BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

IMPLEMENT RUBBER DUCKIES ON AVAILABLE USB DEVICES AND MAKE A PRACTICAL TEST

IMPLEMENTACE RUBBER DUCKIES NA BĚŽNĚ DOSTUPNÝCH USB ZAŘÍZENÍCH A JEJICH PRAKTICKÝ TEST

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **HUNG DO**
AUTOR PRÁCE

SUPERVISOR **Ing. MAREK TAMAŠKOVIČ,**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

147782

Institut:	Department of Intelligent Systems (UITS)
Student:	**Do Hung**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Implement Rubber Duckies on Available USB Devices and Make a Practical Test**
Category:	Security
Academic year:	2022/23

Assignment:

1. Study how the Universal Serial Bus (USB) works and describe it. Next, study and describe how BadUSB and RubberDucky attacks work.
2. Implement RubberDucky on a Raspberry Pico platform working on platforms Microsoft Windows 10 and later and GNU/Linux with kernel 5.15 and later:
    1. The device will behave like a keyboard and execute a desired command in command line (execute reverse shell, blue screen / kernel panic, otherwise specified command executable in the command line).
    2. The device will behave as a virtual USB Hub with a virtual USB memory and a virtual Rubber Ducky device with the above specification.
3. Study and find ways to defend against RubberDucky attacks on Microsoft Windows and GNU/Linux operating systems.
4. Test the functionality of these techniques using your RubberDucky implementation.
5. Evaluate acquired results.

Literature:

- van Woudenberg, J., & O'Flynn, C. (2021). The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks
- USB Specification 2.0, Dostupné z: https://www.usb.org/document-library/usb-20-specification
- USBCaptchaIn: Preventing (un)conventional attacks from promiscuously used USB devices in industrial control systems
- Hou, Hao-Hsun ; 2018, Method for Preventing BadUSB Attack
- NEUNER, Sebastian, Artemios G. VOYIATZIS, Spiros FOTOPOULOS, Collin MULLINER a Edgar R. WEIPPL. USBlock: Blocking USB-Based Keypress Injection Attacks. In: *Data and Applications Security and Privacy XXXII* [online]. Cham: Springer International Publishing, 2018, s. 278-295 [cit. 2022-10-20]. ISBN 9783319957289. ISSN 0302-9743. Dostupné z: doi:10.1007/978-3-319-95729-6_18

Requirements for the semestral defence:
1-2.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Tamaškovič Marek, Ing.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	3.11.2022

Abstract

This thesis deals with computer security attack named BadUSB, implements an example device (Rubber Ducky) and looks for a defense against these types of attack. My task is to analyze the functionality of Universal Serial Bus, communication between the host and device and its shortcomings against BadUSB attacks. For that I implemented a composite USB device on Raspberry Pi Pico using an external open source library TinyUSB. And with a working prototype I present reader a few possible defense mechanisms against these kinds of devices and attacks on Microsoft Windows and GNU/Linux operating systems. The contribution of this thesis is to warn people about the weakness of USB's plug-and-play feature and the danger of connecting unknown USB devices to our computers.

Abstrakt

Tato práce se zabývá problematikou počítačových útoků typu BadUSB, implementací jednoho ze zařízení (Rubber Ducky) a následnou obranou proti těmto typům útoků. Mým úkolem je rozbor funkcionality univerzální sériové sběrnice, komunikace mezi hostem a zařízením a slabiny proti BadUSB útokům. Za tímto účelem jsem implementoval composite (složené) USB zařízení při použití mikrokontroleru Raspberry Pi Pico a externí open source knihovny TinyUSB. S funkčním prototypem prezentuji čtenáři několik možných ochranných mechanismů před těmito druhy útoků na operačních systémech Microsoft Windows a GNU/Linux. Tato práce varuje lidi před nevýhodami USB funkce plug-and-play a nebezpečím při připojování neznámých USB zařízení do našich počítačů.

Keywords

cyber security, USB, Rubber Ducky, BadUSB, Raspberry Pi Pico W, TinyUSB, lwIP, whitelist, payload, scripting language, embedded device

Klíčová slova

kyberbezpečost, USB, Rubber Ducky, BadUSB, Raspberry Pi Pico W, TinyUSB, lwIP, bílá listina, payload, skriptovací jazyk, vestavěné zařízení

Reference

DO, Hung. *Implement Rubber Duckies on Available USB Devices and Make a Practical Test.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Marek Tamaškovič,

Rozšířený abstrakt

Univerzální sériová sběrnice neboli zkráceně USB je populární periferie pro připojování zařízení k počítači. Před vynálezem USB existovalo nespočet periferií s různými tvary konektorů a protokolů. To dosti omezovalo jak uživatele, kteří mnohdy neměli potřebné porty na počítači, tak i vývojáře, kteří se museli rozhodovat mezi použitím dostupných periferií, nebo vytvořením nových. To vedlo k vývoji nové periferie a v roce 1995 byla představena první specifikace USB rozhraní. Rozhraní podporovalo jak datový přenos, tak i napájení. Novinku představovala vlastnost "plug-and-play" neboli "připoj a hraj", která zjednodušila manipulaci s daným zařízením. Po zapojení do počítače je zařízení ihned rozpoznáno a systém automaticky přidělí potřebné ovladače (drivery).

Bohužel tato jednoduchost a důvěra ve všem, co uživatel připojí do počítače skrze USB rozhraní, přinesla nový druh malwaru. V roce 2014, na Black Hat konferenci ve Spojených státech amerických, vědci ze Security Research Labs představili nový škodlivý útok nazvaný BadUSB. BadUSB je USB zařízení, jehož starý firmware byl přepsán a jenž rozšiřuje vlastnost zařízení o novou funkcionalitu (např. klávesnici nebo síťovou kartu). Do zařízení se též předem naprogramuje skript, který je uložen v nedostupné paměti, a po připojení zařízení do počítače začne novou funkcionalitu vykonávat. Běžné antivirové programy nedokážou tento typ útoku detekovat, a proto je tento malware velice nebezpečný.

Tato práce se zaměřuje na implementaci zařízení schopného vykonávat tento druh útoku. Vychází z USB zařízení vytvořeného skupinou Hak5 jménem Rubber Ducky. K vývoji byl použit jednočipový počítač Raspberry Pi Pico řady W, který obsahuje Wi-Fi modul. Implementace je rozdělena do dvou částí: první část je zaměřena na vygenerování firmwaru pro Raspberry Pi Pico a druhá část se zaměřuje na generování (útočných) skriptů a jejich nahrávání na Raspberry Pi Pico zařízení.

Software pro generovaní firmwaru je napsaný v jazyce C a pomocí Pico-SDK, TinyUSB, lwIP a CYW43-driver knihoven. TinyUSB knihovna obsahuje funkce potřebné k definování vlastnosti USB zařízení; v tomto případě chceme, aby se zařízení chovalo zároveň jako klávesnice a zároveň paměťové médium. lwIP knihovna definuje funkce potřebné k vytvoření TCP serveru pro možnou komunikaci s externím zařízením. A CYW43-driver knihovna slouží k inicializaci Wi-Fi modulu na Raspberry Pi Pico. Software též povoluje nastavení automatického vykonávání příkazů stisku kláves ihned po zapojení zařízení do počítače, nebo možnost opětovného vykonání skriptu pomocí tlačítka CapsLock na klávesnici.

Generování a nahrávání skriptů je napsané v jazyce Python. Uživatelé si vytvářejí svoje skripty ve speciálně definovaném jazyce. Ty jsou pak předány programu, který provede lexikální, syntaktickou a sémantickou kontrolu pomocí regulárního výrazu jazyka. Po kontrole mohou být data překonvertována buď do C zdrojového souboru, nebo jsou přeposlána přes TCP spojení bezdrátově.

Druhá polovina práce se zaměřuje na otestování výsledné implementace, rozbor možných škodlivých skriptů a testování dostupných obranných nástrojů na operačních systémech Windows a GNU/Linux. Mezi programy byly zařazeny USBGuard a Kaspersky Security Endpoint. Oba programy používají jiný přístup k potlačení útoku. Výchozí nastavení USBGuardu blokuje všechna nově připojená USB zařízení a pro jeho zprovoznění je potřeba, aby uživatel manuálně zařízení povolil. Kaspersky program na druhou stranu blokuje jen zařízení, která se chovají jako klávesnice. Uživateli se po zapojení klávesnice na obrazovce zobrazí okno, kterým provede autentizaci zařízení.

Hlavním přínosem této práce je upozornění společnosti na tomto druh útoku. Popisuje jeho myšlenku, jak doopravdy funguje a jak se proti němu bránit. Implementací této práce

chci dát veřejnosti možnost experimentovat s modifikovaným zařízením a pochopit tuto problematiku.

Implement Rubber Duckies on Available USB Devices and Make a Practical Test

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. Marek Tamaškovič. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Hung Do
May 9, 2023

Acknowledgements

I would like to express my deepest appreciation to my supervisor Mr. Marek Tamaškovič, for providing me with the opportunity to work on this very interesting topic and for his advice throughout the writing of my thesis.

Contents

Chapter 1

Introduction

Universal Serial Bus (USB) is the most widely-used connector for modern computer systems. It has replaced many older interfaces, such as parallel and serial ports, and has become the standard for connecting devices such as keyboards, mice, printers, and storage devices to computers. The main selling point of the USB interface was the introduction of a „plug-and-play“ feature which removes the need for the users to configure the device. Upon connecting the device to the machine, the system automatically recognizes the device and immediately assigns appropriate drivers.

Unfortunately, as the popularity of the interface grew, new types of malicious attacks against USB began to emerge. In 2014, a group of researchers from Security Research Labs announced a new kind of USB malware called BadUSB. They demonstrated a USB device with modified firmware that could spoof a keyboard, network card, and display. And this type of malware is undetectable by conventional antivirus programs.

The main focus of this work is to design and implement this type of device on the Raspberry Pi Pico board, as well as to evaluate what the finished product is capable of. Numerous defense mechanisms that have been created since the introduction of the malware. So I tested my device against several of them to see how it performed.

The thesis aims to provide readers with an overview of the BadUSB attack security issue so that they can better understand what makes them so dangerous, how they work, and how to defend against them. My implementation, therefore, gives readers a low-cost and simple tool to work with the BadUSB device.

Chapter 2

Universal Serial Bus

Universal Serial Bus (also known as USB) is a peripheral interface used to connect external devices to computers. It defines the specifications of cables, sets of protocols, the speed of data exchange, and the way the host and the device communicate. It can also send power to devices (for example, to charge smartphones).

In this chapter, we will discuss the interface's history, which dates back to 1995. Then we analyze how the USB protocol is structured: how does the host learned about the device, what the communication between the device and the host looks like, and what a USB device needs to act as a keyboard. All the information about USB was drawn from a book USB Complete by Jan Axelson [4].

2.1 History

Before the invention of the USB peripheral, countless kinds of ports were in any shape and size. In the past, every peripheral had its own uniquely shaped connector, a protocol through which it communicated with the computer, and a limited number of devices it could run at once. And that brought many disadvantages.

First, computer manufacturers had to decide which ports to include in the final motherboard. We could usually find ports like PS/2 for connecting keyboards and mice or VGA connectors for connecting monitors on the old machines. But what if the user wanted to attach a device (for example, a scanner or a printer) whose port was not on the machine? They usually had to go outside, purchase dedicated cards, and manually install them afterward. That is something a person without computer experience might have struggled with. And as for developers, during the development of new computer accessories, they had to decide whether to use one of the existing interfaces but run into a risk of being stuck with its original protocols that do not provide enough features the developer needs or design a new interface which is very expensive.

That led to the development of a new interface. In 1995 a group named **USB Implementers Forum** (also known as USB-IF) was formed by these seven companies: Compaq, DEC, IBM, Intel, Microsoft, NEC, and Nortel. They aimed to create an interface with these goals in their mind:

- **Easy to use** – The user doesn't need to configure and set up a device.
- **Fast** – To minimize the delay in communication between the host and device and to be able to transfer

- **Reliable** – To lower the occurrence of errors and automatic error handling.
- **Versatile** – Many kinds of peripherals can use the interface.
- **Inexpensive** – So that the price of a final product can be as low as possible.
- **Supported by all operating systems** – To let developers effortlessly create new drivers.

A year later, USB-IF released the first version of the USB interface called **USB 1.0**. The new interface allowed a user to connect different kinds of peripherals, such as printers, keyboards, mobile devices, and much more, using a single, standardized interface socket. It also lets the user connect and disconnect a peripheral whenever needed without turning off a computer. And at last, a feature called „plug-and-play“ was introduced. It shows the simplicity of the USB – the user plugs the device into the computer and can immediately use it[1].

But it wasn't until the introduction of **USB 1.1** in 1998 that the interface started to be widely used. In that year, the new operating system Windows 98 included support for USB. Version 1.1 also introduced two speeds: Low Speed with 1.5 Mbps and Full Speed with 12 Mbps.

Over the next 20 years, USB has been constantly being developed. In April 2000, **USB 2.0** came out with a new maximum transfer rate of 480 Mbps. It was called High Speed. Eight years later, in November 2008, USB-IF released a new specification for **USB 3.0** with an even faster transfer speed of 5 Gbps (SuperSpeed USB). As for the time of writing this thesis, USB-IF has released the specification for **USB4 2.0** with a maximum transfer rate of 80 Gbps and power delivery of 240 W (48 V, 5 A).

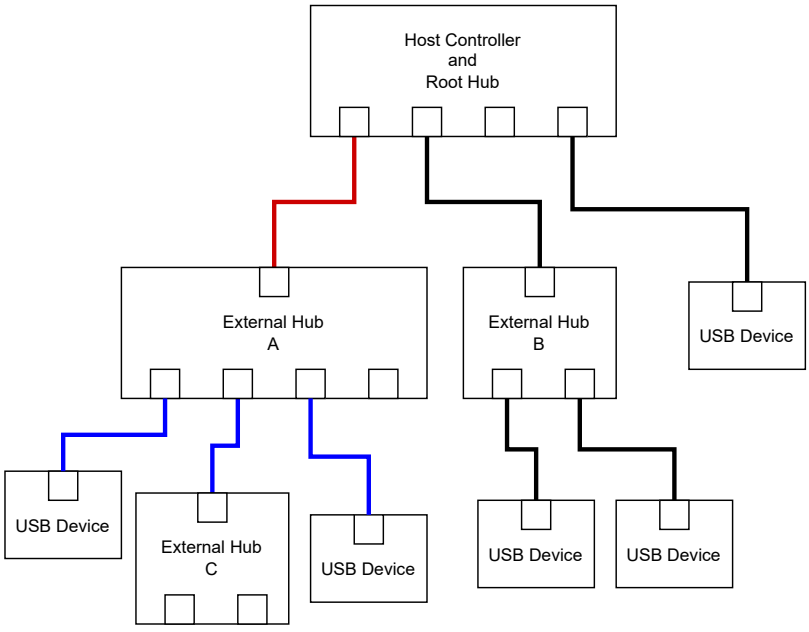

Figure 2.1: An example of USB topology. A red line represents an upstream connection and blue lines downstream connections.

[1]Link to the article about USB history: `https://www.intel.com/content/www/us/en/standards/usb-two-decades-of-plug-and-play-article.html`

2.2 USB topology

USB communication requires two components to work: a host machine with USB support and one or more USB devices. The host machine consists of a USB **host controller**, which manages the communication on the bus, and a **root hub**, which connects external USB devices and USB hubs to the host machine. Together they detect newly attached devices and transfer requests from the host to the device. USB protocol supports up to 127 simultaneously connected devices, including hubs.

Every USB hub creates a star-shaped topology. It typically has two, four, or seven USB ports. Each USB hub has one upstream-facing connector, which the devices use for communicating with the host, and one or multiple downstream connections, each leading to the USB sockets, which are the connected devices use to transfer data to the devices. USB hubs can be connected in series, as shown in Figure 2.1.

Usually, the host machine initiates the USB communication, and the devices are required to respond to incoming requests. The USB specification defines a list of request calls for each device class that the host can send to the device. The device's chip must be able to accept the request and adequately respond to them. That usually requires the device to move data to a buffer to send it back to the host.

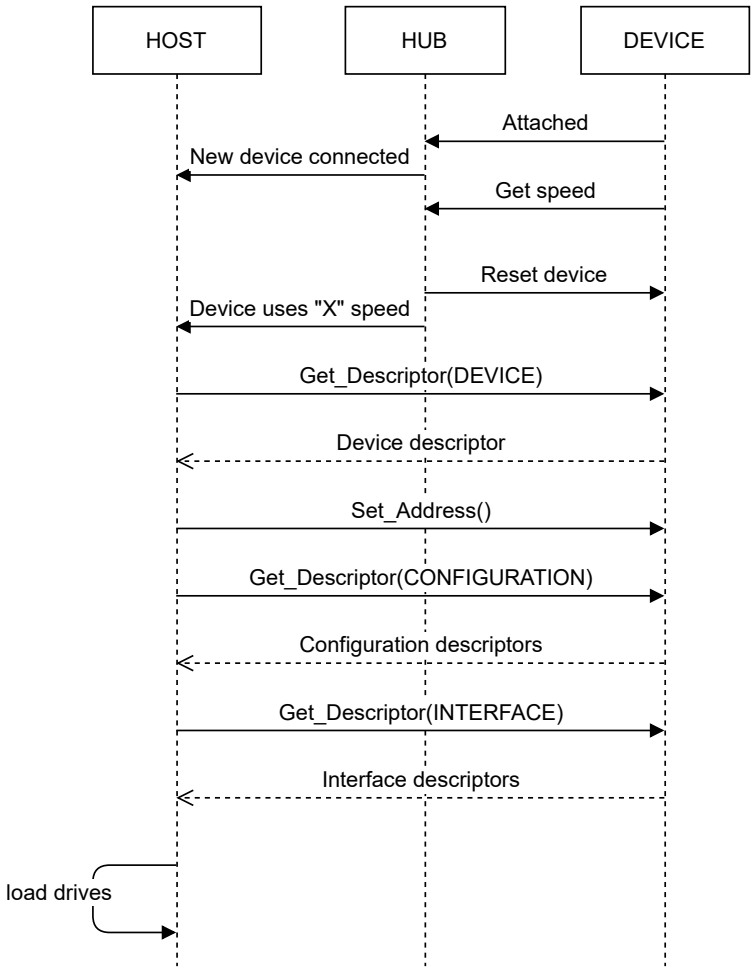

Figure 2.2: USB enumeration process in sequence diagram.

2.3 Enumeration

The interaction between the host and the device starts when the device is connected to the host machine. The host needs to identify what kind of device has been plugged in. The process of learning the device's functionality is called **enumeration**. When the host detects that a new device has been attached, it starts by requesting what speed the device supports for communicating. Depending on the version of the USB, it can support *low speed*, *full speed*, *high speed*, or *SuperSpeed*. The host sends multiple requests to retrieve device descriptors and configuration data (more in Section 2.4). Once all information has been pull, the host assigns an address and loads the device drives. Figure 2.2 depicts the whole series of events.

2.4 Descriptors

As mentioned in the previous section, the host uses an enumeration process to get all descriptors from the device. The **descriptors** contain crucial information that describes the capabilities of the device. Each device must have these four descriptors defined:

- device descriptor,
- configuration descriptor,
- interface descriptor,
- and endpoint descriptor.

Other types of descriptors are not required by the host. If a device supports multiple speeds (full and high speed), it needs to define additional configuration descriptors for each kind of speed (*device_qualifier* and *other_speed_configuration*). Another commonly used descriptor is a *string descriptor*, which allows the host to retrieve descriptive text from the device. Table 2.1 shows a shortened list of descriptor types with their corresponding byte value used in the `Get Descriptor` request.

Descriptor type	Value
device	0x01
configuration	0x02
string	0x03
interface	0x04
endpoint	0x05
device_qualifier	0x06
other_speed_configuration	0x07

Table 2.1: Table of seven most used descriptors and their corresponding identification byte value. In total, there are 18 descriptor types.

Device descriptor

Device descriptor is the first descriptor requested by the host requests upon connecting the device to the host machine. It comprises device identification information such as *Product*

ID, *Vendor ID*, and *Serial Number*, as well as information that the host needs to get further data such as *Device Class* or the *number of configurations*. The host retrieves this descriptor using the `Get Descriptor` request with the parameter byte set to `0x01`.

Configuration descriptor

After receiving the device descriptor, the host proceeds to retrieve configuration, interface, and endpoint descriptors. The configuration descriptor specifies the device's functionalities. The device can support multiple configurations based on the power use, but only one is active at a given time. Each configuration holds information that tells the host how many interface descriptors and endpoint descriptors are present in the response buffer. The host retrieves this descriptor using `Get Descriptor` request with the parameter byte set to `0x02`.

Interface descriptor

An interface descriptor tells the host what the device is capable of doing. The descriptor contains an interface class, subclass, protocol, and number of endpoints. The interface class[2] is what defines the device's functionality. Some interface classes (such as Human Interface Device class, or shortly HID) require additional descriptors to be defined and sent to the host. In the case of HID, its descriptor defines the format of a *report* which is HID means of transporting data between the host and the device. The interface descriptors and their subordinate descriptors are usually sent together with a configuration descriptor request.

Endpoint descriptor

Finally, the endpoint descriptor defines information about the endpoint address. It contains a direction (IN if data are sent to the host, and OUT if data are received from the host) and transfer type. There are, in total, four transfer types:

Control This transfer is primarily used for standard requests, such as the `Get Descriptor` request.

Interrupt Interrupt transfer is implemented on devices that need data to be sent to the host as soon as possible. We can associate interrupt transfers with HIDs.

Bulk This transfer is usually used when the transferring speed is not critical, such as sending data to the printer or reading/writing data to the disk.

Isochronous Isochronous transfer guarantees the delivery of data, but no error correction is present. It is usually used to transfer audio and video in real-time. The device does not re-transmit lost or corrupted data.

Every device must have Endpoint 0 configured for control transfer. The endpoint descriptors and their subordinate descriptors are sent together with a configuration descriptor request.

An example of the device's descriptor structure can be seen in Figure 2.3.

[2] A list of interface classes can be found here: `https://www.usb.org/defined-class-codes`

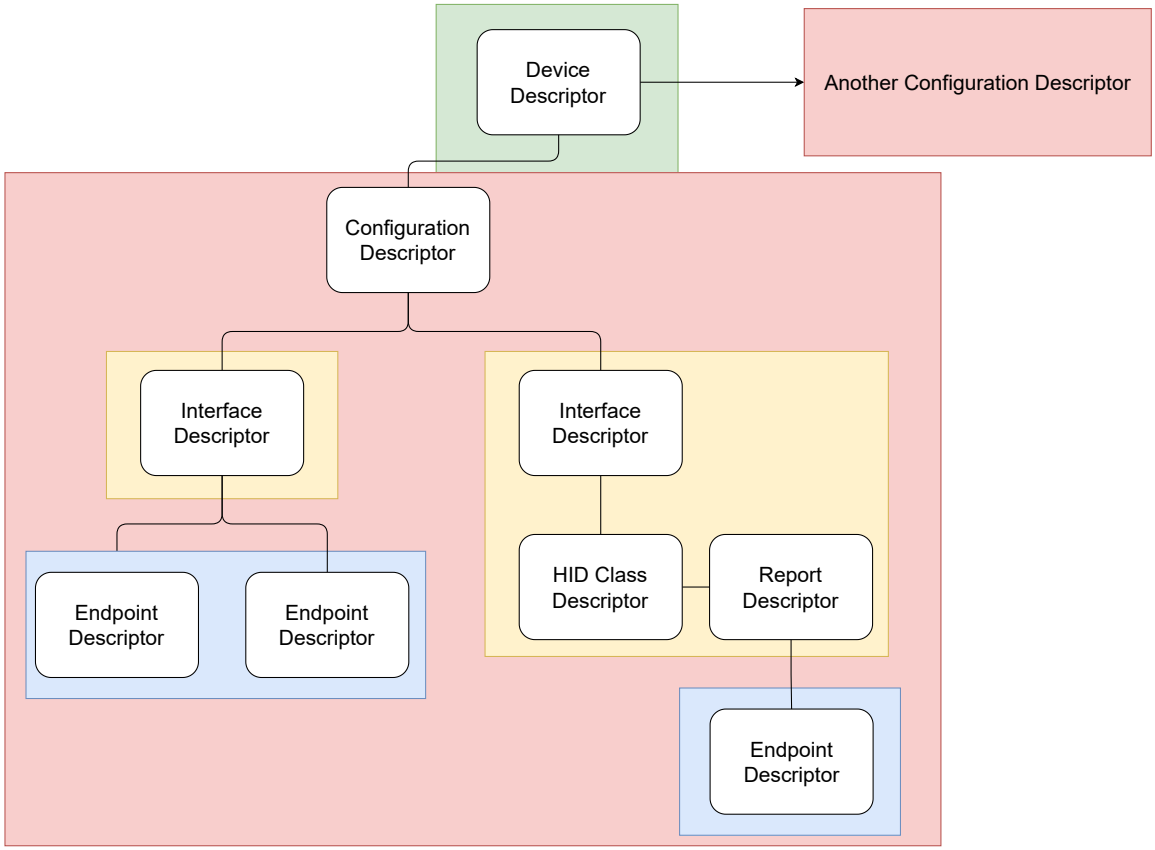

Figure 2.3: A descriptor structure hierarchy of a USB composite device. A green box marks data that can be retrieved with `Get Descriptor(device)` request and a red box is retrieved with `Get Descriptor(configuration)` request. Yellow boxes highlight the interface descriptor and its subordinate descriptors and blue boxes mark out endpoint descriptors.

2.5 USB Device Classes and Human Interface Device

USB introduced device classes to group devices that provide similar functionality. All cameras are designed to capture a video, and all speakers are made to play music. The group can be identified by three bytes named: *Base Class*, *SubClass*, and *Protocol*. The information is stored either in Device descriptor, Interface descriptor, or both[3].

One of the classes is named Human Interface Device (or HID). Peripherals input devices, such as keyboards, mice, or game consoles, are included in the class. Each device in this class must define a **report** structure to communicate with the host. The host pulls this structure during the enumeration phase. The report structure is very flexible, and the user can easily create a new one for its custom device. The host periodically polls from the device IN endpoint. When the device wants to send data to the host, it creates an instance of the report filled with given data and sends it as a reply to the poll token. If not defined differently, the host can send data to the device through the `Set Report` request using control transfer (Endpoint 0).

[3]Defined class codes can be found here: `https://www.usb.org/defined-class-codes`

Chapter 3

BadUSB

BadUSB is a computer security attack that targets peripherals that use USB interfaces. This attack involves modifying a device's firmware to act as a different kind of device, such as a keyboard or a network card. Unlike the usual USB-related attacks, which involve a removable storage device to carry harmful executable files and immediately run them upon plugging into the host machine, BadUSB attacks are immune against antivirus programs since the actual code is stored inside an inaccessible section of memory.

3.1 First appearance

BadUSB was initially revealed in 2014 at the Black Hat conference in the USA. Three security researchers from Security Research Labs, Karsten Nohl, Jakob Lell, and Sascha Krißler, presented a collection of proof-of-concept malicious software that highlighted the security weakness of the USB[8]. They spent months patching the firmware of a thumb drive by listening to the USB communication using a Wireshark[1] sniffing tool, decoding the communication, and creating modified firmware. During the conference, they demonstrated three devices. The first was an infected USB stick used to steal a sudo password on the Linux systems, the second was a USB thumb drive that changed DNS settings in Windows, and the third was an Android device that redirected the network traffic. An original presentation can be found here [15].

The team also briefly discussed a list of five potential defense ideas:

Whitelist USB devices Only selected USB devices will be allowed to communicate with the host. Unfortunately, USB devices have no reliable identifier since not all have a unique serial number. In 2018, Hessam Mohammadmoradi and Omprakash Gnawali presented a work that deals with this problem[14].

Block critical device classes, block USB completely This will reduce the usability of the USB interface. And only a few device classes can be used for abuse.

Scan peripheral firmware for malware Very difficult and not always possible.

Use code signing for firmware updates The main problem with this approach is that billions of old devices that will remain susceptible.

Disable firmware updates in hardware A very simple and effective approach.

[1] Link to official homepage: `https://www.wireshark.org`

3.2 BadUSB devices and attacks

In this section, we will discuss in detail some of the attacks that are related to the BadUSB.

Rubber Ducky

Rubber Ducky is a modified USB thumb drive designed by Hak5 group[2]. The device emulates a keyboard. The group created a scripting language called DuckyScript[3] that is used to create a set of key presses in the form of a payload. The payload is then compiled and stored on the microSD card of the Rubber Ducky device. Once the device is connected to the computer, it will immediately execute the pre-defined keypresses. Another reprogrammable board that can be used for the same purpose is a Teensy USB development board[4].

BadAndroid

BadAndroid is a modified Android device designed to execute an attack over USB. The device emulates a network card (USB Ethernet) which will alter the network routing of the victim's machine. It can change the IP address of the default gateway to the IP address of the Android device, meaning all the network traffic is routed through the Android device (man-in-the-middle attack model). Or it can change the entries of the system's DNS server and, therefore, redirects the communication to the server controlled by the attacker. This attack requires the Android to be rooted[5].

BadUSB Cable

BadUSB cables, such as USBNinja, resemble standard USB charging cables but contain a programmable chip inside. They function similarly to the Rubber Ducky device by emulating a keyboard and carrying a malicious payload[6].

BadUSB-C

The concept of BadUSB-C was introduced in work by Hongyu Lu and his team[13]. The attack uses the Type-C connection's ability to transfer up to 10 Gb of data per second to its advantage. They designed a device that functions as a keyboard and a video capture card. Figure 3.1 below depicts the device's attacking mode. The device can wirelessly transmit the victim's computer screen content and accept keyboard inputs from the attacker. This device extends the capabilities of the standard BadUSB device by allowing the attacker to see the screen content.

Other BadUSB attacks concepts

A BadUSB device can also deliver an attack during the machine's boot phase. It can override an existing computer BIOS with the one stored on the device. The machine will

[2]Link to the product: `https://shop.hak5.org/products/usb-rubber-ducky`

[3]Link to the documentation: `https://docs.hak5.org/hak5-usb-rubber-ducky/duckyscript-tm-quick-reference`

[4]Link to official homepage: `https://www.pjrc.com/teensy/`

[5]Link to the source files: `https://github.com/tst-zdouglas/BadAndroid`

[6]https://mg.lol/blog/badusb-cables/

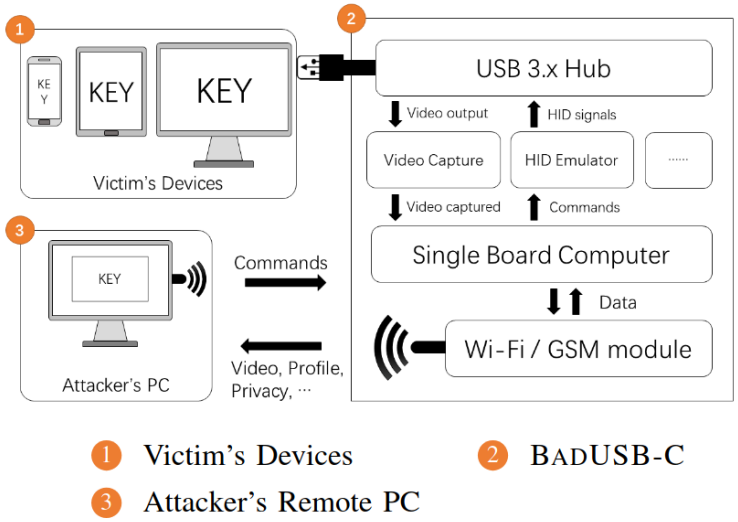

Fig. 3: Attack Model.

Figure 3.1: BadUSB-C's attack model [13].

then boot into the system inside the USB device. This method allows the attacker to execute commands before the actual operating system is loaded. The USB device can also be programmed to brute force the lock screen PIN on an Android phone[7]. The last example is the ability to sniff data from the downstream USB traffic. So if a USB thumb drive is connected to the same hub as the BadUSB device, the device can reconstruct a file that was transferred to the USB thumb drive sent from the host machine.

[7]Link to the payload written in DuckyScript: `https://shop.hak5.org/blogs/payloads/android-pin-brute-force`

Chapter 4

Design and Architecture

In this section, we analyze the key features of this project. The goal here is to break down the whole problem into manageable parts, identify critical issues and come up with a solution.

Since our device is based on Hak5's Rubber Ducky, it must support its main functionalities. The device will be able to execute a series of keystrokes that are stored inside its firmware. The software will provide an easy way to create, generate, and upload these payloads. Together with a keyboard, the device will also include Mass Storage where the users can store a shell basic script or executable file, which they can then run on the host machine.

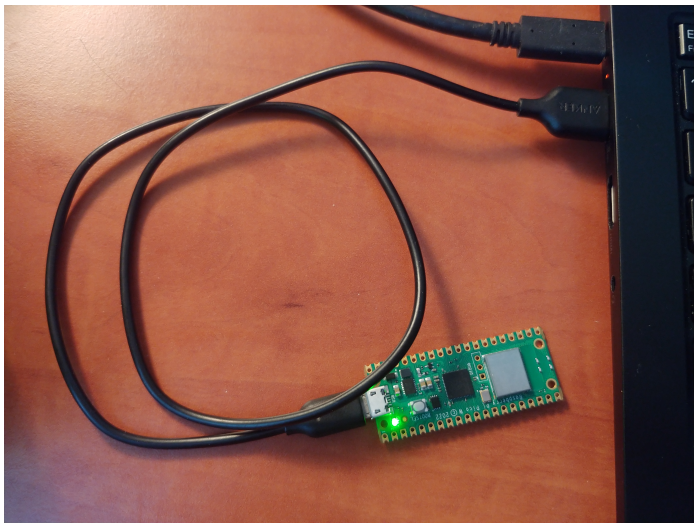

Figure 4.1: A picture of a Raspberry Pi Pico W (the green board) connected to the computer.

4.1 Base device

When designing a device capable of a BadUSB attack, the first thing to consider is which device we should work on. The device needs to have a reprogrammable chip. We also wanted a board that is publicly accessible, inexpensive, and easy to work with. After searching the current market, we came across *Raspberry Pi Pico*, which matches our criteria. **Raspberry**

Pi Pico is a small board with an RP2040 microcontroller chip designed by *Raspberry Pi Foundation*. Its purpose is to encourage people to learn programming and build hardware projects without spending much money on the hardware itself. We chose a newer version of *Raspberry Pi Pico* model *W* (seen in Figure 4.1) as it also comes with a built-in *CYW43439* wireless chip that supports Wi-Fi and Bluetooth. That enables us to add a new use case to our project. And with a well-documented SDK[1] and great support for third-party libraries, this board is a perfect choice for us.

4.2 Custom Rubber Ducky scripting language

First, we need to think of a language that we will use to generate new payloads. The new language has to be intuitive and easy to write in. Since the only thing the devices can produce is keystrokes, we need to find a way to represent each key available on the keyboard. Luckily most (if not all) operating systems come with a keyboard driver preinstalled since it is a commonly used device. For this reason, we don't need to write a driver for our keyboard emulator. USB-IF created a table with a list of supported keys and their IDs[2]. What that means is that we can build our language based on the IDs. That would be great for the machines as all they have to do is upload it directly to our Rubber Ducky device without any processing (apart from converting the ID values to bytes). But unfortunately, that doesn't meet our criteria for the language to be easy to write. After all, typing `0c 11 17 18 0c 17 0c 19 08`[3] feels more like writing a cipher message than a payload.

```
<RD-SCRIPT>           ::= <COMMENT><RD-SCRIPT>   |
                          <DELAY><RD-SCRIPT>     |
                          <PRINTABLE><RD-SCRIPT> |
                          <SPECIAL-COMBINATION><RD-SCRIPT> |
                          <EOL>
<COMMENT>             ::= "#" everything after this is ignored
<DELAY>               ::= "<DELAY " <DELAY-VALUE> ">"
<DELAY-VALUE>         ::= positive whole number
<SPECIAL-COMBINATION> ::= "<" <HOLD-VALUE><MODIFIERS><PRINTABLES> ">" |
                          "<" <HOLD-VALUE><MODIFIERS><SPECIAL-KEY> ">"
<HOLD-VALUE>          ::= positive whole number "-" | ""
<MODIFIERS>           ::= <MODIFIER> "-" <MODIFIERS> | ""
<MODIFIER>            ::= modifier short alias
<SPECIAL-KEY>         ::= "\" <SPECIAL-OR-MACRO>
<SPECIAL-OR-MACRO>    ::= special key name | macro name
<PRINTABLES>          ::= <PRINTABLE><PRINTABLES> | ""
<PRINTABLE>           ::= ASCII printable character
```

Figure 4.2: Language grammar in BNF notation

Another possible way is to map each ID to a key name. That will make it a lot more human-readable. The only thing left to do is to find a way to tell the device to group a set of keystrokes. Without this feature, we would not be capable of producing an upper case `f` since it takes two keys to be pressed all at once: **Shift** and **f** keys.

[1] SDK stands for Software Development Kit

[2] The usage table can be found here: `https://usb.org/sites/default/files/hut1_4.pdf#chapter.10`

[3] If you guessed the word **intuitive** you can call yourself a master of ciphers.

What I designed is a language inspired by a VIM key notation[4]. Its grammar can be represented with a BNF notation seen in Figure 4.2.

The language accepts two types of lexical tokens: *printable keys* and *special combinations*.

Printable keys are a group of keys that can produce an ASCII printable character. In total, there are 95 printable characters in the ASCII table ranging from 32, representing a *space* character, to 126, representing a *tilde* character. However, there are only 48 keys that directly produce a printable character. The second half of the keys also require a *shift* modifier (the exception being a spacebar key which doesn't have a shift counterpart). So a string `Hello World!` produces the following list of keys seen in Figure 4.3:

```
shift+key_h, key_e, key_l, key_l, key_o, spacebar,
shift+key_w, key_o, key_r, key_l, key_d, shift+key_1
```

Figure 4.3: *Hello world!* converted to key presses.

One note here: due to the wide variety of different keyboard layouts, the official key mapping is only compatible with the *US layout*. That means that if the target's machine uses a different keyboard layout than the *US layout*, some keys or key combinations will produce a different string than expected. So, for example, with our machine's keyboard layout set to the Czech QWERTZ, the following set of keystrokes will produce output that the user probably didn't want:

```
Input:     page_123.cz
US layout: page_123.cz
CZ layout: page%+ěš.cy
```

Figure 4.4: Comparing outputs between cs-CZ and en-US layout given the same input string.

A **Special combination** gives the user more control over the key presses. It extends the functionality by adding features that cannot be executed using only *printable keys*. The format of the *special combination* looks as follows:

```
"<" [special_combination_content] ">"
```

Figure 4.5: *Special combination*'s content is wrapped in < and > symbols

There are four features that the user can define within the scope:

- *waiting time* between the keystrokes,
- pressing *non-printable keys*,
- pressing keys with *modifiers* with an option to set a *holding time*,
- force multiple *printable key* pressed simultaneously.

[4]VIM (which is an acronym to Vi IMproved) is a free open-source text editor. It is known among programmers as a text editor filled with keyboard shortcuts which makes coding and writing much faster and more efficient. Unfortunately, VIM has a steep learning curve making it not beginner-friendly. The official documentation for VIM key notation can be found here: `https://vimdoc.sourceforge.net/htmldoc/intro.html#key-notation`

Waiting time or **delay** tells the device how long it has to wait before sending the next series of keystrokes to the host. It is crucial, as omitting it, most of the payloads would fail since the device usually needs to wait until a GUI element loads or a file is downloaded. The format of a delay command is defined as follows: `<DELAY [delay_in_ms]>`. An example usage of the delay command can be seen in Figure 4.6.

```
take <DELAY 5000> dave brubeck
```

Figure 4.6: In this example the USB device will type „take “ string, wait 5 seconds, and finish by typing the string „ dave brubeck“.

Non-printable keys are, as the name suggests, keys that don't produce any printable characters - in other words, all other keys. Of course, there are some notable exceptions. Even though `Keypad_1` or `Keypad_Asterisk` keys all produce a printable character (`'1'`, `'*'` respectively), they are not considered printable keys. The user has to explicitly put them in the *special combination* format to execute them. To differentiate non-printable keys from printable keys, a `\` prefix is attached to the former. Figure 4.7 shows examples of non-printable keys. The parser doesn't distinguish between uppercase and lowercase letters (case-insensitive).

```
<\enter><\SPACEBAR><\BackSpace><\arrow_up><\f12><\num1><\volume_up>
```

Figure 4.7: Examples of non-printable key presses in custom Rubber Ducky script language.

Other additions to *non-printable keys* are supports for *modifier keys* and *holding time*. **Modifier keys** are special keys that temporarily alter the action of a *Normal key* (printable or non-printable key) when pressed together. There are, in total, eight modifier keys:

- left and right `Alt`,
- left and right `Control`,
- left and right `Meta` (also known as Windows, Hyper, Super, or Command key depending on the operating system),
- left and right `Shift`.

We have already encountered a modifier key when discussing printable keys. When we press `Shift` key together with `Key_s`, the *shift* key changes the action of the *s* key to output upper case *S* instead of the lowercase *s* it would generally output. We can also associate modifier keys with keyboard shortcuts, the most famous one being `alt+f4` to close an active window on Windows or `ctrl+s` to save the content of a file. Modifier keys usually do not produce any action when pressed alone (the Meta key being an exception), so they are handled differently when sending keystrokes to the host's machine. In the *special combination* format, modifier keys are placed before Normal keys, as seen in Figure 4.2. Each modifier is identified by the location of the key followed by its starting letter - `'la'` for `LeftAlt`, `'rs'` for `RightShift`, and more. If the user does not specify which one of the keys is meant, the left one will be chosen implicitly - `'m'` will trigger `LeftMeta`, `'c'` will trigger `LeftControl`, and so on. Each key is then separated with a `-` separator.

Holding time, as the name suggests, defines how long a group of keys is meant to be pressed before releasing, measured in milliseconds. The hold delay value is located at the

start of the *special combination*. This field is optional, and the implicit value is set to 0 if not given.

The last thing the *special combination* scope supports is pressing multiple *printable keys* at once. That is especially useful when the user wants to execute a keyboard shortcut that contains two or more Normal keys. For example, we can use Visual Studio Code's[5] keyboard shortcut for closing all files in the editor: `Ctrl+k Ctrl+w`. The language's equivalent of the given keystrokes is `<c-kw>`.

There are some letters that would not work in this format, `<` and `>` being the case. For that, I created a group of *macro keys*. A **Macro key** is an alias to an existing key. It can be an alias to either the Normal key or modifier key and is treated the same way as a *non-printable key* - it starts with an escape backslash character followed by the macro name. So we can use `<\gt>` and `<\lt>` to produce `<` and `>`, respectively.

The last feature of the language is the ability to write *comments*. The comment grammar is inspired by scripting programming languages such as Bash or Python. It starts with `#` symbol, and the characters that follow this symbol are all ignored by the parser until the end of the line. There are no multi-line comments support. If the users want to type a `#` symbol, they put it into the *special combination* scope.

The following Listing 4.1 shows an example of a payload that highlights all the grammar syntax of the language.

```
# ============================================================
# This is a single line comment

# let's open a terminal on Ubuntu using its keyboard shortcut
# and wait 500ms for it to open
<c-a-t><DELAY 500>

# ... now run a command
echo "hello world!"

# ... it doesn't work :( oh wait we must run it first!
<\enter> # yay it outputs hello world!

# lets run some calculation in python now
python -c "a = 2<\enter><#> a = 4<\enter>print(a <\lt> 3)"<\enter>

# the commands from above should produce something like this
# $ python -c "a = 2
# # a = 4
# print(a < 3)"

# now lets find the oldest command run on the machine!
# hopefully 10 seconds will be enough
<10000-\arrow_up><\enter>

# that is the end of our Linux terminal scripting 101 tutorial
```

[5]Visual Studio Code is a popular graphical text editor developed by Microsoft that supports extensions: `https://code.visualstudio.com/`

```
# ============================================================
```

Listing 4.1: Example payload in custom language.

4.3 Communication protocol

Since our development board supports Wi-Fi, we decided to create a communication protocol and a network application based on it. Let our Rubber Ducky device be the *server* and our network application the *client*. We need to define use cases for each side. What follows is a list of requests that we want the client to support:

- Notify the device (server) that the client will send a new payload.
- Send to the device a new series of keystrokes.
- Remove the last sent set of keystrokes.
- Retrieve keystrokes from the device.
- Start the stored payload execution.

The server, on the other hand, will have only two types of responses: an **OK** response, which signalizes that the request was successfully processed, and an **ERR** response with an error message when something unexpected happened while processing the request.

We created a communication protocol on L7 application layer[6] with a diagram shown in Figure 4.8, Figure 4.9, and Figure 4.11. We wanted to simplify the packet's structure as much as possible. The packet header is 4 bytes long, and the rest being filled with the packet's content (payload data).

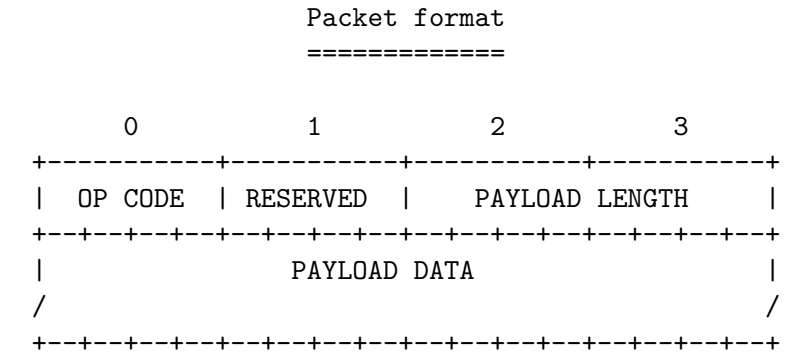

Figure 4.8: Communication protocol's packet format used in the project.

The first byte of the header contains an **operational code**. It tells the server what kind of action we are requesting. It also informs the client about the operation's result. All operational codes are analyzed later in this section. The second byte in the header is **reserved** due to structure padding in the C programming language[10]. The last two bytes in the header store **the size of the payload data**. Two bytes here will give us values ranging from 0 to 65 535, which should be enough to carry the payload.

Next, we can analyze the request packet structure. Our project supports nine *operational codes*, as shown in Figure 4.9. We can divide them into four groups:

[6] More about Internet Network's ISO/OSI model here: `https://www.ietf.org/rfc/rfc1122.txt`

1. work with the USB device's (server's) inner read-write mode,
2. update device payload,
3. analyze the payload stored on the device,
4. run the payload.

```
                              Request
                              =======
Operation codes and their values:
+----+-------------+---------+--------------+--------------------------+
| op |   opcode    | payload |   expected   |       description        |
|    |    name     |  size   |    value     |                          |
+----+-------------+---------+--------------+--------------------------+
| 01 |     SET     |   01    |    00/01     |   Enable/disable device  |
|    |  EDITABLE   |         |              |      read-write mode     |
+----+-------------+---------+--------------+--------------------------+
| 02 |     GET     |   00    |      --      |      Check device's      |
|    |  EDITABLE   |         |              |  read-write mode status  |
+----+-------------+---------+--------------+--------------------------+
| 03 | CLEAR DATA  |   00    |      --      |      Clear device's      | *
|    |             |         |              |     current payload      |
+----+-------------+---------+--------------+--------------------------+
| 04 |  PUSH DATA  |   0d    | key sequence | Send a new key sequence  | *
+----+-------------+---------+--------------+--------------------------+
| 05 |  POP DATA   |   00    |      --      | Remove last key sequence | *
+----+-------------+---------+--------------+--------------------------+
| 06 |  GET DEBUG  |   00    |      --      | Retrieve a key sequence  |
|    |   CURSOR    |         |              |       at the debug       |
|    |             |         |              |     cursor's position    |
+----+-------------+---------+--------------+--------------------------+
| 07 | INC DEBUG   |   00    |      --      | Move debug cursor to the |
|    |   CURSOR    |         |              |    next key sequence     |
+----+-------------+---------+--------------+--------------------------+
| 08 | RESET DEBUG |   00    |      --      |   Reset debug cursor's   |
|    |   CURSOR    |         |              |         position         |
+----+-------------+---------+--------------+--------------------------+
| 09 |     RUN     |   00    |      --      | Tell the device to start | **
|    |  SEQUENCES  |         |              |  executing the payload   |
+----+-------------+---------+--------------+--------------------------+

*  - device must be set to read-write mode to run this command
** - device must be set to read-only mode to run this command
```

Figure 4.9: Set of request's operational codes, their payload, and description.

The first two operational codes fall under the first group. Their purpose is to change or check whether the device is currently in read-write mode. Other actions will depend on it. `SET_EDITABLE` (0x01) sends a single-byte payload of 0x00, which will set the device to *read-only* mode, or 0x01, which will switch the device to *read-write* mode. `GET_EDITABLE` (0x02) retrieves the current mode present on the device.

The second group's functionality is to alter the payload stored on the device. It is required for the device to be in *read-write* mode. Otherwise, the device will respond with

an error message, and the action will be ignored. `CLEAR_DATA` (0x03) action removes the current payload from the device. `PUSH_DATA` (0x04) uploads a new *key sequence* to the device. The payload should contain the *key sequence* data already reshaped to the desired format expected by the device[7]. `POP_DATA` (0x05) removes the last inserted key sequence. Users can retrieve it from the server response payload.

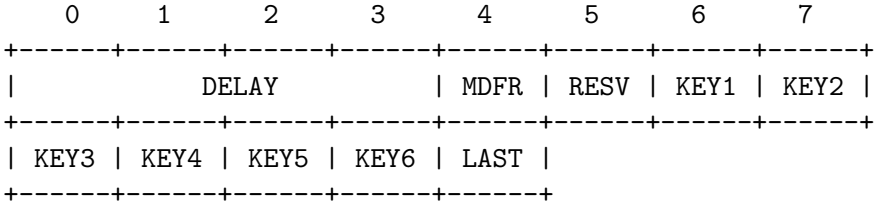

Figure 4.10: Key sequence format.

The third group contains actions that serve as a debug tool for the user. The debug cursor always starts at the beginning of the series. A request with `GET_DEBUG_CURSOR` (0x06) will retrieve a *key sequence* that is under the cursor. To get the following key sequence, the user needs to send `INC_DEBUG_CURSOR` (0x07) request, which moves the cursor to the next one (if available). Finally, the `RESET_DEBUG_CURSOR` (0x08) request moves it back to the beginning.

The last operation on the list is `RUN_SEQUENCES` (0x09) request. As the name suggests, this request will start executing the payload on the device. The only requirement here is that the device must be in *read-only* mode.

Now, we can analyze the last two operation codes, which we can see in Figure 4.11. They are response codes that the server uses to answer incoming requests from the client. The `OK` response is generated when the request has been successfully proceeded. It can carry some data depending on the received request (for example, *key sequence* data as an answer to `POP_DATA` or device mode when received `GET_EDITABLE`). The other type of response is `ERR` which is generated when an error has occurred during the request processing. This response will always carry the error message in its payload.

Response
========

op	name	payload_len	description	payload
0a	OK	varies	Request was successfully processed	data from server
0b	ERR	varies	Request was not correctly processed	error message

Figure 4.11: Set of responses, their content, and description.

[7]The expected format can be seen in Figure 4.10. It is a 13 bytes long structure where *waiting delay* occupies the first 4 bytes, followed by a 1 bitmap of *modifiers* (`MDFR`), a reserved byte, 6 keyboards key IDs (see Section 4.2), and a `LAST` byte, which indicates that this key sequence will be the last one in the series.

Let's end this section with a data flow diagram showcasing a possible communication between the *server* (device) and the *client* (user) shown below in Figure 4.12:

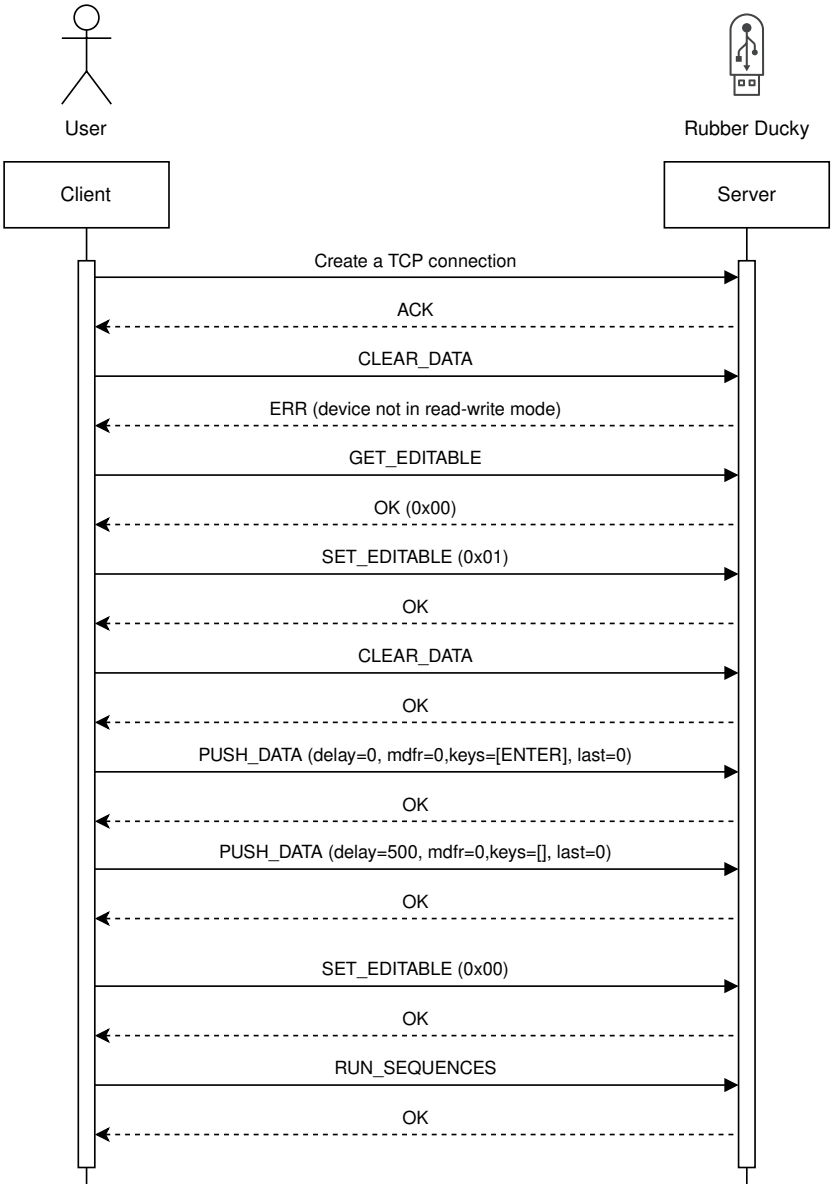

Figure 4.12: Example of the communication between the client and the server. In this example, the user sends two key sequences to the device: `<\enter><DELAY 500>`. We can also see the user making a mistake at the beginning of the communication by sending a `CLEAR_DATA` request when the device is still in *read-only mode*.

Chapter 5

Implementation of Rubber Ducky-like device

The purpose of this section is to familiarize the reader with the actual structure of this project. The goal here is to show the reader how each section of the software is implemented. The design from Chapter 4 will serve as our source, and all implemented parts of this project will be based on it.

The first half of the chapter is dedicated to the Rubber Ducky software, which I wrote using C programming language. All the source codes are present in the `rubber_ducky` directory. The second half of the chapter deals with the Rubber Ducky scripting language and the client application for which I used Python, and the code is located in the `rd_client` directory.

5.1 Used third-party libraries

This project was developed using four open-source an SDK and libraries:

pico-sdk The Raspberry Pi Pico SDK (henceforth the SDK) provides the headers, libraries and build system necessary to write programs for the RP2040-based devices such as the Raspberry Pi Pico in C, C++, or assembly language.[1]

cyw43-driver An open-source library which implements a driver for CYW43xx Wi-Fi/BT SoC.[2]

TinyUSB TinyUSB is an open-source cross-platform USB Host/Device stack for embedded systems, designed to be memory-safe with no dynamic allocation and thread-safe with all interrupt events deferred and then handled in the non-ISR task function.[3]

lwIP lwIP library is a small independent implementation of the TCP/IP protocol suite. The focus of the lwIP TCP/IP implementation is to reduce the RAM usage while still having a full-scale TCP.[4]

[1] `https://github.com/raspberrypi/pico-sdk`
[2] `https://github.com/georgerobotics/cyw43-driver`
[3] `https://github.com/hathach/tinyusb`
[4] `https://github.com/lwip-tcpip/lwip`

5.2 Rubber Ducky device

This section gives an overview of how the Raspberry Pi Pico firmware section is implemented. The class diagram can be seen in Figure 5.1.

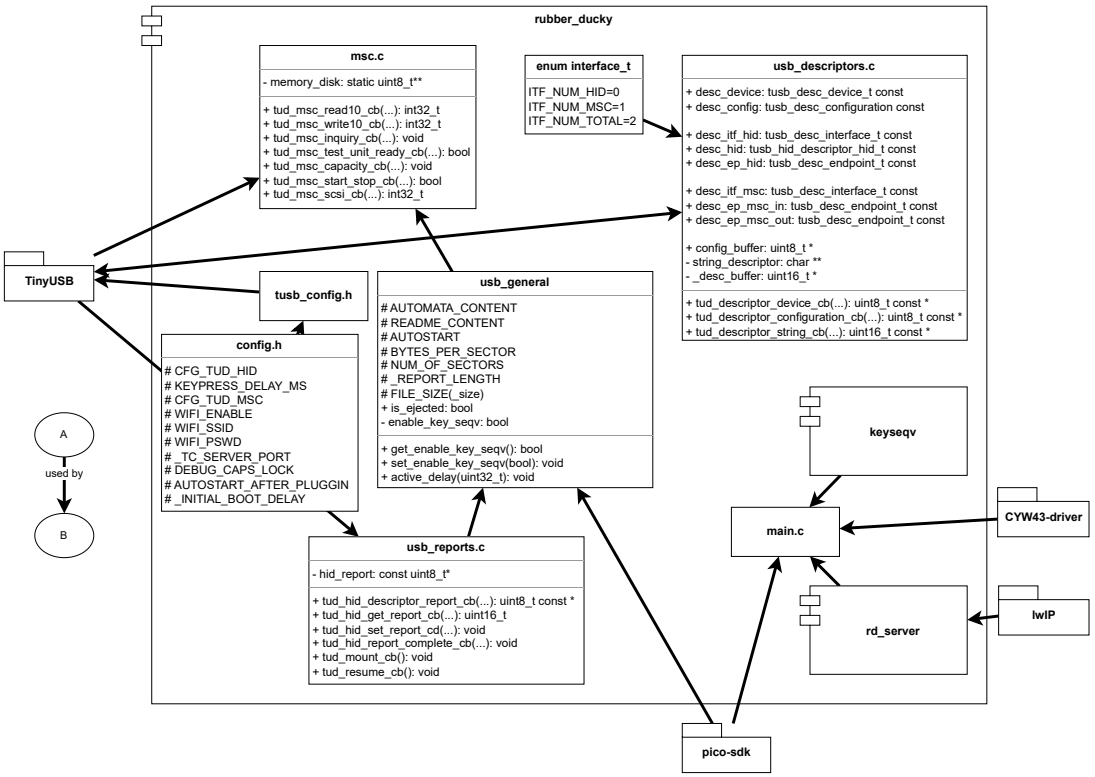

Figure 5.1: RubberDucky module UML class diagram.

5.2.1 USB configuration

In the first part, I needed to configure the USB device for the enumeration process. TinyUSB library requires `tusb_config.h` header file, which defines configuration macros, such as `CFG_TUD_ENABLED`, to set the device as a „device“ and not as a „host“ or `CFG_TUD_HID`, defining how many HID configurations the device will have, and more. Part of the configuration is moved to `config.h` header file. This file contains macros that the user can freely edit. `config.h` is imported to `tusb_config.h` afterwards.

Next, I had to configure the Raspberry Pi Pico board to behave like a keyboard and mass storage. As mentioned previously in Chapter 2, the host identifies the USB device's identity by retrieving its descriptor and configurations. TinyUSB defines 3 callback functions for that:

1. `tud_descriptor_device_cb` – A callback function which the host uses to retrieve *device descriptor*.

2. `tud_descriptor_configuration_cb` – A callback function which the host uses to retrieve every *configuration descriptors* present on the device. It also includes *interface descriptors*, *device class descriptors* (if exists), and *endpoint descriptors*.

3. `tud_descriptor_string_cb` – A callback function which the host uses to retrieve *string descriptors* based on the index.

`usb_descriptors.c` defines all three callback functions. I created instances of descriptors with the structures provided by the library and wrote each callback function's logic to return a pointer to those instances (see an example in Listing 5.1).

```
// device descriptor
tusb_desc_device_t const desc_device = {
    .bLength = sizeof(tusb_desc_device_t),
    .bDescriptorType = TUSB_DESC_DEVICE, // DEVICE constant
    .bcdUSB = 0x0110, // USB1.1
    .bDeviceClass = 0x00,
    .bDeviceSubClass = 0x00,
    .bDeviceProtocol = 0x00,
    .bMaxPacketSize0 = CFG_TUD_ENDPOINT0_SIZE,

    // list of vendors found here: http://www.linux-usb.org/usb.ids
    .idVendor = 0xD0D0, // vendor's id (must be unique)
    .idProduct = 0xCAFE, // product id (must be unique with vendor)
    .bcdDevice = 0x0100, // version

    .iManufacturer = 0x01, // string index of manufacture name
    .iProduct = 0x02, // string index of product name
    .iSerialNumber = 0x00,

    .bNumConfigurations = 0x01 // number of configuration
};

// ...

uint8_t const * tud_descriptor_device_cb() {
    return (uint8_t const *) &desc_device;
}
```

Listing 5.1: Definition of the *device descriptor* and its callback function used in `usb_descriptors.c`.

5.2.2 The keyboard

Since a keyboard is classified as a *human interface device* (HID), I had to define its *report* structure. That is defined in `usb_reports.c` file. A keyboard report consists of three parts:

1. a modifier bitmap, which is sent to the host,
2. an LED bitmap, which is received from the host,
3. and up to 6 keycodes, which are sent to the host.

I created a large byte array named `hid_report`, loaded it with the report byte codes. Then I defined the required HID callback functions. To ease myself during the testing of the

device, I added a debugging feature. If the user turns the Caps Lock LED on and off, the device will start executing the payload again. The algorithm is simple and can be seen in Listing 5.2. In order for this to work, the user needs to set a `DEBUG_CAPS_LOCK` macro to 1 in the `config.h` header file.

```
void tud_hid_set_report_cb(uint8_t instance, uint8_t report_id,
                           hid_report_type_t report_type,
                           uint8_t const* buffer, uint16_t bufsize) {
    // ignore this request
    (void) instance;
    (void) report_id;
    (void) report_type;
    (void) buffer;
    (void) bufsize;
#if DEBUG_CAPS_LOCK
    if (buffer == NULL || bufsize <= 0)
        return;
    if (report_type != HID_REPORT_TYPE_OUTPUT)
        return;

    // reset key sequence after the caps lock LED
    // state changes from on to off
    if (caps_lock_led_on && !(buffer[0] & KEYBOARD_LED_CAPSLOCK)) {
        key_seqv_reset_index_counter(false);
        if (!get_enable_key_seqv()) {
            set_enable_key_seqv(true);
        }
    }
    caps_lock_led_on = buffer[0] & KEYBOARD_LED_CAPSLOCK;

#endif
}
```

Listing 5.2: A snippet of code which implements the payload re-execution.

5.2.3 The mass storage

Mass storage configuration is located, surprisingly, in `msc.c` file. TinyUSB library requires six callback functions to be defined for the mass storage device to run:

- `tud_msc_read10_cb` – Returns a content of a memory sector given an address of the sector.
- `tud_msc_write10_cb` – Updates a content of a memory sector given an address of the sector.
- `tud_msc_inquiry_cb` – Returns Vendor ID, Product ID, and Product revision number of the device.
- `tud_msc_test_unit_ready_cb` – Returns true allowing the host to read and write on the device.

- `tud_msc_capacity_cb` – Determines a disk size.
- `tud_msc_scsi_cb` – Defines the logic of other SCSI commands.

Unfortunately, due to Raspberry Pi Pico's lack of onboard nonvolatile memory, I had to write a static file system from scratch. I chose a **FAT12** file system since it is one of the simplest file systems out there, and, for the proof of concept, a sufficient one. The content of the file system is stored in the `memory_disk` array, which emulates a physical disk. It is represented as a two-dimensional array where the first dimension emulates an array of *sectors*, and the second being the *sector*[5] itself. The first sector contains a **boot table**, the second is a **FAT table**, the third is a **root directory**, and the rest of the disk is filled with file contents. Each part of the file system is described in [12].

5.2.4 The payload

The `keyseqv` directory manages both the payloads and its API. Each key sequence is represented by a `key_seqv_t` structure which can be seen in Figure 5.2. It contains three members: a **delay** that will be applied after sending the report to the host, the **report** with key presses, and a **last item** flag that tells the device not to send any report after this one. The whole payload is stored in `key_seqvs` array and defined in `key_seqv_script.c` source file. This file can be generated by `rd_client` script.

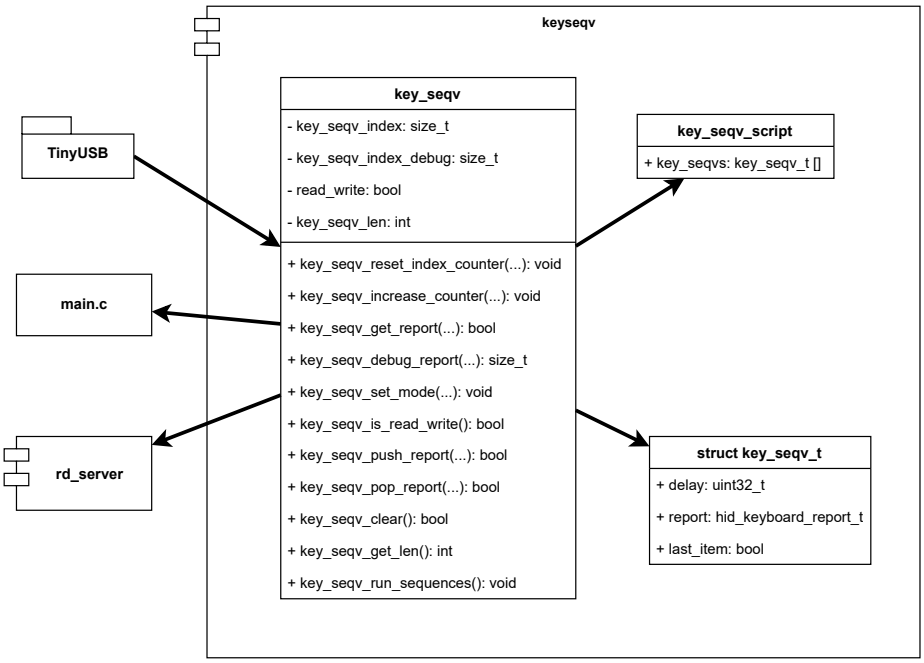

Figure 5.2: Keqseqv module UML class diagram.

Lastly, the `key_seqv.c` source file defines all the functions that operate directly with the `key_seqv_t` structure. That includes functions that update the payload content through the server's API (more in the next section) and those that are used for executing the payload. The two most important functions here are named `key_seqv_increase_counter`,

[5] A sector is an elementary storage unit. It is the smallest number of bytes that the host can retrieve from the disk.

which moves the cursor index to the following item (key sequence) in the array, and `key_seqv_get_report`, which retrieves the current report from the array.

Putting these last two functions together, we can see the pseudocode of the core algorithm of executing the payload in Algorithm 1 shown below:

Algorithm 1: Payload execution algorithm

```
Input: cursor, key_seqvs
device initialization;
while true do
    process USB device tasks;
    retrieve an item from key_seqvs at cursor's position;
    if execution not enabled or item is empty then
        delay = 0;
        report = empty report;
        send a report to host;
    else
        delay = item.delay;
        report = item.report;
        send a report to host;
        if report sent successfully then
            increase the cursor position in key_seqvs;
            wait delay before executing next key sequence;
        end
    end
end
```

5.3 Wi-Fi Access Point and TCP Server

To run a server, we need two things: a port and an IP address of the server. But first, the Raspberry Pi Pico device has to share the same network with the client machine. There are two ways to achieve this: the device can connect to the local Wi-Fi during the booting process or create an access point and let the client connect to its Wi-Fi network. I chose to implement the latter one. The advantage of having the device being an access point is that the server IP address is known – it is the IP address of the network's gateway. The disadvantage is that I need to implement a DHCP server[6] to have a functional network.

For the device to act as an access point, the user needs to set `WIFI_ENABLE` macro, network's SSID (or simply a name), and password, all in `config.h` header file. Once done, Raspberry's software will call `cyw43_arch_enable_ap_mode` during the initialization phase.

After the access point is running, the TCP servers for DHCP and our application are initialized. They are all defined in the `rd_server` directory seen in Figure 5.3. The source codes for the DHCP server are taken over from Raspberry Pi Pico's example page[7]. We

[6]DHCP (Dynamic Host Configuration Protocol) is a network management protocol used to dynamically assign an IP address to any device, or node, on a network, so it can communicate using IP (source: `https://www.techtarget.com/searchnetworking/definition/DHCP`)

[7]DHCP server source codes: `https://github.com/raspberrypi/pico-examples/tree/master/pico_w/wifi/access_point`

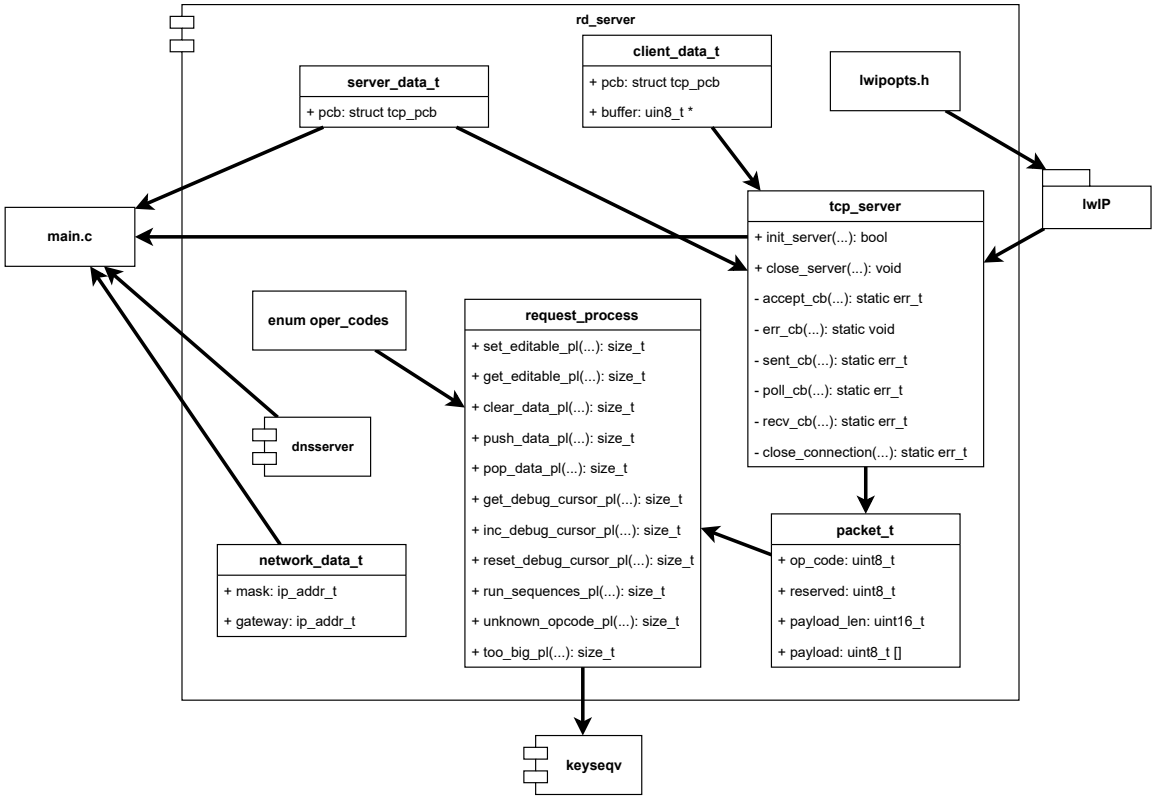

Figure 5.3: RD Server module UML class diagram.

initialize the DHCP server by providing the network's gateway and mask (see Listing 5.3). There is no DNS server in this network since we only want to create communication between the USB device and a client's application, and that can be running in the isolated private network.

```
int main() {
    // ...

    // define network's IP range
    struct network_data_t nd;
    IP4_ADDR(ip_2_ip4(&(nd.gateway)), 192, 168, 4, 1);
    IP4_ADDR(ip_2_ip4(&(nd.mask)), 255, 255, 255, 252);

    dhcp_server_t dhcp_server;
    dhcp_server_init(&dhcp_server, &(nd.gateway), &(nd.mask));

    // ...
}
```

Listing 5.3: A basic network configuration used in the project. In this case, the IP address of the network is **192.168.4.0/30**.

There are two source files that deal with our server application: `tcp_server` and `request_process`. The former defines a TCP socket server using the lwIP library, and the latter defines functions that process requests. The server waits for the connection.

Once it has been established and the server received a request, it extracts its `opcode` and calls the corresponding function from `request_process`. The generated response is then sent back to the client.

5.4 Language parser

This section describes how the parser of the language, defined in Section 4.2, works. Both lexical and syntax analysis of the input is done using regular expression in `KeySeqvParser` class[8]. I use groups in the regex pattern (seen in Figure 5.4) to extract data from the input string. First, the program cycles through the input lines from the file or `STDIN`. Each line's

```
(<DELAY (\d+)>)|(<((?:(\d+)-)?((?:[a-zA-Z]{1,2}-)*)(\\)?([^<>\s]+))>)|(#.*)|([ -~])
```

Figure 5.4: Language's regex pattern.

content is then passed to the `parse_line` function, where it is normalized and added to the `lof_keyseqvs` list for later processing. The parsing algorithm can be seen in Algorithm 2.

Algorithm 2: Processing the input

```
key sequence = empty key sequence;
foreach line in input do
    check line string and extract all groups;
    foreach group in groups do
        if key sequence is full then
            add key sequence to lof_keyseqvs;
            initialize empty key sequence;
        end
        fill data from group to key sequence;
    end
    // store the last pending key sequence if exists
    if key sequence is not empty then
        add key sequence to lof_keyseqvs;
    end
    set last key sequence item in lof_keyseqvs to last;
end
```

The class `KeySeqv` contains information about a single key sequence. It is the same structure as a `key_seqv_t` structure we can find in the Rubber Ducky system (see Subsection 5.2.4). The class also contains a `to_bytes` method which converts the object's content to a series of bytes that complies with the key sequence byte format described in Figure 4.10.

Parser module also implements exception classes for each error case in `error.py`. Figure 5.5 depicts their hierarchy. All exceptions derive from the base class `ParserError`. Six lexical and syntax errors with their corresponding class names are listed below:

[8]This regex pattern was inspired by this project: `https://github.com/lydell/vim-like-key-notation`

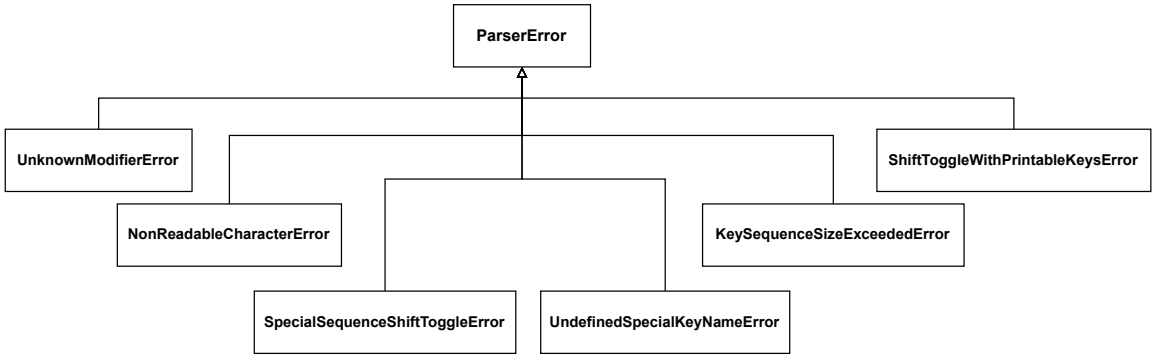

Figure 5.5: Parser exception hierarchy.

NonReadableCharacterError The script contains a character that is not in ASCII encoding or is not printable.

SpecialSequenceShiftToggleError Normal keys in *special combination* uses `Shift` key inconsistently.

UnknownModifierError The *special combination* contains an unknown modifier.

UndefinedSpecialKeyNameError Special key or macro key used in *special combination* is not recognized by the program.

KeySequenceSizeExceededError The number of simultaneously pressed keys exceeded the maximum limit.

ShiftToggleWithPrintableKeysError The use of printable keys and `Shift` key in *special combination* is forbidden.

The whole parser module can be seen in Figure 5.6.

5.5 Client application

There are two modes to run the client application:

CLI mode is a CLI application for creating static payloads. It converts the keystrokes written in Rubber Ducky language to a C source code. The user can use it to replace the existing `rubber_ducky/keyseqv/key_seqv_script.c` file.

Network mode is a CLI application for deploying payloads on the USB device wirelessly using sockets.

Both frontend applications classes (`CliMode` and `NetworkMode`) derive from that base class `BaseMode` as seen in Figure 5.6. Both process the input script in the similar way. They create an instance of `KeySeqvParser` class and feed it with the input data described in the previous section. What differentiate them apart is how they handle the processed data. `CliMode` generates a new C source code file on the `STDOUT` or output file if given. `NetworkMode`, on the other hand, creates a TCP connection with the server and performs a series of requests where it sets the USB device to *read-write* mode, sends `PUSH_DATA` requests for each key

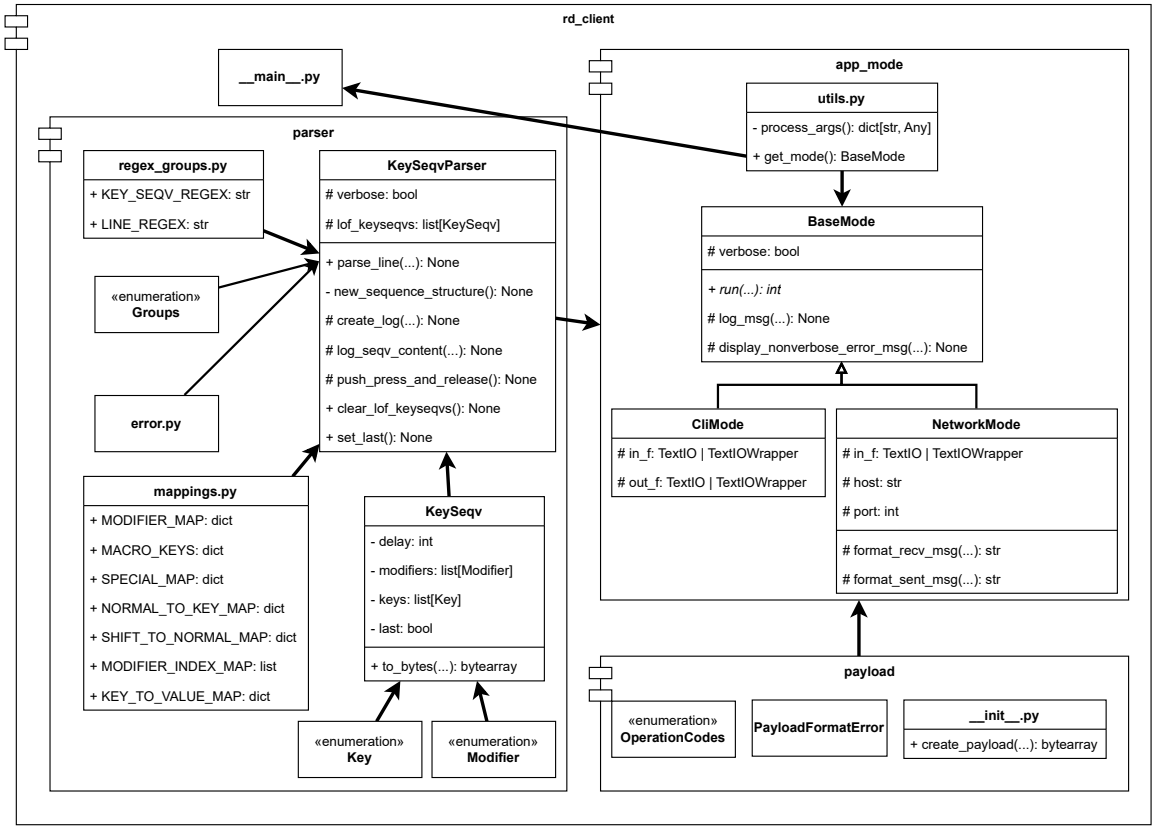

Figure 5.6: RD Client module UML class diagram.

sequence stored in `KeySeqvParser`, then sets the device back to *read-only* mode, and finish it off by sending a `RUN_SEQUENCES` to start executing the payload.

The user can choose the mode by giving the application the corresponding parameter. `-n/--network` flag will toggle the *network mode*. Otherwise, a *CLI mode* is run by default. All of this is handled in `get_mode` function in `utils.py` (shown in Listing 5.4). Lastly, the user can also toggle an option to generate logs by adding `-v/--verbose` flag. That is handled using Python's standard `logging` library.

```
# module rd_client.app_modes.utils

def get_mode() -> BaseMode:
    """Factory function that returns AppMode based on given arguments."""

    args = __process_args()

    # return selected mode
    if args['network']:
        # communication with RubberDucky using network (wifi)
        return NetworkMode(args['input'], args['port'],
                           args['host'], args['verbose'])
    # cli script parsing to C-file source code
```

```
return CliMode(args['input'], args['output'], args['verbose'])
```

Listing 5.4: This snippet of code shows how the frontend mode is chosen.

5.6 Summary

The diagram in Figure 5.7 below summarizes how the Rubber Ducky and client application work. The left one shows the steps the Rubber Ducky device needs to execute before processing the payload. The diagram on the right displays the communication data flow between the client and the server when the user runs `NetworkMode`.

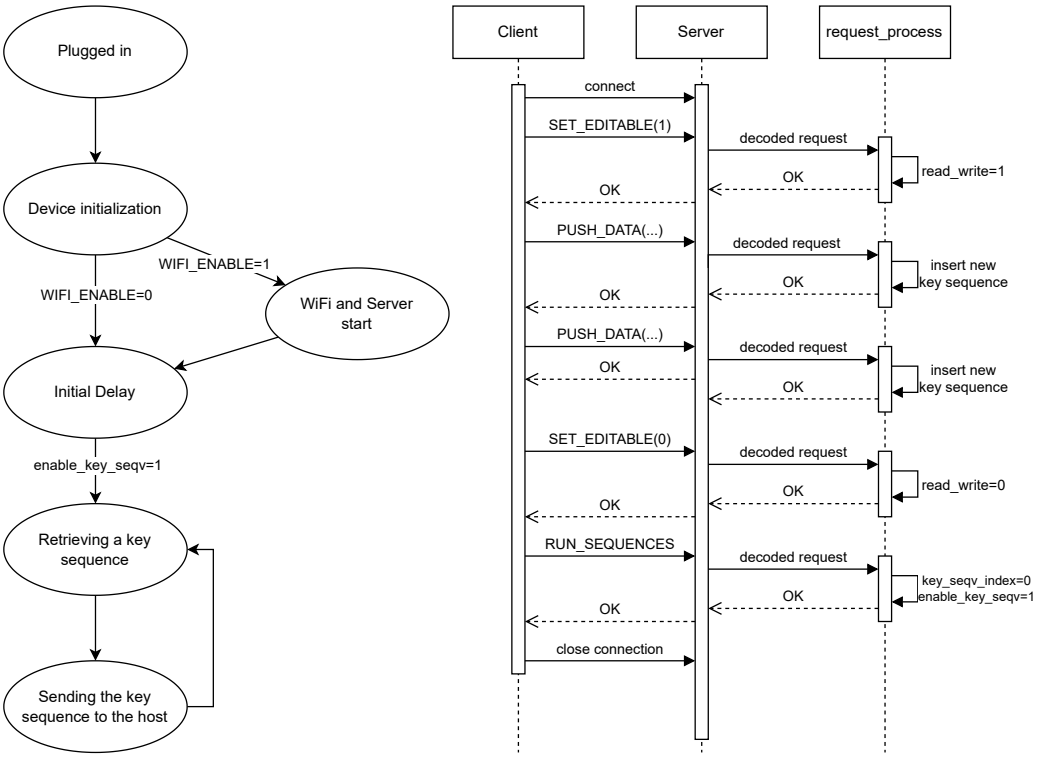

Figure 5.7: Rubber Ducky's state machine and client-server communication data flow.

Chapter 6

Implementation evaluation

In this section I will briefly evaluate my implementation of the Rubber Ducky device. When we connect our device to the machine, the operating system immediately recognizes it as a composite device, and the file system is correctly mounted (shows both „Automata.txt“ and hidden „.README.md“ files). We can check it by opening Device Manager on Windows (or running `lsusb` command on Linux systems). The Vendor ID and Product ID of the device can be seen in Figure 6.1. Both values correspond to the values specified in `usb_descriptors.c` source file.

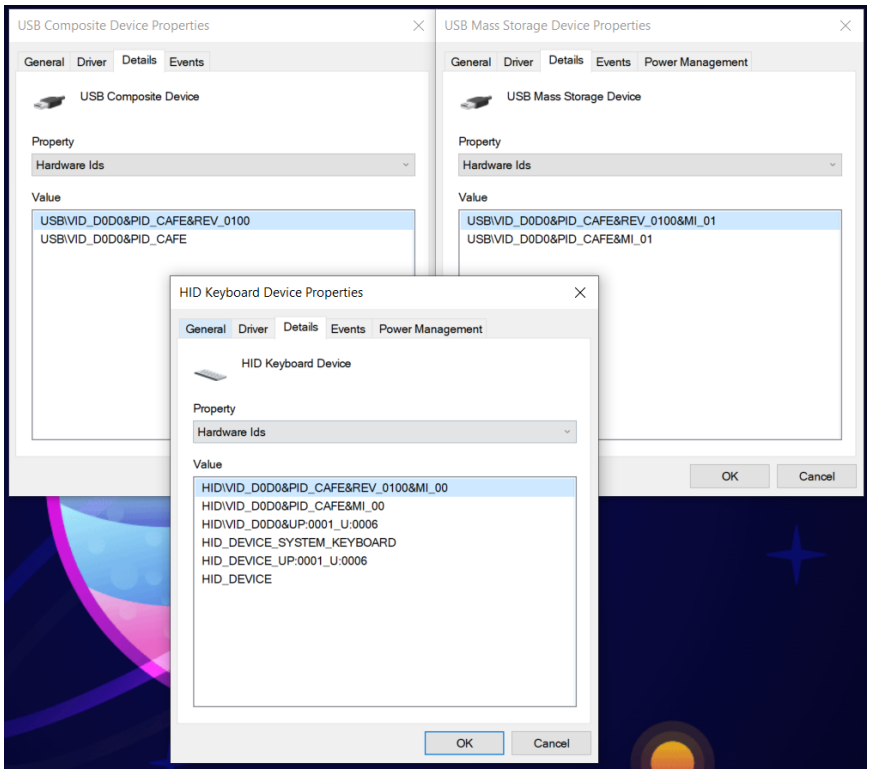

Figure 6.1: Composite device info in Device Manager. `0xD0D0` value for Vendor ID and `0xCAFE` value for Product ID.

The algorithm used to generate the payload is designed to produce as few key sequences as possible (if possible, it will send six keys in one report). Based on the `KEYPRESS_DELAY_MS`

constant, we can test how fast and accurate the device can type. That was tested using the TUI application `tt`[1]. We prepared two test payloads: one with regular three paragraphs long Lorem Ipsum text (2884 characters) and the other with a randomized list of printable ASCII characters (2800 characters)[2]. The mass storage driver was disabled for this test case. Table 6.1 below shows an average typing speed of 20 tests. We can see that the device's accuracy decreases as it types faster. Our Rubber Ducky device struggled a lot on the Fedora system when executing the randomized text at the highest speed, with an average accuracy of less than 94 %. It made most of the mistakes when the Shift key modifier was not properly turned on or off. The table also shows that randomized text slowed typing speed by approximately 2.3 times. We later discovered that running the device on a differently configured system can produce different results. When we ran the randomized text typing test with the `KEYPRESS_DELAY_MS` constant set to 0 on `Manjaro Linux x86_64; kernel 5.15.108` with **i3wm**[3] window manager, the average typing speed and accuracy were both higher (2837.85 WPM and 100 % accuracy).

`KEYPRESS_DELAY_MS`	**Lorem Ipsum**		**Randomized text**	
	WPM	**Accuracy**	**WPM**	**Accuracy**
0	6590.8	99.719 %	2625.05	93.607 %
10	2641.1	99.957 %	1128.5	99.6285 %
20	1321.5	99.981 %	566.1	99.942 %
50	528.95	99.9985 %	226	100 %
80	329.95	100 %	141	100 %

Table 6.1: Typing speed based on the `KEYPRESS_DELAY_MS` constant value. Testing was conducted on `Fedora Linux 36 (Thirty Six); kernel 6.2.9` operating system with AMD Ryzen 7 4700U process and 16 GB of RAM.

Next, we evaluate how accurate the delay feature of our device is. We use the `time` program that is available on most Linux operating systems. Our device executes a `time sleep 10` command in the shell, waits 2 seconds, and then terminates `sleep` command with `Ctrl+C` key shortcut. The time result is then appended to the output file. We repeat the test 20 times and average the results. What we ended up with is a rough estimate of our wait time. According to the results, the average time between starting and terminating the `sleep` program was 2.02s. We can declare that the wait time of our device is accurate to 2 hundredths of a second.

Lastly, we wanted to see how simple it is, to use the device in a real-world scenario. We chose to complete the first training map in the racing game **TrackMania 2020**[4]. The game is completely deterministic, which means that given the same sequence of key presses, it will produce the same result every time. The map only consists of a downhill and three turns (left-right-left) before the finish line. For this test, we disabled autostart and enabled CapsLock debugging features of the device so that we could fully control when we started running our payload. First, we tried a trial and error method, where every time we wrote or updated our payload script, we had to compile a new firmware and flash it into our device. That proved very time-consuming as compiling the code inside the Windows Subsystem

[1]Link to the web page: `https://github.com/lemnos/tt`

[2]Both payloads can be found here: `https://github.com/hungdojan/elastic-quacker/tree/main/tests/typing`

[3]Link to official homepage: `https://i3wm.org/`

[4]Link to official homepage: `https://www.ubisoft.com/en-gb/game/trackmania/trackmania`

Linux was very slow, and unplugging and plugging the device repeatedly was inconvenient. So we changed our approach and enabled the Wi-Fi module in the configuration file. That improved the user experience significantly because, after that, it only took a single command to update the payload. We also learned that the hold delay works correctly, as we needed to hold a `w` key in order to accelerate the car in the game. In the end, it took us around 20 minutes to finish the script, and we completed the track without the car hitting the wall[5].

[5] A final run can be seen here: `https://youtu.be/sDXVFk_Yiyc`

Chapter 7

Malicious payloads

The chapter analyzes the types of payloads that can be installed on our USB device, specifically the ones that were designed to cause damage to the victim's machine. Most attacks require access to the command line application (or terminal). Here are a few examples of payload's notion once access to the command line has been granted:

- update victim's system configuration,
- download and execute a malicious script from the internet, or stored inside the connected device to retrieve sensitive information,
- perform a Denial of Service or BSoD (Blue Screen of Death),
- create a communication backdoor by initializing a reverse shell.

The first item on the list is **updating system configuration**. The attacker can write a payload that, for example, changes the network settings, such as changing the IP address of the DNS server to redirect the requests to the attacker's server, or creates a new symlink or alias of a command (for example, `alias sudo="sudo rm -rf /; sudo` which will, without the user's knowledge, delete the root directory when `sudo` command is called), disable an antivirus program, firewall, or built-in hardware such as a touchpad or keyboard.

Another possible attack involves downloading malware from the internet and running it on the victim's computer (or from mass storage). Keylogger software is one example. **Keylogger** is an application that runs in the background and captures anything the user types on the keyboard. The collected keystrokes can be uploaded to the remote server controlled by the attacker. The attacker can then examine the sent data and extract the user's login credentials. This type of attack is called **Data exfiltration** - a form of attack that involves transferring unauthorized data from a computer. A keylogger can be a software program or a physical device (for example, KeyGrabber[1]). Other data exfiltration methods involve redirecting the user to a fake phishing website.

The attacker can also write a payload that will perform a **Denial of Service** attack or force a **Blue Screen of Death** or **Kernel Panic**. The former attack, Denial of Service, is an attack that is associated with network security. It aims to prevent access to a service or resources [6]. One such attack is known as **Ping of Death**, in which the host machine is overwhelmed with numerous ping requests to a remote server. Once the server's request queue is filled, it will begin to drop new incoming requests, making it unresponsive. **Blue**

[1] `https://www.keelog.com/keygrabber-keylogger/`

Screen of Death (on Windows) and **Kernel panic** (on GNU/Linux), on the other hand, indicate that a fatal error has occurred in the host computer and it is unable to recover from it. This action may result in the victim losing all unsaved data.

Lastly, the attacker can create a payload to get access to the victim's computer. One of the techniques is known as **reverse shelling**. `netcat` is a command-line interface application used by administrators to provide connectivity between two systems. Netcat can operate in either server (listening) or client (creating a connection) mode. The attacker starts a listening shell on the victim's machine and uses his/her machine to connect to it remotely. Unfortunately, the connection will not be established if the victim's computer has a firewall enabled. This problem can be bypassed by creating a reverse shell. The attacker runs netcat in listening mode on his/her machine and uses the victim's machine to connect to it (connecting from the inside out). That will surpass the victim's firewall since it usually only blocks incoming connections [17].

In 2020 and 2021, the cybercriminal group FIN7 started shipping packages containing BadUSB devices to US companies. They were loaded with a malicious payload that gave them access to the victim's network. Once they were in, they deployed the ransomware, such as BlackMatter or REvil, within the network [7] [9].

Chapter 8

Testing defense mechanisms

At the time of writing this thesis, there are many available defense programs on the internet. In this section, I describe how some of them work and whether my device was successful in breaking past any of them. I used the open-source program USBGuard on GNU/Linux and Kaspersky Endpoint Security program on Windows 10.

8.1 Selected programs

USBGuard[1] is a software framework for implementing USB device authorization policies. It was developed in 2015 and has been manage by Red Hat Inc. since then. It consists of two main programs: `usbguard-daemon` and `usbguard`. The former is a service that runs in the background and applies USBGuard policies to each USB device. The service behavior can be configured by editing `usbguard-daemon.conf`. The latter is a command line interface that provides the user with a tool to update the USBGuard policies.

Before I started experimenting, I needed to initialize the service. The instructions were simple: generate an initial policy file using `usbguard generate-policy`, then start the service with `systemctl start usbguard.service`. It was critical to create the rules before starting the service. If not, all USB devices would be blocked by the daemon. The program scans all the USB devices and hubs currently connected to the machine and sets their target to *allow*, meaning they are all whitelisted on the host machine. The testing environment was `Fedora Linux 36 (Thirty Six); kernel 6.2.9`.

The program I chose for Windows operating systems is called **Kaspersky Endpoint Security**[2]. It is a security application that provides computer protection against various types of threads, networks, or phishing attacks. It contains a list of protection components such as File Threat Protection, Web Thread Protection, system scan, and more. What we are interested in is their **BadUSB Attack Prevention**. It works as follows: when a new USB device that emulates a keyboard is connected to the computer, the user receives a pop-up window where he/she/they have to type a 4-digit number displayed on the screen from the connected device. The testing environment for this application was `Windows 10 21H2`.

I tested both software in the following way:

1. Enable only HID class on the Rubber Ducky device.

2. Connect the device to the host machine with the installed and running application.

[1] `https://github.com/USBGuard/usbguard`

[2] `https://www.kaspersky.com/small-to-medium-business-security/endpoint-windows`

3. Observe the application's behavior.
4. Give the USB device access to the system and connect the device again.
5. Enable MSC on the Rubber Ducky device.
6. Connect the device again and observe if something has changed.

The payload present on the USB device opens a terminal or Powershell and runs the `ls` command. I also added a feature where the LED on the board turns on when the device starts to execute the payload.

First, I tested the USBGuard software on Fedora. When I connected the Rubber Ducky device to the machine with USBGuard running, no keystrokes injection happened. In fact, the device did not finish the enumeration process because the LED did not turn on. The system registers that a new device has been connected, but the application immediately blocks it, as seen in Figure 8.1. After I changed the policy for this particular device, the status changed to *allow*, and the payload was executed. Unfortunately, the start of it was trimmed, and only the `ls` command was executed (a new terminal did not open). But that can be resolved by increasing the initial delay.

```
[root@thinkpade14 rebulien_fedora]# usbguard watch
[IPC] Connected
[device] PresenceChanged: id=21
 event=Insert
 target=block
 device_rule=block id d0d0:cafe serial "" name "Rubber Ducky" hash "fYOdXdp4d70OFc0255L+bXvYptG
mtyX+aZZD89zRsbY=" parent-hash "4a4NgfdUaJO43rkCzmWRSeHHR/uUh5+SNsXnhosm9qs=" via-port "1-4" wi
th-interface 03:00:00 with-connect-type "hotplug"
[device] PolicyChanged: id=21
 target_old=block
 target_new=block
 device_rule=block id d0d0:cafe serial "" name "Rubber Ducky" hash "fYOdXdp4d70OFc0255L+bXvYptG
mtyX+aZZD89zRsbY=" parent-hash "4a4NgfdUaJO43rkCzmWRSeHHR/uUh5+SNsXnhosm9qs=" via-port "1-4" wi
th-interface 03:00:00 with-connect-type "hotplug"
 rule_id=4294967294
[device] PolicyApplied: id=21
 target_new=block
 device_rule=block id d0d0:cafe serial "" name "Rubber Ducky" hash "fYOdXdp4d70OFc0255L+bXvYptG
mtyX+aZZD89zRsbY=" parent-hash "4a4NgfdUaJO43rkCzmWRSeHHR/uUh5+SNsXnhosm9qs=" via-port "1-4" wi
th-interface 03:00:00 with-connect-type "hotplug"
 rule_id=4294967294
```

Figure 8.1: Screenshot of USBGuard logging after a new USB device is plugged in.

USBGuard also gave me the option to update the device's policy permanently. Once I whitelisted the Rubber Ducky device, it successfully executed the payload. I also tried to plug it into different ports. I discovered that the policy did not apply to the device if it was connected to a different USB HUB than previously. And lastly, I updated the device's firmware to enable MSC. And again, the USBGuard successfully blocked the device and prevented it from executing the payload.

Kaspersky Endpoint Security program was next to be tested. Upon connecting the Rubber Ducky device, I immediately received a pop-up window, as shown in Figure 8.2. The device did manage to execute the keystrokes, but nothing happened on the host machine since it was „stuck“ in the pop-up window. The Kaspersky program let me input the numbers using my mouse and whitelist the USB device. Unfortunately, I was unable to locate

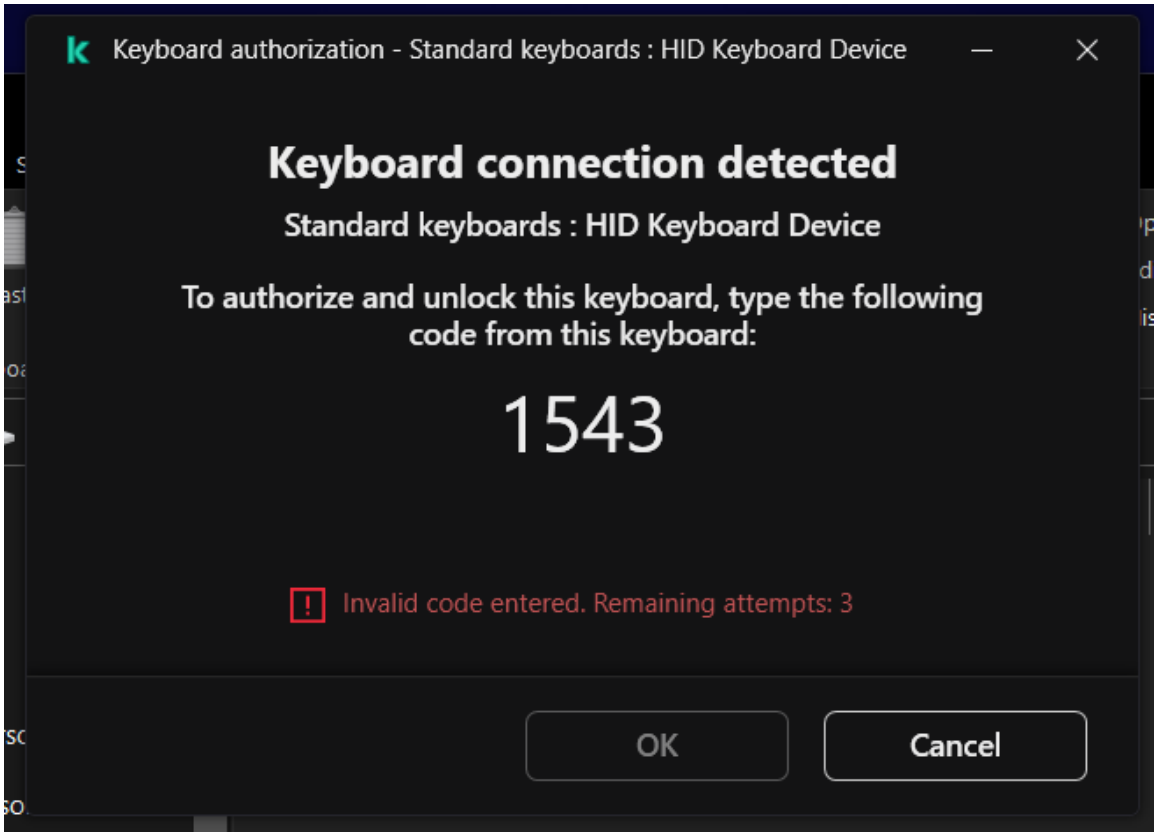

Figure 8.2: Pop-up authentication by Kaspersky Endpoint Security program.

the list of whitelisted devices to remove it from the list. Once the device was whitelisted, it successfully executed the payload just like on the previous test with USBGuard. Interestingly, unlike USBGuard, the Kaspersky application did block all other ports, including those within the same hub. And it also registered the change of firmware when the MSC class was enabled. In Figure 8.3 you can see the reports of the program.

Both programs were very effective against keystrokes injection attacks. The USBGuard was more effective since the device had no real to the system, and all USB-related attacks would have been suppressed (not only the keystrokes injection attack). The disadvantage of USBGuard is that it is not very intuitive to control. All interactions are done using a terminal since there is no official GUI application available. So unless the user is familiar with working with the command line, he/she/they cannot update a device policy[3].

Kaspersky application, on the other hand, gives users a user-friendly GUI application with online documentation. Unfortunately, I was able to break through the defense by sending the authentication PIN through Wi-Fi. So if the attacker has access to the screen, he will also gain access to the victim's machine. Another weakness of the Kaspersky application is that it only covers keystrokes injection attacks. Other types of attacks, such as network card spoofing, will not be detected.

[3]There used to be `usbguard-applet-qt`, but this software is no longer supported. The latest project that provides user-friendly notification pop-ups related to device presence updates is `usbguard-notifier` (`https://github.com/Cropi/usbguard-notifier`)

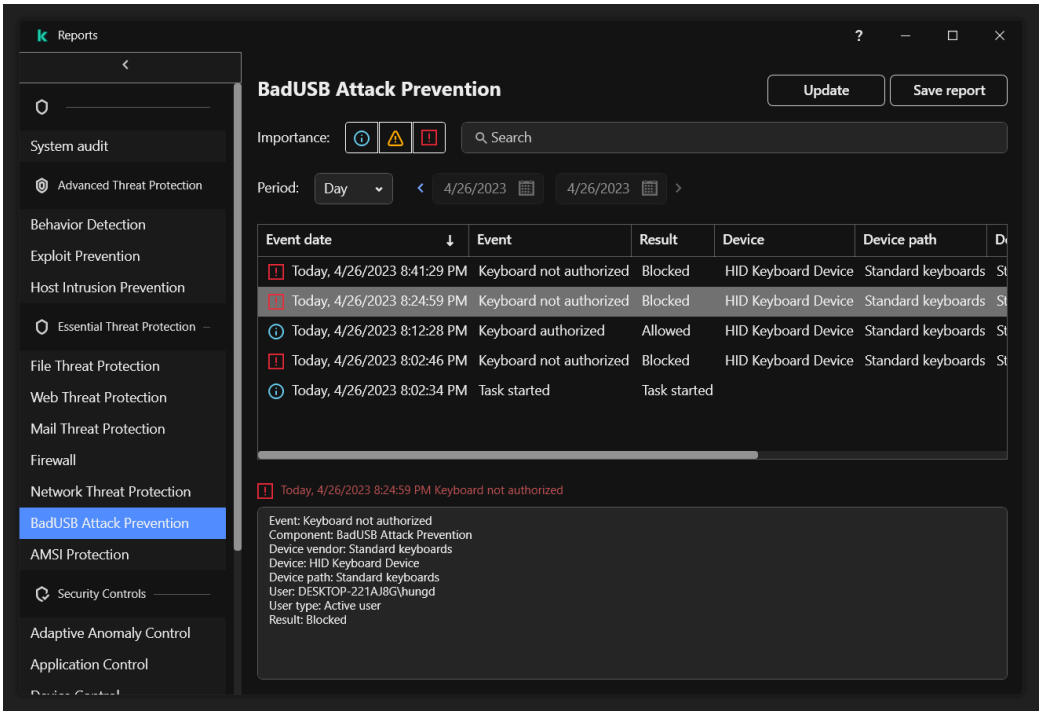

Figure 8.3: The Kaspersky Endpoint Security reports of the test. The reports are sorted from the newest to the oldest. We can see that the first connection (at 8:02 PM) was blocked, then I authorized the device (8:12 PM), tried a different port (8:24 PM), and lastly plugged the device with an updated firmware into the first port (8:41 PM).

8.2 Other defense mechanisms available

As stated in Chapter 7, most attacks require access to the terminal/command line. So another way to protect the host machine is by setting a password for running a command line as an administrator on Windows operating systems. All that is needed is to change the Windows registry[4]. The result can be seen in Figure 8.4. As we can see, before we updated the registry, it was fairly easy to open a terminal (or any other application as an administrator, as the only thing preventing us from doing so was a „Yes/No“ confirmation button. But the new notification pop-up window requires us to enter the username and password (or any other type of authentication).

Another defense mechanism was presented in work [16] by Tian with his team called **GoodUSB** in 2015. They modified a Linux kernel module that maps USB devices to specific whitelisted drivers (for example, an audio device such as headphones registered as a Human Interface Device should only be able to execute a limited number of keys like *Volume Up* and *Volume Down*). Then they introduced a GoodUSB service (called gud), which sits between the host controller and USB drivers. Upon connecting the USB device to the computer, the user is asked to identify the device's functionality. If the device is marked as malicious, GoodUSB service will redirect it to a virtualized USB honeypot where it can be monitored and analyzed. Otherwise, a driver will be loaded. The GoodUSB architecture can be seen in Figure 8.5.

[4]The instructions can be found here: `https://www.manageengine.com/device-control/badusb.html`

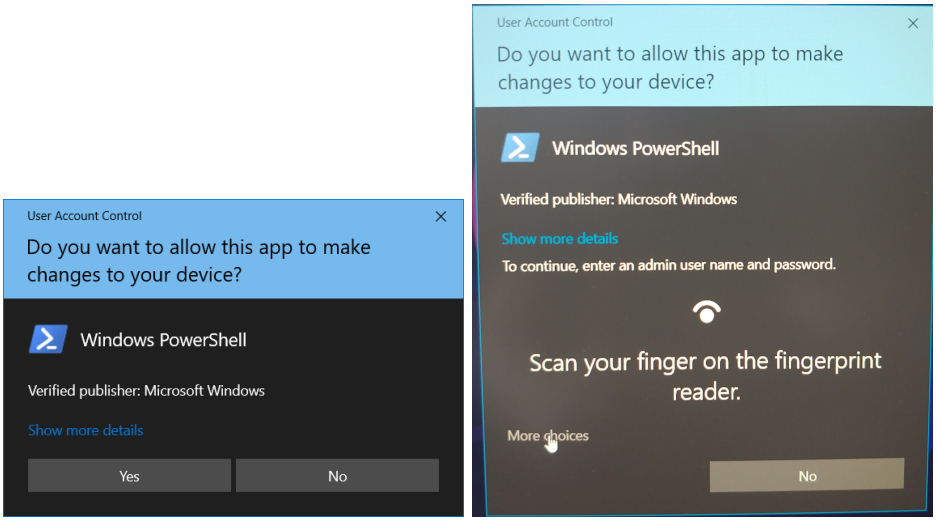

Figure 8.4: The „run as administrator“ pop-up window before (on the left) and after (on the right) the registry value update.

And last defense software I will present here is **Cinch**, a work by Sebastian Angel and his team from The University of Texas at Austin and New York University [2]. Their approach to USB attacks is to create a mediator between a hardware layer host controller and software layer HCI[5]. The USB data are then transferred through a narrow choke point to be analyzed. The Cinch architecture can be seen in Figure 8.5. Cinch uses I/O virtualization hardware to redirect *direct memory access* and interrupts a red virtual machine. The virtual machine then encapsulates and sends USB transfers through the Tunnel to another virtual machine (called Gateway), where it applies all security policies. Once finished, the USB transfer is then sent to the host's HCI (blue machine).

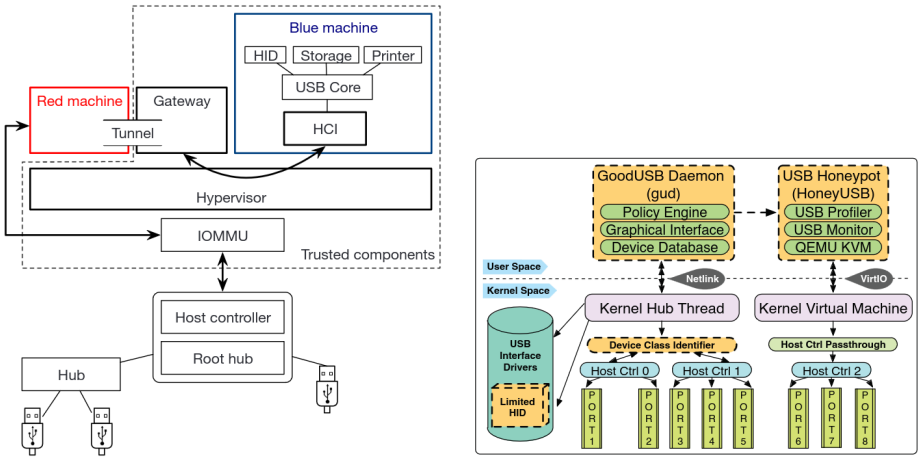

Figure 8.5: Cinch (left)[2] and GoodUSB (right)[16] architecture designs side by side. Both of these pictures were taken from their corresponding papers.

[5]Host Controller Interface, of HCI, is a register-level interface that enables a host controller for USB or IEEE 1394 hardware to communicate with a host controller driver in software. Link to the article: `https://en.wikipedia.org/wiki/Host_controller_interface_(USB,_Firewire)`

Other defense concepts are:

- block the USB device when it starts typing with inhuman speed [11], [3],
- disable firmware updates,
- disable USB drivers on the host machine,
- hardware USB data blocker such as „USB condom“[1] or USG[5],
- and many more...

There are two concepts that occurred to me while I was writing this thesis, and one of them is related to Section 4.2. The notion is that the majority of keystrokes injection payloads rely on the standard US QWERTY keyboard. So the operating system can change the keyboard layout, for example, to CZ DVORAK, when a keyboard connects. That makes the payload fail most of the time because each key is mapped to a different output than on QWERTY. The second idea is to create a mediator that would ignore modifier keys until the user authenticates the device. Without them, the attacker cannot reliably launch the desired applications.

Of course, none of the concepts have yet to be implemented. Changing the keyboard layout may be insufficient and can be bypassed if the attacker guesses the keyboard layout. Furthermore, keyboard layouts only change printable keys, so the attack will remain unchanged if the payload simply consists of modifier and non-printable keys (for example, `Enter`, `Tab`, and arrow keys). The second approach seems safer, but the idea of authenticating a device upon connecting already exists.

Chapter 9

Conclusion

The main objective of this thesis was to create a USB device capable of executing the predefined sequence of keystrokes (payload) which I successfully accomplished. The payload can be statically generated and compiled with the Raspberry Pi Pico's firmware or dynamically deployed after the USB device is connected to the computer. I created a custom language with easy-to-understand syntax for developing payloads. And thanks to Raspberry Pi Pico's inclusion of a Wi-Fi chip, the user can easily control the device wirelessly. The device can also act as a USB thumb drive if enabled.

The BadUSB threads might be undetectable by antivirus programs, but fortunately, the community has already developed numerous effective defense strategies that can protect us against these types of attacks. I tested the implemented device against two programs: USBGuard and Kaspersky Endpoint Security. Both performed well and successfully suppressed the attack. I also discussed a few other possible defense strategies that are currently available.

Even though the core functionality is working pretty well, there are more features that can be added to this project. From supporting simple payloads written in the official DuckyScript to extending the server's API or getting the mass storage fully to work with external hardware (microSD). There are no limitations to what can and cannot be added. I wanted to make this project available to everyone, and so I decided to publish the source code publicly[1]. I hope that this project's development will not end with this thesis.

I created this project for people to experiment and better understand the potential damage that this attack can cause. Though Rubber Ducky is considered a dangerous device, in the end, it is up to us how we will use the device. It can help us automate complex processes and operations that cannot be done by running a script on the computer.

[1] `https://github.com/hungdojan/elastic-quacker`

Bibliography

[1] Al Sibai, N. *You can apparently use a „USB condom“ to protect against the FBI's latest boogeyman* [online]. Futurism, Apr 2023 [cit. 30/04/2023]. Available at: https://futurism.com/the-byte/usb-condom-juice-jacking.

[2] Angel, S., Wahby, R. S., Howald, M., Leners, J. B., Spilo, M. et al. Defending against Malicious Peripherals with Cinch. In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, August 2016, p. 397–414. ISBN 978-1-931971-32-4. Available at: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/angel.

[3] Arghire, B. *Google releases tool to block USB keystroke injection attacks* [online]. Mar 2020 [cit. 30/04/2023]. Available at: https://www.securityweek.com/google-releases-tool-block-usb-keystroke-injection-attacks/.

[4] Axelson, J. *USB Complete: Everything You Need to Develop USB Peripherals*. 3rd ed. Lakeview Research LLC, 2005. ISBN 1931448027.

[5] Doctorow, C. *USG: An open source anti-badusb hardware firewall for your USB port*. Mar 2017 [cit. 01/05/2023]. Available at: https://boingboing.net/2017/03/02/countermeasures.html.

[6] Ericson, J. *Hacking: The Art of Exploitation* [online]. 2nd ed. William Pollock, 2008 [cit. 30/04/2023]. ISBN 1593271441. Available at: https://repo.zenk-security.com/Magazine%20E-book/Hacking-%20The%20Art%20of%20Exploitation%20(2nd%20ed.%202008)%20-%20Erickson.pdf.

[7] Gatlan, S. *FBI: Hackers use badusb to target defense firms with ransomware* [online]. BleepingComputer, Jan 2022 [cit. 30/04/2023]. Available at: https://www.bleepingcomputer.com/news/security/fbi-hackers-use-badusb-to-target-defense-firms-with-ransomware/.

[8] Greenberg, A. *Why the Security of USB Is Fundamentally Broken* [online]. 2014 [cit. 01/01/2023]. Available at: https://www.wired.com/2014/07/usb-security/.

[9] Ilascu, I. *FBI: Hackers sending malicious USB drives & teddy bears via USPS*. BleepingComputer, May 2020 [cit. 01/05/2023]. Available at: https://www.bleepingcomputer.com/news/security/fbi-hackers-sending-malicious-usb-drives-and-teddy-bears-via-usps/.

[10] Kernighan, B. W. *The C programming language*. 2nd ed.th ed. Englewood Cliffs, N.J.: Prentice Hall, 1988. ISBN 0-13-110362-8.

[11] Kerschbaum, F. and Paraboschi, S. USBlock: Blocking USB-Based Keypress Injection Attacks. In: *Data and Applications Security and Privacy XXXII*. Switzerland: Springer International Publishing AG, 2018, vol. 10980, p. 278–295. Lecture Notes in Computer Science. ISBN 9783319957289.

[12] Kholodov, I. *The FAT File System* [online]. Bristol Community Collage, 2010. Available at: http://www.c-jump.com/CIS24/Slides/FAT/lecture.html.

[13] Lu, H., Wu, Y., Li, S., Lin, Y., Zhang, C. et al. BADUSB-C: Revisiting BadUSB with Type-C. In: *2021 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2021, p. 327–338. DOI: 10.1109/SPW53761.2021.00053. Available at: https://doi.org/10.1109/spw53761.2021.00053.

[14] Mohammadmoradi, H. and Gnawali, O. Making Whitelisting-Based Defense Work Against BadUSB. In: *Proceedings of the 2nd International Conference on Smart Digital Environment*. New York, NY, USA: Association for Computing Machinery, 2018, p. 127–134. ICSDE'18. DOI: 10.1145/3289100.3289121. ISBN 9781450365079. Available at: https://doi.org/10.1145/3289100.3289121.

[15] Nohl, K., Krissler, S. and Lell, J. *BadUSB - On accessories that turn evil* [online]. Security Research Labs, 2014 [cit. 01/05/2023]. Available at: https://radetskiy.files.wordpress.com/2014/08/srlabs-badusb-blackhat-v1.pdf.

[16] Tian, D. J., Bates, A. and Butler, K. Defending Against Malicious USB Firmware with GoodUSB. In: *Proceedings of the 31st Annual Computer Security Applications Conference*. New York, NY, USA: Association for Computing Machinery, 2015, p. 261–270. ACSAC '15. DOI: 10.1145/2818000.2818040. ISBN 9781450336826. Available at: https://doi.org/10.1145/2818000.2818040.

[17] Wilhelm, T. *Professional penetration testing : creating and operating a formal hacking lab* [online]. Amsterdam : Boston: Elsevier ; Syngress Publishing, 2010 [cit. 30/04/2023]. ISBN 978-1-59749-425-0. Available at: https://doc.lagout.org/network/2010_professionnal_testing_lab.pdf.

Appendix A

Content of SD card

`build_rd/` Project build directory. Contains files generated by `cmake` application.

- `rubber_ducky/rubber_ducky.uf2` Compiled binary file. The present payload opens BUT FIT official homepage using `firefox` browser on Ubuntu operating system.

`docs/` Document directory. Contains the project documentation and files used to generate the thesis document written LaTeX document.

- `thesis/` Directory with LaTeX source files.
- `configuration.md` Help regarding `rubber_ducky/config.h` macros.
- `rd_script.md` Description of custom language.
- `rd_server_api.txt` Description of packet format and listing of request and response operational codes.

`pico-sdk/` **Pico SDK** framework. Also contains **TinyUSB**, **lwIP**, and **CYW43-driver** libraries.

`rd_client/` Directory containing parser and client network application source files.

`rubber_ducky/` Directory with `rubber ducky` module header and source files.

- `keyseqv/` `key_seqv_t` structure header and source files.
- `rd_server/` TCP server header and source files.

`scripts/` Payloads written in custom language.

`tests/` Test files for `rd_client.payload` module and typing tests.

`README.md` Project's README file. Contains installation instructions and program help.