement Technology Preview*, June 2017.

[48] B. Jain, M. B. Baig, D. Zhang, D. E. Porter, and R. Sion. Sok: Introspections on trust and the semantic gap. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 605–620. IEEE, 2014.

[49] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *NDSS*, 2014.

[50] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. ret2dir: Rethinking Kernel Isolation. In *23rd USENIX Security Symposium*. USENIX Association, 2014.

[51] V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis. kGuard: Lightweight Kernel Protection Against Return-to-user Attacks. In *21st USENIX Conference on Security Symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association.

[52] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In *2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM, 1994.

[53] T. Kittel, S. Vogl, J. Kisch, and C. Eckert. Counteracting Data-Only Malware with Code Pointer Examination. In *18th International Symposium on Research in Attacks, Intrusions and Defenses*, 2015.

[54] T. Kittel, S. Vogl, T. K. Lengyel, J. Pfoh, and C. Eckert. Code Validation for Modern OS Kernels. In *Workshop on Malware Memory Forensics (MMF)*, 2014.

[55] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, 2014. USENIX Association.

[56] Z. Liang, H. Yin, and D. Song. Hookfinder: Identifying and understanding malware hooking behaviors. *Department of Electrical and Computing Engineering*, page 41, 2008.

[57] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *17th Usenix Security Symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association.

[58] L. Litty and D. Lie. Manitou: a layer-below approach to fighting malware. In *1st workshop on Architectural and system support for improving software dependability*, pages 6–11, New York, NY, USA, 2006. ACM Press.

[59] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass, and C. D. McDonell. Linux Kernel Integrity Measurement using Contextual Inspection. In

Proceedings of the 2007 ACM workshop on Scalable trusted computing, pages 21–29. ACM, 2007.

[60] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque Control-Flow Integrity. In *NDSS*, 2015.

[61] D. Oliveira, J. Navarro, N. Wetzel, and M. Bucci. Ianus: Secure and Holistic Coexistence with Kernel Extensions - a Immune System-inspired Approach. In *29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1672–1679, New York, NY, USA, 2014. ACM.

[62] V. Pappas. kBouncer: Efficient and transparent ROP mitigation. 2012.

[63] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, Washington, D.C., 2013. USENIX.

[64] B. D. Payne. Simplifying Virtual Machine Introspection Using LibVMI. `http://libvmi.com/`, October 2012.

[65] B. D. Payne, M. Carbone, M. I. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *IEEE Symposium on Security and Privacy*, pages 233–247. IEEE, 2008.

[66] J. A. Pendergrass and K. N. McGill. LKIM: The Linux Kernel Integrity Measurer. *Johns Hopkins APL Technical Digest*, 32(2):509, 2013.

[67] N. L. Petroni, Jr., T. Fraser, A. Walters, and W. A. Arbaugh. An Architecture for Specification-Based Detection of Semantic Integrity Violations in Kernel Dynamic Data. In *15th USENIX Security Symposium*. USENIX Association, 2006.

[68] N. L. Petroni, Jr. and M. Hicks. Automated Detection of Persistent Kernel Control-Flow Attacks. In *14th ACM conference on Computer and communications security*, CCS '07, New York, NY, USA, 2007. ACM.

[69] N. L. J. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor. In *13th*

USENIX Security Symposium, pages 179–194. USENIX Association, 2004.

[70] J. Pfoh, C. Schneider, and C. Eckert. A Formal Model for Virtual Machine Introspection. In *2nd Workshop on Virtual Machine Security (VMSec '09)*, pages 1–10, Chicago, Illinois, USA, 2009. ACM Press.

[71] J. Pfoh, C. Schneider, and C. Eckert. Nitro: Hardware-based System Call Tracing for Virtual Machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, 2011.

[72] S. Proskurin. Forensic analysis utilizing virtualization on-the-fly. Master's thesis, Technische Universität München, 2016.

[73] N. A. Quynh. Capstone: Next-gen Disassembly Framework. *Black Hat USA*, 2014.

[74] Rekall. Memory Forensics Analysis framework. `http://www.rekall-forensic.com/`, October 2016.

[75] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring. In *Proceeings of the International Conference on Availability, Reliability and Security*. IEEE, 2009.

[76] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *11th international symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[77] J. Rutkowska. Introducing blue pill. *The official blog of the invisiblethings. org*, 22, 2006.

[78] A.-R. Sadeghi, L. Davi, and P. Larsen. Securing Legacy Software against Real-World Code-Reuse Exploits: Utopia, Alchemy, or Possible Future? - Keynote -. In *10th ACM Symposium on Information, Computer and Communications Security (ASIACCS 2015)*, 2015. Keynote.

[79] C. Schneider. *Full Virtual Machine State Reconstruction for Security Applications*. Dissertation, Technische Universität München, 2013.

[80] C. Schneider, J. Pfoh, and C. Eckert. A universal semantic bridge for virtual machine introspection. In *Information Systems Security*, pages 370–373. Springer, 2011.

[81] C. Schneider, J. Pfoh, and C. Eckert. Bridging the Semantic Gap Through Static Code Analysis. In *Proceedings of EuroSec'12, 5th European Workshop on System Security*. ACM Press, 2012.

[82] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *36th IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[83] A. Seshadri, M. Luk, N. Qu, and A. 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, pages 335–350, New York, NY, USA, 2007. ACM.

[84] A. Seshadri, M. Luk, E. Shi, A. Perrig, and L. van Doorn andPradeep Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *twentieth ACM symposium on Operating Systems Principles*, pages 1–16, New York, NY, USA, 2005. ACM Press.

[85] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[86] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In *Presented as part of the 7th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2013. USENIX.

[87] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return address modifications, 2008.

[88] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *2013 IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers (IEEE), may 2013.

[89] Solar Designer. Getting Around Non-Executable Stack (and Fix). Bugtraq Mailing List, Aug. 1997.

[90] A. Srivastava and J. T. Giffin. Efficient Monitoring of Untrusted Kernel-Mode Execution. In *NDSS*, 2011.

[91] D. M. Stanley. *Improved Kernel Security Through Code Validation, Diversification, and Minimization.* PhD thesis, Purdue University, 12 2013.

[92] D. M. Stanley, Z. Deng, D. Xu, R. Porter, and S. Snyder. Guest-Transparent Instruction Authentication for Self-Patching Kernels. In *MILCOM 2012-2012 IEEE Military Communications Conference*. IEEE, 2012.

[93] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy*, SP '13, pages 48–62, Washington, DC, USA, 2013. IEEE Computer Society.

[94] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[95] J. Torrey. MoRE: measurement of running executables. In *9th Annual Cyber and Information Security Research Conference*, pages 117–120. ACM, 2014.

[96] Trusted Computing Group. TPM Specification, March 2011.

[97] V. van der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[98] S. Vogl and C. Eckert. Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture. In *Proceedings of EuroSec'12, 5th European Workshop on System Security.* ACM Press, 2012.

[99] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz. Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data. In *23rd USENIX Security Symposium.* USENIX, 2014.

[100] S. Vogl, J. Pfoh, T. Kittel, and C. Eckert. Persistent Data-only Malware: Function Hooks without Code. In *21th Annual Network & Distributed System Security Symposium (NDSS)*, 2014.

[101] S. W. Vogl. *Data-only Malware.* Dissertation, Technische Universität München, 2015.

[102] Volatility. The Volatility Framework: Volatile memory artifact extraction utility framework. `https://www.volatilesystems.com/default/volatility`, January 2014.

[103] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 380–395, 2010.

[104] R. Wojtczuk. The Advanced return-into-lib (c) Exploits: PaX case study. *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, 2001.

[105] G. Wurster, P. Van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *Security and Privacy, 2005 IEEE Symposium on*, pages 127–138. IEEE, 2005.

[106] B. Zeng, G. Tan, and G. Morrisett. Combining Control-flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *18th ACM Conference on Computer and Communications Security*, CCS '11, pages 29–40, New York, NY, USA, 2011. ACM.

[107] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining trust on virtual calls. In *Annual Network and Distributed System Security Symposium (NDSS)*, 2016.

[108] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[109] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[110] M. Zhang and R. Sekar. Control Flow and Code Integrity for COTS Binaries: An Effective Defense Against Real-World ROP Attacks. In *31st Annual Computer Security Applications Conference*, ACSAC 2015, 2015.

[111] R. Zhang, L. Wang, and S. Zhang. Windows Memory Analysis Based on KPCR. In *2009 Fifth International Conference on Information Assurance and Security - Volume 02*, IAS '09, pages 677–680, Washington, DC, USA, 2009. IEEE Computer Society.