Master's Thesis

Strict separation between OS and USB driver using a hypervisor

Johan Svensson

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

Strict separation between OS and USB driver using a hypervisor

Johan Svensson
johan.svensson692@gmail.com

Advenica AB
Stora Råby Byaväg 88 Lund

Advisor: Martin Hell
Sebastian Nilsson

March 9, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

During 2014, an attack called the *BadUSB attack* surfaced. This attack allows the attacker to reflash the firmware of a USB devices and make it perform malicious tasks. One particularly interesting attack whose source code has been released recently includes modifying a USB flash drive into also acting as a keyboard thus enabling it to send malicious keystrokes.
This thesis presents a modified version of the BitVisor hypervisor along with other possible protection mechanisms and evaluates their efficiency in protecting against this specific kind of attack along with BadUSB attacks in general. In order to test the hypervisor, the initial thought was to construct a BadUSB attack device using the source code made available to the public. When no vulnerable devices were found, emulation of the attack was tried instead. However, emulation did not work either, thus the focus of the evaluation became strictly theoretical. The outcome was that the hypervisor prototype was efficient in protecting against this specific type of BadUSB attack but not against BadUSB attacks in general. The same conclusions were also reached for the other protection mechanisms investigated and evaluated in the thesis.

Table of Contents

Chapter 1

Introduction

Ever since computers became common in the workplace, and later in almost every person's home, there has always been a need to attach external devices such as keyboards, mouses, speakers, and much more. These would be connected through serial ports that came in many varieties using different contacts and transfer speeds. Many of them also did not support plug-and-play which, for the user, was yet another inconvenience.

In 1997, the first computers using the Universal Serial Bus (USB) came onto the market. Even though USB at first had issues with its transfer speed, which was later addressed in USB 1.1, USB solved many of the problems with other serial ports. The transfer speeds are variable, the USB protocol supported plug-and-play, and it was based strictly around a single type of contact. From that point, USB has evolved from supporting everything from keyboard and mouses to gadget devices such as USB-connected cup warmers and reading lamps.

During 2014, the concept of reflashing a USB chip's firmware and using it as an attack vector came into discussion. This knowledge, along with some creativity, was put together into a concept called **BadUSB** which will be one of the main focuses of this thesis. To shortly summarize, BadUSB involves reprogramming a USB device e.g. a USB flash drive thus changing the firmware. By doing this it is possible for an attacker to e.g. send malicious keystrokes, spoof DNS records. How would one go about protecting against this? There already exist a few alternatives that is capable of protecting against BadUSB to a certain degree.

A concept that has not yet been tested is whether it is possible to use virtualization to protect against BadUSB attacks. Since virtualization involves separating the operating system (OS) from the hardware, and therefore the USB device itself, this could potentially be possible to accomplish, even without modifying the OS in any manner.

The concept will be tested by designing and implementing a hypervisor prototype with protection against BadUSB along with a device that can perform a specific form of BadUSB attack, sending malicious keystrokes while functioning normally. However since, there were no vulnerable devices available and due to time constraints, the testing itself was abandoned and the focus of this reported is strictly theoretical.

1.1 Goals

The overall goal of this thesis is to investigate BadUSB attacks and whether a hypervisor, through separation of the USB stack from the rest of the OS, can protect against all, or a subset of, BadUSB attacks.

To do this, virtualization techniques, more specifically paravirtualization and full virtualization, among with open-source alternatives utilizing these techniques will be investigated. Furthermore, the investigated open-source alternatives will be used as a starting point for a hypervisor prototype, capable of protecting against BadUSB attacks, which is then designed and implemented.

The original goal was for the hypervisor prototype to be tested with and without the protection activated and the results then compared to other types of protections, more specifically *TMSUI*, *GoodUSB*, and *Udev*. However, due to difficulties in testing the hypervisor prototype, later detailed in the report, a theoretical analysis was performed instead.

1.2 Definitions and abbreviations

OS Operating System

USB Universal Serial Bus

Udev The device manager of the Linux kernel.

Attack vector - A path or means that can be used by an attacker to e.g. infected a computer.

DNS Spoofing - The act of spoofing i.e. adding a false DNS record thus making the requester visit e.g. a mock-up website.

UHCI Universal Hardware Controller Interface - a USB chipset standard that supports USB 1.X

EHCI Enhanced Hardware Controller Interface - a USB chipset standard that supports USB 1.X and USB 2.0

XHCI Extensible Hardware Controller Interface - a USB chipset standard that supports USB 1.X, USB 2.0 and USB 3.0

URB USB Request Block - a data structure used to describe a request/reply sent by either a USB host or controller.

IOMMU Input/Output Memory Management Unit - a type of hardware unit that connects a **DMA**-capable I/O bus to the main memory.

DMA Direct Memory Access - a technique where a device on the motherboard can access the computer's memory without getting the CPU involved.

qHD Queue Head Descriptor - A data structure used in the host controller to describe the head of a queue.

qTD Queue Element Transfer Descriptor - A data structure used in the host controller to describe an element of a queue.

Low-speed - A mode introduced in USB1.X supporting transfer speeds up to 1.5 Mbps.

Full-speed - A mode introduced in USB1.X supporting transfer speeds up to 12 Mbps.

High-speed - A mode introduced in USB2.0 supporting transfer speeds up to 480 Mbps.

Frame/Microframe - Base unit of time used during USB communication. Consist of 125 μs in the USB 2.0 standard and 1 ms in the USB 1.X standard [1, p. 36].

Chapter 2
Background

2.1 USB - The Universal Serial Bus

The first standard of the Universal Serial Bus, USB 1.0, was released in January of 1996. Since then, there have been another 4 major revisions/additions to the USB standard. These are in order of release: USB 1.1(August 1998), USB 2.0 (April 2000), USB 3.0 (November 2008), and USB 3.1 (January 2013). Common for all standards is the introduction of new (higher) transfer speeds. Additionally, each standard is typically associated with a new type of host controller which is defined in a separate standard. The host controller and parts of their inner workings will be explained further in Section 2.1.4 but before these can be fully understood, the USB communication protocol must be explained.

2.1.1 The USB Communication flow

The Universal Serial Bus in its most basic form consist of a host controller, later described in Section 2.1.4, and one or more USB hubs that are connected in a tiered-star topology. One of the hubs is physically built into the bus and is referred to as the *root hub*. The hubs acts as *concentrators* i.e. they turn what would only be a single port into multiple ports.

The port(s) typically consist of 4 lines, 2 for power and 2 for data transmission. The communication itself is however split into logical channels called *pipes* [1, p. 33 - 34]. Each *pipe* corresponds to an *endpoint* on the device. Due to this, it is quite common that the terms *endpoint* and *pipe* is used interchangeably since a *pipe* always correspond to a single *endpoint*. Since different devices have different requirements on the amount of data that is to be sent as well as the timing of the data, there are 4 different endpoint categories.

Isochronous endpoint

Isochronous endpoints provide continuous and steady bandwidth. This is especially useful when transmitting data for e.g. voice or video. This endpoint category provides error-detection but does not guarantee delivery. It does however guarantee that an attempt at transmission will be made within a certain timeframe [1, p. 21]. Since this kind of endpoint is not used in the solution, the workings of isochronous endpoints will not be further detailed.

Control endpoint

Control endpoints are used for status and command communication. These packets are sent *best effort* i.e. the delivery is not guaranteed [1, p. 21]. One of the most important uses for Control endpoints is USB enumeration, later detailed in Section 2.1.3.

Interrupt endpoint

The interrupt endpoint provide the functionality to transfer small amounts of data within a certain timeframe. This is used for e.g. USB mouses and USB keyboards [1, p. 21].

Bulk endpoint

Bulk endpoints provide the functionality to transfer large bursts of sequential data. This is typically used for e.g. printers, scanners, and most notably to send data to and from the flash memory in USB flash drives. It also provides reliability in that it includes error-detection as well as a (limited) number of retries for resending. Bulk data, however, does not guarantee that the data will be transmitted within a certain time frame since it uses spare bandwidth when the bus is not busy sending [1, p. 21].

Endpoints, Interfaces and Functions

Besides from the *Default Control Pipe*, also known as *Endpoint zero*, the endpoint(s) are organized into interfaces. Both the amount of available endpoints and interfaces may vary for a device depending on its function, or *configuration*. Exactly how the configuration of the device is done will be described further in Section 2.1.3.

2.1.2 The USB protocol

The USB protocol is centered around transmitting data using packets. This is done using a master/slave topology i.e. the attached *USB device*, the slave, sends/receives data when the *USB host*, the master, tells it to do so.

All data that is sent in either direction is always associated with (at least) three packets: A **Token** packet that controls the direction of the data flow, a **Data** packet containing the actual data and a **Handshake** packet that reports the outcome/status of the transmission. These 3 stages can be used to divide a majority of the available packets into the categories Token, Data and Handshake [1, p. 196]. Sending a Token, a Data, and finally a Handshake packet is also referred to as a **bus transaction**.

All packets follow a similar format although some packet might have additional fields. The common 3 fields in all packets are **Synchronization**, **PID** and **End-of-Packet** (EOP). The Synchronization field is, as the name suggests, used for synchronization. More specifically it is used to synchronize the device's clock with the clock of the host controller. When running in low- or full-speed mode

this field is 8 bits while it is 32 bits in high-speed mode [1, p. 37].

The PID field is used to determine what kind of packet is being sent. All available packet types can be seen in Table A.1, located in Appendix A.1. Finally, there is also the End-of-Packet field which as the name may suggest marks the end of the packet.

The Token category

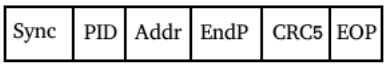

Figure 2.1: Token Packet

The **Token** category consists of *IN*, *OUT*, *SETUP*, and *SPLIT* packets, the latter of will be described in the next section. *IN* and *OUT* packets are used for flagging incoming/outgoing transmissions. *SETUP* on the other hand is used to send controls messages. This includes configuration of the device as well as probing for information about the device. This will be more thoroughly explained in Section 2.1.3. As shown in Figure 2.1, **Token** packets also contain the address (Addr) of the device, which endpoint (EndP) the data should go to along with an error detection code, CRC5, used on the address field and the endpoint field [1, p. 199].

The Data category

Figure 2.2: Data Packet

Once the direction of the transmission has been established, the transmission continues with a packet of the **Data** category. The basic outline of these are always the same, with the *PID* field being the only difference, as shown in Figure 2.2.

In its most basic form, only the *DATA0* PID is used to send data. *DATA1* is used by non-isochronous transactions when Data Toggle Synchronization is required. By alternating the PID between *DATA0* and *DATA1*, it is possible to make sure that the data arrives in the correct order [1, p. 232]. The other two PIDs, *MDATA* and *DATA2*, are used in isochronous transactions while split-isochronous transactions only use *DATA2*.

Split transactions are used when transmitting data to a device running at a speed different from the hub it is connected to. A split transaction begins with a special token called a **SPLIT** token which are used in Start-split transactions (SSPLIT) and in Complete-split transaction (CSPLIT). Start-split transactions

begin with a SPLIT token and is followed by the *Token* that the host controller wants to send to the device. If the host controller is sending data, as is the case when sending a *SETUP* or *OUT* token, the data is also sent directly after the *token* [1, p. 441]. The hub will then respond with an *ACK* if the transaction was received successfully.

The hub will now perform the transaction in the same manner as the host controller would have done if it was connected directly to the device. The host controller will during this time wait for the transaction to be completed. Once the host controller thinks that the transaction should have been completed, it sends a *SPLIT* token followed by the original *Token*. The hub will then respond with the data, in case of a *IN* token, or simply with an *ACK* handshake if it was a *SETUP* or an *OUT* token. The hub may also respond with 4 other handshake packets to indicate an error of some kind:

- *NYET* - The hub has not received a response from the device yet [1, p. 476 - 479].
- *NAK* - The hub is unable to perform the operation due to lack of buffers [1, p. 481] or the device is simply not ready to send yet.
- *ERROR* - The hub encountered a transaction error during an **interrupt** transaction [1, p. 498].
- *STALL* - The device sent a *STALL* [1, p. 463, 507, 531] or 3 consecutive transaction errors between the hub and the device have occurred during a **bulk** or a **control** transactions [1, p. 477].

The Handshake category

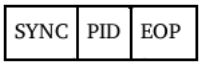

Figure 2.3: Handshake Packet

The **Handshake** category consists of 5 kinds of packets that are used to communicate the outcome/status of the transmission. As shown in Figure 2.3 they only consist of the three fields common for all packets i.e. they communicate their message simply by being sent. The 5 available messages are:

- *ACK* - The packet was received successfully [1, p. 206].
- *NAK* - The device cannot communicate any data [1, p. 206]. This may be due to several reasons e.g. there is no more data to send, the device is busy processing data already in flight.
- *STALL* - The endpoint has encountered an error of some kind that the USB host needs to address [1, p. 207].

- *NYET* - Used in high-speed mode (USB 2.0+) either as a response to the PING protocol or by a hub as a response to a non-completed full/low-speed transaction [1, p. 207].
- *ERR* - Used in high-speed mode (USB 2.0+) for reporting an error in a high-speed hub for a full/low speed bus [1, p. 207].

2.1.3 USB enumeration

Before an attached USB device can be used, the OS requires information about the device e.g. what kind of device it is, what manufacturer made it. The process of gathering this information is referred to as **USB enumeration** and is done so that the correct driver can be loaded by the OS. In order to fully understand how the USB enumeration process works, the usage of *SETUP* transactions must first be explained.

The SETUP transaction

A *SETUP*-transaction begins with a *SETUP*-packet and is followed by a **Data**-packet consisting of 8 bytes divided into 5 fields. These are:

1. bmRequestType (byte 0) - Used to determine the nature of the request e.g. the direction of the data, what type of request, and who the recipient is.
2. bmRequest (byte 1) - Tells what specific kind of request is made. All possible requests are listed in Table 2.1.
3. wValue (byte 2:3) - Depends on the kind of request.
4. wIndex (byte 4:5) - Depends on the kind of request.
5. wLength (byte 6:7) - The length of the data stage. If this field is 0 there is no additional data in the response.

The response begins with the host sending an *IN*-packet followed by a *DATA0*-packet sent by the device. This packet will consist of the same header as in the previous stage. Additionally it will also contain the data of length **wLength** that was requested. Finally the data is, if the transaction is error-free, acknowledged with an *ACK*-packet.

If the **Maximum Packet Size** is less than the size of the requested descriptor, the *IN* transactions are repeated until the whole descriptor has been sent to the host controller. The host controller will then acknowledge the descriptor by sending a empty *OUT* transaction.

The USB enumeration process

The enumeration process begins when a device is plugged into a USB hub connected to the host controller. The host controller learns about the newly attached device through an interrupt endpoint that report the hub's status [2, p. 31].

At this point, the host controller will issue a *GET_PORT_STATUS* request. The information sent in this request gives the USB host controller information

Table 2.1: USB Device Requests

bRequest	wValue	wIndex	wLength
GET_CONFIGURATION	Feature selection	Interface endpoint	1
GET_DESCRIPTOR	Desc Type and Index	0 or Lang Index	Desc Length
GET_INTERFACE	0	Interface	1
GET_STATUS	0	Interface endpoint	2
SET_ADDRESS	Device address	0	0
SET_CONFIGURATION	Configuration value	0	0
SET_DESCRIPTOR	0	0	0
SET_FEATURE	Feature selector	0	0
SET_INTERFACE	Alternate setting	Interface	0
SYNCH_FRAME	0	Endpoint	2

whether the device can run in Full-speed or only in Low-speed [2, p. 31].

The host controller resets the port of the new device by issuing a *SET_PORT_FEATURE* request to the hub. During the reset, the hub can detect whether High-speed is supported. If this is the case, this mode is enabled [2, p. 31].

The host controller will now issue one or more *GET_PORT_STATUS* requests until the device leaves the reset state [2, p. 31]. When this occurs, the device can be communicated with using the default address 0. The host controller will do an initial probe of the device by requesting its Device Descriptor. Using the Device Descriptor, the host controller can find the **Maximum Packet Size** [2, p. 32]. The port is then reset again.

The device is assigned a device number which in the future will be the content of the *Addr* field until the device is detached or encounters an error forcing it to reset. The assignment is done by sending a *SET_ADDRESS* request (see Table 2.1) with the device address in the *wValue* field [2, p. 32].

When the device has been assigned an address, the probing and configuration of the device can begin. The host will send a series of *GET_DESCRIPTOR* requests. The first request is typically for the Device Descriptor which will give some initial information about the device (device class, vendor id, product id, etc.). If there are available configurations and interfaces, these will typically also be probed by requesting the Configuration and/or Interface Descriptors in the same way as the Device Descriptor [2, p. 32].

After the probing is completed, a *SET_CONFIGURATION* request containing the configuration number is issued [2, p. 32]. When this is completed, the USB enumeration process is done and the device is ready to be used.

2.1.4 The USB host controller

The underlying hardware that actually sends and receives the packets to and from the device is called the USB host controller. The software communicates with the hardware using a register-level interface on top of the hardware.

During the development of USB, there has in total been 4 different kinds of standards for interfaces that handle this communication. These 4 standards are, in order of earliest to latest, OHCI (USB 1.1), UHCI (USB 1.X), EHCI(USB 2.0),

XHCI (USB 3.X). The currently most used standards are EHCI and XHCI where EHCI is still present in many computers and XHCI being introduced to many new computers. A common design choice in many motherboards is to have multiple host controllers, often one XHCI controller and one EHCI controller. This section will however focus on the EHCI standard, since the XHCI standard will not be used in this thesis.

The EHCI standard

The EHCI bus is, just like many other buses, found and identified by the software using PCI configuration registers which are memory-mapped registers. These are used to identify the USB host controller and its capabilities [4, p. 7]. Among the different registers, one of the most relevant for understanding the EHCI standard is the *USBBASE*-register. The *USBBASE*-register contains the base address of the *Host Controller Register Space*. This is where the registers that concern the actual operations of the USB host controller are located.

The first 20 bytes of this area of the memory are called the *Host Controller Capability Registers* [4, p. 13]. These consist of a version number, structural parameters (number of supported ports, port routing rules, port power control etc), capability parameters (64 bit addressing capability, parameters regarding schedules etc), and an optional field that helps the controller keep track of what port is mapped to what companion host controller (in case there are more than one).

In order to keep track of where the *Operational registers* begin, there is also a field called *CAPLENGTH* that holds the offset to the *Operational registers* relative to the *USBBASE*. The *Operational registers* can be divided into 9 different kinds of registers which all are 32 bits.

Asynchronous transactions

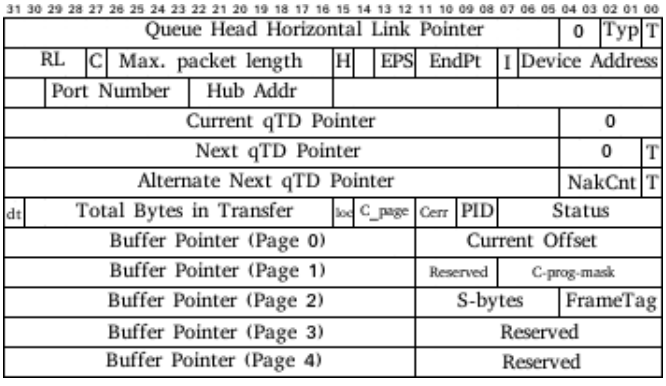

Figure 2.4: Queue Head Format

The asynchronous list is used for sending bulk transactions and control transactions. These are represented using 2 vital data structures, **Queue Heads** (QH)

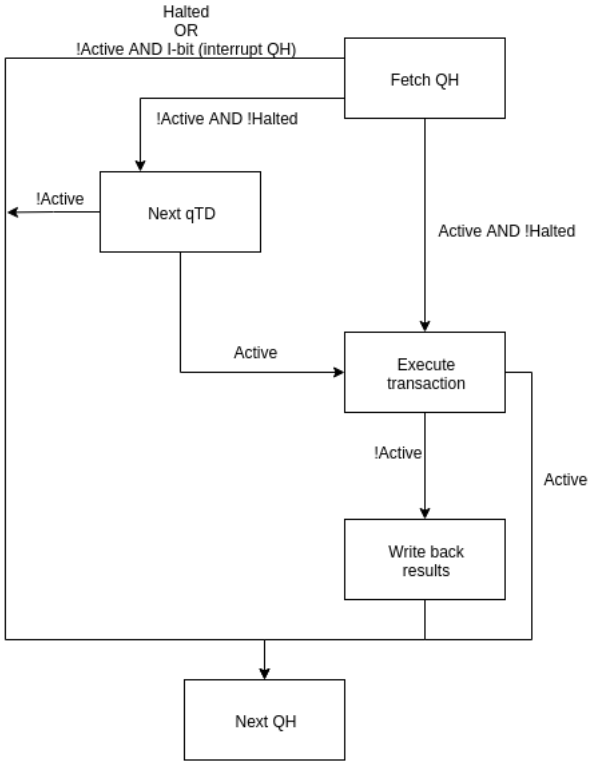

Figure 2.5: Asynchronous list state machine

and **Queue Element Transfer Descriptors** (qTD). *Queue heads*, as depicted in Figure 2.4, consists mostly of fields used for addressing and the current status of the link. Another important part of the QH is the overlay area, which consists of a total of 6 buffer pointers. These are used as intermediate storage when performing transactions stored in a qTD.

The qTD(s) are used for representing the transactions themselves [4, p. 71]. Each qTD contains 2 pointers that points to other qTDs in the queue (or itself if the qTD is alone in the queue). Each qTD also contain 5 buffer pointers, each pointing to a buffer used for storing data of each *bus transaction*. Finally, the qTD also contain status- and option information regarding the link e.g. *PID*, *Interrupt on Complete*, *Total Bytes to Transfer*.

The asynchronous list itself consists of Queue heads linked together in a circular list. The current element of the list is pointed to by the **ASYNCLISTADDR** register of the USB host controller's *Operational registers*. How the list itself operates will not be described in much detail since it is not needed for understanding the thesis.

The list is operated by fetching QHs from the list and looking at each QH's qTD(s), mainly at each qTD's *Active* bit and *Halted* bit, as described in Figure 2.5. What is not shown in the Figure is how the traversal stops. In order to make sure that the list is not traversed more than once, the Reclamation bit of the

USBSTS register and the **H-bit** of each QH is used. Initially, the Reclamation bit is set to 1. As the list is traversed and the head of the queue, having its **H-bit** set to 1, is found the Reclamation bit is set to 0. Once this element is traversed again the host controller determines the asynchronous list to be empty and stops its traversal. It is also worth mentioning that if the Reclamation bit is set to 0 and a transaction is executed, the Reclamation bit will be set to 1 again.

The EHCI host controller also consists of more parts, e.g. the periodic list, which are left out since these are of no relevance to the thesis. The asynchronous list, however, is a vital part of the emulation implementation, as later described in Section 3.6.2.

2.2 Virtualization using a hypervisor

The concept of virtualization in computing involves creating a virtual aspect of a computer which may range everything from storage devices to entire hardware platforms, the latter of which will be the focus of this section. The hardware platform that is to be virtualized is the x86-platform which is the most common used by personal computers today.

The virtualization is performed using a **hypervisor**. A hypervisor is, in its most basic form, a minimalistic OS that enables one or more operating system(s), also called **guests**, to run alongside one another. The task of the hypervisor is to act as a scheduler for the CPU(s) i.e. deciding which guest gets to use the CPU next along with controlling access to the hardware of the computer e.g. the main memory, the USB host controller.

Hypervisors also come in two different types depending on when it starts execution. The first kind, **hosted hypervisors**, are hypervisors that are hosted by another OS while the second kind, **native hypervisors**, are hypervisors that begin their execution before any other OS. The key difference between these two types of hypervisors is the amount of control they have of the hardware. Since the native hypervisors lay directly on top of the hardware it has more or less full control over the hardware. By using a native hypervisor it is possible to create a clear separation between the OS and the rest of the hardware which in turn is one of the main goals of this thesis.

The most common native hypervisors in use today use 2 different techniques to perform virtualization. These are **full virtualization** and **paravirtualization**. To shortly summarize, paravirtualization addresses the problems of virtualization by changing the guests' kernels, thus taking a more software-based approach. Full virtualization, on the other hand, has to allow the guest to run unaltered. Because of this limitation, modern full virtualization solution often relies on hardware assistance (hardware-assisted virtualization) to improve performance.

The two chosen open-source software alternatives, **Xen** and **BitVisor**, (mainly) uses paravirtualization and full virtualization respectively (Xen supports full virtualization as well but performance is by far better when using paravirtualization). This section will describe how both of these techniques go about solving the issues of virtualization but before jumping into this, the common issues that all virtualization solutions must address has to be put into context.

2.2.1 General issues in virtualization

The goal of the virtualization solutions that will later be presented is to create a virtual version of the x86 platform. What this means is that (almost) all aspects of the x86 platform needs to be emulated by the hypervisor to the guest. Since the task of virtualizing an entire hardware platform is very extensive this has been simplified into 3 aspects that are deemed to be the most important for understanding how virtualization works. These aspects are **privileged instructions**, **memory**, and **interrupts**. **Device I/O** is also a very important aspect of virtualization, especially considering the goal of this thesis (since the USB host controller is essentially a I/O device). This will however be explained later in Section 3.5.1.

Privileged instruction

Privileged instructions are special instructions in the CPU's instruction set that require the CPU to be in a state where it has the appropriate permission to execute the privileged instruction. On the x86 platform this feature is enabled when the CPU is in **protected mode**. The other mode, where the feature is deactivated, is called **real mode**.

When the CPU is in the **protected mode**, the processor may be at 4 different privilege levels also called **rings**. These are numbered from 0 to 3 where 0 has the most privilege and 3 has the least privilege.

In virtualization, this feature plays a very large role in causing the separation between the guest OS and the hypervisor. By letting the hypervisor run in ring 0 it is possible for the hypervisor to limit the amount of control that the guest has over e.g. the main memory along with other hardware. However, it is not uncommon that operating systems also use this feature so that an application runs in ring 3 while the kernel runs in ring 0. Because of this, the hypervisor must emulate different privilege levels in some manner so that the OS can function normally.

Memory

Even when using virtualization memory is accessed in the same manner as it is always done, by giving the CPU an instruction telling it to read and/or write to a certain memory address. Example:

```
;Store the value 5 at memory address 0x8000
MOVL $5, 0x8000
```

In order to understand the solutions fully, the basic mechanics of memory management must first be explained. Basically all operating systems use virtual memory i.e. it has a *virtual memory space* that maps to a *physical memory space* residing in the actual RAM.

Since translating individual memory addresses are quite difficult, the memory is divided into chunks, also called **pages**, consisting of usually 4096 bytes. In order to access the memory the CPU uses a **MMU** (Memory management unit) whose task is to translate the virtual address entered by the CPU into an actual physical address. Some implementations have the **MMU** built into the CPU but

for simplicity it is assumed that this is not the case. Keeping track of the virtual to physical mapping in memory is done using a **page table**. A *page table* is a table containing mapping between a virtual page and a physical page.

This becomes an issue in virtualization since both the hypervisor and the guest will want to use this functionality. To solve this, there needs to be a mechanism that does not allow the guest to access certain parts of the memory or alternatively there needs to be a mechanism that keeps track of virtual addresses relative to the hypervisor and guest respectively.

Interrupts

An interrupt is a mechanism used by both hardware and software to signal the CPU that an event that needs attention has occurred. When an interrupt occurs, the CPU suspends the current activities by stopping whatever it is doing, saving its state (all registers of the CPU), and finally jumping to the **interrupt handler**, a function that handles the interrupt.

The main issue with interrupts in virtualization is that the hypervisor needs to control the interrupts. What this means in practice is that the hypervisor has to have control over the **IVT** (Interrupt vector table), which controls which function should handle which interrupt/error/exception. The hypervisor must also control the enabling/disabling of interrupts by controlling the interrupt flag (**IF**) as well as controlling interrupt masking (turning individual categories/types of interrupts on/off).

2.2.2 Full virtualization

Full virtualization on the x86 platform was for a long time implemented but impractical to use, mainly due to performance factors. The first major step towards full virtualization becoming a viable option was achieved in 2005 - 2006 when hardware-assisted virtualization was introduced to the x86-platform through the implementation of AMD-V for AMD processors and Intel VT-x for Intel processors. Both of these techniques were at their introduction not very groundbreaking i.e. full virtualization did not become a viable option over night [5]. Over time however, both of the techniques have evolved in making full virtualization a possibility performance-wise.

Privileged instructions

Before hardware-assisted virtualization became possible, a technique called binary translation was used. Binary translation involves replacing all privileged instructions in the guest's kernel, upon loading it into memory, with a corresponding hypervisor call. This way, all privileged instructions are handled by the hypervisor.

Binary translation was for a long time the only viable method to be used in full virtualization until x86 processors supporting Intel VT/AMD-V (hardware-assisted virtualization) came onto the market. The Intel VT solution to the issue of privileged instructions is to treat the execution of these instructions as interrupts.

When a privileged instruction occurs, the hardware saves the guest's current context/state in a *Virtual Machine Control Structure* so that it can later be restored when the privileged instruction is done executing. Control is then handed over to the hypervisor which then executes the privileged instruction and then hands the control back to the guest [6].

Memory

In full virtualization, the concept used to address the issue of memory management is called **nested page tables**. Nested page tables includes adding an extra *page table* for each guest where a virtual address in the guest maps into a (still) virtual address in the guest's extra page table which is then in turn mapped to the physical memory. The shadow page table(s) themselves are either managed by the hypervisor or edited by the guest and then validated by the host depending on the implementation [7].

From the beginning, this was strictly software-based which caused some issues with performance. Since then, support for doing this in hardware has been added to both Intel VT and AMD-V. When done in hardware, this is referred to as **EPT** (Extended Page Table(s)) and when done in software it is referred to as **shadow paging**.

Interrupts

Interrupts in full virtualization are usually handled using some sort of *trap-and-emulate* scheme where the interrupts is intercepted by the hypervisor, handled, and then sent virtualized to the guest.

In Intel VT, the problem is solved in exactly this manner. Control is handed over to the hypervisor once an interrupt occurs. The hypervisor then handles the interrupt and injects a *virtual interrupt* into the guest once the guest resumes execution [8]. Another aspect of the interrupt handling that must be addressed is regarding the control of the *interrupt flag* (**IF**) and the control of masking/unmasking interrupts. Both of these actions are considered to be *privileged operations* i.e. the control is automatically given to the hypervisor when these events occur thus the problem is handled.

2.2.3 Paravirtualization

Paravirtualization, virtualization involving changing the guest system in some manner, was first used as early as 1972 in a virtualization OS made by IBM. Since then, both commercial native hypervisors e.g. VMware and open-source hypervisors e.g. Xen has adopted this concept. Since Xen is one of the most widely used open-source solutions to x86 paravirtualization, with approximately 10 million users [15], many of the solutions to the issues of x86 virtualization will be directly related to Xen.

Privileged instructions

The *paravirtualized* solution to the issue of privileged instructions was for a long time easier and more efficient than full virtualization. Since it is possible to modify the kernel, all privileged instructions are replaced with a call to the hypervisor. This was for a long time a much better solution to the issue than e.g. binary translation and also for some time the hardware-assisted implementation. In recent years this has however changed and nowadays the hardware-assisted implementation is in most cases slightly faster than the *paravirtualized* implementation [9].

Memory

As discussed in the section about full virtualization, memory management which is software-based uses a concept that is called *shadow paging*. In paravirtualization it is also entirely possible to use *shadow paging*. However, since this has already been explained it may be worth mentioning how **Xen**, one of the software alternatives that will later be examined, deals with memory management [9].

Xen takes a completely different approach and grants the guests direct read access to the hardware page tables. The updates the guest wishes to make are instead batched, validated, and finally written to the page table by the hypervisor [9].

Interrupts

The solution to interrupts in paravirtualization is usually solved by using an event-system that is integrated into the guest's kernel. The specifics on how this is designed and implemented can be found by the reader by e.g. looking at the documentation of **Xen** [9].

Chapter 3
BadUSB

The BadUSB attack was first presented by a group of German computer security analysts at Blackhat USA 2014. The attack consists of reprogramming the firmware of a USB device in order to make it perform malicious tasks [14]. The attacks shown during the presentation include emulating a keyboard, DNS spoofing by emulating a USB network adapter, and adding a virus to the boot sector of a USB drive. Additionally, it was also proposed that it might be possible to update BIOS and take control over it, inject viruses/malicious code into files on a USB flash drive, etc.

In this thesis, there is a limitation to only one kind of BadUSB attack which is an attack that involves reprogramming a USB flash drive into having an extra keyboard interface which is used to send keystrokes for e.g. spawning a reverse-shell or something similar. This chapter will in detail explain how this kind of BadUSB attack works as well as possible means of protection/mitigation. A hypervisor prototype that will provide some protection against, at least this kind of, BadUSB attack will be designed using an open-source alternative and then implemented. Both the attempt at building a BadUSB device along with the attempt at emulating the attack will also be detailed.

3.1 Autorun malware and Stuxnet

Although the concept of reprogramming USB devices had not really been fully considered, the use of a USB device as an attack vector is far from new. One particularly interesting approach is exploitation of the Windows autorun feature. The autorun feature enables an attacker to make executables run upon plugging in the device. This enables the attacker to infect the computer with malware, viruses, trojan, rootkits, etc. sometimes even without the user noticing it [10]. Once the computer is infected, it is not impossible that other USB flash drives attached to the computer also become infected, thus spreading the virus further.

One notable example of a worm that utilized this, among other techniques, to spread itself is the **Stuxnet** worm. **Stuxnet** surfaced in 2010 [11] and targets Windows systems with the purpose of infecting and controlling *programmable logic controllers* (PLC). The specific kind of PLC it wanted to target was a type of controller that controls some form of spinning e.g. pumps, centrifuges, etc. By changing the rotational frequency and masking the sensor values, it could effec-

tively destroy the machine controlled by the PLC by increasing and decreasing its rotational speed to speeds that it cannot manage [12]. Beside the autorun feature, the Stuxnet worm also uses RPC (Remote Procedure Call) once it is inside the facility in order to spread itself further throughout the network.

3.2 Reprogramming of USB firmware

Before BadUSB was presented, the idea of reprogramming USB devices had already been considered. The example that comes the closest to BadUSB can be seen in a presentation from Shmoocon 2014 by Richard Harman [13]. The attack involves using *Phison MPALL*, software used for recovery of Phison USB controllers, to change settings of a Phison USB device. This can potentially be used for reprogramming a USB flash drive in the same way as with BadUSB but instead it was suggested how to produce secret partitions on the USB device.

3.3 The BadUSB attack

The BadUSB attack consists of two stages, reprogramming the device's firmware and attaching the USB device to a computer, thus making the attack execute. The reprogramming of the firmware will later be detailed in Section 3.6.1. The second stage of the attack can be summarized as [14]:

1. The device is detected and enumerated as described in the Section 2.1.3.
2. During the enumeration process, the device reports itself having a total of 6 endpoints, 1 for control/status (endpoint 0), 3 for transferring data to and from the USB flash drive, and 2 for emulating a keyboard that can perform the actual attack.
3. Once *SET_CONFIGURATION* has been sent and acknowledged, the device can be used. By this time, the partition(s) on the USB flash drive can be accessed by the OS and the malicious keystrokes are sent.

The **Device Descriptor** (Table A.2) sent will have its reported *Device Class* and *Device Subclass* set to **0x00**. By doing this, the device can report that it supports multiple classes. This is then followed by the **Configuration Descriptor** (Table A.3) consisting of 2 *Interface Descriptors* (Table A.4), one consisting of 3 endpoints and the other one consisting of 2 endpoints, both shown in Figure 3.1. More specifically, the first interface is the interface to the flash drive and the other interface is the interface to the emulated keyboard used in the attack.

The flash drive's interface consist of 3 endpoints where 2 endpoints are *Bulk endpoints*, one going **IN**, and one going **OUT**. The final endpoint of the interface is an *Interrupt endpoint* going in the **IN** direction. The emulated keyboard's interface consist of 2 endpoints, one being a *Interrupt endpoint* going **IN** as well as a *Control endpoint* used as a dummy endpoint.

The setup described allows the USB flash drive to still function as a normal USB flash drive yet being fully capable of performing the attack. It is worth mentioning that this specific setup is not a requirement for the attack to work.

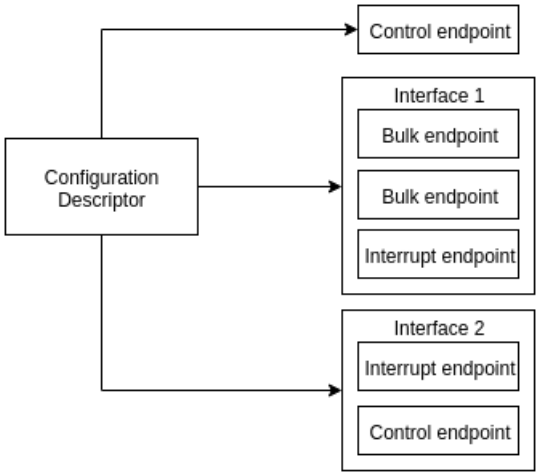

Figure 3.1: BadUSB device configuration descriptor

One approach could be to simply make the USB flash drive only act as a keyboard and perform the attack every time it is plugged in which would be more than sufficient in order for the attack to work. The process of reprogramming the USB device's firmware and replacing it with a custom firmware will be discussed further in Section 3.6.1.

3.4 Protection against BadUSB

Currently there exists 2 security schemes that have been specifically designed with BadUSB in mind. In this section, both of these security schemes, **GoodUSB** and **TMSUI**, will be explained. Since the target system is Linux, using Udev to filter out malicious devices will also be examined to investigate whether it also can be used to protect against BadUSB.

3.4.1 Udev

All devices that are attached to the computer, e.g. Hard drives, USB devices etc. are all managed by device drivers that run in the kernel. Since some of these devices might need to communicate with applications running in userspace, outside of the kernel, any changes to the device(s)' state has to be passed on.

Udev can be regarded as a layer that handles communication between the kernel and userspace. This is mainly done by handling device node files. These files are claimed by device drivers running in the kernel and may then be written to and read from in order to pass information between userspace and the device driver running in the kernel.

Udev can also be used for filtering out devices. When a USB device is plugged in, Udev gets access to almost all of the information supplied by the device during enumeration. By changing the configuration file(s) of Udev, it is possible to execute

a script upon a device being attached to the computer. This filter can also be customized to filter out certain interfaces depending on their class. For the attack used in this thesis, as described in Section 3.3, the best approach would be to have a script execute when booting up the kernel. The script would set the kernel to block all new HID devices from being attached and used (keyboards and mouses belong to this category). When the computer is shut down, another script re-enabling new HID devices would run so that the devices are allowed during boot. An example of such a rule is:

```
ACTION=="add", ATTR{bInterfaceClass}=="03" RUN+="/bin/sh
    -c 'echo 0 >/sys$DEVPATH/../authorized'"
```

The rule states *"If an interface with class **0x03** (HID interface) is found, write a 0 to the file 'authorized' in the device's corresponding sysfs folder"*. **Sysfs** is a virtual file system used by the Linux kernel to export various information about devices attached to the system, including USB devices. The variable **$DEVPATH** corresponds to the device's path relative to the */sys* directory e.g. */bus/usb/devices/<device number>*. Finally, writing a 0 to the file *authorized* in the device's corresponding folder will disable the device from being used.

This rule needs to be enabled once the actual mouse and keyboard has been enumerated and can be used. The options is between enabling/disabling the rule during boot/shut down or during login/logout. In order to make the implementation more fail-safe it is probably a better alternative to enable the rule after login/disable the rule after logout. The rule may be put into a configuration file e.g. *usbblock.rules*. Enabling the rule could then simply be done using the *sed* command (stream editor) as such:

```
sed -i 's/#//' /etc/udev/rules.d/usbblock.rules; udevadm
    control --reload-rules
```

What the command does is essentially 2 actions. First, it finds all commenting characters, the character "#" in this case, and removes it/replaces it with nothing. Secondly, it forces the Udev service to reload its set of rules, thus enabling the rule (since the rule is now uncommented). Disabling the rule is done as such:

```
sed -i 's/^/#/' /etc/udev/rules.d/usbblock.rules;
   udevadm control --reload-rules
```

This command does essentially the same actions but rather than replacing "#" with nothing, it inserts a "#" character at the beginning of every line of the file. Since there is only one line in the file, this means that the rule is now seen as a comment thus making the rule not load when the set of rules are reloaded. Worth noticing is that the idea and implementation for this came from a thread on StackExchange [20].

3.4.2 GoodUSB

GoodUSB is a recently presented and implemented scheme [21]. It is implemented for Linux and consists of 3 vital componenets: A graphical interface, an

emulated USB honeypot, and a policy engine, also called a **mediator**.

At the center of the design is the **mediator**. Its purpose is to decide whether a device can be trusted or not. In order to do this, a graphical interface is used. The **mediator** is connected to a modified Linux kernel. Using a netlink, the **mediator** gets access to the information supplied by the device during enumeration. This information is then shown to the user in using the graphical interface. The user will then select a profile from a list. Each profile describes a kind of device and what interfaces it should support, presented in a simple fashion so that even the most uninitiated users can understand. Using the user feedback together with the information received during enumeration, the **mediator** can decide whether the device can be trusted or not [21].

If the device can be trusted, the **mediator** routes the USB traffic to the actual OS and the device, as usually. In the case the **mediator** decides that the device cannot be trusted, it is instead redirected to the emulated **USB honeypot**. The **USB honeypot** is a virtual machine (VM) running using **KVM**, **Kernel-based Virtual Machine**. **KVM** is a module/kernel extension that is capable of turning a regular *Linux kernel* into a hypervisor with the use of hardware-assisted virtualization based on Intel VT and/or AMD-V [22]. Within the **VM**, a USB host controller, is emulated using **QEMU**, a *hosted hypervisor* which is capable of emulating hardware [23]. Using the VM along with the emulated USB host controller the **mediator** can safely redirect insecure devices to the emulated USB host controller and monitor them, in an attempt to profile them so that these devices can be identified easier in the future.

3.4.3 TMSUI

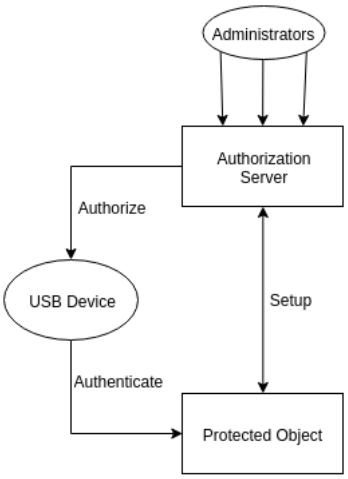

Figure 3.2: Overview of TMSUI

TMSUI, Trust Managment Scheme of USB storage devices for Industrial control systems, is a proposed scheme [25] that can be used to protect terminals connected to *industrial control systems* (ICS) from a wide variety of attacks ranging e.g.

Stuxnet, BadUSB. The system consists of an Authentication Server, one or more administrators, and protected objects (PO) e.g. terminals, all depicted in Figure 3.2.

The Authentication Server

The authentication server (AS) is a centralized server that is responsible for keeping track of a lot of the data related to protected objects and administrators.

Besides generating public parameters, it is also used by administrators to generate their authorization key consisting of an asymmetric key pair computed using an unspecified key derivation function. The key is derived using the public parameters, the master key seed generated by the *AS*, and the hash of the administrator's password. This key is then encrypted using a randomly generated key. The generated key is then encrypted using a randomly generated key based on a PRNG (Pseudo Random Number Generator) with the master key seed and hash of the administrator's password as input. The result of the encryption is referred to as the administrator's *blob*, which is saved by the *AS* until it is needed for authorization [25].

Protected Objects

Protected objects are terminals that are in need of protection since they serve a vital function e.g. they are connected to an industrial controller. All *PO* are required to have a *TCM* (Trusted Cryptography Module) chip. This chip, capable of performing cryptographic operations, stores an asymmetric key pair. The identity will then consist of a public key exported from the *TCM* along with other information about the *PO*, e.g. its function, location. This information is what will identify the *PO* in a later stage.

Authorization of a device

A device is authorized by an administrator. This process begins with the administrator plugging the device into the *AS*. The device is scanned for anomalies using e.g. antivirus software. If the anti-virus software does not come up with anything dangerous, the device is given a unique identifier derived from its e.g. manufacturer, type, usage, etc.

The identifier is searched for in the *Revocation list*, located in the *AS*. Given that the device can not be found on the *Revocation list*, the *AS* will decrypt the administrators authorization key using the storage key derived in Section 3.4.3.

The administrator will then decide which *PO* the device will be granted access to. The public key of the *PO* along with the expiration timestamp and the device's identifier are then hashed into a digest. This digest along with the public parameters of *AS* are then used as input to a function generating a signature using the administrator's private authorization key. A quadruple consisting of the expiration date, the public key of the *PO*, the ID of the authorizing admin, and the generated signature, usually denoted as σ, is then written to the USB device as a read-only and hidden document/file.

Authentication of a device

When a device is plugged in to a *PO*, the process begins with the *PO* extracting the identifier of the device along with the public key of the *PO* the device is authorized for. The *PO* will first see if the device has been revoked by the *AS*. If the device is not revoked and the expiration date has not yet passed, the validity of the signature will be computed.

The validity of the signature is computed by essentially repeating the process described earlier but rather using the administrator's public key. That is, the digest is computed using the expiration date, the public key of the administrator, and the identifier of the device. Verification is then be performed by using the signing function's corresponding verification function with input consisting of:

1. The administrator's public authorization key
2. The signature generated during authorization, σ
3. The public parameters supplied by the *AS*

If the verification is successful the device is granted access.

3.5 The hypervisor prototype

With the previously explained protection mechanisms against BadUSB in mind, a hypervisor prototype capable of protecting against BadUSB will now be designed and implemented. The only protection mechanism, that in any way seems viable and can be implemented, is to use some kind of device filtering.

Just as *GoodUSB* is capable of filtering out devices by looking at the information provided by both the device and what the user expects the device to do, it is possible to implement something similar in a hypervisor. In the case of the hypervisor, it is however not a viable option to get information from the user mainly due to trust. The goal of the hypervisor is to create a protection scheme based on separation between OS and USB. If the hypervisor design was to introduce a mechanism that break the separation, the whole core idea of the thesis would be pointless.

3.5.1 Using USB in virtualization

USB is as described earlier a kind of bus that uses memory-mapped registers to communicate with the OS and its applications. For the sake of simplicity, it will be assumed that the USB host controller in question is integrated into the motherboard i.e. it is not externally connected, even though roughly the same principle would apply.

The USB host controller does use interrupts, although it is intended for very specific purposes e.g. to communicate that a very important packet has arrived. In the general case, the USB host controller is polling-driven i.e. the USB host controller driver running in the OS checks the registers of the USB host controller periodically, to detect changes from the last polling.

The registers of the USB host controller are **memory-mapped** i.e. the USB

host controller and the software shares a part of the memory which they use to communicate. In the normal case where only the OS is running the OS would read and write to the part of the memory without any interference. In many implementations of USB host drivers in virtualization software, this is also the case. The hypervisor's USB host driver would simply let all the data pass through to one or more guests without interfering with the data [24]. These kind of drivers are referred to as **passthrough drivers**.

Since the chosen approach to protect against BadUSB is filtering of devices, it is no longer possible to simply let the data pass between the guest OS and the USB host controller freely. It is required to introduce a mechanism that can intercept data coming from the USB host controller. Using the intercepted data, it should then be possible to determine whether a device has ill intent and block access to the guest.

This can be done it two different ways. One option is to let the guest have access to the original *memory-mapped registers* of the USB host controller and to have the hypervisor simply observe what is happening. When BadUSB device is attached and detected, the hypervisor would then intervene and block the device.

The second option would be to use a variant of *nested page table*, as described in Section 2.2. Instead of letting the *memory-mapped registers* be fully visible to the guest, it is also possible to create a *shadow buffer* which would essentially be a copy of the original *memory-mapped registers*. The hypervisor would then go through all information coming from the USB host controller, filter the information, and then copy all that it thinks is appropriate to the guest's shadow buffer. With this concept in mind, the software alternatives will now be examined.

3.5.2 Choosing a software alternative

Before a hypervisor prototype capable of protecting against BadUSB can be constructed, an open-source hypervisor must be chosen as a base for the prototype since implementing a hypervisor from scratch is very time consuming. The two open-source alternatives that will be examined is **Xen**, which is to a large extent based around paravirtualization, and **BitVisor**, which uses full virtualization and uses *hardware-assisted virtualization* (Intel VT).

Xen

Xen was created as a research project at Cambridge University in the late 1990s consisting only of its hypervisor. In 2002, the Xen hypervisor became open-source and in the fall of 2003, the first public release of Xen was announced. During the first decade of the millenia, Xen gained a lot of support from e.g. Red Hat, Novell and Sun which greatly helped the improvement of Xen [16].

Xen relies heavily around paravirtualization, even though full virtualization relying on hardware-assisted virtualization is supported. As described earlier, modifications of the kernel is required in order to run paravirtualized hypervisors. For Xen, there exists modified kernels for Linux, NetBSD, FreeBSD, and OpenSolaris [17]. *Xen* has 2 modes where it relies heavily on paravirtualization. These are **PV**, which only uses paravirtualization, and **PVH**, which uses some

hardware-assistance features; more specifically hardware implementations for privileged instructions and nested page tables [17].

USB in Xen can be done in 2 different ways. A guest in Xen, referred to as *domU* (domain unprivileged), may either use the **PVUSB** interface or an emulation-based approach involving **QEMU**. In the case of PVUSB the interface consists of a front-end and a back-end interface where the back-end is required in *dom0*, the host, and the front-end is required for *domU*. The front-end and the back-end use a special protocol, specifically made for PVUSB, to commmunicate [18].

The other option, using QEMU, runs in *dom0* and exposes an emulated host controller to the guest running in *domU*. Worth mentioning is also that in order to use this approach, it is necessary to use **HVM** i.e. full virtualization using hardware-assistance e.g. Intel VT.

Even though none of the techniques involves the use of a single driver but rather a few essential components, the techniques can still be considered to be *passhrough drivers*, as previously described in 3.5.1.

Bitvisor

BitVisor is a hypervisor which was created as a part of the SecureVM project in Japan. Its purpose is to prevent information leakage from desktop/laptop PCs by applying security policies to the guest OS. The design goal was mainly to have a slim and fast hypervisor which could apply security policies to the guest OS without loosing too much performance.

BitVisor is implemented to use full virtualization using hardware-assisted virtualization through Intel VT for the x86 platform. It relies heavily on these features for e.g. execution of privileged instructions as described in Section 2.2.2. One very important aspect of the implementation relates to how memory management is handled.

Since BitVisor is designed to, among other features, be able to enforce encryption on hard drives and on USB flash drives it does, unlike many other hypervisors, have a **para-passthrough driver**. A *para-passthrough driver* is a USB host controller driver that is capable of keeping track of the state of the USB host controller and its attached devices, and when necessary, intervene. This is implemented using the concept of a **shadow buffer**, just as described in Section 3.5.1.

From what was explained in Section 3.5.1 it is more than reasonable to consider BitVisor as the best candidate since it already has some of the necessary mechanisms required to perform device filtering, as previously suggested. It was mainly because of this that BitVisor was chosen as the base to build the prototype from.

3.5.3 Design specification

With the BitVisor hypervisor as a base, the hypervisor prototype can now be implemented. In order to understand the prototype design, a few concepts of BitVisor has to be explained.

BitVisor uses a *para-passthrough* USB host controller driver. The driver will

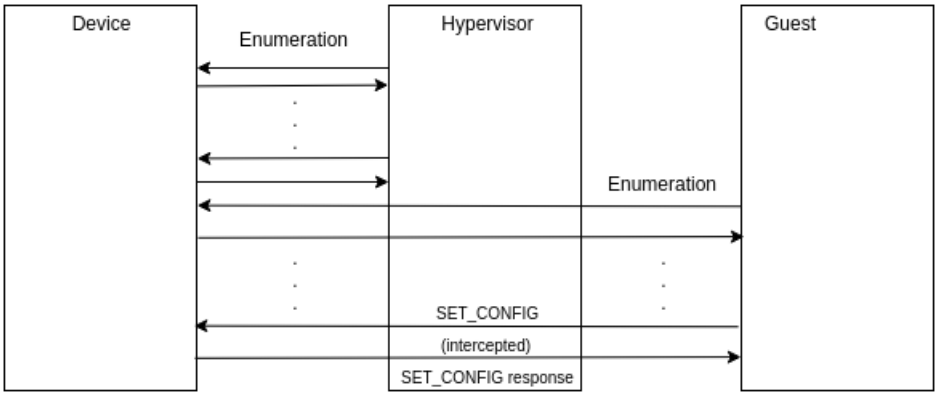

Figure 3.3: Hypervisor's operational model

keep track of the state of attached USB devices and intercept the data sent between the device and the OS. During the enumeration process, the hypervisor will gather information about the device. It will look at all of the descriptors sent (Device Descriptors, Interface Descriptors, etc.) and use data structures to create a representation of a device and its capabilities.

As decided in the previous section, implementing a device filter is probably the best solution. The most simple filter would only filter to the class given in the *Device Descriptor* which will be set to **0x00**. This is however not sufficient since this filter is far too simple and would result in too many false positives. The other option is to filter on the combination on interfaces, more specifically the combination of a HID interface and a Mass Storage interface. If the hypervisor is to be modified so that it can detect the disallowed combination, it is necessary to understand how all of the data structures relate.

The 4 types of descriptors, **Device Descriptors**, **Configuration Descriptors**, **Interface Descriptors**, and **Endpoint Descriptors**, are all stored in the hypervisor in the form of a data structure, referred to as *usb_device*, which describes the device.

```
struct usb_device {
  struct usb_device *next, *prev;

  struct usb_bus *bus;

  struct usb_device_descriptor descriptor;
  struct usb_config_descriptor *config;

  void *dev;    /* unused */

  spinlock_t lock_dev; /* device address lock */

  u8 devnum;

  /* driver internal use */
  u64 portno;
  struct usb_device *parent;  /* for HUB cascade */
  struct usb_host *host;
  u8 bStatus;

  /* specific device handler if registered */
  struct usb_device_handle *handle;

  struct uhci_hook *hook; /* FIXME */
  int hooknum;
  u8 speed;

  struct usb_config_descriptor cdesc;
  size_t l_ddesc, l_cdesc;
  u8 serial[256];
  u8 serial_len;
  u8 ignore;
};
```

The *usb_device* data structure contains a *device descriptor* and a *configuration descriptor* which are allocated directly into the data structure. The *interface descriptor(s)* are however allocated separately and pointed to by a pointer in the *configuration descriptor*. When a new *interface descriptor* is to be allocated, a variant of *realloc*, which reallocates a memory segment into a new memory segment of a different size, is used to make room for the new *interface descriptor*. What this means for the implementation is that the *interface descriptors* will be written in memory in a sequence, thus making it easy to read all *interface descriptors* by simply reading the number of interfaces used by the configuration and casting pointer(s), incrementing the size of an *interface descriptor* for each pointer cast.

The *interface descriptors* held by the hypervisor have their information filled in by the hypervisor performing its own enumeration process, as shown in Figure 3.3. This process begins with allocation of all data structures and then performed

in the same manner as described in Section 2.1.3.

With all of the gathered information, it is now possible do determine whether the new device is potentially malicious. Once a disallowed device has been found, it must be blocked. The blocking itself is implemented by adding an extra field to the *usb_device* data structure called *ignore*. When a device is determined to be a potential BadUSB device, this field is set to non-zero which in turn will trigger the blocking mechanism. The blocking mechanism involves blocking all actions that the device may perform. This is implemented using 2 lines of code which checks the value of the *ignore* field and, if it is non-zero, the function call is not allowed to complete. One example of this is the *usb_control_msg*, as shown below. Certain parts, e.g. related to UHCI and debug printing, have been left out. Example:

```
static inline int _usb_control_msg( ... )
{
  struct usb_request_block *urb;
  struct usb_ctrl_setup csetup;
  u64 start;
  int ret;

  start = get_time();

  csetup.bRequestType = requesttype;
  csetup.bRequest = request;
  csetup.wValue = value;
  csetup.wIndex = index;
  csetup.wLength = size;

  /* If the ignore bit is set we ignore the device. */
  if (dev->device->ignore)
    return -1;

  urb = dev->host->op->
    submit_control(dev->host, dev->device, ep, pktsz,
              &csetup, NULL, NULL, 0 /* no IOC */);
  if (!urb)
    return -1;

  ret = _usb_async_receive(dev, urb, bytes, size, start,
      timeout * 1000);

  dev->host->op->deactivate_urb(dev->host, urb);

  return ret;
}
```

The method shown above is used for injecting a control transaction into the asynchronous list. This method is mainly used during enumeration. The lines below are similarly added to all other methods used by the hypervisor's USB host con-

troller driver, which in turn results in the blocking of the device when the *ignore* field is set to non-zero, as later seen in Section 4.1.

```
if (dev->device->ignore)
  return -1;
```

3.6 Testing the hypervisor prototype

In order to be confident in that the hypervisor actually protects the guest OS as was first theorized, a USB device that can perform the BadUSB attack is required. To construct such a device, requires a USB flash drive that uses a specific chipset called **Phison PS2251-03**. The reason for this is not because the chipset has any particular vulnerability but rather that there already exists software [19] that can be used for reprogramming this particular chipset and making it perform the attack.

3.6.1 Building an attacking USB flash drive

Finding a vulnerable device with the specific chipset Phison PS2251-03 proved more difficult than first though. A few devices were bought and tested. Of all the devices listed as vulnerable only a few of them were available for purchase in Sweden. The devices that were bought were *SanDisk Ultra 16Gb*, *DataTraveler G4 8Gb*, and *DataTraveler G4 16Gb*. 0,The chipset of the devices' firmware were checked using *ChipEasy*, a free program that can read a variety of information about USB devices and display it in a convenient manner. By using this program, it was concluded that none of the devices available had the correct chipset.

The list of vulnerable devices was found on an information page in the GitHub repository where the code for the attack is available [19]. Other users have also had the same issue of finding vulnerable devices. To summarize the reports/discussions, it seems as if most of the manufacturers have turned their back on the PS2251-03 chipset and chosen to use PS2251-07 instead. Whether this is an easy way to mitigate the potential damage the old chipset could do to their customers or a way for the manufacturers to reduce costs, if it happens to be so that the PS2251-07 chipset is cheaper, is hard to know.

Naturally, more products from a wider variety of sellers could have been bought but judging from the discussions/reports on GitHub regarding other purchases it did not seem worth the time and resources to pursue further. Even though no vulnerable USB flash drive could be found, it is worth briefly mentioning how reflashing a vulnerable device would be done:

1. Extract the firmware from the USB flash drive and inject the payload into the firmware. Another alternative is to compile the reverse-engineered firmware.

2. In order to reflash, the USB flash drive has to be in boot-mode. This can be done either by sending commands to the chip or by using short-circuiting pin 2 and 3 on the USB chip. The second method is less convenient since it

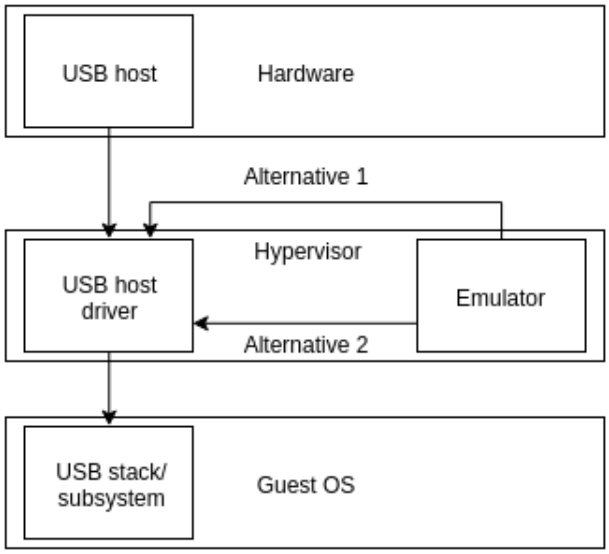

Figure 3.4: Emulation vs. using a real device

would require the attacker to open the USB flash drive and seal it again in order to perform the attack undetected.

3. When the USB flash drive is in boot-mode, a burner image is sent to the chip. This image is loaded into memory and the chip awaits a new firmware image.
4. The reprogrammed firmware image is sent to the chip and the chip is re-flashed. The chip is then restarted by resetting the USB port.
5. The USB flash drive will now perform the USB-attack every time it is plugged in into a computer.

The burner image was found on a Russian website specializing in reprogramming USB controller chips and the software for flashing, including the firmware, is available at GitHub [19].

Since it proved to be harder than expected to create a USB device capable of a BadUSB attack, emulation was to be used instead so that the hypervisor prototype could be tested.

3.6.2 Emulation of the attack

Before looking into specifics on how the BadUSB attack would be implemented, it is first necessary to establish the core components necessary to make the emulation work. As illustrated in Figure 3.4, there are basically 2 alternative approaches that can be implemented with the major difference being whether the result of the test can be seen in the guest.

Both solutions are based around modyifing the asynchronous queue managed by the *para-passthrough driver* running as a part of the hypervisor. The

synchronous queue, as previously described in Section 2.1.4, will contain **Queue Head(s)** which in turn link to one or more **Queue Element Transfer Descriptors** (qTD), both previously described in Section 2.1.4. The core idea behind both emulation designs is to allocate and inject a QH along with a few qTDs.

In the first alternative, the goal with this is to make the guest believe that a new device has been attached. Once the guest believes there is a new device, it will begin enumeration and from there the emulator simply has to intercept and inject the response data.

The second alternative is more oriented towards restricting the emulation to only include the hypervisor itself i.e. the guest is never aware that the attack is ongoing. The biggest consequence of this approach is that it is harder to make a reference test since the results are not obviously visible in the guest.

Since the overall goal of the thesis is to test whether the hypervisor through filtering can actually stop the attack, implementing a full emulation of the entire attack was chosen. How this was done will now be detailed.

The initialisation (starting) of the emulation is implemented as a function inside the hypervisor which can be accessed through a hypercall, a call directly to the hypervisor, coming from the guest. Hypercalls are already implemented in the hypervisor and is used mainly for debugging. By adding a new hypervisor call, it is possible to start the emulation from the hypervisor using a simple program that runs from the guest.

The emulation process begins with the creation of a **USB Request Block** (URB), a data structure used by the EHCI USB host controller driver to represent transactions. From there, a QH is allocated along with a **SETUP qTD** and an **OUT qTD**. Both of these *qTDs* are to be used for placing a (fake) *SET_ADDRESS* request within the asynchronous queue. The thought behind this is use the *hook process* implemented in the hypervisor which looks at incoming and outgoing data before it is passed to or from the guest. When the fake *SET_ADDRESS* request is found, the function for enumeration of a new device is called in the hypervisor. With this in combination with installing a custom-made hook that only looks at the address of the emulated device, it is possible to inject data before anything is passed to the guest. This way, it will seem to the guest as if there is an actual device present and it will be handled in a normal manner.

Chapter 4
Results

4.1 The hypervisor prototype

Implementation of the hypervisor prototype was one of the easiest parts of the entire thesis. Before implementing the interface filter, another filter was implemented to test the blocking mechanism.

The filter first implemented involved blocking *Mass Storage Devices* i.e. USB devices reporting their device class code as **0x08**. A small test was then conducted by simply inserting a USB flash drive into the computer running the hypervisor prototype along with a guest running Linux. The goals were to examine whether the blocking mechanism worked properly i.e. the device was inaccessible for the OS, and that the implementation did not affect performance too much. The rate of performance inhibition was tested in the most simple manner available, by performing simple tasks e.g. opening a web browser, etc. This basic test was performed both before, during, and after the USB flash drive had been plugged in. From what could be gathered from just observing the response time of the guest OS, the implementation's effect on the guest OS's performance seemed to be minimal, thus the conclusion could be drawn that the blocking mechanism could be used.

4.1.1 Emulation of a BadUSB attack

The emulation part of the implementation was probably one of the most difficult parts of the entire thesis, mainly due to it not being expected from the beginning.

As stated in the section discussing the design of the emulation, Section 3.6.2, the chosen alternative was to implement the attack to propagate all the way to the guest. Building the response handler proved quite easy since all the response data could be found in the GitHub repository [19]. There was also the issue of assigning the device an address number which would already be taken. The solution to this was to simply assign a static address number to the device. By observing the numbers allocated by the guest's USB stack, it was set to 8 since the numbers are allocated by looking at the first free device number in sequence starting from 1 (since 0 is used for the *SET_ADDRESS* request).

Even though some of the obstacles were overcome, the emulation was never completed to a state where it would work properly. The issues were mostly related

to the inner workings of the hypervisor not being completely understood, which can be attributed to this part of the thesis being unexpected and also a bit too complicated to finish.

One issue with the chosen approach was the synchronization of the guest and the hypervisor. Since the goal was to make the attack be performed in the guest, the approach of injecting a *SET_ADDRESS* request into the hypervisor proved to not be sufficient. In the regular case of a physical device being attached, it would be the guest that assigns an address to the device and not the hypervisor, thus the guest would never recognise that the device is connected.

This realisation made the implementation far more complicated. If the original goal was to be met, with the attack propagating all the way to the guest, the guest would have to be the initiator of the emulation. From the beginning, the USB subsystem of the Linux kernel was examined to investigate whether this maybe could be solved from userspace. The conclusion was that in order for the emulation to be started from the guest it would be necessary to modify the Linux kernel. Understanding the kernel to a point where it can be modified to the extent necessary, so that emulation can be performed, was considered out-of-scope for this project. Because of this, the idea of emulating the attack was abandoned. In hindsight, it would have been better to just implement the other approach, where the emulation is restricted to only take place inside the hypervisor. However, due to this major setback, the comparison of solutions will be strictly theoretical.

4.2 Comparison of solutions

In order to compare the alternative solutions to the hypervisor solution, a definition of what makes a solution good is first needed. Naturally this is no easy task since the intended user of the solution may have different requirements and also prioritizes these requirements in different ways. Because of this, a few requirements that surely will be prioritized differently depending on the user have been chosen. These are:

1. Human interaction - Does the solution require the end-user to interact in some way?
2. Ease of use - How easy is the solution to use for the end-user?
3. Level of confidence - How often does the solution give false negatives/false positives? If there is human interaction, what is the risk of human error?
4. Integrity of the firmware - Can the integrity of the firmware be trusted?

With these criteria in mind, the hypervisor prototype along with the other described alternatives from Section 3.4 will be examined.

4.2.1 Udev

Filtering by configuring Udev is a reasonably simple method that, at least for some BadUSB attacks, provide the OS with a certain degree of protection. The Udev filtering method does not require any human interaction for the end-user. Likewise,

deploying the solution to an organization, e.g. a large company, is fairly simple. Regarding the level of confidence in the solution, and filtering in general, it relies heavily on the reported information. In this specific case of the BadUSB attack there is a clear anomaly, since there are 2 odd interfaces, that can be observed only from the information provided by the device. This is however not the general case for BadUSB. One example of a BadUSB attack that cannot be filtered is the case where a USB flash drive is changed to only act as a keyboard with one single (HID) interface. This is mainly because a single HID interface consisting of a keyboard is very common which means that there is no real anomaly to filter on. Another issue is that there is no check for the integrity of the firmware. Altogether it is however a quite effective method especially considering how simple it is to deploy and use.

4.2.2 GoodUSB

Overall, the GoodUSB solution matches most of the criteria described at the beginning of this section. The interface shown to the user is fairly simple to grasp and the risk of an attack getting through undetected is quite low. Since the solution uses human interaction to address the problem, the solution can be considered to address the integrity of the firmware somewhat, with the human interaction being the only drawback. This enables e.g. a corrupt employee to knowingly let the device perform its attack by giving it the permission to do so. The risk of such an attack being successful can be considered fairly high, especially if the attack involves a USB device pretending to be another kind of device e.g. a USB flash drive pretending to be a USB keyboard. Besides from that, there is also a concern that the *profiles* used by the software to identify USB devices might sometimes be a bit too restrictive and block legit devices i.e. it might produce false-positives.

4.2.3 TMSUI

Even though the TMSUI scheme was designed with the ICS systems in mind, it can still be considered for use in a regular environment e.g. within a company. The TMSUI scheme also fulfils most of the criteria defined earlier. The scheme is essentially based around manually controlling and detecting a compromised device before allowing it into the system. This can in some sense be considered to establish some trust in the firmware. Also, even though the act of granting access is manual, the authentication of a device is automated. Because of this, the solution can be deemed to be *semi-automated*. Additionally, the solution might also be considered to be quite simple to use, with the only real hassle being that administrators have to authorize the devices before use.

The important question regarding this method is whether all allowed devices can be trusted after they have been granted access. When the device is plugged into the *AS*, it is quite clear that the device is not compromised but after this process it is quite impossible to know where the device has been. From what can be gathered from the description of the TMSUI, there exists no mechanism that properly addresses this. This can naturally be addressed by the organization using the scheme by having special security protocol that e.g. a device cannot leave the

building until it has been plugged into the *PO* it was approved for, or something similar.

Another troubling aspect is the use of a *TCM*. Even though the use of a *TCM* is a good choice security-wise, it has some drawbacks regarding the costs. Naturally, the **Trusted Platform** approach would be better but it is not a necessity for the scheme to function properly.

4.2.4 The hypervisor prototype

Since it was never possible to construct a BadUSB device, nor implement emulation of an attack, it is still uncertain whether the hypervisor prototype actually provides protection against the BadUSB keyboard emulation attack as well as BadUSB attacks in general.

Similarly to the Udev solution, the hypervisor prototype uses filtering to protect against BadUSB. Just as with Udev there is a difficulty in filtering out devices that do not send information containing a clear anomaly. Regarding the ease of use, the hypervisor prototype is fairly simple to deploy in a larger company and requires little to none interaction from the user. The biggest drawback is probably performance-wise. Even though no real comparison was made, it is fair to assume that the hypervisor prototype has a larger impact on performance since it involves a lot more complex operations.

Chapter 5
Conclusions and Future Work

Even though the hypervisor prototype should, in theory, provide the guest OS with some protection against the specific attack as described in Section 3, the problem persists. It was never tested, but it is fair to assume that the hypervisor itself does not provide any protection against other attacks since the filtering is too specific.

None of the solutions presented in the previous chapter was able to fulfil all of the criteria, even though some were very close. From what could be gathered the most realistic solution in a real life scenario would probably be *GoodUSB* since it can protect against many kind of attacks, at least attacks that could be thought of for now.

However, if the evaluation should be very strict, i.e. any unfulfilled criteria is deemed as insecure, none of the solutions are satisfactory. This is mainly due to the fact that the core issue, that of the integrity and authenticity of the device's firmware, is never sufficiently addressed.

This leaves one difficult question unanswered: How would a system with perfect security against BadUSB attacks be designed? As previously, mentioned the core issue of the BadUSB attack is that there is no way of knowing whether it is possible to trust the information coming from the device, thus making the BadUSB attack very hard to detect. One very simple, yet effective, solution is to simply disable firmware reflashing to all USB devices. This does however contradict the whole idea of having firmware updates for e.g. recovering a damaged device or to update the firmware due to any other reason.

Another interesting approach would be to introduce some form of firmware signing scheme. This does however require significant additions/modifications to the USB devices that are to be used along with modification of computers/terminals e.g. in the form of special drivers.

5.1 Suggestion for Future Work

BadUSB is an attack that still has a lot of potential to do damage, especially if someone were to invest time and resources into developing an exploit based around it. In the future it would be very interesting to investigate design and implementing a software signing scheme for USB devices. In this section a brief description of a signing scheme that might address the problem will be detailed. The setup is based around storing a public-private key pair on the USB device.

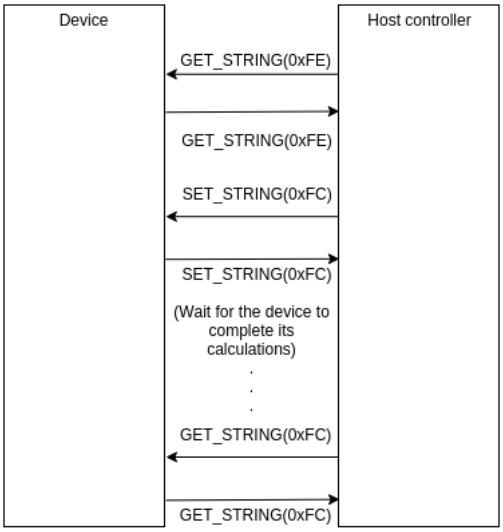

Figure 5.1: Suggestion for a verification scheme for USB devices

The host controller will require a special driver in order to be able to handle the device, since new steps are added to enumeration. The extra steps involves first sending a *GET_STRING* of a custom index containing the public key of the device. The host will generate a short string which is either encrypted or unencrypted. This is sent to the device by sending a *SET_STRING* request with another custom-index, containing the generated string. The string is then signed or encrypted, depending on which approach is chosen. The host may then retrieve the plaintext/signature by sending another *GET_STRING* with the same index as the earlier *SET_STRING* request. Using the contents of this request the host can then finally determine if the signature/plaintext is valid. This process is also described in Figure 5.1.

References

[1] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Phillips *Universial Serial Bus Specification rev 2.0*, April 27, 2000

[2] Robert Murphy, *USB 101: An introduction To the Universal Serial Bus 2.0*, n.d, [Online], Available: `http://www.cypress.com/file/134171/download`

[3] Howard, John S, et al. *Enhanced Host Controller Interface Specification for Universal Serial Bus*, *March 12, 2002*, [Online], Available: `http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/ehci-specification-for-usb.pdf` [Accessed: Jan. 4, 2016]

[4] Howard, John S, et al. *Enhanced Host Controller Interface Specification for Universal Serial Bus*, *March 12, 2002*, [Online], Available: `http://www.intel.com/content/dam/www/public/us/en/documents/technical-specifications/ehci-specification-for-usb.pdf` [Accessed: Jan. 4, 2016]

[5] Adams, Keith, and Ole Agesen. "A comparison of software and hardware techniques for x86 virtualization." ACM Sigplan Notices 41.11 (2006): 2-13.

[6] Robert Murphy, *Understanding Full Virtualization Paravirutalization, and Hardware Assist*, n.d, [Online], Available: `https://www.vmware.com/files/pdf/VMware_paravirtualization.pdf`

[7] Johan De Gelas, *Hardware Virtualization: the Nuts and Bolts*, March 17, 2008, [Online], Available: `http://www.anandtech.com/show/2480/10`

[8] Uhlig, Rich, et al. *Intel virtualization technology* Computer 38.5 (2005): 48-56.

[9] Barham, Paul, et al. *Xen and the art of virtualization.* ACM SIGOPS Operating Systems Review 37.5 (2003): 164-177

[10] Thomas, Vinoo, Prashanth Ramagopal, and Rahul Mohandas. *The rise of autorun-based malware.* McAfee Avert Labs., McAfee Inc (2009).

[11] Falliere, Nicolas, Liam O. Murchu, and Eric Chien. *W32. stuxnet dossier.* White paper, Symantec Corp., Security Response 5 (2011), p. 6

[12] Farwell, James P., and Rafal Rohozinski. *Stuxnet and the future of cyber war.*, Survival, vol.53. no 1, pp.23-40, 2011

[13] Richard Harman, *Controlling USB Flash Drives: Expose of Hidden Features, 2014*, [Online] Available: `https://www.youtube.com/watch?v=7RqANUhVnl4` [Accessed: Jan. 27, 2016]

[14] Nohl. Karsten,Krißler. Sacha,Lell. Jakob, *BadUSB - On accessories that turn evil*, `https://srlabs.de/blog/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf`

[15] Unknown author, *Why The Xen Project?, n.d.*, [Online] Available `http://www.xenproject.org/users/why-the-xen-project.html` [Accessed: Feb. 19, 2016]

[16] Unknown author, *Xen Project HIstory, n.d.*, [Online] Available: `http://www.xenproject.org/about/history.html` [Accessed: Jan. 5, 2016]

[17] Unknown author, *Xen Project Software Overview, n.d.*, [Online] Available: `http://wiki.xen.org/wiki/Xen_Project_Software_Overview` [Accessed: Jan. 4, 2016]

[18] Unknown author, *Xen USB Passthrough, n.d.*, [Online] Available `http://wiki.xenproject.org/wiki/Xen_USB_Passthrough` [Accessed: Feb. 19, 2016]

[19] Caudill, Adam, *BadUSB source, n.d.*, [Online] Available: `https://github.com/adamcaudill/Psychson` [Accessed: Jan. 4, 2016]

[20] Unknown author, *How to prevent BadUSB attacks on Linux desktop?, Aug. 9, 2014*, [Online] Available: `http://security.stackexchange.com/questions/64524/how-to-prevent-badusb-attacks-on-linux-desktop` [Accessed: Feb. 22, 2016]

[21] Tian, Dave, Bates, Adam, Butler, Kevin *Defending Against Malicious USB Firmware with GoodUSB*, Proceedings of the 31st Annual Computer Security Applications Conference, Pages 261-270. ACM, 2015.

[22] Unknown author, *Kernel Virtual Machine, n.d.*, [Online] Available: `http://www.linux-kvm.org/page/Main_Page` [Accessed: Feb. 22, 2016]

[23] Unknown author, *QEMU/Devices, n.d.*, [Online] Available: `https://en.wikibooks.org/wiki/QEMU/Devices` [Accessed: Feb. 22, 2016]

[24] Abramson, Darren, et al. *Intel Virtualization Technology for Directed I/O* Intel technology journal 10.3 (2006).

[25] B. Yang, D. Feng, Y. Qin, Y. Zhang, and W. Wang. *TMSUI: A Trust Management Scheme of USB Storage Devices for Industrial Control Systems.* Cryptology ePrint Archive, Report 2015/022, 2015

[26] Shinagawa, Takahiro, et al. *Bitvisor: a thin hypervisor for enforcing i/o device security."* Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. ACM, 2009.

Appendix A

Appendix

A.1 USB PID Types

Table A.1: USB packet types

Type	Name	Description
Data	DATA0	Even-numbered data packet.
Data	DATA1	Odd-numbered data packet.
Data	DATA2	Used when doing high-bandwidth data transfers (USB 2.0+).
Data	MDATA	Used when doing high-bandwidth data transfers (USB 2.0+).
Handshake	ERR	Split transaction error (USB 2.0+).
Handshake	ACK	Data packet accepted.
Handshake	NAK	Retransmit, packet not accepted.
Handshake	NYET	Data is not ready yet.
Handshake	STALL	Fatal error, transfer is impossible. Begin error recovery.
Special	PRE	Low bandwidth preamble.
Token	IN	Used by the host to prepare the device for transmitting packets using a DATAx packet.
Token	OUT	Used by the host to prepare the device it for receiving packets using a DATAx packet.
Token	PING	Check if device can accept data (USB 2.0+).
Token	SETUP	Address for host-to-device control transfer
Token	SOF	Start of frame marker which is sent every ms.
Token	SPLIT	High-bandwidth split transaction, introduced in USB 2.0+.

A.2 USB Descriptors

Table A.2: Device Descriptor

Field	Value	Description
bLength	0x12	The length of the descriptor in bytes.
bDescriptorType	0x01	Set to *0x01*.
bcdUSB	0x0002	USB version
bDeviceClass	0x00	Device class
bDeviceSubClass	0x00	Device subclass
bDeviceProtocol	0x00	Protocol code.
bMaxPacketSize	0x40	Maximum packet size
idVendor	0xFE13	2 bytes describing the device's vendor.
idProduct	0x0152	2 bytes describing the vendor's device.
bcdDevice	0x1001	Release number of the device.
iManufacturer	0x0000	Index of the manufacturer's string descriptor.
iProduct	0x00	Index of the product's string descriptor.
iSerialNumber	0x00	Index of device's serial number.
bNumConfigurations	0x01	No. possible configurations.

Table A.3: Configuration Descriptor

Field	Value	Description
bLength	0x09	The length of the descriptor in bytes.
bDescriptorType	Constant	0x02 for Configuration Descriptor.
wTotalLength	Number	Total length of the data returned.
bNumInterfaces	Number	Number of interfaces used by the configuration.
bConfigurationValue	Number	What number that should be sent in **wValue** as part of a *SET_ CONFIGURATION*-request.
iConfiguration	Index	Index of the configuration's string descriptor.
bmAttributes	Bitmap	Bit 7 - Must be set to 1 Bit 6 - Is self-powered Bit 5 - Supports remote wake-up Bit 4:0 - Must be set to 0
bMaxPower	Number	Maximum power consumption expressed in 2 mA units.

Table A.4: Interface Descriptor

Field	Value	Description
bLength	0x09	The length of the descriptor in bytes.
bDescriptorType	Constant	0x04 for Interface Descriptor.
bInterfaceNumber	Number	The interface's number/index.
bAlternateSetting	Number	Value for alternate setting.
bNumEndpoints	Number	The number of endpoints used by this interface.
bInterfaceClass	Class	Class code of the interface.
bInterfaceSubClass	SubClass	Subclass code of the interface.
bInterfaceProtocol	Protocol	Protocol code of the interface.
iInterface	Index	Index of the interface's String Descriptor.

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-486

http://www.eit.lth.se