Diploma Thesis

Evaluating Security Aspects of the Universal Serial Bus

Moritz Jodeit

Department of Informatics
University of Hamburg, Germany
http://www.informatik.uni–hamburg.de

First reader: Prof. Dr. rer. nat. Joachim Posegga
Second reader: Prof. Dr. Dieter Gollmann

December 16, 2008

Abstract

The widely–used Universal Serial Bus provides a physical attack vector, which has only been considered in the past sparingly. Not only the lack of public research, but also the lack of suitable tools makes it really hard to assess the security of provided USB ports. We first break down the different attacks against the USB architecture into different categories. After describing some theoretical attacks, we present our implementation of a USB fuzzer. To demonstrate its effectiveness of finding new vulnerabilities in USB stacks and device drivers, we use it to fuzz test the USB mass storage class driver of the various operating systems. The vulnerabilities we find support our claim that the USB architecture provides a real attack vector and should be considered when assessing the physical security of computer systems in the future.

Acknowledgements

This work would not have been possible without the help of many others. Special thanks to Martin Johns for the guidance and the initial idea to work on this subject. I would also like to thank Jeremias Reith for bouncing ideas back and forth when things got a bit stuck. Finally, I would like to thank Miriam Tenten for her patience and tolerance while working on this thesis paper, which helped me quite a bit in staying focused.

I would like to dedicate this thesis paper to my family.

Contents

Chapter 1

Introduction

The Universal Serial Bus (USB) is a widely–used serial cable bus for connecting different peripherals to a host computer. With the introduction of the Certified Wireless USB (CWUSB) extension [1], USB can even be used on the wireless link. The scenario of a malicious USB device has not really been considered until now[1]. Additionally, it is currently really difficult to assess the security of all components participating in the USB implementation of the host. This is particularly true for the USB stacks and device drivers.

One possibility for a first evaluation is the development of a USB fuzzer. A device such as this would mostly conform to the USB specification, but deviate from it in different places, trying to trigger bugs in the host's implementation of the USB protocol. Additionally, such a device could act in conformance to the USB specification, but violate the assumptions of the host about usual USB devices. This could be used to find vulnerabilities, which are based on deviation from wrong assumptions. A possible difficulty with such an approach is the actual design of the malicious USB device. Since most of the available USB chips are probably trying to prevent the user from violating the USB specification, a special–purpose hardware design could be required.

The largest potential for possible vulnerabilities inside host implementations of the USB protocol, might be offered by USB device drivers. The security at the application–layer gets better and better over time. Vulnerabilities like simple stack–based buffer overflows and format string bugs are becoming less frequent. The availability of different fuzzing tools and frameworks decreases the time it takes, to find new implementation vulnerabilities even more. Device drivers on the other hand, are often developed by third–party companies and could vary largely in quality. The design goal of maximum performance combined with the fact, that device drivers are often developed under strict time frames, could easily result in security aspects being neglected.

Furthermore, a vulnerability inside a device driver has much higher consequences than a vulnerability inside some application running in user–mode, since most device drivers are running in kernel–context. Any exploited vulnerability inside such a device driver could allow an attacker to execute his code in the context of the kernel and thus gain full control over the attacked system.

[1] A notable exception being the presentation "Plug and Root" by Darrin Barrall and David Dewey from SPI Dynamics at the Black Hat USA conference in 2005.

1.1 Overview

This thesis is a presentation of different security aspects of the Universal Serial Bus. We provide an initial overview of the USB support in some of the major operating systems and describe in detail, how new USB devices are enumerated and how device drivers are loaded. Subsequently, we show some real world scenarios, how attacks might be accomplished by an attacker. After motivating the feasibility of attacks using the Universal Serial Bus, we will develop a classification of possible attacks and demonstrate some related attacks for each category. We will present our implementation of a USB fuzzer and investigate the effectiveness of our approach by fuzz–testing a USB device driver from different operating systems. We'll conclude with a list of our findings and some of the limitations of our approach.

1.2 Organization

We begin Chapter 2 with the introduction of some preliminary concepts, required for the understanding of the remaining thesis. It covers an overview of the USB protocol, followed by an introduction to the most common classic software vulnerabilities. Finally, the concept of fuzzing is explained. Chapter 3 provides an overview of the USB support in some of the major operating systems. Chapter 4 then shows different real–world attack scenarios, and how the Universal Serial Bus might be used for attacks. Following those scenarios, we classify attacks against the USB architecture in four different classes. Some possible attacks are described for each class. In Chapter 5, we present our implementation of a USB fuzzer to demonstrate some of the afore mentioned attacks. We conclude the chapter with some thoughts about how to build a malicious USB device in hardware. Chapter 6 lists some results found by fuzzing the USB stack of different operating systems. In Chapter 8, we list some limitations of our current implementation and show areas, where further research is needed. We conclude this thesis in Chapter 7.

Chapter 2

Technical Background

This chapter provides the technical background information necessary to understand the remaining thesis. First, Section 2.1 provides a detailed overview of the USB standard. Starting with the USB architecture, all the remaining main concepts of the USB standard relevant to this thesis are described. Section 2.2 then introduces the most common classic software vulnerability classes, used by attackers to compromise systems. After introducing stack–based buffer overflows, we delve in the internals of heaps and explain heap–based buffer overflows and double free vulnerabilities. Integer–related bugs and format string vulnerabilities are covered at the end. Finally, Section 2.3 introduces the concept of fuzzing, used throughout the thesis to find most of the low–level vulnerabilities.

Even though this chapter provides the foundation for the remaining thesis, the well–versed reader may of course skip this chapter in parts or entirely.

2.1 Universal Serial Bus

The Universal Serial Bus (USB) is a serial cable bus for the connection of a wide range of peripherals to a host computer. It can connect devices such as mice, keyboards, printers or flash drives. USB is intended to replace many of the serial and parallel legacy buses such as RS–232 or PS/2. One of the design goals of the USB architecture was to provide a standard bus interface, which could be shared by different kinds of devices. USB allows the attachment and removal of devices without rebooting the computer, also known as *hot swapping*. Power can be provided by the host to low–consumption devices without the need for an external power supply.

USB is standardized by the USB Implementers Forum (USB–IF). This industry standards body includes leading companies such as Hewlett–Packard, Intel, LSI, NEC and Microsoft. The latest USB specification as of this writing is the USB 3.0 specification [2], which was released at the end of 2008. Since the release of the 3.0 specification was just a few days before the deadline for this thesis, this thesis is based on version 2.0 of the specification [3]. Despite this fact, lots of the core concepts of the Universal Serial Bus haven't changed with USB 3.0, so most concepts introduced by version 2.0 are still valid for version 3.0. If not otherwise noted, we will always refer to version 2.0 of the USB specification in the following.

The USB specification defines three different speed modes to fulfill the needs of different classes

of devices. The supported speed modes are low–speed, full–speed and high–speed mode with transfer rates of 1.5 Mb/s, 12 Mb/s and 480 Mb/s respectively. The low–speed mode is mainly used for interactive devices such as mice or keyboards, while the full–speed and high–speed mode is mostly used for isochronous data transfers, implemented by audio or video devices.

The following sections will go into more detail about the USB specification. Note, that the discussion will stay focused on the logical software layer, because this is what we are mostly examining in this thesis. Mechanical and electrical USB details will be mostly skipped. Interested readers may refer to Chapters 6 and 7 of the USB specification [3] for more details.

2.1.1 Architecture

The USB system can be divided into three separate parts. These are the USB devices, the USB host and the USB interconnect, which connects all USB devices with a single USB host.

USB devices are either *hubs* or *functions*. A USB hub is a special device that provides one or more attachment points to the bus, while a function provides a specific capability, such as a mouse or an external hard disk drive. Each USB device has a unique device address, which is assigned by the host on attachment time. Before a unique address is assigned, devices have a default address of zero. Multiple functions can be combined together with a hub into a single physical package. This is called a *compound device*. All contained functions, including the hub itself, have separate device addresses. Related USB devices are grouped into device classes, which are described in Chapter 2.1.7.

The USB host is the central point in the USB architecture. It interacts through the host controller with the rest of the USB system. Only a single USB host per bus is allowed. Tasks of the host include the management of all transfers, detection of device attachment and removal and configuration of new devices. Additionally, it may supply power to attached devices. All transfers are initiated by the USB host. The USB host includes a *root hub*, which provides one or more attachment points.

The physical USB interconnect is organized as a tiered star topology, as shown in Figure 2.1 At the top of the topology is the USB host. All other devices are connected to the host through the root hub. In this case, two functions and another hub are connected directly to the root hub. All other functions and hubs are connected to the hub at tier 2. Hubs form the stars in the topology and can be used to increase the number of attachment points for additional devices. Functions can be connected directly at the root hub or any other hub. Due to physical constraints, the maximum number of tiers is limited to seven.

2.1.2 Communication Flow

This section introduces some of the main concepts of the USB communication flow. Figure 2.2 shows the logical connection of a USB device connected to a host. Each component will be described below in detail.

Endpoints

Each USB device can have several *endpoints*. The number and types of endpoints depends on the purpose of the device. An endpoint is a source or sink of a communication flow on the bus. Each

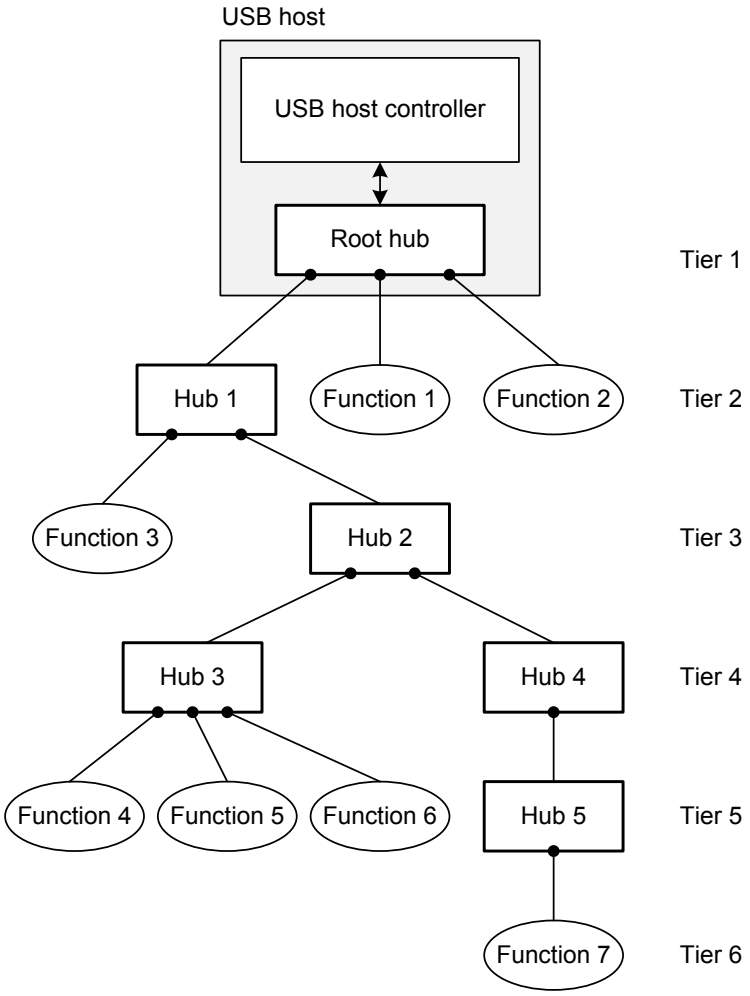

Figure 2.1: Universal Serial Bus topology

endpoint has a four–bit *endpoint number* and an associated *endpoint direction*. The endpoint direction can be either IN or OUT and is seen from the perspective of the host. An IN endpoint is used to transfers data from the device to the host, while an OUT endpoint is used by a device to receive data from the host. The endpoint direction together with the endpoint number are called the *endpoint address*. Figure 2.2 shows seven endpoints abbreviated as EP. The direction of the endpoint is specified in parentheses.

Pipes

Endpoints are connected to software on the host using *pipes*. Pipes can be either unidirectional or bidirectional. Where the latter are implemented using two endpoints with the same endpoint number but different direction. Figure 2.2 shows two bidirectional pipes. Both pipes connected to EP0 and EP1 are bidirectional pipes. The pipes connected to the remaining endpoints are

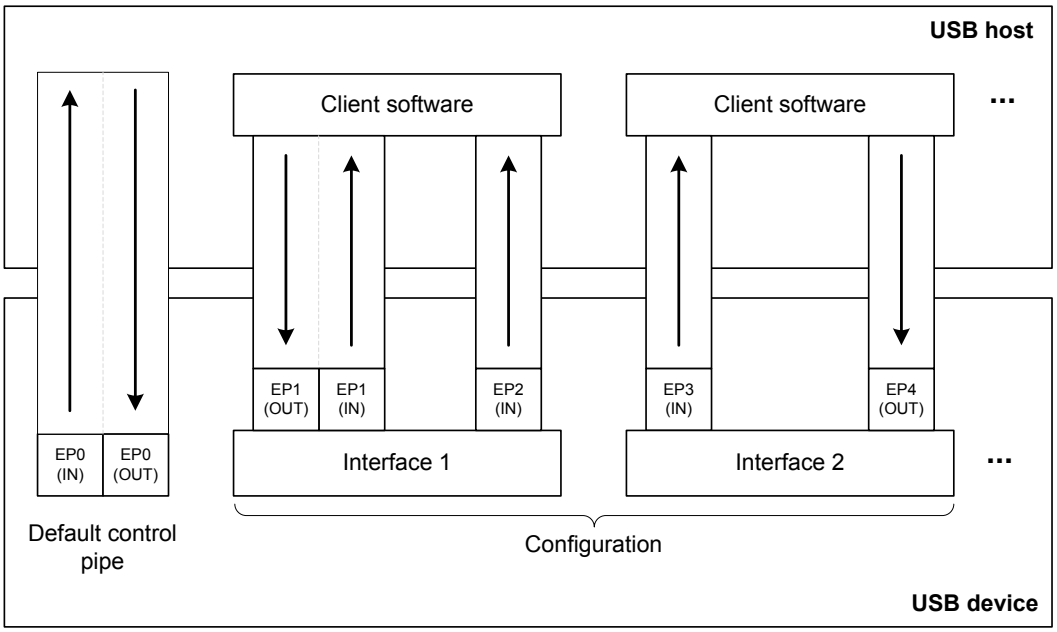

Figure 2.2: Logical connection between a USB device and a host

unidirectional pipes. There are two different kinds of pipes, differing only in the structure of data transferred. *Stream pipes* are used for the transmission of data without any USB–defined structure. The communication flow in a stream pipe is always unidirectional. They are mainly used for pure data transfers. *Message pipes* on the other hand have an imposed structure on the data transferred over them. A successful communication flow over a message pipe always starts with a request from the host, followed by a data transfer in one direction and finally a status transmission to indicate the success of the data transfer. Hence, message pipes are always bidirectional and require the IN and the OUT endpoint with the same endpoint number. Once a device is powered up, one message pipe that is used by the host for the configuration of the device always exists. This pipe is called the Default Control Pipe. It is always connected with endpoint number zero.

Interfaces

Multiple pipes are grouped together to form an *interface* as can be seen in Figure 2.2. Interfaces represent a single specific feature, which is offered to the host. One interface could provide a mass storage device, while another one might by used to provide a human interface device, such as a mouse. The exact purpose of each pipe in an interface, and the protocol used to communicate over them, can either be specified in a device class specification as introduced in Section 2.1.7 or in a vendor–specific definition. Most USB devices only provide one single interface at a time, but it's possible for a USB device to provide multiple interfaces. Such a device is called a *composite device*. All interfaces can operate independently and provide their service at the same time, while the device only has a single device address.

Configurations

A USB device can provide one or more *configurations*. Each configuration consists of one or more interfaces. Compared to interfaces, only a single configuration can be active at any one time. The host can choose between each provided configuration. It can even switch between configurations, while the device is attached.

2.1.3 Bus Protocol

USB uses a token–based protocol, which employs polling. The host initiates all transfers and checks on a scheduled basis whether a device needs to be served. In the following, we provide a top–down overview of the USB protocol.

Transactions are used to transfer data on the bus in one direction. Data is either transferred from the USB host to a USB device (upstream) or from a USB device to the USB host (downstream). A transaction consists of multiple *packets*. Each packet consists of 8–bit bytes with the least–significant bit first. All packets start with a synchronization field SYNC, which is used by the electrical layer of the host to align incoming data with the local clock. After the SYNC field, each packet contains a one byte packet identifier (PID). The packet identifier specifies the type of the packet. Packets are grouped into packet classes. Three different packet classes are defined. Table 2.1 lists the packet classes and some associated packets together with their encoding and a description. For a complete list of all packet types, please refer to the USB specification [3].

Packet class	PID name	PID value	Description
Token	OUT	1000	Asks the function to receive data from the host.
	IN	1001	Asks the function to send data to the host.
	SETUP	1011	Initiation of a control transfer by the host.
Data	DATA0	1100	Even data transfer.
	DATA1	1101	Odd data transfer.
	DATA2	1110	Used for high–speed isochronous endpoints with high bandwidth.
	MDATA	1111	Used for high–speed isochronous endpoints with high bandwidth and also used in split transactions.
Handshake	ACK	0100	Acknowledgement of the received Data packet.
	NAK	0101	Negative acknowledgement send by a function.
	STALL	0111	Indicates inability to send/receive data.

Table 2.1: USB packet classes

Token packets are sent on a scheduled interval by the host. A transaction always starts with the transmission of a token packet. Token packets are only sent by the host and never by a device. Each token packet consists of a device address and endpoint number, in addition to the PID field. Inside OUT and SETUP packets, those two fields are used to uniquely identify the endpoint, which will retrieve the following data packet. IN packets use those fields to identify the sender of the following data packet.

Data packets are sent in response to the reception of a token packet. They contain a data field that can hold up to 1024 bytes of data. When a device receives an OUT token packet, it reads the following data packet from the host. If an IN token packet is received by the device, the

following data packet is sent from the device to the host. The different data packets listed in Figure 2.1 are used for different transaction modes not covered in this thesis. Please refer to Chapter 8 of the USB specification for more details.

Handshake packets are used to return the status of a data transfer. The transmission of a handshake packet finishes a transaction. Handshake packets don't have any additional packet fields. ACK handshake packets are send by the receiver of a data packet to signal the successful retrieval of the data. NAK packets are used by a function to either signal, that it couldn't read the data send by the host of the OUT transaction or it didn't have any data to be send for the IN transaction. In both cases, it only signals a temporary condition. NAK packets are never send by the host. STALL packets are send by a function to indicate the inability to transmit or receive data.

Every USB device connected to the bus reads the initial token packet and decodes the device and endpoint address therein. If the device address matches, the device selects itself for the current transaction. The source of the transaction then sends the data in a data packet to the destination. If no data is available at present, it indicates this by sending a NAK handshake packet instead. The destination acknowledges the reception of the data packet with an ACK handshake packet. The absence of an expected response indicates a failure. Token and data packets are protected by a cyclic redundancy check (CRC).

All fields except for the packet identifier (PID) are covered by the CRC checksum in both packets. If the CRC doesn't match on the receiver side, the packet was corrupted and is ignored by the receiver. The four–bit PID field of every packet is not covered by the packet CRC checksum. It's already protected by the following four–bit check field, which is generated by performing a one's complement of the PID field.

2.1.4 Transfer Types

The USB specification defines four different data transfer types, which determine, how data is to be transferred over a pipe between the host and the endpoints on the device. Each transfer type has different characteristics and thus fulfills the needs of different types of functions. Additionally, each transfer type determines the type and order of performed transactions. The decision, which transfer type is used for which pipe is made at development time of a device. The transfer type can't be changed afterwards. The various characteristics of each transfer type will be described below.

Control Transfer

Control transfers can be used for configuration, command and status requests to a device. They are only used on message pipes. Hence, the Default Control Pipe uses control transfers too. The control transfer is the only transfer type which has an imposed structure on the data to be transferred over the pipe. A control transfer always starts with a SETUP packet from the host to the device, which contains the actual request. The SETUP packet is followed by zero or more data packets, which transfer the data in the requested direction. Finally, a handshake packet is sent from the function to the host, confirming the transfer. Control transfers are handled as a "best effort" transfer. If there is free bus time, control transfers are scheduled. However, neither bandwidth nor latency is guaranteed, although data delivery is guaranteed to be without loss.

Bulk Transfer

Bulk transfers are used for stream pipes with high bandwidth demands. This transfer type is usually used to transfer large amounts of data. Any bandwidth available can be used. But no guarantee for bandwidth or latency is given. Bulk transfers are handled the same way as control transfers. When there is free bus time, bulk transfers can happen, but control transfers have a higher priority than bulk transfers and therefore, are given priority. In the case of a bus error, transfers are retried and the delivery of the data is guaranteed. Bulk transfers are, for example, used with USB flash drives to transfer the data.

Interrupt Transfer

Interrupt transfers are used for devices that need infrequent access to the bus, but with guaranteed maximum service periods. Interrupt transfers are only used in stream pipes and thus are always unidirectional. An example for a device using an interrupt transfer is a USB mouse. Coordinate changes are only generated when the mouse is moved, but changes must arrive at the host within a guaranteed time–frame.

Isochronous Transfer

Isochronous transfers guarantee a specific USB bandwidth with bounded latency. The endpoint on the device specifies its required bus access period. Additionally, the data transferred over the pipe is guaranteed to have a constant data rate, as long as enough data is provided by the sender. Isochronous transfers are only used on stream pipes and thus are always unidirectional. The obvious result is that transmission errors are only indicated to the receiver of a transaction. Only full–speed and high–speed devices can use the isochronous transfer mode. Isochronous transfers are for example used for the transmission of audio or video streams.

2.1.5 Descriptors

Descriptors are data structures that are provided by devices to describe all of their attributes. When a device is attached to the bus, the host reads the descriptors from the device and configures the device based on the read descriptors. The USB specification defines some standard descriptors which are described below. There may be additional class– and vendor–specific descriptors. All standard descriptors have a similar structure. They start with a one–byte length field, which describes the total length of the descriptor. The length byte is followed by a one–byte type field, which describes the type of the descriptor. The rest of the descriptor structure depends on the specific descriptor type. All standard descriptors are interleaved and provided in serial form to the host. Figure 2.3 shows the descriptors as they would be provided by the sample USB device from Figure 2.2. The different standard descriptors are described below:

Device Descriptor

The device descriptor describes some general information about the device. All of the information that may change with different configurations or interfaces is described in separate descriptors.

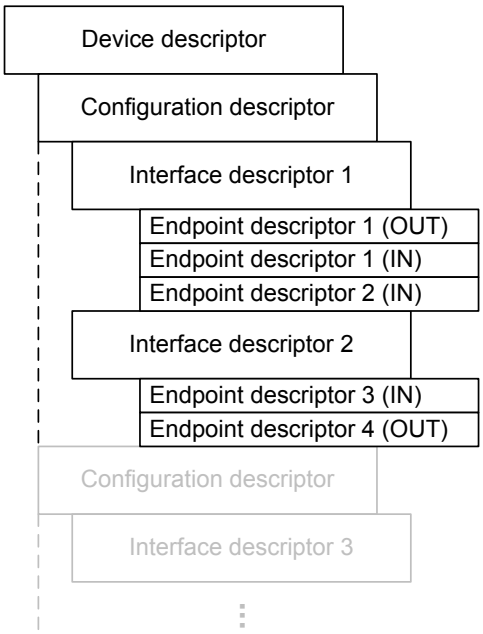

Figure 2.3: Interleaved descriptors provided by the sample USB device from Figure 2.2

There is only one device descriptor per device. It contains things like the number of configurations, maximum packet size and the vendor and product IDs. The device descriptor is the first descriptor read by the host as illustrated in Figure 2.3.

Device_Qualifier Descriptor

The device_qualifier descriptor is provided by devices that have different device information depending on the speed they are operating at. For example, a high–speed device may have different settings when operating as a full–speed or low–speed device. The settings for the current operating speed are returned in the device descriptor. The device_qualifier descriptor may only be used by the host to request settings for operating speeds that it isn't currently using. The device_qualifier descriptor includes the same information as the device descriptor except for the things that don't change with different operating speeds such as the vendor and product ID.

Configuration Descriptor

The configuration descriptor describes a specific configuration of the device. There is one configuration descriptor for each configuration. A configuration descriptor describes one or more interfaces. One of the first fields of the descriptor is a configuration ID, which is used by the host, when it is selecting this specific configuration. Additional fields include the number of interfaces provided by the configuration and optional references to string descriptors describing the configuration using text strings. When the host requests the configuration descriptor, all related interface and endpoint descriptors are returned as well. Figure 2.3 shows all the descriptors, which are returned with the configuration descriptor. The configuration descriptor is

followed by the interface descriptor of the first interface, followed by all endpoint descriptors for the endpoints of the interface. These endpoint descriptors are followed by possibly more interface descriptors and their respective endpoint descriptors.

Other_Speed_Configuration Descriptor

The other_speed_configuration descriptor describes the configuration of a high–speed device. It may only be requested by the host, when operating at one of the other speeds. The structure of the other_speed_configuration descriptor is the same as the configuration descriptor. It allows the host to get details about the high–speed configuration, even though it is currently operating at another speed.

Interface Descriptor

The interface descriptor describes a specific interface of a configuration. It provides zero or more endpoint descriptors. Interface descriptors can't be accessed directly, but are always returned as part of a configuration descriptor. Fields of the interface descriptor include the number of endpoints, class and subclass code and an interface protocol, which may refer to a device class specific protocol.

Endpoint Descriptor

Endpoint descriptors describe the properties of endpoints. The host uses that information to make sure; it can handle the bandwidth requirements of all endpoints. The endpoint descriptors, like the interface descriptors, can't be accessed directly. They are always returned as part of an interface descriptor. Fields of an endpoint descriptor include the type and address of the endpoint, the maximum packet size and the required polling interval, which is used by the host to contact the endpoint. Every endpoint has a corresponding endpoint descriptor, with one exception: endpoint number zero does not have an endpoint descriptor. The maximum packet size of endpoint zero is defined in the device descriptor and everything else is implicitly defined by the USB specification.

String Descriptor

The string descriptor is a special case, insomuch that it isn't returned directly by the device on attachment time. All other device, configuration and interface descriptors may reference strings from an optional string descriptor using string indexes. If no string descriptor is present on the device, all references must be null. The actual strings in the string descriptor are represented using Unicode UTF–16 encodings as defined in The Unicode Standard [4]. Strings can be represented in multiple languages inside the same string descriptor. When the host requests a string descriptor, it supplies a 16–bit language ID with it. String index zero is a special case. It returns a string descriptor with an array of all supported language IDs supported by the device.

2.1.6 Bus Enumeration

Bus enumeration is the process, in which the host learns about the attached USB device and loads the corresponding device drivers. Only after a successful device enumeration can software running on the host communicate with the attached USB device. During the enumeration process, devices transition from one state to another. The USB specification defines six different device states [5]:

1. Powered
2. Default
3. Address
4. Configured
5. Attached
6. Suspended

Except for the last two states, a USB device usually transitions through all those states during the enumeration process. Although a specific order is not defined, a usual enumeration process starts with a device in the Powered state and ends when the device reached the Configured state.

Enumeration starts, when a new device is attached to a hub. This can be either the root hub or any other hub in the USB topology. Each hub provides an interrupt IN endpoint, which is used to inform the host about newly attached devices. The host continually polls on this endpoint to receive device attachment and removal events from the hub.

When a new device was detected by the hub, the host is notified about this event. At this point, the hub port is still disabled and the device is in the Powered state. In response to the status notification from the hub, the host queries the hub to receive the exact cause of the status notification and to receive the actual port number, where the device was attached. Then, the host waits for at least 100 milliseconds to let the device settle and until the power gets stable.

The host then sends a request to the hub to enable and reset the port. In response to this request, the port is enabled and the device switches to the Default state. In this state, the device can answer to requests sent from the host to the default address zero. The host then starts sending standard USB requests through the Default Control Pipe to endpoint zero of the device. Since the host only enumerates one device at a time, it is guaranteed, that only one device will answer to requests addressed to the device address zero.

The first thing, the host requests from the device, is the device descriptor. This descriptor must be read first, because it contains the maximum packet size of the Default Control Pipe, which is needed for further communication. Because the host doesn't know the maximum packet size before reading the device descriptor, it only reads the first few bytes of it, which contain the maximum packet size.

Although not required by the specification, some implementations issue another port reset to the hub at this point to make sure that the device is at a known state afterwards. After the device was reset, the host controller assigns a unique address to the device and sends a request to the device to change its address. After the device changed its address, it switches to the Address state and henceforth, only answers to requests to the newly assigned address.

Using the newly assigned address, the host now sends another request for the device descriptor. This time, equipped with the knowledge of the maximum packet size, it reads the entire device descriptor. After reading the device descriptor, one or more configuration descriptors, as specified in the device descriptor, are read by the host. By reading a configuration descriptor, all associated interface and endpoint descriptors are returned as well. After the host reads all the descriptors, it tries to find a matching device driver. This process is highly dependant on the used operating system and is described in detail for every major operating system in Chapter 3.

After a matching device driver was found and loaded, it's the task of the device driver to select one of the provided device configurations. The device driver selects one of the configurations based on its own capabilities and the available bandwidth on the bus and activates this configuration on the attached device. At this point, the device is in the Configured state. That means, that all interfaces and their endpoints of the selected configuration are set up and the device is ready for use.

When a hub doesn't provide any power to a connected device, it is in the Attached state. This condition can either be forced by the host or it could happen due to a detected over–current condition. In the Attached state, a device can't communicate with the host.

During normal operation the host is constantly sending token packets on the bus. A device can enter the Suspended state, when no activity has been seen on the bus for at least 3 milliseconds. In this state, the device should try to limit its power consumption to a minimum.

2.1.7 Device Classes

USB devices can be categorized into groups of different device classes [5]. A device class describes devices that have similar attributes and services in common. The grouping of devices into different device classes has the benefit that only one device driver must be written per device class instead of writing a new driver for every developed device. Operating systems can provide class drivers, so that the vendor of a USB device doesn't have to provide separate drivers. In addition, the device class specification itself can be used as a reference when building new USB devices of a specific class.

There are device class specifications for a lot of device classes, such as audio devices, human interface devices (HID) or mass storage devices. USB hubs are a special case. They are not described in a separate class specification. Their description is part of the USB specification [3] itself. This decision was made because the USB architecture can't work without at least one USB hub, the root hub. Each class specification assigns at least one class code, which can be used by the USB device to identify itself as a device of that class. Depending on the device class, the class code can be provided at two different locations by the device. It can either be specified in the device descriptor or in the interface descriptor. Some device classes allow the class code to be provided in both descriptors. For a list of approved class specifications [6] and the descriptors, where the class code can be specified, see Table 2.2.

All device class specifications are based on the Common Class Specification. This specification defines all the things that should be included in a class specification document and how it should be organized. Typical things included in a class specification are:

- Number of endpoints and their use
- Values in standard descriptors

Class	Descriptor usage
Audio	Interface
Communications Device	Device/Interface
Human Interface Device (HID)	Interface
Still Image Capture	Interface
Printer	Interface
Mass Storage	Interface
Chip/Smart Card	Interface
Content Security	Interface
Video	Interface
Personal Healthcare	Device/Interface
Device Firmware Upgrade	Interface
IrDA Bridge	Interface
Test and Measurement	Interface

Table 2.2: USB class specifications

- Class–specific descriptors and their structure
- Interfaces
- Control requests

Additionally, the structure of the data transferred by a class device may be defined in the class specification. For example the class specification for Human Interface Devices (HID) defines the format of the *report data structure*, which is used to transfer events such as mouse movements or clicks to the host. However, the data transferred by a class device can also be specified in a different standard, as it's the case with mass storage devices, where SCSI [7] commands are transferred.

A class specification has to satisfy two claims: it should allow a manufacturer to build a new device for that specific device class and developers should be able to write a device class driver with the help of the class specification.

2.2 Software Vulnerabilities

This chapter will introduce some of the most common classical software vulnerability classes that can be used by an attacker to compromise a system. Each class is first introduced, then the method of exploitation is described.

2.2.1 Buffer Overflows

Buffer overflows happen, when a process writes more data to a fixed–length buffer in memory than it can hold. Adjacent memory regions will be overwritten by the data exceeding the buffer length leading to undefined behaviour. Depending on the location of the buffer in memory and the content of adjacent memory regions, the consequences can range from simple program crashes to the execution of arbitrary attacker–supplied code.

The root cause of buffer overflows is insufficient bounds checking. If an attacker can supply data to a process that is used as the source of a memory copy operation, and the process doesn't check for the data fitting into the buffer, a buffer overflow condition can occur. C and C++ are two of the most common programming languages plagued by buffer overflows because they have no built–in protection for out of bounds memory access.

Two specific buffer overflow classes will be described in more detail in the following two sections. Starting with stack–based overflows and then descending to the slightly more complicated heap–based overflows.

Stack–based Overflows

Stack–based overflows are buffer overflows, where the buffer resides on the execution stack. Figure 2.4 illustrates a simplified memory layout of a process in the IA–32 architecture [8]. The kernel of the operating system resides in the lower memory addresses. The memory of the actual process is organized in multiple segments. The read–only text segment holds all the instructions of the process to be executed. The data and bss segments hold initialized and uninitialized data respectively. For example, static variables are stored in those segments. The heap holds dynamically allocated memory buffers. Data on the heap must be allocated and freed explicitly by the application. The stack is used for local variables, arguments and return values of functions. In addition, control data of active functions is placed on the stack. This control data includes the return address of the current function, where program execution should continue after returning from the current function. The stack is organized as a LIFO (last in, first out) structure. That is, elements can only be added or removed from the top of the stack. In contrast to the other segments, the heap and stack segments don't have a fixed size. They are automatically enlarged by the kernel when more space is needed. The heap grows towards the high memory addresses, while the stack grows towards the low memory addresses. The operating system makes sure that the heap and the stack don't clash.

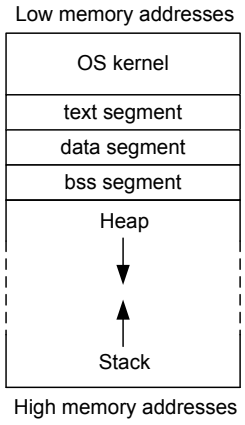

Figure 2.4: Process memory layout

The stack is composed of multiple *stack frames*. Each stack frame corresponds to an active function, which has not yet returned. There is one register which always points at the top of the stack. It's called the *stack pointer*. The bottom of the current stack frame can be

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void
5  parse(char *ptr)
6  {
7          char buf[128];
8
9          strcpy(buf, ptr);
10         printf("%s\n", buf);
11 }
12
13 int
14 main(int argc, char **argv)
15 {
16         if (argc != 2)
17                 return 1;
18         parse(argv[1]);
19         return 0;
20 }
```

Listing 2.1: Simple strcpy(3) overflow

indicated by another register, called the *base pointer*. Memory inside the current stack frame can be addressed relative to the base pointer. To modify the stack, the IA–32 architecture implements two instructions called `PUSH` and `POP`. The `PUSH` instruction decrements the stack pointer and stores the source operand on the stack at the new address of the stack pointer. The `POP` instruction reads a value from the stack into the destination operand and increases the stack pointer.

On the IA–32 architecture, a function is called using the `CALL` instruction. This instruction saves the instruction pointer of the next instruction after the `CALL` on the stack, before jumping to the given function. When program control gets transferred to the function, it starts by executing a *function prologue*. This function prologue is responsible for setting up a new stack frame and saving the registers to be used onto the stack. A typical function prologue as generated by a compiler first saves the current base pointer EBP on the stack. It then sets the current value of the stack pointer ESP as the new value for the base pointer, effectively starting a new stack frame. By decrementing the stack pointer, the size of the stack frame can be increased to reserve some space for local variables. At the end of a function, the *function epilogue* restores the values of the old stack and base pointer, and issues the `RET` instruction, which transfers program control back to the address located on top of the stack.

Listing 2.1 shows a small C program, which is vulnerable to a simple stack–based overflow. It first makes sure that exactly one command line argument was passed and then calls the `parse()` function with a pointer to the first argument. The `parse()` function then copies the string to a local buffer `buf`, which can hold up to 128 bytes. Since the `strcpy(3)` function doesn't perform any bounds checking, this can overflow the local buffer, if the length of the first command line argument exceeds the length of the buffer.

Figure 2.5 shows the stack layout just before calling the `strcpy(3)` function. At the bottom of

the stack is the stack frame of the `main()` function. It contains the arguments of the function pushed on the stack in reverse order and at the top of the stack frame, the return address, where the `main()` function returns to.

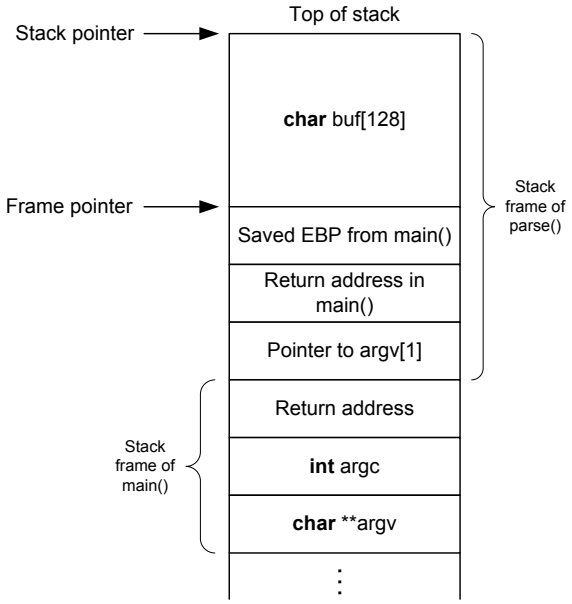

Figure 2.5: Stack layout just before the strcpy(3)

After `main()`'s stack frame, the stack frame of the `parse()` function begins. It begins with the arguments passed to the `parse()` function, which in this case is only a pointer to the first command line argument `argv[1]`. After that comes the return address, where the `parse()` function will transfer program control back to when finished. This return address was pushed onto the stack by the `CALL` instruction. The return address is followed by the saved base pointer, which was stored there by the function prologue of `parse()`. Just next to the saved base pointer at the top of the stack resides the local buffer `buf[128]`.

When the program in Listing 2.1 is called with an argument of 128 characters or more, the call to `strcpy(3)` starts to overflow the adjacent memory region. Starting with the saved base pointer, the return address and everything after that is overwritten by the `strcpy(3)` overflow. So by crafting a specially formed command line argument, the attacker can control other values on the stack. When running the program with a command line argument of exactly 136 "A" characters, the buffer is filled by the first 128 characters and the remaining 8 characters overwrite the saved based pointer and the saved return address on the stack. Figure 2.6 shows the stack, just after the `strcpy(3)` function returned.

Note that a single byte of the pointer to `argv[1]` is also overwritten by a `NUL` byte. This is because the `strcpy(3)` function always NUL–terminates the copied string. When the `parse()` function returns, its function epilogue first pops the saved base pointer from the stack and stores it as the current base pointer. Since it was overwritten with four "A" characters, it is now `0x41414141`. After that, the `RET` instruction pops the saved return address from the stack and transfers program control to it. The saved return address was overwritten as well, so program control is now transferred to the address `0x41414141`. Since this address is likely not mapped in

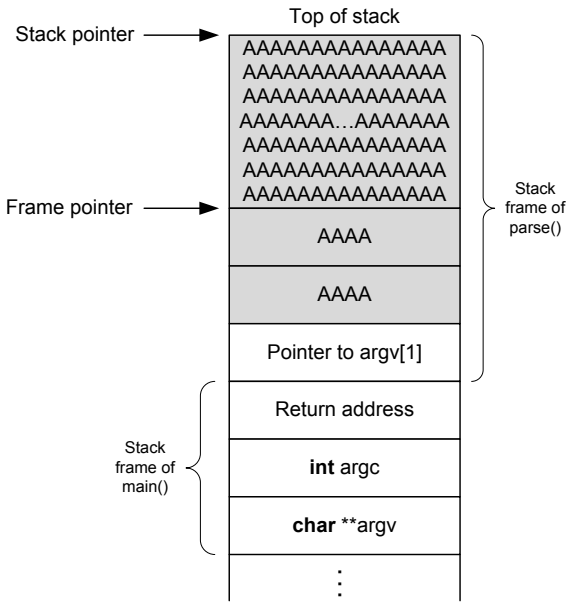

Figure 2.6: Stack layout after a strcpy(3) overflow

memory, this results in a segmentation violation.

One way an attacker could exploit this vulnerability is by putting his own code to be executed at the start of the buffer and then overwrite the saved return address with the memory address, where his own code resides on the stack. This way, when the `parse()` function returns, it transfers program control directly to the attackers supplied code, which then gets executed on the stack [9].

Heap–based Overflows

Heap–based overflows are buffer overflows where the buffer resides on the heap of a process. The heap is a dynamically sized memory region, managed by some heap allocator which resides in the user–mode. In contrast to the stack, the heap can be used by applications to store data, the length of which isn't known until run–time. By utilizing the help of the kernel, the heap allocator reserves some big block of memory in the heap segment of the process and provides smaller chunks of it to the process on demand. Additionally, it can free the requested chunks again, if requested by the application. While trying to keep good performance, the heap allocator also tries to avoid fragmentation [10].

Common heap allocators include RtlHeap, which is used on Windows–based operating systems, Doug Lea's Malloc [11], which is included in the GNU C Library and PHK malloc [12] which is mainly used on BSD–derived operating systems.

The general problem, which makes heap–based overflows universally exploitable, is the in–band storage of management information. This includes lists of used and free blocks and the different sizes of memory blocks. When an attacker overflows a buffer on the heap, this management data is modified, which can lead to arbitrary memory overwrites, as will be demonstrated below.

```
1 #define INTERNAL_SIZE_T size_t
2
3 struct malloc_chunk {
4         INTERNAL_SIZE_T prev_size;
5         INTERNAL_SIZE_T size;
6         struct malloc_chunk *fd;
7         struct malloc_chunk *bk;
8 };
```

Listing 2.2: Structure of the boundary tag

In the following, the general functionality and exploitation of a heap allocator will be described on the basis of the original Doug Lea's Malloc. Doug Lea's Malloc, or *dlmalloc* for short, provides some library functions for managing the heap memory. These include `malloc(3)`, `free(3)`, `calloc(3)` and `realloc(3)`. These allow a process to request some parts of memory and to free it again. The heap allocator separates the memory of the heap into multiple *chunks*. Each chunk represents an allocated or free part of the memory. Each chunk starts with a *boundary tag*, which is basically some management information used by the heap allocator and is hidden inside the implementation. The structure of a boundary tag can be seen in Listing 2.2.

The `prev_size` member describes the size of the previous chunk, while the `size` member describes the size of the current chunk. Since all chunks are 8 byte aligned, the 3 least significant bits of the `size` member are always 0. These are used to store two status bits. The low–order bit is `PREV_INUSE`, which indicates, if the previous chunk is allocated or free. The second–lowest order bit is `IS_MMAPPED`, which indicates, if the underlying memory was obtained by the `mmap(2)` system call. The size fields are followed by `fd` and `bk`, which are pointers to the next and the previous chunks respectively. Note, that the next and previous chunks are not necessarily the physical adjacent ones.

The use of the different members of the boundary tag depends on if the current and the previous chunk is free or allocated. The `prev_size` member is only used by dlmalloc, should the previous chunk be a free chunk. The forward chunk pointer `fd` and the backward chunk pointer `bk` are only used if the current chunk is free. In allocated chunks, the user data starts directly after the `size` member. See Figure 2.7 for two adjacent heap chunks, where a free heap chunk is followed by an allocated one.

All available free chunks are stored in *bins*. There are multiple fixed–width bins, each storing free chunks of a different specific size range. The chunks themselves are sorted by a decreasing size sequence inside their bins. Each bin represents a single circular doubly–linked list, which is empty at program start. Free neighbour chunks are always coalesced to form the largest possible free chunk and to prevent fragmentation. New chunks are searched for in a smallest–comes–first, best–fit order.

When an application requests some heap memory by the use of one of the aforementioned library functions, dlmalloc needs to find some free chunk and remove it from the bin. The `unlink()` macro is used for this purpose. When the application no longer needs the memory, it calls the `free(3)` function, causing dlmalloc to add the respective chunk back to a bin. This is done using the `frontlink()` macro. The `unlink()` macro is shown in Listing 2.3.

The `unlink()` macro first reads the backward chunk pointer of the chunk to be unlinked and stores it in `BK` in line 2. The forward chunk pointer is stored in `FD` in line 3. To remove the chunk

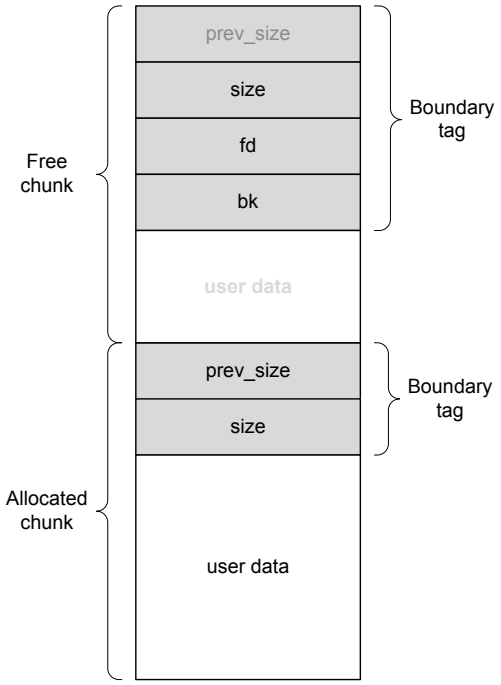

Figure 2.7: Two adjacent heap chunks

```
1 #define unlink(P, BK, FD) {                                          \
2         BK = P->bk;                                                  \
3         FD = P->fd;                                                  \
4         FD->bk = BK;                                                 \
5         BK->fd = FD;                                                 \
6 }
```

Listing 2.3: dlmalloc unlink() macro

from the doubly–linked list, first the backward pointer of the next chunk `FD->bk`, which pointed to the chunk `P` to be removed, is set to the previous chunk `BK` in line 4. The same is done for the forward chunk pointer of the previous element `BK->fd` in line 5, effectively unlinking chunk `P`. This process is illustrated in Figure 2.8.

Both the `unlink()` and the `frontlink()` macros can be used by an attacker to exploit a heap–based overflow to execute arbitrary code. We are only demonstrating the exploitation using the `unlink()` macro. Exploitation using the `frontlink()` macro works in a similar way. Interested readers may refer to [13].

When an attacker overflows a buffer on the heap, he starts overwriting the boundary tag of the following chunk. By modifying the boundary tag, the attacker can effectively construct a fake chunk, with values of his own choosing. By tricking dlmalloc to process this fake chunk with the unlink() macro, an arbitrary value chosen by the attacker can be written at an arbitrary location in memory. This is enough for an attacker to execute arbitrary code.

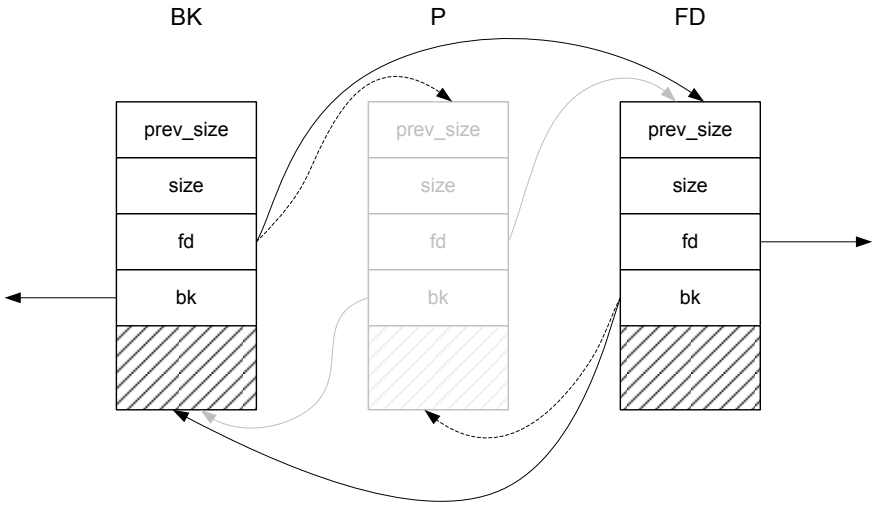

Figure 2.8: Removing a chunk from the doubly–linked list

The memory location to overwrite, subtracted by 12, can be stored in the forward pointer `P->fd` of the fake chunk. The need for the constant to be subtracted will become clear shortly. The value to be written at that memory location must be placed in the backward pointer `P->bk`. When the `unlink()` macro tries to unlink the fake chunk, it first reads the backward pointer `P->bk`, which is now our value to be written, and stores it into `BK`. Next, the forward pointer (which is the memory location minus 12) is stored in `FD`. Line 4 of the `unlink()` macro now stores the value of `BK` at `FD->bk`. Since the `bk` member of the boundary tag structure is exactly at offset 12, it assumes the value from `FD`, which was the desired memory location minus 12, and adds 12 to it. This effectively writes the value from `BK` at the desired address.

One way to accomplish arbitrary remote code execution is to overwrite some function pointer in memory with the address of some attacker–supplied shellcode in the hope that this function pointer will get used in the future. The attacker would store the address of the function pointer inside the forward pointer and the address of the shellcode inside the backward pointer. However, when the last line of the unlink() macro gets executed, it would write the forward pointer `FD`, containing the address of the function pointer, directly in the middle of the shellcode. To overcome this small hurdle, the first thing the shellcode should do, is to jump over this location, to prevent modification by the `unlink()` macro.

Most of the common heap allocator implementations where hardened recently to make exploitation as described above more difficult by introducing some kind of cookies [14], guard pages [15] or completely storing the management information out–of–band [16]. Although completely protecting the management information may prevent the universal exploitation demonstrated above, other attacks can still be feasible from case to case. Modifying sensitive data, such as function pointers, in the same chunk or overflowing inside the adjacent neighbour chunk might lead to exploitable conditions, even when management information is stored out–of–band.

2.2.2 Double Frees

Double free vulnerabilities happen when the application mistakenly frees a heap memory block twice. This can happen when calling the `free(3)` function twice with the same pointer. These kind of bugs often happen in error conditions and under other exceptional circumstances.

When a memory chunk gets freed by the heap allocator, it is either coalesced with another free neighbour chunk or it is linked inside a bin by the `frontlink()` macro. The `frontlink()` macro just finds the point in the bin, where the chunk should be linked. It then modifies the forward chunk pointer of the chunk before and the backward chunk pointer of the chunk behind to point to the new chunk. The forward and backward chunk pointers of the new chunk are set to the neighbour chunks accordingly.

When `free(3)` is called a second time on the same pointer, dlmalloc tries to free the chunk again. If no free neighbour chunk is found, the `frontlink()` macro is used again on the same chunk. This effectively makes the forward and backward chunk pointer of this chunk point to itself. If thereafter, the attacker can trick the application into requesting memory of the same size, dlmalloc tries to unlink the doubly freed chunk from the bin by using the `unlink()` macro. Since the forward and backward chunk pointer point to the chunk itself, the `unlink()` macro has no effect (see Listing 2.3), effectively leaving the chunk linked in the bin. Now the application holds a pointer to the data portion of the chunk, which is still linked inside a bin as a free chunk. As can be seen in Figure 2.7, the data portion of an allocated chunk directly starts at the offset, where the forward and backward pointers are stored in a free chunk. If the application writes data at the first 8 bytes of the returned pointer, it overwrites the forward and backward chunk pointers. Since the chunk is still linked in the bin, those pointers are still used by dlmalloc.

To exploit such a vulnerability, an attacker can use the same methods, as used for the exploitation of heap–based overflows as demonstrated in Section 2.2.1. To write an arbitrary value at an arbitrary address in memory, the attacker just overwrites the forward chunk pointer with the desired address subtracted by 12 and sets the backward chunk pointer to the value to be written. By tricking the application into requesting another chunk of the same size, dlmalloc again uses the `unlink()` macro to unlink the doubly–freed chunk. But this time, the forward and backward chunk pointers were modified by the attacker, leading to the execution of arbitrary attacker–supplied code.

2.2.3 Integer–related Bugs

Integer–related bugs can happen, when the programmer isn't fully aware of the slightly complicated rules of integer arithmetic and type conversion in the C family of languages. Assumed an attacker is able to influence an arithmetic operation, depending on the code, he might be able to leverage the unexpected behaviour to his benefit.

Unsigned integers are stored in memory by their respective binary sequence. For storing signed integers, usually the twos complement system is used. The most significant bit in memory represents the *sign bit*. It indicates the signedness of the stored integer. The sign bit is accompanied by a number of *value bits*, which are used to store the actual integer without the sign. The number of value bits depends on the type of the integer. A sign bit of 0 indicates a positive integer, while a value of 1 indicates a negative one. The value of positive integers can be read directly from memory by just reading the value bits. This can't be done for negative integers. To get the value of a negative integer, the value bits first have to be converted. This is done by

inverting all the bits and then adding one to the result. This method is used, because it can be implemented efficiently in hardware.

The actual range of representable numbers of a type depends on the number of value bits. For example, an 8–bit unsigned integer can use all 8 bits as value bits. So 2^8 possible values can be represented which results in the range of representable numbers from 0 to 255. A twos complement 8–bit signed integer on the other hand, needs one bit for the sign bit. This leaves 7 bits as value bits. Thus, 2^7 positive and 2^7 negative values are possible. With the sign bit set to 0, positive numbers in the range from 0 to 127 can be represented. When the sign bit is set to 1, negative numbers in the range from -128 to -1 can be represented. So an 8–bit signed integer can store values ranging from -128 to 127. In general, a twos complement signed integer of width X can represent values in the range of -2^{X-1} to $2^{X-1} - 1$ [17].

When doing some arithmetic operation on an integer, it is possible to yield a result, which would exceed the boundaries of the representable numbers of the actual integer type used. This is called an *arithmetic boundary condition*. Normal addition, subtraction, multiplication and even division can result in values, which can't be stored in the underlying type and thus lead to arithmetic boundary conditions. Two specific arithmetic boundary conditions are integer overflows and integer underflows, where the resulting value is either too large or too small to be stored in the integer respectively.

The effect of an arithmetic boundary condition depends on the signedness of the type. Unsigned integers are subject to the rules of modular arithmetic. That means that all results are taken modulo 2^X, where X is the bit width of the integer. Increasing the largest possible value by one, results in a wrap–around to the smallest possible value 0. Likewise, decrementing the smallest value 0 by one results in a wrap–around to the largest possible value.

The effect of arithmetic boundary conditions on signed integers is a little bit different. The largest possible positive value is a value, where the sign bit is 0 and all the value bits are 1. When this value is incremented by one, the sign bit gets flipped and the result is a large negative number. Similarly, when the smallest possible negative number gets decremented by one, the sign bits wraps around and the result is the largest possible positive number.

When arithmetic boundary conditions are not taken into account by the programmer, unexpected results may happen. When an attacker is able to influence an arithmetic operation to provoke an arithmetic boundary condition, he may be able to influence program flow in a certain way. This can often lead to a security compromise of the application. C and C++ are most prone to these kinds of problems, because the programmer has to manage low–level details such as dynamic memory allocation and he is responsible for storing only as much data into a buffer, as it can hold. These tasks require arithmetic operations in most cases. By influencing one of these arithmetic operations, an attacker could trick the application to operate on memory outside the allocated region. This can easily result in buffer overflow conditions, which can be exploited by an attacker as demonstrated in Section 2.2.1.

Listing 2.4 shows a simple example of an integer overflow using an unsigned integer. The program first makes sure, that exactly one command line argument was given to the application. In line 14, the length of the first argument is stored in the unsigned short variable `size`. After the length of the string is known, the length of the buffer to be allocated is calculated in line 16. The one additional byte is used for the terminating NUL byte, which will be appended in line 22. Line 18 then allocates a buffer on the heap using the calculated buffer size. Following the allocation of the buffer, the provided command line argument will be copied to the allocated buffer in line 21 using the `memcpy(3)` function.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int
6  main(int argc, char **argv)
7  {
8          char *buf;
9          unsigned short buflen, size;
10
11         if (argc != 2)
12                 return 1;
13
14         size = strlen(argv[1]);
15
16         buflen = size + 1;
17
18         if ((buf = malloc(buflen)) == NULL)
19                 return 1;
20
21         memcpy(buf, argv[1], size);
22         buf[size] = '\0';
23
24         printf("%s", buf);
25         free(buf);
26
27         return 0;
28 }
```

Listing 2.4: Example of an exploitable integer overflow

This example may seem harmless, but is actually an exploitable integer overflow. Assume, that the `size` variable is a 16–bit variable. When the program is run with a first command line argument of exactly 65535 (0xffff) characters, which is the largest possible value representable in a 16–bit unsigned short integer, the arithmetic operation `size + 1` in line 16 will result in a value, which can't be stored in a 16–bit unsigned short integer. The result of that expression will wrap around to the number 0, which will then be used to allocate the buffer on the heap. Later in the call to `memcpy(3)`, the value of `size` (0xffff) is used to copy memory from the second command line argument to the zero–sized heap buffer, which results in memory corruption.

But integer overflows and underflows are not the only problems, which can be exploited by an attacker. Type conversion bugs are another class of bugs, which are really tricky to spot in the source code of an application. These kind of bugs often lead to security vulnerabilities as well.

Type conversion is the process of the compiler, by which it converts an object of one type to another. There are basically two kinds of type conversion. The first one is explicit type conversions. This happens, when the programmer explicitly used casts to convert from one type to another. The second kind of type conversion is implicit type conversion. As the name suggests, implicit type conversion is done by the compiler in the background without the programmer explicitly asking for it. It tries to make things work as expected.

Similar to arithmetic boundary conditions, type conversion can result in bugs, where variables can hold values, unexpected by the developer of an application. This again can lead to unexpected results and may allow an attacker to compromise the security of the application.

A complete description of type conversion bugs and their security relevant consequences can be found in [17].

2.2.4 Format String Vulnerabilities

Format strings are used in a couple of ANSI C functions. Those functions take a variable number of arguments and one of the first arguments is the format string. It describes the number and types of the following arguments. Common format string functions include the `printf(3)` family of functions, `syslog(3)`, `err(3)` and a lot of other ones. Format strings are used to convert the remaining arguments of different C data types to their string representation. If an attacker can influence the format string in certain ways, a format string vulnerability arises.

Format strings consist of normal text and embedded *format specifiers*. One format specifier refers to one argument of the function and indicates the data type of it. Some common format specifiers can be seen in Table 2.3.

Format specifier	Output	Argument type	Passed by
%d	Signed decimal integer	int	value
%u	Unsigned decimal integer	unsigned int	value
%x	Unsigned hexadecimal integer	unsigned int	value
%s	Pointer to array of characters	const char *	reference
%n	(Number of characters written so far is stored)	int *	reference

Table 2.3: Some printf(3) format specifiers

Arguments to the format string functions are passed on the stack. Each format specifier in

```
1  #include <stdio.h>
2
3  void
4  logmsg(char *msg)
5  {
6          printf(msg);
7          printf("\n");
8  }
9
10 int
11 main(int argc, char **argv)
12 {
13         if (argc != 2)
14                 return 1;
15         logmsg(argv[1]);
16         return 0;
17 }
```

Listing 2.5: Simple format string vulnerability

the format string is replaced in the output with the string representation of one argument from the stack. Thus the number of format specifiers in the format string determines, how many arguments are read from the stack.

Listing 2.5 shows a simple format string vulnerability. The first command line argument `argv[1]` of the application is passed to the `logmsg()` function where it is directly used as the format string for `printf(3)`. As long as no format specifiers are found in the string, the application shows the expected behaviour, just printing the given argument on the console. But when format specifiers are found in the string, the application tries to receive the respective arguments from the stack. Since the format string is the only argument passed to the `printf(3)` function, arbitrary data is read from the stack.

An attacker has multiple choices, when exploiting such a vulnerability. Information disclosure is one of the easiest ones. If the format string function outputs the string somewhere, where the attacker can read it, he can specify multiple `%x` format specifiers and thus display the content of the stack. Sensitive information such as return addresses or possible stack cookies could be learned this way. But not only values from the stack can be displayed. Even values from arbitrary memory locations can be obtained, by using one of the format specifiers, which expect their argument to be passed by reference. In most cases, the format string itself is located somewhere on the stack. By supplying enough "dummy format specifiers", the format string itself can be reached. The desired memory address can then be placed inside the format string. By using a format specifier such as `%s` at that point, all the characters at the desired address are displayed.

Another simple attack is to just crash the application. This can be easily done by putting multiple format specifiers into the format string, which expect their argument passed by reference. When the `printf(3)` function tries to dereference the respective pointers on the stack, chances will be high, that there are addresses, which don't represent valid addresses to mapped memory, resulting in a segmentation violation and the termination of the process.

But a format string vulnerability can even be exploited to gain arbitrary code execution for the attacker [18]. The special `%n` format specifier can be used for this purpose. It is special in so far, that it doesn't produce any output in the resulting string, but it stores the number of the currently written bytes inside its corresponding argument on the stack. Like in the information disclosure attack above, the attacker uses a few format specifiers for padding to reach the actual format string on the stack. Since the `%n` format specifier expects a pointer to an integer on the stack, as stated in Table 2.3, an arbitrary memory address can be put into the format string, which is then used by the format specifier to store the current number of written characters. The number of written characters can be influenced to a certain extend, since the attacker can just prepend more characters in the format string. By setting the minimum field width of the format specifiers to a large value, the number of written character can even be more increased. But this is not enough, to write a complete 4 byte value, which is needed e.g. to overwrite a pointer in memory. To overcome this limitation, one byte at a time can be written. Since the `%n` format specifier expects a pointer to an integer, it always writes 4 bytes. By using four single `%n` format specifiers with each address on the stack addressing one of the 4 single bytes, each least significant byte of the value can be written separately, always overwriting the next 3 bytes. This effectively stores an arbitrary 4 byte value at an arbitrary address. The 3 bytes behind that 4 byte value are overwritten too, but that can be neglected in most cases.

2.3 Fuzz–Testing

The process of finding new vulnerabilities in software can be categorized into two approaches, namely white box testing and black box testing. One popular white box technique is source code auditing. As the name implies, the auditor has full access to the software's source code. Although a big advantage, depending on the code size, it can also be a really complex and time–consuming task leading to high costs.

An alternative to source code auditing is fuzz–testing or fuzzing, which is a black box approach. Fuzzing provides unexpected or faulty input to a running process in trying to reach corner cases in the code and trigger exceptions or undefined behaviour [19]. The advantage of fuzzing compared to a white box approach is the simplicity and short time, in which new vulnerabilities may be found without much effort. The missing source code might even be a benefit, because when testing the application the auditor doesn't make any assumptions about the underlying source code, which the developers of the application might have done. This can easily happen, when doing a source code audit and leads to gaps in testing. Fuzzing is a good technique for catching the low–hanging fruit implementation bugs, resulting in crashes of the application. But it doesn't come without a cost. There are several vulnerability classes, which are not likely to be found by fuzzing, such as design logic bugs or other bugs which are not resulting in obvious crashes or exceptions. Compared to a white box approach such as source code auditing, where you could read the source code from start to end, in fuzzing it is harder to get complete code coverage of the whole application, because your input to the application may never reach some code paths.

2.3.1 History

The fuzzing technique was first developed in 1989 at the University of Wisconsin–Madison by Professor Barton Miller and his Advanced Operating Systems class [20]. Inspired by the observation, that sometimes, noise on a dial–up phone line was causing programs to crash in the

current login session, they developed a simple fuzzer, which was used to test the robustness of different UNIX applications. Their fuzzer just passed random strings to each UNIX application and looked for core files afterwards, which would indicate a program crash. When an application crashed or just hung, it had failed the test.

In 1999 work began at the University of Oulu on the PROTOS test suite [21]. They took a little bit more systematic approach than the one taken by the group around Professor Barton Miller. Instead of generating random data, they analyzed some protocol specifications and created hand–crafted packets, which were expected not to be handled correctly by implementations. This approach took considerable more time to create all the packets, but could be used to test lots of different implementations afterwards.

The development of the PROTOS test suite set the stage for the development of lots of different fuzzers in the coming years. Besides all the stand–alone fuzzing scripts, some fuzzing frameworks where created, which allowed to easily write new fuzzers without much effort. One of them will be introduced in the next section.

2.3.2 Block–Based Fuzzing

Block–based fuzzing was first presented by Dave Aitel in 2002 [22] and resulted in *SPIKE*, a fuzzer creation kit based on a C API. Protocol data is structured as blocks, which contain a length field and a queue of bytes. The length field is dynamically updated in each fuzzing run, when the amount of bytes inside the block changes. This way, blocks can be nested and intertwined, without worrying about the length field of each block. In addition to length fields, checksums and other fields, which change as well with the modification of the data, can also be dynamically adjusted. This makes it really easy to stack multiple protocols on top of each other without worrying about the lower layer protocols.

2.3.3 Input Data Creation

There are generally two possibilities to create the fuzzing input for an application which can be broken down into two categories of fuzzers. *Generation–based* fuzzers create their test cases from scratch by modelling a specific protocol or file format, while *mutation–based* fuzzers are modifying valid data samples to construct new test cases from that [19]. Generation–based fuzzers can have a much higher code coverage, because they can implement each and every detail of a protocol or file format. But since they are basically re–implementing the whole protocol or file format, it is a really time–consuming process. Mutation–based fuzzers can be developed very quickly, because the only thing needed is a valid data sample. The quality of the results then depends on the number of different data samples. Protocol or file format details, which are not used in one of the data samples, are obviously not tested.

Chapter 3

USB Support in Selected Operating Systems

Most of the common operating systems today come with some kind of USB support. This chapter gives an overview of the USB support in some of the major operating systems. Section 3.1 introduces the USB support in Linux–based operating systems. It is followed by Section 3.2, which deals with USB support in Apple's latest operating system OS X. Finally, Microsoft Windows XP and Windows Vista are introduced in Section 3.3 and 3.4 respectively.

For each operating system, the driver architecture is described first. Based on this information, the process of enumeration, i.e. what happens when a new USB device is connected to the system and how drivers are loaded, is then described. Finally, a list of all class drivers that come pre–installed with each operating system is given.

3.1 Linux

Linux first introduced some preliminary USB support with the 2.2 series of the kernel. The first complete USB support then came with the 2.4 series, of which some parts were back ported to the 2.2 branch [23]. This section will focus on the latest stable 2.6 series of the kernel as of this writing.

3.1.1 Driver Architecture

The Linux kernel has support for two different kinds of USB device drivers. The first ones are what most people would associate with the word "device drivers". These drivers run on the host system, which gives the host the capability to talk to external USB devices. Throughout this section, we will refer to those drivers as *device drivers*.

The second kind of supported device drivers are drivers running inside some USB device to control the behaviour of the device. These drivers are usually used in USB devices that are implemented as embedded Linux systems. To distinguish their name from the device drivers running on the host, these drivers are called *USB gadget drivers*. In the following two sections,

both kinds of drivers will be introduced starting with the device drivers running on the host.

USB Core Subsystem

The main component of the USB architecture in the Linux kernel is the *USB core subsystem* [24]. It connects the device drivers with the host controller drivers. Figure 3.1 gives an overview of the USB core subsystem and its connected components.

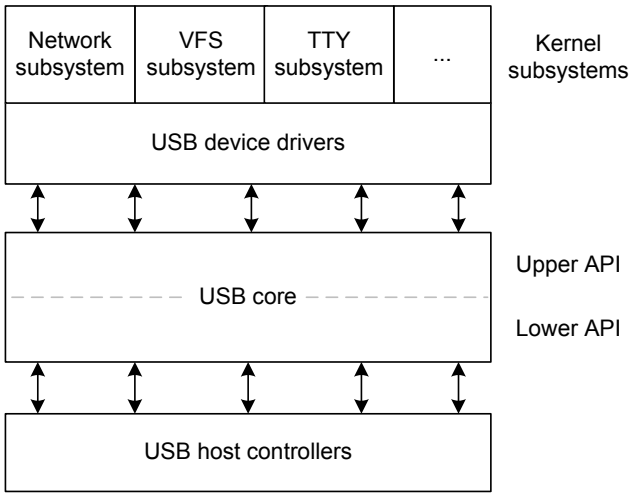

Figure 3.1: Linux USB core subsystem

The USB core has different responsibilities inside the driver architecture. It provides utility functions used by device drivers and host controller drivers to make their code as simple as possible. Additionally, it includes USB device drivers with an abstract interface to transparently access the hardware through any host controller connected to the USB core subsystem. The USB core can be divided into a lower part and an upper part with each part providing some specific API for the connected drivers. The lower part is used by the host controller drivers, while the upper part connects to the actual USB device drivers. The device drivers are free to utilize every kernel subsystem to provide their service.

Data is passed between the different drivers in the kernel using a data structure called the *USB Request Block* (URB). URBs are sent and received asynchronously by the actual USB device drivers to and from the endpoints of the devices. On their way to the endpoints, they pass through the USB core subsystem and host controller drivers. Each endpoint has a queue of URBs. This allows the device driver to submit new URBs before other URBs for the same endpoint have finished. The completion of a URB is signaled by the use of callback functions. URBs contain all of the information necessary to perform a specific data transfer. Figure 3.2 shows the transfer of an URB from the device driver to the host controller driver. First, the URB is created by the USB device driver. It's then assigned to a specific endpoint on a specific device. The URB is then transferred to the URB core, which is responsible to pass the URB to the correct host controller driver. The host controller driver processes the URB and carries out the USB transfer with the USB device. The device driver which initially submitted the URB is notified of the successful completion of the transfer by the use of a callback function.

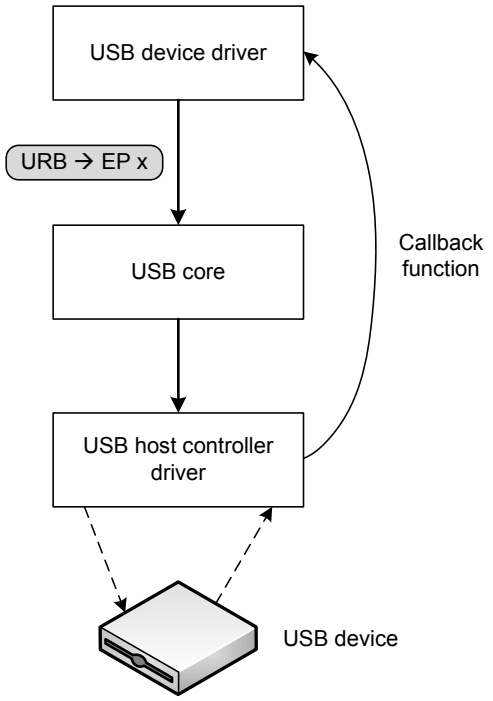

Figure 3.2: URB transfer passing through the USB core subsystem

Device drivers register at the USB core subsystem. Each device driver needs to provide some functions that are used by the rest of the USB core subsystem to control the device driver. Entry points to the functions are provided in the *USB device driver structure*. This structure is passed by each USB device driver to the USB core when it registers itself. Figure 3.3 shows some of the more important parts of the device driver structure.

The first field specifies the name of the USB device driver and is just some text string. The next two fields are function pointers to functions provided by the device driver. These two functions are used by the subsystem as entry points into the driver. They must be provided by every USB device driver. The `probe()` function is called whenever a new device is attached to the bus, which the respective driver should handle. Inside the `probe()` function, a device driver can do some last checks to see if the corresponding device can be really handled by this driver. After that, the device driver is responsible to create a new data structure for this instance of the device inside this function. The `disconnect()` function is called whenever a device is disconnected, which was previously bound to the driver. The device driver needs to deallocate all resources that were used for this instance of the device in this particular case. The next field is a function pointer to an `ioctl()` function provided by some device drivers. This function is only provided by device drivers that want to communicate with user–mode applications directly (see the description of the *usbfs* file system below). The last two fields of the device driver structure are pointers to additional structures. The first one points to a structure of supported file operations, which is used by some device drivers providing file access. The second structure is called the device ID table. Its purpose is to define, which devices should attach to this driver. It is described in detail in Section 3.1.2.

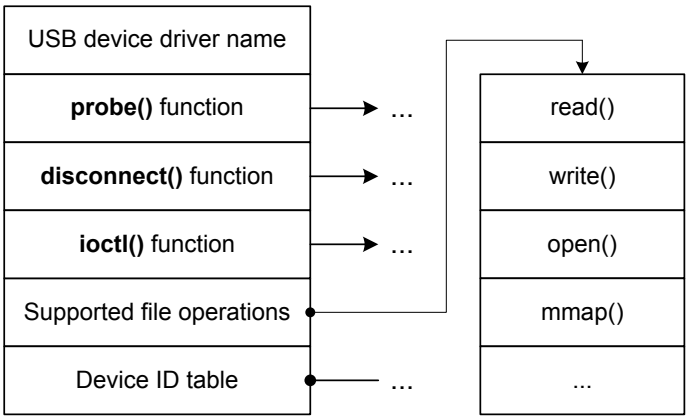

Figure 3.3: USB device driver structure

The USB core subsystem also offers a few functions to the device drivers. These include functions for selecting configurations and interfaces or functions for receiving different kinds of USB descriptors.

Host controller drivers also register at the USB core subsystem. They are responsible for handling the hardware details of performing a USB transaction and receive URBs from the device drivers through the USB core subsystem and perform the respective transaction. When they successfully finished the requested transaction, they signal the completion to the device driver by running the associated callback function.

Most USB device drivers are kernel–mode drivers. They are either statically linked into the kernel or compiled as separate kernel modules that can be dynamically loaded into the kernel. However, user–mode USB device drivers are possible as well. They utilize the *USB device file system* (usbfs). This is a file system that provides all the needed hardware details of attached USB devices to user–mode applications. The usbfs file system can be mounted at some directory such as `/proc/bus/usb/` where it represents each attached USB device as a group of filenames. Even USB devices, which don't have a corresponding device driver loaded, show up in the usbfs file system. By using `ioctl()` requests, a user–mode process can perform USB operations on the respective devices. Normally, user–mode processes don't access those files directly but use some USB user–mode libraries such as libusb [25] or jusb [26] instead.

Linux–USB Gadget API Framework

The Linux–USB Gadget API Framework [27] is a device driver framework inside the Linux kernel, which allows to program the USB device instead of the host. This framework makes it possible for embedded devices running Linux to act in the USB *device* (slave) role. It's available in the Linux kernel's 2.6 series as well as in version 2.4.23 and later. Embedded devices only need a USB peripheral controller to make use of the framework. Figure 3.4 shows a Linux–based USB device connected to a USB host. The USB peripheral controller is basically the complement of the USB host controller. There are no restrictions with respect to the kinds of devices that can be developed.

The framework is organized in three layers inside the kernel. From top to bottom they are:

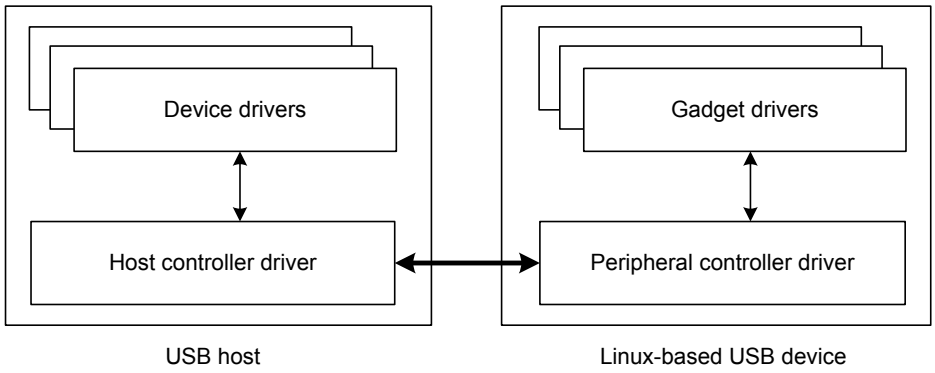

Figure 3.4: Linux–based USB device connected to a USB host

1. Upper Layers
2. Gadget Drivers
3. Peripheral Controller Drivers

The peripheral controller drivers represent the lowest layer in the kernel. This layer is the only one that talks directly to the hardware of the peripheral controller and abstracts all the hardware details. The peripheral controller driver is responsible for handling all of the endpoint data transfers between the peripheral controller and the gadget drivers. Only a small subset of the standard USB control requests are handled by the peripheral controller driver itself. Everything else, especially control requests to receive descriptors or set configurations, are passed to the higher layer gadget drivers.

Gadget drivers implement hardware–neutral USB functions. The bottom part of the gadget drivers utilize the peripheral controller drivers underneath. Responsibilities of the gadget drivers include the handling of setup requests from the host and returning descriptors and configurations. All endpoints are configured by the gadget driver, which also manages all IN and OUT transfers for the endpoints. Additionally, the gadget drivers are responsible for setting configurations and interfaces. The top part of the gadget drivers either connect to one or more of the upper layers in the kernel or directly to user–mode.

The upper layers represent every kernel layer that can be used by a gadget driver. This can be the network subsystem, the file system, or any other protocol stack inside the kernel. The gadget drivers use those upper layers to implement their real functionality. The upper layers act as producers and consumers of the data that passes through the gadget driver.

The Gadget API Framework has support for lots of different peripheral controllers. Among multiple highly–integrated processors, there is also a software–only controller called *dummy_hcd*. When using this controller driver, any loaded gadget driver will attach to the local system. Its main purpose is to allow developers to build and test gadget drivers, without the need to buy a hardware peripheral controller.

The framework already includes some gadget drivers, such as a file backed storage driver, a serial driver, or a MIDI driver. The usual gadget driver is running in the kernel. However, it's possible to build gadget drivers running completely in user–mode. For this purpose, the *gadgetfs* kernel

driver is included. It provides a file system, which when mounted, can be used by user–mode processes to communicate with the Gadget API Framework.

3.1.2 Enumeration

When a USB device driver is loaded, it registers with the underlying bus driver. The bus driver keeps a list of currently registered device drivers. Through the process of registration, the USB core is informed about some details of the device driver, which allows the USB subsystem to integrate the driver and make use of it. Information passed to the USB core include:

- A short name of the device driver for purposes of identification
- A `probe()` callback function, which is called for every attached device that is handled by this driver.
- A `disconnect()` callback function, which is called for every device that is detached from the bus.
- A *device ID table*, which is used to identify the devices that should be handled by this device driver.

The device ID table consists of multiple *match–specifier entries*. Each match–specifier entry specifies some information to match against and a bit field that indicates which provided information should be matched against.

A really simple match–specifier in the device ID table only includes a single vendor and product ID and a bit field that specifies only the vendor and product IDs should be matched against. Yet match–specifiers are not restricted to those elements (see below for a complete list). Even wildcard match–specifiers could be created, which would match against any USB device attached. In such a case, the drivers `probe()` function should check if the device should really be handled by this driver.

Matching is done in order of appearance in the device ID table. So entries appearing at the top of the table have a higher priority than entries appearing at the bottom. Device–specifiers range from really specific ones to the most general ones. In the following, we will examine the possible match–specifiers starting with the former ones.

The most specific match–specifiers use data from the USB device descriptor. This includes the vendor ID, product ID, and possible product revision numbers. These kind of match–specifiers are mostly used to assign product–specific device drivers as opposed to class drivers. Slightly more general matches are against the device class, the device subclass, or the device protocol numbers. Those match–specifiers are used for single–function devices where each interface doesn't have its own class. The most general matches are against the interface class, the interface subclass, or the interface protocol numbers. These matches let the driver bind to every interface of a multi–function device.

For more information about the matching process, the USB core function `usb_match_id()` inside the Linux kernel sources can be consulted. It contains all of the logic pertaining to the process of identifying devices supported by a device driver.

When a new USB device is attached to the bus, the bus driver iterates over its list of registered device drivers to find a matching driver that should service the attached device. Each interface

provided by the USB device could need a different driver. Thus, the matching process is done separately for every provided interface. The USB core does the actual matching process by using the device ID table provided by the device drivers during the process of registration. When a match is found, the provided `probe()` function of the device driver is called. Inside the `probe()` function, device drivers have a last chance to check if they really want to attach to the given device. The device driver then allocates and initializes some device structure used for this instance of the device and sets up that state of the device to make it functional. This includes setting up and starting the endpoints.

USB device drivers statically linked into the kernel are automatically registered in the USB core and thus are automatically used when a supported device is attached. Kernel modules on the other hand, are only registered when they are loaded into the kernel. This prevents USB devices from working when they attached to a system where the corresponding kernel module was not previously loaded. To deal with this problem, the *Linux hotplugging subsystem* was introduced. It is enabled by the kernel compile switch `CONFIG_HOTPLUG`. When the Linux hotplugging subsystem is activated, the kernel creates the new file `/proc/sys/kernel/hotplug` inside the proc file system. This entry contains the pathname to the kernel hotplug helper program, which is a user–mode application chosen by the user. This is usually just set to the default kernel hotplug helper `/sbin/hotplug` or left blank on newer Linux distributions as explained below. This user–mode application can be invoked by any kernel subsystem when the kernel wants to report configuration changes. The first argument passed to this application is always the name of the kernel subsystem that received the initial event. In the case of the USB subsystem, this is just the string `usb`. Additional arguments and environment variables can be set as needed but are all optional. In the case of USB, configuration change events are generated for the attachment and removal of USB devices. Table 3.1 lists all environment variables set for the called user–mode application.

Environment variable name	Content
ACTION	“add” or “remove”
PRODUCT	USB vendor, product, and version codes (hex)
TYPE	Device class codes (decimal)
INTERFACE	Interface 0 class code (decimal)
DEVICE*	Pathname of the device in the USB device file system
DEVFS*	Mount point of the USB device file system

Table 3.1: List environment variables passed to the kernel hotplug helper program

Environment variables marked with a trailing asterisk are only set if the USB device file system *usbfs* is configured in the kernel. The called kernel hotplug helper is usually configurable in user–mode. Depending on what application is used, tasks include loading needed kernel modules and invoking driver–specific setup scripts.

Since letting the kernel call into user–mode directly was not considered such a good design, starting with kernel version 2.6.14 a new mechanism that was based on the *udev* system [28] was introduced. This new mechanism should replace the old functionality. Instead of letting the kernel call some user–mode application directly, a kernel netlink socket was used to send out kernel device events. The user–mode daemon `udevd` listens on the netlink socket and acts upon receiving those events. `udevd` is highly configurable by the use of certain rules. The system administrator can add new rules that perform different tasks. There are multiple things a rule could do. Different actions could be performed, such as loading a new kernel module with a

needed driver. Additionally, events can just be passed on to some other subsystem, such as the *Hardware Abstraction Layer* (HAL) [29]. HAL builds upon the udev system to maintain a complete picture of the various hardware connected to the system. It provides some hooks, which can then be used by system– and desktop–level software to act accordingly. For example, when a new USB mass storage device is attached to the system, the device could be automatically mounted and a new window could be displayed showing the content of the file system to the user.

There are a few other solutions for automatically loading needed kernel modules, which are either based on the Linux hotplugging subsystem or are using other mechanisms to detect new device attachments. Nevertheless, the mechanism explained above is currently the most common one in use. Which mechanism is used to notify the user–mode of hardware device changes and which specific applications are used to handle the USB device attachment and detachment events is highly dependant on the Linux distribution used.

3.1.3 Supported Class Drivers

Every USB device driver that is either statically compiled into the kernel or is configured as a loadable kernel module can be used on a Linux system. Which drivers are actually enabled depends on the Linux distribution. Because of that, we list any USB class driver included in the Linux kernel. Device drivers marked as EXPERIMENTAL in the Linux kernel are indicated by a trailing asterisk. This could hint that such drivers are not so widely used since Linux distributions may choose to keep those drivers disabled. Table 3.2 lists all USB class drivers [30] included in the Linux kernel 2.6.24.

Class specification	Driver name	Class code
Hub class	(Part of `usbcore`)	0x09
Human interface devices (HID)	`usbhid`	0x03
Communications device class, ACM	`cdc-acm`	0x02
Communications device class, Ethernet	`usbnet`	0x02
USB Audio class	`snd-usb-audio`	0x01
MIDI device class	`snd-usb-audio`	0x01
Printing class	`usblp`	0x07
Mass storage class (MSC)	`usb-storage`	0x08
IrDA Bridge device class	`irda-usb`*	0xfe
Bluetooth class	`hci_usb`	0xe0

Table 3.2: USB class drivers included in Linux kernel 2.6.24

Some USB classes are supported by user–mode drivers, which can be obtained elsewhere. These are not included in the list. In addition to the USB class drivers, the Linux kernel contains a lot of other vendor USB drivers. For a complete list of supported drivers, refer to the kernel configuration of a recent Linux kernel.

3.2 Mac OS X

This section will focus on the latest release of Apple's operating system Mac OS X.

3.2.1 Driver Architecture

While Mac OS X is in parts based on Mac OS 9 and FreeBSD, the device driver model was completely redesigned. The reason for this was the lack of desired features in both device driver models. The device driver model of Mac OS 9 was lacking support for operating system features introduced with Mac OS X, such as memory protection, preemptive multitasking or multiprocessing. The device driver model of FreeBSD offered those features but was lacking other features considered important for Mac OS X. Those features included automatic device configuration, driver stacking, power management and the dynamic loading of device drivers [31].

The *I/O kit framework* was introduced with Mac OS X, which promised to provide all of those features. It is an object–oriented framework that can be used to develop device drivers on Mac OS X. It is provided as part of the *kernel development kit* (KDK). The I/O kit framework can be used to develop kernel–mode drivers and the corresponding user–mode applications to access those drivers. It consists of multiple components, including different frameworks, libraries, tools and other resources for creating device drivers for Mac OS X. The following frameworks and libraries are provided:

- *Kernel/IOKit* is a library used for the development of kernel–mode device drivers. Its headers can be found at `Kernel.framework/Headers/IOKit`.
- *Kernel/libkern* is a library containing classes, which can be used for general kernel programming tasks. It provides commonly needed functions such as arithmetic/logical operations guaranteed to be atomic, byte swapping routines and classes for common data structures. The I/O kit is based on the libkern library. The headers for the libkern library are located at `Kernel.framework/Headers/libkern`.
- *IOKit* is a framework, which is used to develop device interfaces. It can be found at `IOKit.framework`.

The I/O kit framework is implemented in a restricted subset of C++, which is suitable for use in a multithreaded kernel environment. Its complete source code is available as part of the Darwin project [32]. The I/O kit consists of multiple components that are described in the following sections.

I/O Kit Families

An I/O kit family is a C++ class providing software abstraction to multiple devices of a specific type. There are I/O kit families for all kinds of devices. A separate I/O kit family exists for USB devices. I/O kit families include code for common tasks, which basically needs to be performed by every device driver of a specific class.

Drivers

There exist basically two kinds of driver objects in the I/O kit framework.

Device driver objects are the actual drivers for specific hardware devices or services. By the use of inheritance, device drivers become members of specific I/O kit families, while new device drivers can concentrate on the device specific tasks instead of duplicating the work of all other

device drivers of the same class. Driver objects are created as kernel extensions (KEXT) and are stored in most cases in the extensions folder `/System/Library/Extensions`.

Nub objects are the other kind of driver object in the I/O kit framework. They provide communication channels for device driver objects to the underlying bus. A nub is an object published by the family of the communication bus where the device is attached. Device drivers inherit from a family class and communicate through the use of a nub object. It's the main way of communication for the device driver. Every form of I/O between the device driver and the bus passes through the nub object. Nub objects are sitting between every driver object and connect them. Figure 3.5 gives a simplified overview of the driver objects involved when a single USB device is attached to a USB host controller that is connected to the systems PCI bus. The device driver objects are all connected by the use of nub objects.

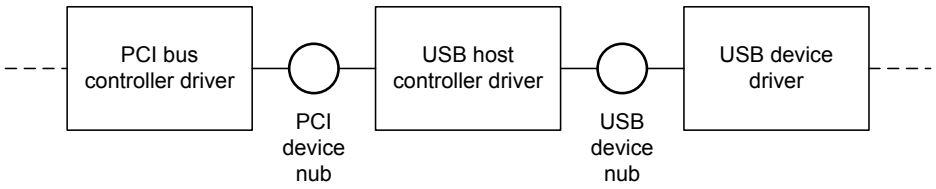

Figure 3.5: Driver objects connected through nub objects

Another responsibility of the nub object is to match newly attached devices against drivers to load the correct driver. See Section 3.2.2 for all details related to the loading of device drivers.

I/O Registry and I/O Catalog

The *I/O registry* is a dynamic database that records every driver object and the relationships between all objects. Each driver has to register in the I/O registry to work with the I/O kit. When a new USB device is attached or removed from the system, the I/O registry is immediately adjusted to reflect the current situation of loaded drivers. The I/O registry only resides in system memory. It is not stored on disk nor is it archived over reboots. The I/O registry is structured as an inverted tree. Each node of the tree represents one of the driver objects. It is either a device driver or a nub driver object. The I/O registry can be accessed from user–mode through functions exported by the I/O kit. User–mode processes use this functionality to communicate with attached devices.

The *I/O catalog* is another dynamic database working closely together with the I/O registry. It contains all device drivers, available on the whole system. Currently loaded and unloaded drivers are stored in the I/O catalog.

Device Interfaces

Most device drivers run in kernel–mode. To access an attached USB device from user–mode, the kernel provides so called *device interfaces*. A device interface is a plug–in interface between the kernel and a user–mode process. It adheres to the standards of the plug–in architecture, which is defined by the *Core Foundation Plug–in Services* (CFPlugIn). The kernel acts as the host of plug–ins and provides well–defined I/O kit interfaces to those. A few plug–ins (device interfaces) are provided by the I/O kit framework, which can be used by user–mode processes. Figure 3.6

shows the communication between a user–mode application and a kernel device driver through a device interface.

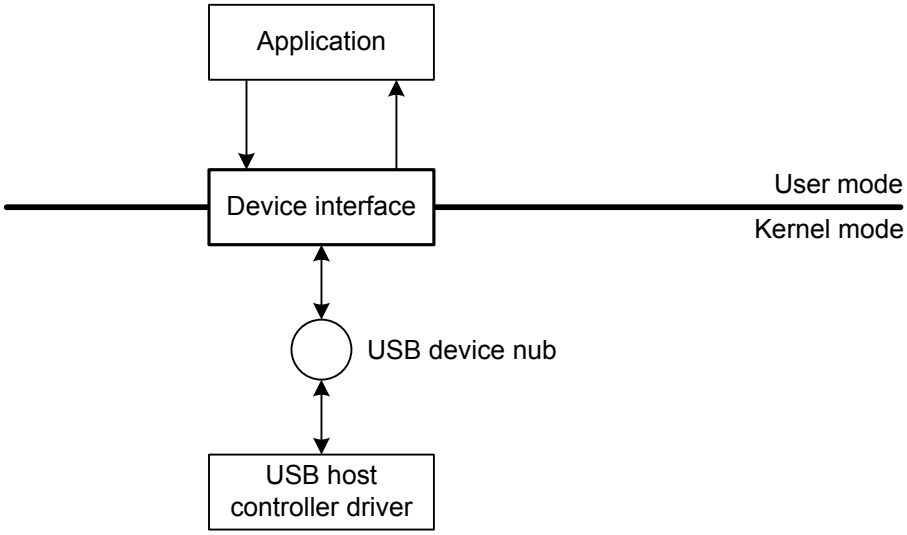

Figure 3.6: Application controlling a USB device from user–mode

To obtain a device interface, a user–mode process does a search in the I/O registry. This process is called *device matching*. A device interface provides a user–mode process with a table of function pointers. By calling those functions, a user–mode process can exchange data with the kernel. Inside the kernel, a device interface can be considered just another driver object. The nub object of the respective bus provides the device interface.

Another possibility for user–mode processes to communicate directly with attached devices are *POSIX device files*. But this mechanism is not used for USB devices and thus not further discussed.

3.2.2 Enumeration

One task of the nub objects is to provide matching services, which allows the I/O kit to assign the correct driver to an attached device. Starting with Mac OS X, device drivers are only loaded when a corresponding device is attached. Before Mac OS X, device drivers were pre–loaded whether or not a device of that kind was already attached.

To allow the I/O kit to match attached devices against available device drivers, each device driver provided as a loadable kernel extension must provide one or more *personalities*. A personality specifies the kind of devices that can be handled by the driver. The personality is stored in *XML matching dictionaries* that are defined in the *information property list* (Info.plist) inside the KEXT bundle of a driver. The information property list stores contents, settings and requirements of the driver. XML matching dictionaries are structured as key–value pairs of XML tags. The `<key>` tag specifies the name of the key. The value is specified inside tags that identify the type of the data. Examples are `<integer>42</integer>` or `<string>Some string </string>`. There are a few important keys that must be specified in each device driver. The `IOProviderClass` key is one of those. Its value identifies the nub class where the driver should

attach.

All match keys of a personality must match in order to match the whole personality. There are several causes for a driver to have multiple personalities. There could be multiple versions of a device that should be handled by the same driver. Another reason for multiple personalities could be a USB device with multiple interfaces where a separate personality would be provided for each interface. Although a device driver can have multiple personalities, the match of a single personality forces the loading of the corresponding driver.

Figure 3.1 shows an example of a XML matching dictionary with two personalities. It's a shortened version directly taken from the AppleUSBAudio driver. The first personality called `AppleUSBAudioControl` would match if both of the USB descriptor fields `bInterfaceClass` and `bInterfaceSubClass` are set to 1. The second personality also includes the product and vendor IDs for the matching process.

When a new USB device is attached to the system, the USB host controller driver detects the event and creates a new nub for each detected USB device. The I/O kit then starts the device match process. Values needed for the matching are extracted from the USB descriptor. The matching process consists of three steps. The process starts with a list of lots of potential drivers and is slowly decreased by every step until the perfect match is found. Each step of the device matching process is explained below:

1. **Class matching:** First, the I/O kit requests a list of all device drivers with the correct class type from the I/O catalog. In the case of a USB device, all device drivers with a class type of "USB" are put into the pool of potential device drivers.

2. **Passive matching:** The personality of each remaining device driver is matched against the attached device. Every driver that doesn't match is removed from the list.

3. **Active matching:** Every remaining driver probes the attached device to check if it can serve it. This is done by calling the drivers `probe()` function with a reference to the nub of the attached device. The `probe()` function communicates with the device through the nub to decide if the driver really can control the device. The driver then returns a *probe score* that indicates the ability of the driver to control the device. The probe score is a signed 32–bit integer, which is initialized to a default value in the personality of a driver or set to 0, if it's not defined there. It can be adjusted by the driver inside the `probe()` function. The highest probe score represents the best match. After the `probe()` function of every remaining driver is called, the I/O kit sorts all device drivers using their probe score.

After the last step of the driver matching process, the I/O kit chooses the device driver with the highest probe score and tries to start it. If the driver is successfully started, it is added to the I/O registry and the matching process is done. When starting the driver fails, the driver with the next best probe score is attempted to be started. This process continues until a driver successfully starts up or no more potential drivers are remaining, in which case no driver is loaded.

3.2.3 Supported Class Drivers

Mac OS X includes several USB class drivers that are provided as kernel extensions. These are loaded when a matching device is attached at the system. Table 3.3 lists all USB class drivers included in Mac OS X 10.5.

```
1  <key>IOKitPersonalities</key>
2  <dict>
3          <key>AppleUSBAudioControl</key>
4          <dict>
5                  <key>CFBundleIdentifier</key>
6                  <string>com.apple.driver.AppleUSBAudio</string>
7                  <key>IOClass</key>
8                  <string>AppleUSBAudioDevice</string>
9                  <key>IOProviderClass</key>
10                 <string>IOUSBInterface</string>
11                 <key>bInterfaceClass</key>
12                 <integer>1</integer>
13                 <key>bInterfaceSubClass</key>
14                 <integer>1</integer>
15         </dict>
16         <key>AppleUSBTrinityAudioControl</key>
17         <dict>
18                 <key>CFBundleIdentifier</key>
19                 <string>com.apple.driver.AppleUSBAudio</string>
20                 <key>IOClass</key>
21                 <string>AppleUSBTrinityAudioDevice</string>
22                 <key>IOProviderClass</key>
23                 <string>IOUSBInterface</string>
24                 <key>bConfigurationValue</key>
25                 <integer>1</integer>
26                 <key>bInterfaceNumber</key>
27                 <integer>0</integer>
28                 <key>idProduct</key>
29                 <integer>4353</integer>
30                 <key>idVendor</key>
31                 <integer>1452</integer>
32         </dict>
33 </dict>
```

Listing 3.1: Example of a XML matching dictionary containing two personalities

Class specification	Driver name	Class code
Bluetooth class	`AppleUSBBluetoothHCIController`	0xe0
IrDA Bridge device class	`AppleUSBIrDADriver`	0xfe
Hub class	`AppleUSBHub`	0x09
Human interface device (HID)	`IOUSBHIDDriver`	0x03
Communications device class, Ethernet	`AppleUSBCDC`	0x02
Communications device class, Serial	`AppleUSBCDC`	0x02
Mass storage class (MSC)	`IOUSBMassStorageClass`	0x08
USB Audio class	`AppleUSBAudio`	0x01
MIDI device class	`AppleMIDIUSBDriver`	0x01

Table 3.3: USB class drivers included in Mac OS X 10.5

In addition to the provided class drivers, OS X provides several vendor drivers. A complete list of all provided kernel extensions can be found in the directory `/System/Library/Extensions`.

3.3 Windows XP

The first Microsoft Windows operating system with reliable USB support was Microsoft Windows 98 [5]. With each new release, support was improved and more class drivers were added. Although this section focuses on Windows XP, lots of concepts that are the same on other versions of Microsoft Windows are explained as well. This is especially the case with Windows Vista, in which only a few things have changed (see Section 3.4 for the differences).

3.3.1 Driver Architecture

The foundation of the Windows driver architecture is the Microsoft Windows I/O system [33]. It consists of multiple components that work together to manage all kinds of hardware devices and provide interfaces to user–mode applications. Figure 3.7 shows some of the components of the I/O system and their interaction. Each component is described in more detail below.

I/O Manager

The I/O manager is a main part of the I/O system. Each I/O request passes through the I/O manager. Communication takes place using *I/O request packets* (IRPs). IRPs are data structures that describe single I/O requests. Instead of passing all arguments related to a request separately, such as buffer addresses and buffer sizes, they are stored inside an IRP and only a single pointer to this IRP is passed inside the I/O system from component to component. Figure 3.8 shows the path of a typical I/O request from a user–mode application to the hardware device and back.

When a user–mode application requests an I/O operation, this request first passes through a subsystem DLL. The subsystem DLL communicates with the I/O manager in kernel–mode, which then allocates an IRP that represents the requested I/O operation. The I/O manager sends the IRP to the respective device driver, which extracts the data from the IRP and starts the I/O operation with the hardware device. When the operation is complete, the device driver calls back into the I/O manager passing back the IRP. By doing this, the device driver acknowledges

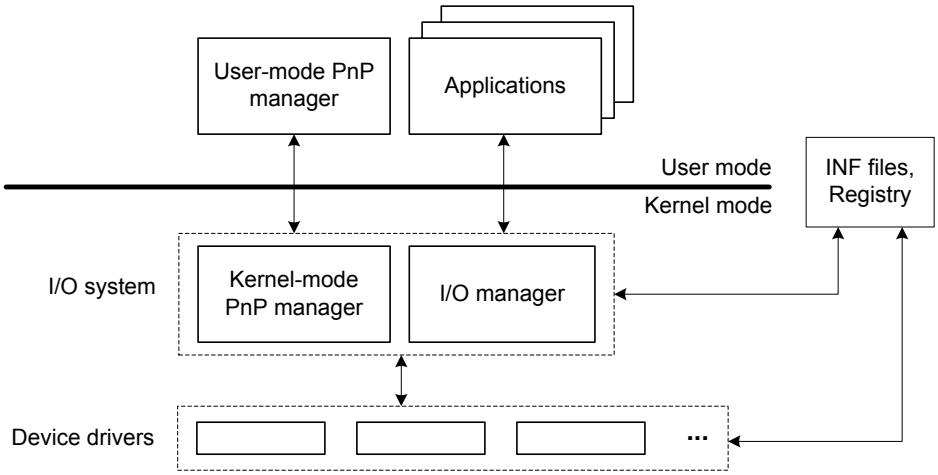

Figure 3.7: Microsoft Windows I/O system

the completed operation. When only a single device driver is involved, as in Figure 3.8, the I/O manager marks the operation as complete and notifies the user–mode application. If more drivers are involved, a driver can also request that an IRP should be passed to another driver. This request is sent to the I/O manager, which then forwards the IRP to the desired additional driver.

In addition to the previously mentioned tasks, the I/O manager also provides some I/O support functions to the device drivers. Device drivers can make use of those to easily handle their I/O processing. This can simplify the actual code of the device driver.

Device Drivers

Windows uses the *Windows Driver Model* (WDM), which is a framework for the development of unified device drivers on the Microsoft Windows operating system. It was introduced with Windows 98 and Windows 2000 and was intended to ease the development of new device drivers and to provide a unified driver model. WDM device drivers are source–compatible and in many cases binary–compatible too. There are basically two possibilities to create a WDM device driver:

- The *Windows Driver Kit* (WDK) [34] is a fully integrated device development system. It contains everything needed to develop all kinds of drivers on the Windows platform. For the development of device drivers it contains the *Driver Development Kit* (DDK), which can be used to create WDM device drivers.
- Another possibility is to use one of the available driver toolkits such as WinDriver USB Device toolkit from Jungo Ltd. or DriverWorks from Compuware Numega [5]. These toolkits try to ease the development of a new device driver. Very simple device drivers with no special requirements can sometimes be created even without any programming at all.

Inside the operating system, device drivers are represented by two kinds of objects. These are *driver objects* and *device objects*.

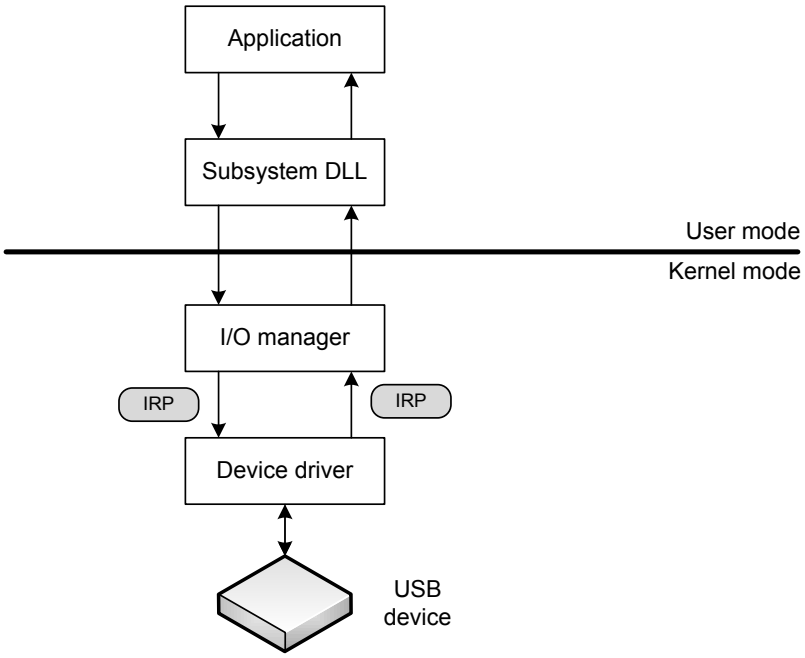

Figure 3.8: Path of a user–mode I/O request through the I/O system

Driver objects represent a single driver inside the operating system. The whole interface of the driver is provided through this kind of object. When a new driver is loaded into the system, the I/O manager creates the corresponding driver object. There is always only one driver object per driver.

Device objects on the other hand represent a physical or logical device. The characteristics of a device are stored inside the device object. As the name suggests, device objects exist for each device attached to the computer. They are allocated by the *Add-device* routine (described below) of a function driver whenever a new device is detected by the PnP manager. Every device object has a pointer back to its corresponding driver object. This allows access to the driver interface of a specific device object.

There are three different types of WDM drivers:

1. *Bus drivers* are responsible for managing a bus such as the Universal Serial Bus. Their tasks include the detection of new device attachments and their removals. The bus driver reports those events to the PnP manager, which then handles the events.

2. *Function drivers* handle the actual devices. Every function driver exports a unified driver interface to the operating system that is utilized by the rest of the I/O system. The driver interface includes an initialization routine (typically named *DriverEntry*), which is called directly by the I/O manager when the driver is loaded. Whenever a new device is detected, the PnP manager calls the *Add-device* routine of the driver interface that then allocates a new device object. The main functionality of the function driver is provided through multiple dispatch routines exposed through the driver interface. Dispatch routines receive

an IRP generated by the I/O manager and perform their designated task with the data inside the IRP. Common dispatch routines include `open()`, `close()`, `read()` and `write()`.

3. *Filter drivers* can optionally be used to influence the behaviour of a device or driver. They sit above or below a function driver and any data passing through the function driver passes through the filter drivers as well. By modifying the data, filter drivers can influence aspects of the communication.

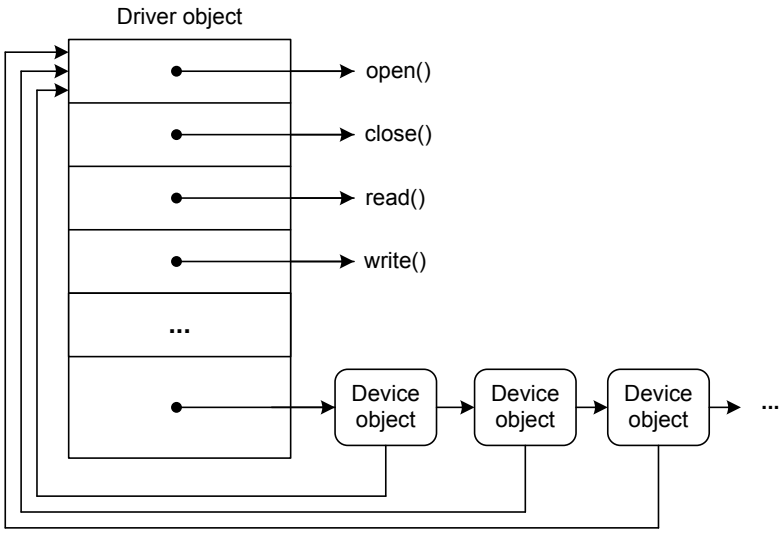

Figure 3.9: Driver object and its exposed driver interface

Plug and Play (PnP) Manager

The PnP manager is the component inside the I/O system that is responsible for detecting and adapting to changing hardware configurations. It works together with the I/O manager and the USB bus drivers. The PnP manager is responsible for installing and loading device drivers when a new device is detected. The PnP manager is split up into a user–mode and a kernel–mode component, as can be seen in Figure 3.7.

When a new device is detected on the bus, the USB bus driver notifies the PnP manager of this event. The kernel–mode PnP manager then checks if an appropriate driver is available on the system. If a driver is found, the kernel–mode PnP manager instructs the I/O manager to load the driver. If no driver is found, the kernel–mode PnP manager instructs the user–mode PnP manager to install a matching driver. Before a loaded device driver actually starts talking to the hardware, the PnP manager has to send a *start-device* command to the PnP dispatch routine of the driver. After that, the device driver is fully functional.

Driver Installation Files

Driver installation files are used before a driver is used the first time. They are often called INF files, because they are stored with the `.INF` file extension. INF files basically describe a device or

a class of devices and how to install their respective driver. The most important items included in an INF file are:

- Source and target location of driver files
- Device driver files to be installed
- Registry modifications to be done

In addition, some co–installer DLL files can be specified optionally. They are executed after the driver is successfully installed. They can be used to customize the device installation process. For example, a configuration window could be displayed to give the user a chance to do some post–setup configuration.

Every device driver that comes with Windows is listed in at least one INF file. INF files are text–based files and are by default located in the `%SystemRoot%\inf` directory. Even INF files for third–party drivers are stored in this directory. When the driver is installed, the INF files are copied there. They can be identified by their name of `oem*.inf`, where `*` is an increasing number starting with 0. Every INF file has a corresponding *precompiled INF* file, which has the file extension `.PNF`. The PNF files are machine–readable forms of the INF files, which allow faster lookups.

When the PnP manager searches for a new driver to be installed, by default it only looks in the above directory for a matching INF file. But additional paths can be specified by modifying the registry key `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\DevicePath`. Even network drives could be added there, which would allow the installation of new device drivers from a network share.

Registry

When a new device is detected for the first time and the device driver is installed, some information about the device is stored in the registry. This allows Windows to detect that a device driver was already installed and load the driver directly the next time the device is attached. Information about installed USB device drivers is spread across multiple registry keys [5]. The most important ones are described below.

Hardware keys store information about all devices detected since the system was initially installed. They are stored below the USB enumeration key `HKLM\SYSTEM\CurrentControlSet\Enum\USB`. This key contains sub–keys that the vendor ID and product ID are each a part of. Below these sub–keys are the actual hardware keys. Each hardware key corresponds to one instance of a device with the same vendor ID and product ID. The names of the hardware keys represent the serial numbers of the devices.

A serial number is an optional per–device unique 64–byte string that is stored in the device descriptor and is used by the operating system to distinguish different instances of the same device.

Devices which have a USB serial number only create a single hardware key the first time they are attached. When the same device is attached at another port, it's identified as the same by it's serial number. Devices without a serial number create a new hardware key each time they are attached to a port they were not attached before. This is because the host has no possibility to

differentiate the device from another instance of the same device because all the USB descriptors are the same.

Each hardware key has a `Class` value that describes the class of the device as a string, and a corresponding `ClassGUID` value that references a *class key*. Class keys are stored below the registry key `HKLM\System\CurrentControlSet\Control\Class`. Optional entries inside the class key can influence what users will see when the device is installed. If the `NoInstallClass` entry is present and not set to 0, users are not required to manually install a driver. If the entry `SilentInstall` is set, installers will never show pop–ups that require a response to the user. The `UpperFilters` and `LowerFilters` entries can be used to specify filter drivers, which apply to all drivers of this class.

Each hardware key also references a *driver key* with the `Driver` entry. Driver keys are referenced by a class GUID followed by a device instance number. The device instance number matches a driver key. Driver keys are stored as sub–keys in the corresponding class key. They store information about the drivers assigned to the instances of devices in that class. Driver keys contain the name of their INF file that reference the device driver files.

The last registry key of interest is the *service key*. Service keys are stored below the `HKLM\System\CurrentControlSet\Services` key. Service keys store information about the driver files. This includes the locations where they are stored and how to load them.

Accessing USB Devices From User–Mode

User mode applications access USB devices by using a file handle. To obtain the file handle, the application must open a specific symbolic link corresponding to the desired interface of a device. USB device drivers expose one or more interfaces identified by a *globally unique identifier* (GUID), which is a 128–bit value. To obtain the name of one of the symbolic links, the application can call one of the Plug and Play setup functions (`SetupDi*`) with the desired GUID as an argument. These functions return a device object that contains the name of the symbolic link. The filename can be opened using the Windows function `CreateFile` to obtain the needed file handle. Using the obtained file handle, a user application can then communicate with the device using Windows functions like `ReadFile()`, `WriteFile()` and `DeviceIoControl()`.

3.3.2 Enumeration

The enumeration process starts when a new device gets attached to the bus. The device attachment is detected and the descriptors are read from the device by the USB bus driver. Using values from the read device descriptor, a *device identification string* is constructed. This string is reported to the PnP manager, which does a lookup in the registry for a hardware key.

If the device was attached before, the previously stored hardware key is found. By using the `ClassGUID` value from the found hardware key, the PnP manager locates the class key in the registry. The hardware key together with the class key give the PnP manager all the information needed to load the corresponding driver. Lower–level filter drivers can be specified in the `LowerFilters` value of the hardware key or class key and will be loaded before the function driver is loaded. The `Service` value in the hardware key specifies the actual function driver to be loaded. Upper–level filter drivers can be specified in the `UpperFilters` value of the hardware key or class key and will be loaded after the function driver is loaded. All of these drivers are referenced by their service key and are loaded by the PnP manager.

If the device was not attached before or was attached at another USB port where it wasn't attached before and doesn't provide a serial number, PnP manager doesn't find a hardware key in the registry and delegates the task of finding a driver to the user–mode PnP manager.

The user–mode PnP manager first tries to automatically install a driver. This is done by searching through all the INF files to find a matching driver.

Windows first uses the device identification string to look for a matching *device ID*. The device ID is constructed by the HUB driver out of the vendor ID, product ID and the revision number from the USB device descriptor. Device IDs have the following form: `USB\VID_xxxx&PID_yyyy &REV_zzzz`. `xxxx`, `yyyy` and `zzzz` represent the vendor ID, product ID and revision number in hexadecimal form respectively. Note that the revision is optional and can be omitted. If a device has multiple interfaces and a different driver for each interface, a `&MI_ii` can be included in the device ID, when `ii` specifies the interface number as set in the `bInterfaceNumber` field in the interface descriptor.

When no matching device ID is found, Windows looks for a *compatible ID* match. Compatible IDs are constructed out of the class code, subclass code and the protocol code from the device descriptor. Compatible IDs have the following form: `USB\CLASS_aa&SUBCLASS_bb&PROT_cc`. `aa`, `bb` and `cc` correspond to the `bDeviceClass`, `bDeviceSubClass` and `bDeviceProtocol` fields in the device descriptor respectively.

Device IDs are usually used for third party vendor drivers, while class drivers supplied with Windows are usually found through a compatible ID match.

To find either a device ID or a compatible ID match a ranking system is used. Every found match is assigned a rank. Lower ranks indicate a better match. Signed drivers receive lower ranks than unsigned ones. See Table 3.4 for possible match priorities.

Rank	Signed INF file	Kind of match
0	Yes	Device ID matches hardware ID
1	Yes	Device ID matches compatible ID
2	Yes	Compatible ID from device matches hardware ID
3	Yes	Compatible ID from device matches compatible ID
4	No	Any match

Table 3.4: Device driver match priorities

When no match is found and the currently logged on user has administrator privileges, PnP manager launches `Rundll32.exe` to execute the *Hardware Installation Wizard*, which gives the user a chance to specify the location of a matching INF file. When the user doesn't have administrator privileges or no user is currently logged on, the user–mode PnP manager defers the installation until a privileged user logs on.

When the best match is an unsigned driver, operating system settings decide, what will happen. Windows either prevents the installation of the driver, warns the user and gives him the choice what to do, or just installs the unsigned driver without any warning. In the default setting, Windows XP will warn the user and give him the choice to install the driver anyways.

After Windows found a matching INF file, the actions in the INF file are executed and eventually any existing class or device co–installer DLLs listed in the INF file are run by the Hardware Installation Wizard. During this process, some information about the device is stored in the registry in the hardware key and the class key [5]. After the driver is successfully installed, the

kernel–mode PnP manager instructs the I/O manager to load the driver.

3.3.3 Supported Class Drivers

Windows XP comes with a number of pre–installed class drivers [35]. Table 3.5 lists all class drivers that are available on a default Windows XP installation. Driver names marked with a trailing asterisk are not available in all installations of Windows XP. The CCID class driver `usbccid.sys` ships with Service Pack 2 and the video class driver `usbvideo.sys` is only available from Windows Update and thus can only be expected to be present when a video class device was successfully attached before.

Class specification	Driver name	Class code
Bluetooth class	`bthusb.sys`	0xe0
Chip/smart card interface devices (CCID)	`usbccid.sys`*	0x0b
Hub class	`usbhub.sys`	0x09
Human interface devices (HID)	`hidusb.sys`	0x03
Mass storage class (MSC)	`usbstor.sys`	0x08
Printing class	`usbprint.sys`	0x07
Scanning/imaging (PTP)	`wpdusb.sys`	0x06
Media Transfer (MTP)	`wpdusb.sys`	0x06
Audio class	`usbaudio.sys`	0x01
Communications device class, Modem	`usbser.sys`	0x02
Video class (UVC)	`usbvideo.sys`*	0x0e

Table 3.5: USB class drivers included in Windows XP

Windows XP does not only provide USB class drivers in its default installation. Other non–class drivers are provided as well. For a complete list of all provided USB drivers see the `%SystemRoot%\inf` directory.

3.4 Windows Vista

This section describes the USB support in Windows Vista. Although some things have changed compared to previous versions of Windows, most of the changes only concern the API to build new device drivers. The implementation inside the kernel has mostly stayed the same. So instead of duplicating all the information, this section will only focus on the differences in Windows Vista compared to previous versions of Windows. Please see Section 3.3 for a thorough description of all the remaining parts of the USB architecture that are common to both Windows XP and Windows Vista.

3.4.1 Driver Architecture

Windows Vista is also based on the *Windows Driver Model* (WDM) just like previous versions. The *Windows Driver Kit* (WDK) can be used to develop device drivers. But with Windows Vista, the WDK includes a new framework called the *Windows Driver Foundation* (WDF) [36]. The WDF is an object–oriented, event–driven device driver model that greatly simplifies the

development of new device drivers by providing simpler driver interfaces. Common driver tasks such as Plug and Play (PnP) or power management that had to be handled by each WDM device driver itself, are handled automatically in the WDF framework. This reduces the lines of code and makes WDF drivers easier to debug. WDF drivers are compatible with Windows 2000 and later versions. Under the hood, the WDF framework communicates with the operating system by using the old WDM interfaces. WDF drivers don't talk directly to the operating system components such as the PnP manager or the I/O manager. Instead, they are talking to the WDF framework, which then communicates on behalf of the driver with kernel components of the I/O system as explained in Section 3.3.1. The WDF consists of two frameworks that can be used to create different kinds of device drivers. Both frameworks are described below.

Kernel–Mode Driver Framework

The *Kernel–Mode Driver Framework* (KMDF) allows the development of kernel–mode drivers that conform to the WDF model. Kernel–mode drivers are used in cases in which the driver needs to handle interrupts, perform DMA operations, or needs access to other kernel–mode resources.

The KMDF was designed as a replacement for the WDM for driver developers. It supports nearly all the devices and device classes the WDM supports. Many of the WDM sample drivers included in the DDK where converted to KMDF drivers that are smaller and less complex. The KMDF defines an object–oriented driver model. Common driver constructs are represented as objects, which KMDF device drivers can utilize. Drivers can only interact with those objects using the provided methods and properties. Some of those objects are created by the framework, while some are explicitly created by the device driver itself. Device drivers utilize callback functions to get notified of important events. The framework handles memory allocations and important driver data structures. This allows the driver to concentrate on the device–specific details. Additionally, the framework introduces *counted Unicode strings* to help prevent string–handling errors such as buffer overflows that were introduced in Section 2.2.1. All KMDF interfaces are designed to make the isolation of kernel–mode drivers possible in the future. Such an isolation would allow a driver to run in a protected environment where crashes off the driver would not affect the whole system.

All KMDF drivers have a similar structure. They all have a `DriverEntry` function, which is the primary entry point of the driver. Every PnP–capable device driver has an `EvtDriverDeviceAdd` callback that is called when the PnP manager enumerates a new device for this driver. Additionally, KMDF drivers have one or more `EvtIo*` callbacks that are called to handle different kinds of I/O requests. Those are the minimum requirements for a KMDF device driver. Everything else is handled by the framework. Device drivers can have additional code if they want to support more device–specific features.

User–Mode Driver Framework

The *User–Mode Driver Framework* (UMDF) allows the development of device drivers that are running completely in user–mode. This presumes that the driver does not need direct access to hardware or kernel resources such as interrupts or DMA. The benefit of a user–mode driver is the separation from the kernel. A crash in the driver doesn't directly affect the stability of the kernel and thus doesn't affect the whole system.

The UMDF tries to ease the development in the same way as the KMDF does. It offers intelligent

defaults so that a driver developer only needs to write code for the device–specific functionality of the device. The framework provides everything else. Figure 3.10 illustrates the UMDF architecture and shows two loaded UMDF device drivers.

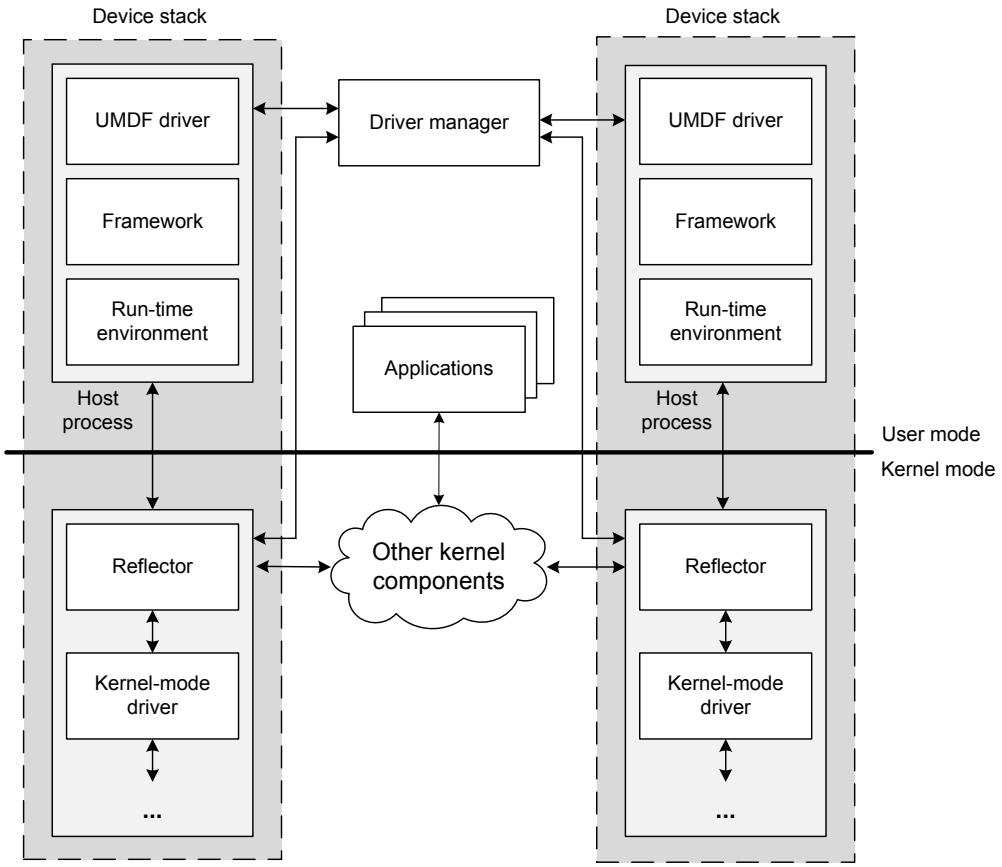

Figure 3.10: UMDF architecture

Every UMDF driver runs in a separate *host process* in user–mode. The host process runs with the security credentials of the `LocalService` account. It contains the whole user–mode device stack, which is separated into three parts:

1. UMDF driver
2. Framework
3. Run–time environment

The UMDF driver is an in–process component object model (COM) [37] running inside the host process and controlling the hardware from user–mode. Every host process contains the framework, which provides the device driver interface (DDI) exposed to user–mode drivers. It is linked in into the host process as a dynamic linked library (DLL). The run–time environment is responsible for the host process. It manages the user–mode thread pool, loads the UMDF driver,

and constructs and destroys the user–mode device stack. All I/O requests are passing through the run–time environment that dispatches those requests. In addition it communicates with the other components such as the *reflector* and the *driver manager*.

The driver manager is a Windows service, which is responsible for creating and managing all the driver host processes. The driver manager service is started the first time a device is attached, which is handled by a UMDF driver. The driver manager also responds to messages from the reflector.

The reflector is at the top of the kernel–mode device stack. Implemented as a WDM filter driver, it connects the top of the kernel–mode device stack with the bottom of the user–mode device stack. I/O requests are forwarded between kernel–mode drivers and the user–mode driver host process by the reflector. Additionally, the reflector forwards other events such as power or PnP messages from the kernel to the driver host process. This allows the UMDF driver to participate in the driver enumeration process and to respond to PnP events accordingly.

Applications in user–mode can't talk directly to UMDF drivers. They access UMDF drivers the same way they would access other device drivers. They perform I/O requests by using the standard Microsoft Win32 File I/O API.

3.4.2 Enumeration

Device enumeration works similarly here to how it does on Windows XP. Although there are some differences, especially in regard to the driver signing process, those changes are not relevant for this thesis and thus are skipped. Please refer to Section 3.3.2 for details on USB device enumeration on Windows XP and Windows Vista.

3.4.3 Supported Class Drivers

Windows Vista comes with the same USB class drivers as provided by Windows XP (see Table 3.5 for the complete list). All class drivers marked with a trailing asterisk, which where not available on all installations of Windows XP, are now provided with any default Windows Vista installation.

For all the USB non–class drivers provided by Windows Vista, please refer to the Windows Vista driver store, which holds all device drivers that can be installed without user intervention when an appropriate device is attached. The driver store can be found in the directory `%SystemRoot%\system32\DriverStore`.

Chapter 4

Attack Vectors

This chapter first introduces different real–world attack scenarios in Section 4.1, which should make it clear that the Universal Serial Bus can be utilized for attacks against the host system. Section 4.2 then classifies attacks in different categories. Each category is first described in detail, followed by some related attacks.

4.1 Attack Scenarios

In the case of the USB 2.0 standard [3], an attacker needs physical access to a system. This might change, however, with the Certified Wireless USB (CWUSB) extension [1] that introduces wireless USB. Although nearly every system can be broken into with enough physical access, USB ports represent a special case. Often the system itself together with human interface devices such as keyboards and mice are protected against unauthorized access. On the other hand, USB ports are often considered safe to be provided to the user. In some cases, USB ports must even be provided to the user to accomplish the task of the respective system. Hardware security tokens are one example, which are small hardware devices that are used by users to authenticate to computer systems or even physical buildings. Different vendors such as VeriSign, Aladdin or SafeNet offer many different variants of such tokens. Lots of these tokens are USB–based and thus a USB port must be provided in order for a potential attacker to have a chance to attach a malicious USB device. The problem is that such authentication solutions are implemented in the first place to secure systems, which have a higher need for security. By making the USB port available to untrustworthy users, this opens up an attack vector that might give an attacker a way into critical systems.

Another example is kiosk print systems that can be found at public places such as shopping malls. Those systems either allow customers to print their own photos right away or to commission them for later pickup. Photos can be provided by the customer on different kinds of storage mediums. In most cases, those systems offer a USB port for USB mass storage devices. Kiosk print systems are unattended and in most cases not actively monitored and thus allow an attacker to attach malicious USB devices.

If the attacker is an employee of the company he is trying to attack, he has lots of possibilities to unobtrusively attach malicious USB devices. In most cases, office computers are in freely

accessible for employees. The fact that most computers are locked by a login screen or screensaver doesn't matter all that much, because lots of the attacks mentioned in Section 4.2 even work without any user logged in. Additionally, an employee is equipped with lots of information about the inner workings and processes inside a company, which makes it even easier for him to be unnoticed.

In lots of cases, the attacker himself doesn't need direct physical access but can get his malicious USB device attached to the USB port of a system by other means. People with legitimate physical access to a system could be paid or bribed to act in the interest of the attacker. An example could be any employee or facility staff member that might have a financial interest. People could be instructed to add malicious USB devices to some systems or replace existing devices with modified ones.

Instead of bribery, people with legitimate physical access could also be tricked to attach an attacker–supplied device. When it comes to physical access, social engineering works very well. When the goal of the attacker is just to get one of his USB devices attached to any computer system inside a company, just placing a few attractive or interesting looking USB devices in the form of USB flash drives in front of the company building might work. An employee finding one of those could eventually take it with him and stick it into his office computer. When the target of the attack is the computer system of a specific employee, sending a malicious USB device by mail might work. Depending on how much money the attacker has available for the attack, the USB device can be in original package and could have diverse appearances, ranging from a simple USB flash drive to an exclusive mobile phone with USB connectivity.

Another interesting case, in which people could be tricked to attach a malicious USB device to a system of interest is digital voting systems using so–called *voting pens*. In February 2008, the Free and Hanseatic City of Hamburg, Germany wanted to introduce electronic voting via electronic voting pens for the state parliament election. Due to previous changes to the election law, counting votes for this election was expected to be complicated and time consuming. The digital voting pen provided an opportunity to speed up vote counting. Each voting pen was equipped with a small camera. The paper to be used for the election was filled with a special pattern that was recorded by the camera of the pen, and then stored inside the pen. After the voter finished voting, the pen was given back to the election supervisor, who in turn attached the pen to a USB docking station that was connected to a computer system. The computer system read the votes from the digital pen and stored them in anonymous form inside the system. The critical point is the connection of the voting pen with the USB docking station. The USB docking station is basically a USB hub and the communication directly takes place between the voting pen and the computer system. An attacker could either replace or modify the voting pen given to him, which would then get attached to the host system storing all the votes. A successful attack might then be used for election fraud.

4.2 Classification of Attack Methods

Different methods of attack can be categorized based on what part of the system is being attacked at which level. We separate attacks into the following four categories:

1. Logic attacks
2. Application–level attacks

3. USB stack and device driver attacks
4. Kernel subsystem attacks

Figure 4.1 gives an overview of the different components of the USB architecture.

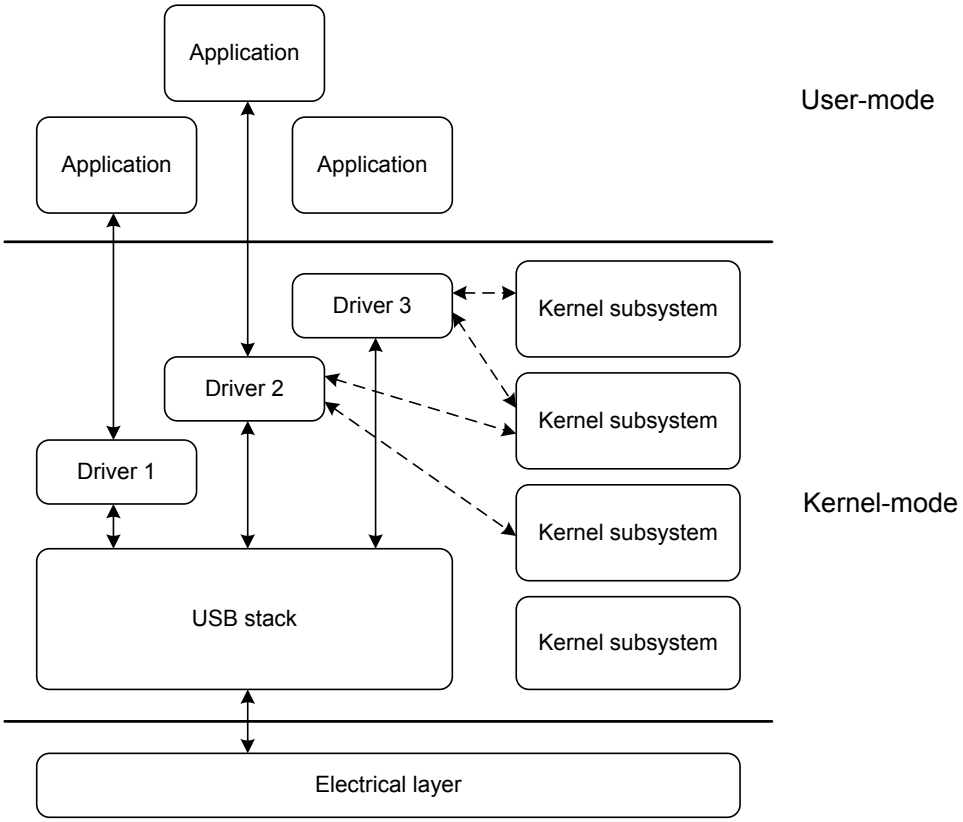

Figure 4.1: Relation between components of the USB architecture

At the bottom, we have the electrical layer. Its purpose is to encode and decode the electrical signals on the wire. Please note that we don't consider attacks on the electrical layer. The electrical layer connects directly to the USB stack, which is responsible for handling protocol details of the USB protocol. Each device driver registers itself at the USB stack. The only way a USB device driver can communicate with an attached device is through the USB stack. Attacks against the USB stack and the attached device drivers are described in Section 4.2.3. To provide their service to an attached device, in many cases device drivers don't run in isolation but communicate with other kernel subsystem components. For example, a USB network card driver makes use of the network subsystem, while a mass storage device driver utilizes the I/O subsystem in the kernel. Attacks against kernel subsystems are described in Section 4.2.4. Additionally, USB device drivers are not only connected to different kernel subsystems. To provide the interaction with a user, applications running in user–mode can communicate with different USB device drivers. Thus data coming from a malicious USB device can even reach applications running in user–mode. Attacks making use of this fact are described in Section 4.2.2. Logic attacks don't attack any specific components of the USB architecture, but try to

abuse the trust relationships between the host and different devices.

4.2.1 Logic Attacks

Logic attacks don't exploit any implementation bugs but abuse the fact that the host puts some trust into the data it receives from different USB devices. Logic attacks are attacks that utilize the default behaviour of the USB architecture and its implementation in the operating system to archive remote code execution on the attacked system. Instead of exploiting any implementation bugs, the underlying technology is used in a creative way to archive the attackers goal. Some possible logic attacks are described below.

Malicious HID Devices

Human Interface Devices (HID) are devices that interact with a human being. HID devices are specified in the Device Class Definition for Human Interface Devices. Obvious examples for HID devices are mice or keyboards. But devices such as USB headsets or data gloves also fall into the USB HID device class. Most operating systems have a class driver for USB HID devices that handles every connected USB HID device.

USB HID devices are really simple USB devices. In addition to the mandatory control pipe, they only require a single interrupt IN pipe. The interrupt IN pipe is used to transfer the data from the device to the host. Transferred data is structured in *input reports* that can contain the relative position change for a mouse or a pressed key code for a USB keyboard. While using the HID device, most of the communication is unidirectional through the interrupt IN pipe.

It's obvious that if an attacker can attach any USB device to a system, he may just connect a standard mouse or keyboard to perform some action. But HID devices can be constructed in any form and behaviour that deviates from the expected behaviour of usual mice or keyboards. For example, a device could be constructed that automatically performs some mouse movements or key presses after it is attached to a system. It could be constructed to look like a standard USB flash drive for example. Such a device could be used for passive attacks, in which people are tricked to attach the device to their computer. By programming such a malicious HID device to perform the right actions, code execution could be possible just by attaching the device. The simple protocol HID devices use further eases the actual implementation of such a device.

Windows AutoRun

The Windows AutoRun feature is a mechanism to automatically perform an action upon the insertion of a new removable media such as a CDROM or a USB flash drive. It is used to automatically install, configure, or launch programs supplied on a CDROM or flash drive when the new medium is detected. To accomplish this task, an executable can be specified, which usually resides on the media itself. This executable is automatically launched when the media is detected.

A similar feature to the AutoRun mechanism is the Windows AutoPlay feature. When a new media is detected, the content of the media is determined. Based on the content type, different actions could be performed by the AutoPlay mechanism:

- Content could be played automatically. This is the case for audio CDs and other examples.

```
[AutoRun]
open=cmd.exe
shell\doit\command=calc.exe
shell\doit=Do your math
```

Listing 4.1: Simple example of a Windows AutoRun file

- A dialog box could be displayed prompting the user to choose between several actions. Depending on the number of different contents found on the media, the user can either choose from one or multiple handler applications that are appropriate for the given content.
- A standard folder view of all files could be opened automatically.

In contrast to the Windows AutoRun feature, no executable that is executed automatically without user intervention can be specified. More details about the use and configuration of AutoPlay can be found at the Microsoft Developer Network [38].

A medium that wants to utilize the Windows AutoRun feature must provide a `autorun.inf` file. This file can also be used to adjust the AutoPlay behaviour. In particular, it can be used to adjust which options are displayed to the user. Depending on the content of the `autorun.inf` file, either the AutoRun or the AutoPlay mechanism is used, when the new media is detected. The `autorun.inf` file must be placed in the root directory of the medium. The file can consist of multiple sections. Every section is introduced by the name of the section enclosed in square brackets. Each section can have multiple key/value pairs. One section is indispensable for the functioning of the AutoRun feature; The `[AutoRun]` section. This section can contain different keys of which the most important ones are described below. For a complete description of the `autorun.inf` file format, see the MSDN documentation [39].

open: The `open` key specifies the path and the application that should be launched by AutoRun when the medium is inserted.

shellexecute: This key is similar to the `open` key, but in addition to applications this key can also specify data files that are opened by their default handler application. The specified value is directly passed to the Windows `ShellExecuteEx` API function.

shell: This key can be used to specify a default command for the shortcut menu of the drive. The shortcut menu is displayed when the user right–clicks on the drive's icon. The value of the `shell` key is a *verb* that identifies the corresponding menu command and it's description. The verb must be defined previously in the `autorun.inf` file. The `shell\`*`verb`*`\command` key specifies the executable that will be run for that verb. The optional `shell\`*`verb`* key specifies a description of the command that is displayed in the shortcut menu. If this key is missing, *`verb`* is displayed instead. For both keys, *`verb`* is just a name for this specific entry. The interesting thing is that the default command for the shortcut menu is not only used for the menu but is also automatically executed when the user double–clicks on the drive's icon. Although not fully automatic, this may execute a binary when the user doesn't expect it.

Listing 4.1 shows a simple `autorun.inf` file that just tries to open a new command line interpreter when the media is mounted. The last two lines add the command "Do your math" to the shortcut

menu. Selecting the command from the shortcut menu or double–clicking on the drive's icon will both startup the Windows calculator `calc.exe`.

According to Microsoft [40], the AutoRun feature is disabled for USB storage devices completely on Windows XP and Windows Vista. If the AutoRun mechanism is enabled or disabled for other classes of devices, it depends on a specific registry key. The registry path `HKCU\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\` contains the key `NoDriveTypeAutoRun`, which indicates the classes of devices when Windows doesn't look for an `autorun.inf` file. Each bit of the value indicates a driver class to be excluded from AutoRun. Table 4.1 lists the meaning of every bit starting with the least significant bit.

Bit number	Bitmask constant	Description
0	DRIVE_UNKNOWN	Unknown drive
1	DRIVE_NO_ROOT_DIR	Devices without a root directory
2	DRIVE_REMOVABLE	Removable disks such as floppy disks
3	DRIVE_FIXED	Fixed disks, which can't be removed from a drive
4	DRIVE_REMOTE	Mapped network drives
5	DRIVE_CDROM	CD–ROM drives
6	DRIVE_RAMDISK	RAM disks
7	–	Reserved for future use

Table 4.1: NoDriveTypeAutoRun bits

The default value on Windows XP is `0x91`. That means, in addition to USB mass storage devices, only the `DRIVE_REMOTE` and the `DRIVE_UNKNOWN` classes are disabled for AutoRun. Starting with Windows Vista, the `NoDriveTypeAutoRun` registry key does not exist anymore in default installations but is implicitly set to `0xff` effectively disabling AutoRun completely.

Although the AutoRun mechanism is not enabled for USB storage devices, there is a way to make AutoRun work with a USB device on Windows XP. Windows XP supports the AutoRun mechanism for the `DRIVE_CDROM` driver class. So to use the AutoRun feature as an attack vector, a USB CD–ROM drive that comes with a `autorun.inf` file that executes a malicious executable when attached to the system can be constructed. This attack can be easily implemented even without special hardware or programming by modifying a U3 smart drive.

The U3 standard [41] is a proprietary standard for USB smart drives, which allows launching applications directly from the USB drive without prior installation. U3 smart drives differ from standard USB flash drives in that they come with a pre–installed Windows application called the U3 Launchpad. This Windows application is automatically started when the device is attached. The U3 Launchpad allows the launch of applications or installation of new U3 compatible applications from the Internet. To automatically start the U3 Launchpad, the U3 smart drives are constructed as composite USB devices. They attach as a CD–ROM drive and as a mass storage device at the same time. The CD–ROM drive shows up as a read–only ISO9660 volume, which contains the `autorun.inf` file for starting the U3 Launchpad. The mass storage device is used for storing additional applications and data.

A U3 flash drive can be modified to not execute the U3 Launchpad applications but instead any attacker–chosen payload when it is attached. This attack is not new. Instructions on how to modify such a device can be found on the Internet [42]. For our proof–of–concept attack, we used a 1 GB SanDisk Cruzer Micro drive. The ISO9660 volume can't be modified directly, but SanDisk provides the SanDisk Launchpad Installer (`LPinstaller.exe`), which allows re–installation of

the U3 smart drive. It can be obtained from their website at `http://u3.sandisk.com`. The installer fetches an ISO9660 image from the SanDisk server when it is run. This image will be installed as the CD–ROM volume. The `%SystemRoot%\system32\drivers\etc\hosts` file on Windows can be modified to point the DNS name `u3.sandisk.com` to an IP address, where we setup a web server and provide a modified ISO image. The installer fetches the original image from the SanDisk website[1]. We can download that ISO image and modify it to make any desired changes to the `autorun.inf` file to load our own payload. After the modified ISO image was stored at the web server, just running the SanDisk LaunchPad installer will fetch the modified ISO image and store it inside the smart drive.

This gives us a device that looks like a normal USB flash drive but actually presents a CD–ROM drive to the system, which is then allowed to make use of the AutoRun feature on Windows XP. Windows Vista can't be attacked this way, but it still provides the AutoPlay feature. So an `autorun.inf` file with the `shell` keyword could be constructed to get code executed when the user double clicks on the drive's icon as mentioned above.

USB Packet Sniffer

The Universal Serial Bus uses a token–based packet protocol in which the host initiates all the communication. With the USB standard up to version 2.0, every device connected to the same bus can potentially see all the packets sent by the host to other devices. The host controller sends the token packets at regular intervals on the bus. Amongst other things, the token packet contains a device address and the direction of the following transaction. Each attached device decodes the device address and does a match against its own. If the device address matches, the device selects itself for the forthcoming transaction. Whether the device address is the source or the destination of the following data transfer, it is indicated by the direction specified in the initial token packet. The source then either sends a data packet to the destination or signals that it has no data to be transmitted by sending a NAK handshake packet. After the data was transferred, the destination acknowledges the successful reception with an ACK handshake packet. Because of the broadcast nature of the protocol, the token packet and the data packet of OUT transactions are sent to all USB devices connected to the same bus.

According to [5], if the USB device receives a packet in which the device address doesn't match its own, the device just ignores the communication. The device address match process is in almost all cases implemented in hardware. When a packet with a matching device address is received, the USB device stores the data in a receive buffer and triggers an interrupt. Since it's the USB device itself that decides if a packet was destined for it or not, nothing prevents a USB device from accepting every packet regardless of the set device address. A USB device that acts like a normal USB device according to the USB specification but also stores every received packet that doesn't match its own device address could be built. This leads to interesting possibilities for an attacker.

To provide USB functionality, a system includes a USB host controller. But depending on the system, more than one host controller may be present. Every host controller provides a separate bus. With only a single host controller, all connected devices have to share the bandwidth on the bus. Multiple host controllers can mitigate this potential bottleneck.

Since all the data packets of OUT transactions to other devices on the same bus can be read, files transferred from the computer to any connected USB hard disk or flash drive could be

[1] http://u3.sandisk.com/download/apps/lpinstaller/isofiles/PelicanBFG-autorun.iso

eavesdropped. Any documents printed on a USB printer, while a USB sniffing device is connected, could be silently sniffed as well. This all presumes that the USB connector to which the USB sniffing device is attached and the one used for the device to be sniffed are connected to the same bus.

All those scenarios target USB devices connected to external USB ports of a system. But external devices are not the only devices using USB. Most notably wireless components, such as IEEE 802.11 or Bluetooth modules, are often internally connected to the USB bus. If the external USB connectors are attached to the same bus as used by internal components, this could enable an externally connected device to record all the outgoing wireless communication of the host. Despite the use of encryption on the wireless link, data can be captured in plain text. The encryption is in most cases directly implemented on the wireless hardware itself, which happens after the data was already captured.

USB 3.0 implements a dual–bus architecture to stay backward compatible with USB 2.0 devices. The old USB 2.0 bus is augmented by another bus called the *SuperSpeed* bus, which offers most of the new features of USB 3.0. One change on the SuperSpeed bus compared to the old one affects the flow of communication. Instead of broadcasting all packets to all enabled ports, packet traffic is explicitly routed to the receiving device [2]. Packets are equipped with additional routing information that is used by the hubs in the decision to which downstream port a packet should be routed to reach the final device. While sniffing on the SuperSpeed bus doesn't seem to be possible, devices using the old bus are still susceptible to the sniffing attack.

4.2.2 Application–Level Attacks

The host can serve a USB device in two different ways. Some USB devices are only served by the operating system running on the host. Other USB devices are handled by the operating system but then communicate with a user–mode application running on the host. Examples for USB devices only talking to the operating system are e.g. simple keyboards or mice. Devices only talking to a user–mode application are e.g. USB measurement instruments. But for most devices, the distinction between the two types is a fuzzy one. Lots of USB devices are handled by the operating system, but can get in contact with other user–mode applications through various mechanisms detailed below.

Every user–mode application handling data received from an external USB device increases the potential attack surface. But what differentiates user–mode applications from other kernel components handling the data, is that exploitation techniques for user–mode applications are far more researched than the same techniques for kernel–mode. An attacker can use the usual exploitation methods he's familiar with to exploit flaws in user–mode applications. This decreases the time for the development of a working exploit and makes user–mode applications communicating with USB devices an attractive target.

In the following we will go into the details of some USB devices and how they can get into contact with different user–mode applications.

Apple iTunes and iPods

All Apple iPods have been equipped with a USB connector since the third generation of iPods. They attach as usual USB mass storage devices to a system. To manage the media files stored on the iPod, the popular iTunes application can be used. The file system contains some control

data for the iTunes application and the actual media files. When the iTunes software is installed on a Windows system, the `iTunesHelper.exe` application is running in the background waiting for the attachment of an iPod. Once an iPod is attached to the system, the `iTunesHelper.exe` detects this event and starts up the actual iTunes application. The iTunes application then reads the control data stored on the file system of the iPod and does its job. Reading and parsing the control data is exactly the point, where vulnerabilities could exist due to the nature of parsers. If the iTunes parser for those control files had a bug, this could potentially be exploited with a modified USB iPod device.

OS X Quick Look

Quick Look is a technology introduced in Mac OS X 10.5, enabling applications such as the Finder or Spotlight to either display thumbnails or preview images of different kinds of files [43]. When an application is showing a listing of files to the user instead of presenting the user with only a file name, some meta information and an icon indicating the file type, Quick Look replaces the icon with a small thumbnail representing the content of the file. This makes it easier for users to get an idea of the real content of files.

Quick Look can be used to display two representations of documents: thumbnails and previews. Thumbnails are just a replacement for the usual icon. They are a little bit larger than icons and give the user a notion of the content. The thing that makes thumbnails interesting for our discussion is the fact that really little user intervention is required to display them. Just opening the finder in some directory shows the generated thumbnails of those files.

The other representation displayed by Quick Look are previews. Previews are larger than thumbnails. They can be requested by the user to quickly catch a glimpse of the content of some document without opening the application associated with that document type.

The Quick Look architecture consists of consumers and producers as illustrated in Figure 4.2.

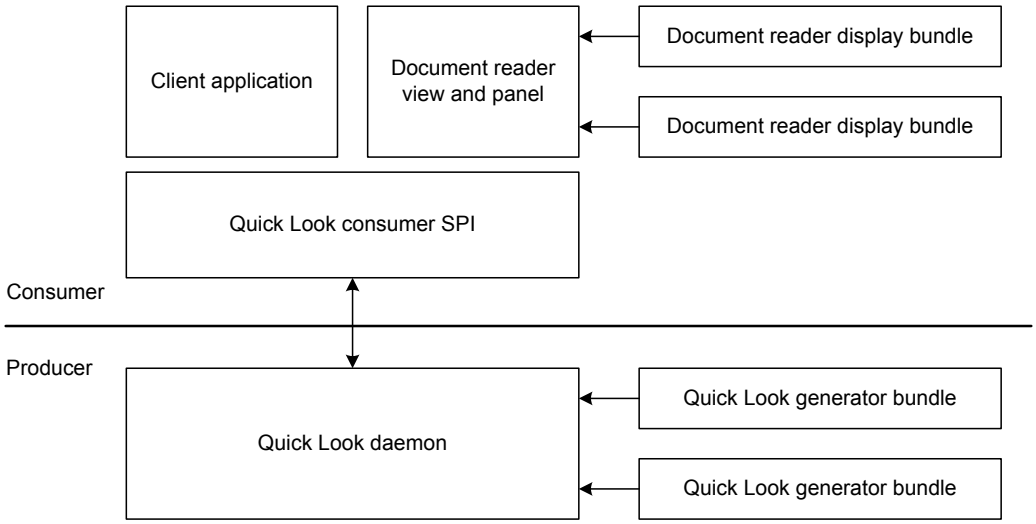

Figure 4.2: Quick Look architecture

The consumer in this architecture is any client application that wants to display thumbnails or

previews of some documents. The producer on the other hand provides the requested image to the consumer. The consumer consists of three different components. The main component of the consumer is the client application itself. The document reader view and panel are usually embedded inside the client application. They are used to display the preview content together with some optional controls to manipulate the preview. The document reader view only provides the place to display the preview content. The task of actually displaying the preview content inside the view is delegated to a display bundle. Each display bundle is responsible for one native document type. The consumer system programmatic interface (SPI) provides the interface between the consumer and the producer. It is used by the client application to request thumbnails or previews, and by the producer to provide the requested content back to the client application.

The producer is the Quick Look daemon `quicklookd` that offers an extendable plug–in architecture. Some file types are natively supported by Quick Look, which means that those file formats can be parsed to build thumbnails or previews of the content. The plug–in architecture allows third–party applications to provide their own Quick Look generator bundles. Those bundles can parse the application–specific file formats and can provide thumbnails and previews on request.

What makes the Quick Look architecture a good attack surface for attacks over USB is the fact that a simple USB mass storage device can be used to store a malformed document. When the USB device gets attached to the computer and the Finder is opened to look at the content of the device, the document is automatically parsed by the Quick Look daemon in the background. Due to the nature of parsers, it is very likely that a Quick Look generator for one or more file formats may contain a bug. Those bugs could be triggered by a prepared document on the USB flash drive. Depending on the bug, code execution might by possible just by viewing the content of the attached USB flash drive.

Strictly speaking this attack is not a real USB attack. But it makes it clear how the attack surface increases when taking into account the attachment of untrustworthy USB devices. In addition to Quick Look, there are lots of other technologies which could be used for similar attacks, such as the metadata indexing service of Spotlight [44] or the AutoPlay feature mentioned in Section 4.2.1. Basically getting any slightly complicated code (e.g. parsers) process data provided by an untrustworthy USB device could provide a chance for an attacker to compromise the system.

4.2.3 USB Stack and Device Driver Attacks

By communicating with an attached USB device, data provided by the USB device is handled by different software components. Malicious or unexpected data could trigger software vulnerabilities inside the implementation, which an attacker could potentially exploit to compromise a system. This section first describes two of the main components of the USB architecture inside operating systems and why they might be interesting targets for an attack. Finally, we conclude with an example of a device driver that has the potential to be attacked.

The USB stack handles all the USB protocol details and loads the corresponding device drivers for attached devices. The benefit of attacking the USB stack itself is that no specific USB device driver must be installed on the system to be attacked. Even hardened systems with only a minimum of needed USB device drivers installed could be exploited through the use of a USB stack vulnerability. A specific version of an operating system with a vulnerability inside the USB stack would be enough.

USB device drivers on the other hand handle specific USB devices. Potentially every USB device driver (kernel or user–mode) can be attacked. The significant number of different USB drivers

makes them a target worthwhile to be considered. Additionally, lots of USB device drivers are developed by third–party companies. It is to be expected that the quality of different USB device drivers could differ vastly from driver to driver. Although third–party device drivers might be an interesting target because of their differing code quality, the downside is that they might not be installed on every system. USB device class drivers on the other hand come with most operating systems pre–installed and thus can be expected to be found on lots of different systems.

One example for a device class driver installed on nearly every version of Microsoft Windows is the USB mass storage class driver `usbstor.sys`. All device drivers in Microsoft Windows responsible for managing a specific storage device form together the *Windows storage stack* [33]. Among other drivers, it contains *hard–disk storage drivers*. These drivers are responsible for the low–level management of found storage devices. The hard–disk storage drivers are started by the I/O manager (see Section 3.3.1) and are structured in a class/port/miniport architecture, illustrated in Figure 4.3.

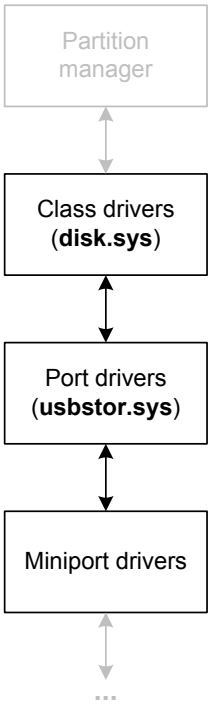

Figure 4.3: Windows hard–disk storage drivers

The storage class drivers implement common functionality, which can be used by all storage devices of a specific device type independent of the bus they are attached to. Figure 4.3 shows the `disk.sys` class driver that handles disks. Port drivers implement the functionality needed for a specific bus. The `usbstor.sys` port driver that handles USB storage devices is shown in Figure 4.3. Microsoft supplies the storage class drivers and the port drivers. Miniport drivers are supplied by third parties and are used to handle hardware–specific details.

Because of the complexity of the Mass Storage Class (MSC) specification and all the underlying protocols and different driver components involved, this class driver might provide an interesting

target for attacks.

4.2.4 Kernel Subsystem Attacks

The USB stack and device drivers are not the only components inside the kernel that are processing data received from external USB devices. This section shows that other components of the kernel can also get in contact with data from external USB devices, although they might not be associated with the USB protocol at a first glance. To demonstrate our point, we close this section with an example of a device driver not related to USB that could get easily in contact with data supplied by USB devices.

One design goal of USB that is even reflected in its name by the word "universal" is its multi–purpose use. USB can be used to connect lots of different kinds of devices to a host computer. Most USB device drivers don't work in isolation. Depending on the type of device, the USB device driver on the host might make use of multiple other kernel components or subsystems to serve the device. This is illustrated in Figure 4.1. Examples for some subsystems, USB devices might communicate with, are:

- Disk subsystem
- Network subsystem
- Audio/video subsystem
- Protocol stacks

The last item illustrates the point that basically any new class of communication device attachable through USB increases the attack surface by another protocol stack. Communication devices for protocols like IrDA, 802.11 or Bluetooth all exist as USB variants. If a host has the respective device drivers installed, the Universal Serial Bus offers the possibility for potentially malformed data to reach those protocol stacks.

So data arriving on the bus can traverse through all those different kernel components. With usual USB devices this is not a problem, because most of them at least minimally adhere to some standard so the kernel knows more or less what to expect from specific USB devices. But an attacker can build a USB device that deviates from the expected behaviour and sends malformed data. This presumes that all those kernel components do enough validation of the data they receive. There might be problems with kernel components that where not designed with the thought in mind, and that potentially malformed data could reach them. USB provides an attacker with the ability to reach those kernel components. By providing malformed data to those components, using a custom–build USB device, an attacker might be able to trigger bugs inside the kernel, leading to vulnerabilities. Exploiting any of those vulnerabilities, could result in arbitrary code execution inside the kernel and thus allow an attacker to compromise the system.

One example for a kernel component, which is not directly related to the USB protocol, is the `disk.sys` class driver already mentioned in Section 4.2.3. This driver is responsible for managing all kinds of disk–based storage devices. Although it is not directly related to the USB protocol, USB mass storage devices make use of it to provide the user the ability to access the file system of such USB devices. So a malicious USB device identifying itself as a mass storage device can now easily provide arbitrary malformed data to the host, which is then handled by the `disk.sys`

class driver. Depending on the data provided, it might not be handled correctly in all cases and might lead to vulnerabilities.

Chapter 5

Implementation

In Chapter 4, we started off to list potential attack vectors against the Universal Serial Bus. To actually demonstrate that the mentioned attacks are not only of a theoretical nature, we needed a way to implement some of them. Since there was no available solution to assess the security of USB stacks, we started by writing a simple USB fuzzer implemented inside a peripheral controller driver of the Linux–USB Gadget API Framework introduced in Section 3.1.1. This allowed us to fuzz the communication of every available Gadget driver provided with the framework. Although this worked and led to most of the results in this thesis paper, it has the big disadvantage that only device drivers with corresponding Gadget drivers can be tested. The development of a new Gadget driver is very time consuming and is comparable with the development of a new USB device driver.

We needed a more universal approach, which would allow us to assess the security of more USB device drivers without the restriction of available Gadget drivers. Despite the fact, that our implementation is focused on fuzzing, being able to use it for other attacks besides fuzzing would be beneficial too. This resulted in the implementation described in this chapter.

In Section 5.1 we show the layer at which we are fuzzing and explain our decision. After listing all the prerequisites we expect from our fuzzer in Section 5.2, we describe the general design in Section 5.3. Section 5.4 then goes into detail about how each component of the fuzzer was implemented.

5.1 Layers to be Fuzzed

The first things we have to get straight is the layer at which we want to fuzz. There are basically two possibilities where we could intervene in the flow of communication. We could either fuzz on a packet level or we could fuzz the endpoint data traveling through the USB pipes.

Fuzzing on a packet level has the benefit that we are fuzzing at a really low layer, but this also means that we don't have much context of the data we are fuzzing. So it's more of a blind fuzzing approach. Another thing to keep in mind when fuzzing at the packet level is the fact that each packet is protected by a cyclic redundancy check (CRC). We have to recalculate the CRC after each modification. If we skip this step, the CRC won't match anymore and the host will in most cases just ignore the packet [3]. Although fuzzing at this level could potentially reveal

vulnerabilities in the lowest–level components involved, such as the host controller firmware itself, it isn't the ideal place for our goal of finding vulnerabilities in the USB stack and device drivers of a host.

Fuzzing the endpoint data that traverses through USB pipes is another possibility. Fuzzing at this level has the big advantage that we can choose to only fuzz specific pipes. Depending on the concrete device, each pipe with its corresponding endpoint is used for a specific task. Fuzzing specific pipes allows narrowing down the code, which is being fuzz–tested. For example, the default control pipe (endpoint 0) is used with every USB device to exchange control and status information. So by fuzzing the default control pipe, there is a good chance that discovered vulnerabilities are inside the USB stack or in the device driver code responsible for handling the attachment of a new device. Fuzzing other pipes could reveal vulnerabilities in other kernel subsystem components, which are using the data from those pipes. In addition, we don't have to care about any USB checksums, because those will only be created on a packet level. Fuzzing on a pipe level modifies the data, before it is split into multiple packets. For the above mentioned reasons, we have decided to fuzz at the USB pipe level.

5.2 Implementation Prerequisites

We have the following four prerequisites for our fuzzer:

1. Fuzzing should be automatic. We don't want to manually attach and detach devices. As such, automatic and repeated attachment and detachment is a must.
2. Our fuzzer should be able to send malformed or invalid data in the hope to trigger some bugs in the software implementation of the host. This implies, that the data sent might deviate from the USB specification in some cases. So the underlying software shouldn't restrict us, in what we are able to send.
3. The fuzzer should be implemented in software. To actually find new vulnerabilities, a software–solution would be beneficial, since changes to the fuzzer could be done in a more flexible way. In particular, a software solution in user–mode would ease the fuzzing process even more and provide better control over the whole process. Although we strive to implement the fuzzer in software, a hardware solution is needed later for the implementation of the final attack. It would be a benefit, if the build fuzzer could directly be transformed into a hardware proof–of–concept device without many modifications.
4. To fuzz the code of specific USB device drivers, the USB fuzzer should be able to emulate different kinds of devices without much development effort. For example, to fuzz the code of a HID device driver, the fuzzer should attach to the host claiming to be a HID device such as a mouse or keyboard and then start to send malformed data. The fuzzer shouldn't be restricted to different classes of devices.

5.3 Design of the Fuzzer

We have to choose between a generation–based and a mutation–based fuzzer. The problem with a complete generation–based USB fuzzer is that development is very time consuming. To

successfully communicate with a single USB device driver on the host, simply providing the correct vendor and product ID inside the USB device descriptor is not enough. The emulated USB device has to act in conformance to the expectations of the device driver. This means it has to provide the expected USB descriptors, provide the expected endpoints and has to send valid data on each pipe connected to the endpoints. If good code coverage is desired, development of such an emulated USB device is nearly equivalent to the development of a complete USB driver. This would have to be done for every driver class to be tested. A mutation–based USB fuzzer on the other hand can rely on the communication from a real USB device and just manipulate the valid communication. So a mutation–based fuzzer is the ideal choice for quickly getting some first results.

To build a mutation–based USB fuzzer, we need to let a real USB device talk to a device driver on the host. Then we can both actively manipulate selected transfers while the communication takes place or we could just log the communication of the device and replay it at a later point in time.

So our man–in–the–middle approach can be separated into three components:

1. Receiving Component
2. Processing Component
3. Device Emulation Component

The receiving component is responsible for acquiring the initial USB packets. It either receives the raw USB communication from an attached USB device or reads in a stored flow of communication, which was recorded beforehand. All USB packets are just forwarded to the processing component.

The processing component conducts the optional modification or analysis of the USB communication. This is where the actual fuzzing can be implemented. The processing component can also record a flow of communication and store it for replaying at a later point in time. The processing component passes all the USB communication to the device emulation component.

The device emulation component forwards the USB communication it received from the processing component to a connected host system. It basically acts like the real USB device.

The separation into those three components makes our approach really flexible. Although we select specific technologies to implement each component, each component can be easily replaced with another implementation, should the need arise.

5.4 Implementation of each Component

In the following, the implementation of each component will be described in more detail, starting with the device emulation component.

5.4.1 Device Emulation Component

For the device emulation component, we need a way to emulate USB devices in software. There are basically two more or less widely known frameworks for this task: The Microsoft Device Simulation Framework [45] and the Linux–USB Gadget API Framework [27]. Both frameworks

allow the emulation of USB devices on the local machine without additional hardware. Together with a virtualization solution such as VMware or Microsoft Virtual PC, this would allow us to emulate malicious devices locally and to test the device drivers of an operating system running as a guest inside the virtual machine. Although this would be a convenient solution, it is not a viable option for something like a USB fuzzer. Simulated USB devices are first enumerated by the host operating system. Only when the host successfully detects the device is it claimed by the virtualization solution to finally pass it on to the guest operating system. When the host operating system doesn't successfully enumerate the device, e.g. because of some fuzzed USB descriptors, the guest operating system doesn't even get a chance to see the device. So we effectively fuzz the host instead of the guest operating system. Even if the host successfully enumerates the USB device, the device is first claimed by the virtualization solution which means another software layer, which can cause problems if malformed data isn't handled properly. We learned this the hard way when VMware crashed while we where trying to fuzz–test the USB stack of an operating system running as a guest inside VMware.

The easiest way to overcome all such problems is a hardware solution. The USB device is still emulated in software on one machine, but instead of letting the emulated device attach to the same machine, hardware is used to let it attach at another physical machine.

While the main purpose of the Microsoft Device Simulation Framework is to emulate devices in software to actually develop and test device drivers, the Linux–USB Gadget API Framework is intended for the development of device–side drivers, mainly used on embedded systems. In particular, it comes with good support for hardware chips, which can be used to actually implement the USB fuzzer in software but connect it to the host through a hardware peripheral controller. Since the Linux–USB Gadget API Framework has support for lots of small on–chip controllers, an exploit for any found vulnerability could be easily implemented in any imaginable form of enclosure using an embedded Linux system together with one of those peripheral controllers.

Another benefit of the Gadget API Framework in addition to the good hardware chip support is the availability of its source code. Should the need arise; it gives us the freedom to make any needed modifications.

So we finally decided to implement the device emulation component using the Gadget API Framework. To actually let the emulated USB device attach to another host, one of the many supported USB peripheral controllers has to be selected. We decided to use the NetChip net2280 controller. It's a high–speed USB 2.0 peripheral controller, which supports bulk, isochronous and interrupt transfers. It provides six different endpoints that can all be arbitrarily programmed. One of the main benefits of this controller is its availability as a PCI development board. This way, it can easily be connected to any standard PCI slot and provide a USB 2.0 peripheral port. Using this peripheral controller allows us to stick to using standard PC hardware for our implementation, invalidating the need for a special–purpose hardware design.

5.4.2 Processing Component

The processing component should give the user the freedom to do any desired modifications or analysis of the raw flow of USB communication. In addition, the processing component could be used to record the communication of a USB device and replay the communication at a later point in time. Since these tasks require the greatest amount of user–intervention, the processing component is implemented completely in user–mode. By adding corresponding API bindings, the processing component could be implemented as a library and be utilized by third–party tools

such as fuzzing or exploitation frameworks [46].

One important aspect of the processing component is the mechanism used to reproduce any discovered crashes, when doing random–based fuzzing. In our previous implementation, we just recorded the packet number of the packet which was modified by the fuzzer together with the changed values and their offsets inside the packet. To reproduce a found crash, we just re–attached the same device and then re–applied the same modifications as done in the previous attachment. This procedure has the disadvantage that we are blindly relying on the packet number sent by the device, to decide which packet needs to be modified. If, for any reason, the number or order of packets differs between two attachments, we are modifying the wrong packet when trying to reproduce the first attachment.

Another possible approach to reproduce a found crash is to use the replay mechanism to just replay the whole communication of the device from the respective attachment, which triggered the crash. However, despite the fact that it has the same problems as the aforementioned mechanism, should the host send other packets than those in the previous attachment, there is yet another problem. Replaying the communication of devices with simple protocols, like HID devices, works perfectly. Mass storage devices on the other hand are one class where replaying doesn't work without some modifications. Trying to replay the communication of a mass storage device works up to the point, where the device starts to exchange SCSI [7] commands with the host at which point the communication breaks. This problem could eventually be overcome by analyzing packets sent by the host and applying some slight modifications to the replayed data.

Currently, only the latter approach is implemented in the processing component, but the mechanism for reproducing device attachments used in the previous implementation could be easily integrated into the processing component.

5.4.3 Receiving Component

For the receiving component, we need access to the original communication of a USB device. There are multiple places inside the Linux kernel where we could grab the raw USB packets. Packets could be intercepted directly from the host controller driver before they are moved to any responsible device driver. Although this could be possible with a small modification to the kernel, we would still need a way for the processing component running in user–mode to access and modify the flow of communication.

Luckily, the Linux kernel already provides a mechanism to pass all the USB packets down to user–mode. This mechanism is provided by the USB device file system introduced in Section 3.1.1. To retrieve the descriptors of an attached USB device, the corresponding device files inside the mounted USB device file system can be read. Communication with a device takes place using `ioctl()` calls on the desired device file. For easier access to all the offered features, the libusb library wraps those operations in an intuitive API, which can be used by the processing component.

We are using the old but latest stable version 0.1 of the library. Although stable, one of the shortcomings of this version is the fact that it doesn't support isochronous transfers. Isochronous transfers are mostly used by video and audio devices. The missing support in the receiving component prevents us from testing those devices using our implementation. A completely redesigned version 1.0 of the library is already in development and currently in beta stadium. This new version promises to add lots of new features missing in version 0.1, including the support for isochronous endpoints. When the new version gets more stable, the receiving component of

our implementation should be re–implemented using libusb 1.0.

5.5 Implementation Details

The final architecture of the implementation is shown in Figure 5.1. The USB fuzzer running in user–mode is connected to the receiving component through the libusb library, which uses the USB device file system under the hood. The USB fuzzer communicates with any attached USB devices through this link.

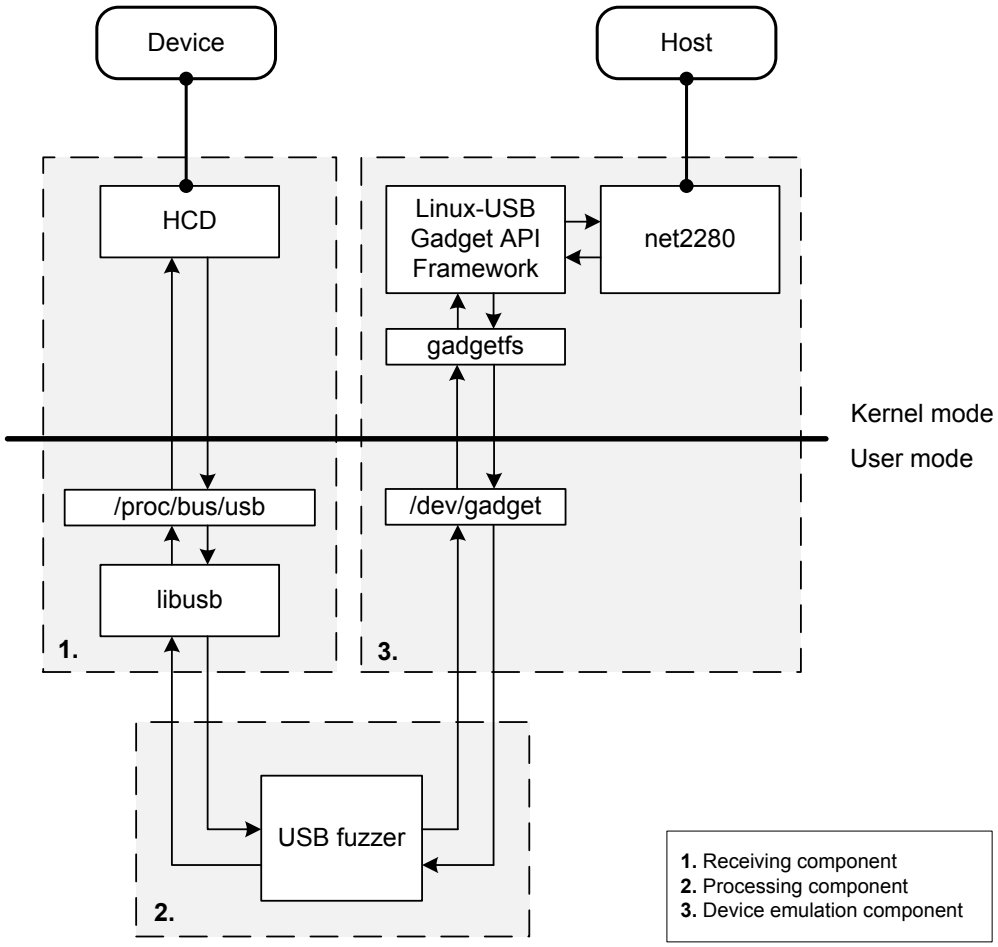

Figure 5.1: USB fuzzer architecture

The connection to the device emulation component is implemented by the gadgetfs driver. Provided as a kernel module, it provides user–mode processes with access to the Gadget API Framework. It provides a new file system, which is usually mounted on the `/dev/gadget/` directory. A single device file for the peripheral controller itself and several endpoint device files are provided. By reading and writing these files, a user–mode process can communicate with the Gadget API Framework. The Gadget API Framework uses the peripheral controller driver for the NetChip

net2280 card, which is then connected to another host. This link is used by the USB fuzzer to emulate the USB devices.

The USB fuzzer first initializes the libusb library and checks, if any device drivers running on the same host where already loaded for the attached USB device. If any device drivers are found, they are detached. This is needed, so that the device can be claimed. After claiming the attached USB device, we first read all the descriptors from the device using the libusb API.

We extract information about each provided endpoint from the read descriptors. Before actually writing the descriptors to the control endpoint device file of the gadget peripheral controller, they must be slightly modified. This is required because the Gadget API Framework expects the descriptors in another order. In addition, the Gadget API Framework has some limitations that must be worked around in order to successfully attach some USB devices. See Chapter 8 for some of the limitations of the Gadget API Framework.

Once we have written the descriptors to the control endpoint device file, the Gadget API Framework starts to attach a new device to the connected host. The USB fuzzer now just listens on the control endpoint, waiting for requests from the host. Despite some control requests that must be handled separately to make the man–in–the–middle approach possible, any received control requests are decoded and forwarded to the attached USB device using the libusb API. For IN transactions, we read the requested number of bytes from the device and forward the response to the host. For OUT transactions, we read the requested number of bytes from the host and forward them to the attached USB device.

When receiving a `Set_Configuration` control request, we set up all the endpoints of the requested configuration by writing the descriptors, we read from the USB device to the gadgetfs device files. Any data read from the real USB device is forwarded through these endpoints and any data coming from the host, is forwarded to the device.

If fuzzing is enabled, all the IN data transfers can be fuzzed. We implemented a simple random–based fuzzer, which randomly changes some bytes, while trying to set the most–significant bit more often in the hope to trigger signedness issues in the code, which is being fuzz tested.

5.6 Hardware Implementation

Since our implementation is based on the Linux–USB Gadget API Framework, implementing a malicious USB device in hardware is really straight forward. The Gadget API Framework offers support for many different device chips, of which most of them are highly integrated SoC (System–on–Chip) processors. The complete list of supported chips can be found at the project homepage [27].

We used the NetChip net2280 because it offered a development board as a PCI card, which simplified our implementation since we could use standard PC hardware for the development. Another benefit of using the Gadget API Framework is that any implemented driver should be usable with any of the supported chips with minimal to no changes at all. That means that any implementation based on the Gadget API Framework, which was developed using the net2280 card, could be easily ported to one of the other supported chips. So any proof–of–concept could be ported with minimal effort to a small hardware exploit.

Let's suppose, an exploitable vulnerability was found through the use of our fuzzer. One of the easiest ways to build a hardware exploit is to utilize one of the mechanisms to reproduce device

attachments described in Section 5.4.2. The required modifications of the send packets could be adjusted from a proof–of–concept crashing the host to a working exploit, which would execute attacker–supplied code on the host.

The first approach mentioned in Section 5.4.2, which just re–attaches the device while applying the necessary modifications to the respective packets, could be used to implement an exploit in hardware. Yet it has the disadvantage, that the device to be re–attached is a real device, which would have to be included in the final hardware solution. Although possible, this has some size constraints for the build device.

The second approach, where the whole communication of a device attachment is just replayed, could also be used to implement a hardware exploit. This approach has the benefit that the original USB device doesn't have to be present. The only problem with this approach is the aforementioned restriction to only some device classes.

Whatever replay mechanism is used to build an exploit in hardware, they still both have one problem in common: a change in the flow of communication from the host might break the communication. To overcome this problem, it's of course possible to build a hardware exploit by writing a new gadget driver, which just tries to trigger the specific vulnerability. Although this is more time consuming than relying on the replay mechanism, this course of action allows a developed hardware exploit to be made much more reliable and built in nearly any form.

A hardware implementation is even easier for the mass storage device based attacks mentioned in Section 4.2.2. Any device providing the USB mass storage functionality can obviously be used for those attacks.

Chapter 6

Results

Our first approach to building a USB fuzzer was the integration of the actual data mutation process inside a peripheral controller driver of the Linux–USB Gadget API Framework. This way, we could fuzz all device drivers for which a corresponding gadget driver existed. Most of the results of this thesis paper were found using this implementation, but since the implementation restricted us to only a few device classes, we started to implement the new architecture described in Chapter 5. It provides a good deal more flexibility. Since the new implementation was still in its final stage of development, the new implementation was not used to reproduce all of the findings, but it can be expected that the implementation from Chapter 5 will be able to reproduce the results from the previous implementation. Nothing changed between both implementations in the way we are doing the actual data mutation. Both implementations use exactly the same random–based fuzzer to fuzz IN transactions. There are only two relevant changes in the new implementation. Instead of fuzzing IN transactions of a virtual gadget device emulated in software, we are now fuzzing IN transactions of real USB devices connected to our system. The second change concerns the place where we are fuzzing. Compared to our previous implementation, the new design allows us to do the fuzzing in user–mode instead of in the kernel. To get the data back into the kernel, we need the gadgetfs kernel module. Although this means one additional layer, which the mutated data must traverse, it should not influence the fuzzed communication in any way. So, the new implementation should be able to produce the same results, but in addition, also increase the number of USB device drivers that can be tested.

In the following, we will describe our findings using the previous implementation. We used the file–backed storage gadget driver to test the mass storage device class drivers of all operating systems listed in Chapter 3. We built a 1 GB file system image file with a NTFS partition and stored some files on it. This image file was used as the file system for the emulated mass storage device. We repeatedly let the emulated device attach at the host, waited for at least 5 seconds to let the device be enumerated and then detached the device again. For each attachment, some IN transactions on all USB pipes from the device to the host were fuzzed by randomly replacing some bytes with random ones while the most–significant bit was set more often in the hope to trigger some signedness issues. The packet number, together with the offsets and values modified by the fuzzer, were stored such that discovered crashes could be reproduced. Once a crash was found, the same gadget driver was just loaded again, but instead of blindly fuzzing all IN transactions, the recorded modifications were simply re–applied.

While testing Windows XP SP2, multiple crashes where encountered. Some of them couldn't be

reproduced using our approach described in Section 5.4.2. This was either because the host sent packets other than those in the enumeration before or because of the nature of the crash. Some crashes where most likely some kind of race conditions, which obviously couldn't be reproduced easily. For fuzzing Windows XP, we had to adjust the fuzzer a little bit. It turned out, that fuzzing on endpoint 0 disabled the USB functionality for the whole USB controller. After a few runs of attaching and detaching emulated USB devices, the attachment of new devices were no longer detected. The USB functionality was restored only after a reboot of the host. For this reason, we excluded the Default Control Pipe from being fuzzed.

We actually found two reproducible crashes. The first one was found inside the USB mass storage class driver `usbstor.sys` that was introduced in Section 4.2.3. We found out that it was possible to trigger a bug check inside the kernel leading to a kernel crash, by sending malformed data to the host. Analysis of the crash revealed that it happened due to a double–free of kernel pool memory. Although exploiting a double–free vulnerability in kernel–mode is harder as the equivalent in user–mode (see Section 2.2.2), exploitation could still be possible [47]. This vulnerability shows that attacking USB device drivers is not only a theoretical concept, but is also feasible in practice.

The second crash was found inside the disk subsystem, more precisely inside the `disk.sys` class driver introduced in Section 4.2.3. A bug check was triggered when the host tried to read the partition table of the attached USB mass storage device using the `disk.sys` function `DiskReadPartitionTableEx`. The crash probably happened due to some kind of kernel memory pool corruption. A more thorough analysis of the vulnerability would be needed to decide if it could be exploited for code execution. Independent of the exploitability of this particular vulnerability, this case shows that other kernel components not obviously related to the USB protocol can also provide an attack surface. The USB port is just used as an entry point to reach other components inside the kernel.

Fuzzing Mac OS X 10.5 led to varying behaviour. When the time between attachment and detachment of the USB device was chosen too small, repeatedly re–attaching the device, led to a complete lockup of the system. Although no kernel panic screen was shown, only a reset of the system restored the behaviour. One other kernel panic was produced by our fuzzer. Unfortunately we were not able to reproduce it using our approach.

Fuzzing the Linux kernel 2.6.24 and Windows Vista did not lead to any crashes. This might indicate slightly better code quality but is of course no guarantee for anything. More thorough testing must be performed in the future to get a better picture. Additionally, other class drivers than the mass storage class driver should be tested. We are expecting to find more bugs, especially when using the new implementation to fuzz some third–party vendor drivers, which the old implementation didn't allow us to do.

Besides the fuzz–testing, we also implemented one of the logic attacks from Section 4.2.1 using our implementation. We implemented a malicious HID device, by first recording the flow of communication of a default USB keyboard connected to a Windows XP system. We manually opened the Windows run dialog and typed a command. The stored communication was then just replayed at another system, effectively executing the desired command.

Chapter 7

Conclusion

This thesis paper has presented different security aspects of the Universal Serial Bus architecture and the software that implements it. We first gave an overview of the USB support in some of the major operating systems and described in detail the process of device enumeration and how new device drivers are loaded. Each of the mentioned operating system was accompanied by a list of USB class drivers that come pre–installed with the respective operating system and thus, provide a potential attack vector for default installations.

After the reader was familiarized with the operating system specific details, the awareness for attacks against the Universal Serial Bus was raised by some real world scenarios. Each scenario demonstrated a way which might be used to attack a system using a provided USB bus. We developed a classification to categorize all the possible attacks. Consequently, we described different attacks for each mentioned category and even implemented some of them.

To actually prove the feasibility of some of our mentioned theoretical attacks against the USB device drivers and stacks, we developed a USB fuzzer. Our implementation is based on three different components which allow for easy replacement of each component and make the implementation universally usable even for other tasks than fuzzing. We demonstrated the universal usability by implementing one of the previously mentioned logic attacks using our implementation.

Finally, we presented the results of fuzz–testing the USB mass storage class driver provided by all previously mentioned operating systems. Amongst several crashes we found a vulnerability inside a USB class driver provided with every version of Windows XP. An additional crash was found inside the disk subsystem of Windows XP, supporting our claim that other kernel subsystems might prove to be fruitful targets as well.

All those results support our conclusion that the Universal Serial Bus provides a real attack vector that should be taken into account when assessing the physical security of a system.

Chapter 8

Future Work

Although our implementation provoked lots of different crashes in the different operating systems tested, a high number of them where not reproducible. This can be attributed to the way we are trying to reproduce a previous device attachment as described in Section 5.4.2. Both approaches described could be improved by taking more information about the underlying protocol into account and applying some slight modifications to the send packets. This way, the mechanism to reproduce discovered crashes could be made more resilient against changes in the flow of communication. Since the mechanism to reproduce device attachments can also be used to implement an exploit in hardware, this would improve the reliability of build exploits as well.

Another shortcoming of our implementation is related to the third–party software in use. The first thing concerns the libusb version used to implement the receiving component. As described in Section 5.4.3, it doesn't support isochronous transfers and thus, prevents us from testing video and audio devices. When libusb 1.0 becomes more stable, the receiving component should be re–implemented based on libusb 1.0.

Some remaining problems with our implementation are related to the Gadget API Framework used for the device emulation component. The used *file system gadget driver* gadgetfs is still marked EXPERIMENTAL in the Linux kernel. This has led to multiple dead–locks of the emulating machine during the development of the fuzzer. Some other minor problems with the gadget framework are rooted in the fact that it modifies certain fields of the descriptors written to it. Since we are just forwarding the descriptors received from a USB device, this behaviour is not desired. For example, the gadgetfs module always sets the release number field (`bcdUSB`) inside the device descriptor to Version 2.0. This is even done for devices that only adhere to Version 1.0 of the specification. In addition to resetting the release number field, which doesn't lead to problems in every case, the gadget framework always sets the `wMaxPacketSize0` field of the device descriptor to the value 64, even if a device reported another value for this field. Trying to forward the communication in such a case has led to various problems.

Due to these problems, we might need to consider in the future whether it makes sense to re–implement the device emulation component using another technology. We don't need any intelligence inside the device emulation component. We only need a way to send and receive raw USB packets. One idea is to alienate a usual USB host controller as a raw packet sending/receiving device. If that would be possible, the implementation could abandon the gadget framework and there would be no need for a hardware peripheral controller anymore. Future research in

this direction will show if this is possible.

Besides the problems mentioned that are related to the implementation, more research into the field of USB security is generally required. USB security is a really broad field. Not every aspect could be covered in detail in the limited time frame provided for this thesis. We tried to give a good overview of security aspects related to the Universal Serial Bus. The following are areas where we think that further research should be conducted in the future.

One area of interest is certainly the Certified Wireless USB (CWUSB) extension [1]. One of the design goals of the wireless USB specification is to keep the current software infrastructure including all the USB device drivers intact. Wireless USB basically only provides the wireless transport mechanism for the USB protocol to remove the need for any cables. To guarantee comparable security, as is the case with wired USB, it implements mutual authentication and encryption between the host and wireless USB devices. What makes wireless USB so interesting is the fact that attacks can now potentially be performed over the air without the need for physical access. The mechanisms used for authentication and encryption should be analyzed for their effectiveness.

Another area which will be in need of more research in the future is the USB packet sniffing attack described in Section 4.2.1. We only described the theoretical possibility of this attack. A proof–of–concept could be created to demonstrate the feasibility of this attack. The next logical step after being able to sniff the communication destined for other devices is the impersonation of these. We need to investigate whether it's possible for a USB device to actively spoof responses of other connected devices. This would allow a malicious USB device to silently replace the data transferred from other connected USB devices to the host. For example, while the malicious USB device is attached, files copied or executed from an attached USB flash drive could be replaced when accessed from the host.

Bibliography

[1] Agere, Hewlett-Packard, Intel, Microsoft, NEC, Philips, and Samsung. *Wireless Universal Serial Bus Specification 1.0*, May 2005. `http://www.usb.org/developers/wusb/` [2008/11/04].

[2] Hewlett-Packard, Intel, Microsoft, NEC, ST-NXP Wireless, and Texas Instruments. *Universal Serial Bus Specification 3.0*, November 2008. `http://www.usb.org/developers/docs/` [2008/11/17].

[3] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips. *Universal Serial Bus Specification 2.0*, April 2000. `http://www.usb.org/developers/docs/` [2008/07/26].

[4] *Unicode Standard, Version 5.0*. Addison-Wesley Professional, 2006.

[5] Jan Axelson. *USB Complete: Everything You Need to Develop Custom USB Peripherals*. Lakeview Research, 2001.

[6] USB Implementers Forum, Inc. Approved Class Specification Documents. `http://www.usb.org/developers/devclass_docs#approved` [2008/12/09].

[7] Technical Committee T10. *SCSI Primary Commands - 2 (SPC-2) 5.0*, September 1998.

[8] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, April 2008. `http://www.intel.com/products/processor/manuals/` [2008/07/16].

[9] Aleph One. Smashing The Stack For Fun And Profit. 1996. `http://www.phrack.org/issues.html?id=14&issue=49` [2008/06/28].

[10] anonymous. Once upon a free()... 2001. `http://www.phrack.org/issues.html?issue=57&id=9#article` [2008/08/03].

[11] Doug Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html` [2008/07/29].

[12] Poul-Henning Kamp. Malloc(3) revisited. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 36–36, Berkeley, CA, USA, 1998. USENIX Association.

[13] Michel "MaXX" Kaempf. Vudo malloc tricks. 2001. `http://www.phrack.org/issues.html?issue=57&id=8#article` [2008/08/02].

[14] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th USENIX Large Installation Systems Administration Conference (LISA)*. USENIX Association, 2003.

[15] Bruce Perens. Electric fence. `http://directory.fsf.org/project/ElectricFence/` [2008/07/27].

[16] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the International Conference on Information and Communication Security (ICICS 2006)*, Raleigh, North Carolina, U.S.A., December 2006.

[17] Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[18] Scut / team teso. Exploiting format string vulnerabilities. September 2001. `http://packetstormsecurity.org/papers/unix/formatstring-1.2.tar.gz` [2008/07/03].

[19] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[20] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[21] Oulu University Secure Programming Group. PROTOS - Security Testing of Protocol Implementations. `http://www.ee.oulu.fi/research/ouspg/protos/` [2008/08/01].

[22] Dave Aitel. The Advantages of Block-Based Protocol Analysis for Security Testing. 2002.

[23] Linux USB Project. `http://www.linux-usb.org/` [2008/08/23].

[24] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[25] Libusb Project. `http://libusb.sourceforge.net/` [2008/08/28].

[26] jUSB Project. `http://jusb.sourceforge.net/` [2008/08/28].

[27] Linux-USB Gadget API Framework. `http://www.linux-usb.org/gadget/` [2008/08/24].

[28] Greg Kroah-Hartman. udev - a userspace implementation of devfs. In *Proceedings of the Linux Symposium*, pages 249–257, 2003.

[29] Hardware Abstraction Layer (HAL) Specification. `http://www.freedesktop.org/wiki/Software/hal` [2008/08/29].

[30] Linux USB Device Driver Support. `http://www.linux-usb.org/devices.html` [2008/08/28].

[31] Apple Inc. *I/O Kit Fundamentals - Hardware & Drivers*, May 2007. `http://developer.apple.com/documentation/DeviceDrivers/Conceptual/IOKitFundamentals/` [2008/09/12].

[32] Apple Open Source Darwin Releases. `http://www.opensource.apple.com/darwinsource/` [2008/09/16].

[33] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[34] *Microsoft Developer Network: Windows Driver Kit.* http://msdn.microsoft.com/en-us/library/aa972908.aspx [2008/08/13].

[35] Microsoft Windows Hardware Developer Central USB FAQ. http://www.microsoft.com/whdc/connect/usb/usbfaq_intro.mspx [2008/08/13].

[36] Penny Orwick. *Developing Drivers with the Windows Driver Foundation.* Microsoft Press, Redmond, 2007.

[37] Microsoft Component Object Model Technologies. http://www.microsoft.com/com/default.mspx [2008/09/24].

[38] Microsoft Developer Network: Using and Configuring AutoPlay. http://msdn.microsoft.com/en-us/library/bb776829(VS.85).aspx [2008/09/18].

[39] Microsoft Developer Network: Autorun.inf Entries. http://msdn.microsoft.com/en-us/library/bb776823.aspx [2008/09/17].

[40] Microsoft USB Storage - FAQ for Driver and Hardware Developers. http://www.microsoft.com/whdc/archive/usbfaq.mspx [2008/09/18].

[41] U3 Standard. http://www.u3.com [2008/09/30].

[42] Wesley McGrew. Hacking U3 Smart USB Drives. http://www.mcgrewsecurity.com/research/hackingU3/ [2008/09/30].

[43] Apple Inc. Quick Look Programming Guide. February 2008. http://developer.apple.com/documentation/UserExperience/Conceptual/Quicklook_Programming_Guide/Quicklook_Programming_Guide.pdf [2008/10/07].

[44] Apple Inc. Spotlight Overview. May 2007. http://developer.apple.com/documentation/Carbon/Conceptual/MetadataIntro/MetadataIntro.pdf [2008/10/07].

[45] Microsoft Windows Device Simulation Framework. http://www.microsoft.com/whdc/devtools/DSF.mspx [2008/11/08].

[46] The Metasploit Project. http://www.metasploit.com/ [2008/11/12].

[47] Kostya Kortchinsky. Exploiting Kernel Pool Overflows. June 2008. http://www.immunityinc.com/downloads/KernelPool.odp [2008/09/30].

List of Tables

List of Figures

Listings

Declaration

I hereby declare that I created this thesis myself, without any outside help and without using other means of research than those listed in the attached bibliography. All quotes — both literal and according to their meaning — from other publications have been marked accordingly.

I agree to the public display of this thesis in the department's library.

Hamburg, May 7, 2009,

(Moritz Jodeit)