Noroff University College

Bachelor's Degree Project

Orthrus: An open-source platform towards automated forensic data collection

Maxine Brandal Vågnes

#20340

First Supervisor: *Prof. Barry Irwin, B.Sc. B.Sc (Hons.) M.Sc. Ph.D.*

Second Supervisor: *Emlyn Butterfield BSc (Hons), PGCHE, MSc, SFHEA*

Year of Study: *2017-2020*

NOROFF UNIVERSITY COLLEGE

STATEMENT OF ORIGINALITY

This is to certify that, except where specific reference is made, the work described in this project is the result of the investigation carried out by the student, and that neither this project nor any part of it has been presented, or is currently being submitted in candidature for any award other than in part for the Bachelor's degree from Noroff University College.

Signed:

Maxine Brandal Vågnes

Abstract

HID injections have been a much-discussed attack vector, but relatively little research has been done in regards to creating an open-source platform that would enable this attack vector to be used for forensic purposes. Previous research is mainly concentrated on specific attack vectors under the umbrella of HID injection and not to deliver a platform which could enable such techniques to be executed in practice.

This paper presents the results from work done towards an open-source platform which enables users to do HID injections, with a focus on applications in the forensic field, such as automated data collection. The research bases its research process on what has previously been done both in terms of formal research and open-source projects.

The primary aim of this research was accessibility and how this influences one's design choices whilst creating an open platform geared towards digital forensics.

Acknowledgements

Firstly, I want to thank the dedicated open-source community for continuously improving the world by making their code public so that people may learn from their work. You are truly an inspiration to many.

I will be forever grateful for the detailed and constant help I have gotten from my supervisors; prof. Barry Irwin and Emlyn Butterfield. Without their help and encouragement, the project would not have been.

I want to thank Johannes Tomren Røsvik for helping me with my code, coming with suggestions and corrections for my thesis and for also making the logo of the project, as seen in appendix B.

Finally, I would like to thank my family and my fiancée for putting up about my rants about CircuitPython and LaTeX.

Contents

List of Figures

List of Tables

Listings

Just because something is going to break in the end, doesn't mean that it can't have an effect that lasts into the future.

Tom Scott

1

Introduction

As more digital crime takes place, there is a potential opportunity for devices that make digital forensic investigations quicker and more accessible to conduct; especially for those who are not experienced in this field. This project describes the development of a prototype hardware platform and accompanying software framework called *Orthrus*. Its aim is to simplify the process by working towards an open-source platform that can facilitate this for digital forensics investigators, especially first responders. Orthrus can also prove to be valuable for penetration testing, offensive security as well as other branches of the information security world.

The Orthrus prototype uses keystroke injection via USB HID injection to allow the user to inject scripts and collect data in ways that make it a predictable and relatively easy affair as well as cross-platform. Seeing that as close to 75% of corporations does not control removable devices in a sufficient manner (Fabian, 2007), this approach is quite certain to

work in many cases, even without prior modification to the computer system. Also, it is not unjustified to deduce that one will have an even higher success rate with consumer systems as the techniques discussed in this paper may not be something that most users expect to harden their system against.

One also has to remember that being a first responder and having knowledge in the information security field is not a given. Any help that first responders can get is a step towards a more efficient process and an easier job for people in this profession.

1.1 Aims

The main aim of Orthrus is to be accessible. That is, however, not a singular issue. Rather, for Orthrus to be considered accessible to the demographic that this project is dedicated to, a number of points have to be met.

1.1.1 Accessible

One of the main aims of Orthrus is to build a platform that is easy to start using for those who are not highly skilled in programming and scripting. In other words, Orthrus has to be accessible. While many people in the information security field might know basic programming and scripting, making one's own tools or having to write convoluted code in low-level languages can be tedious and waste much valuable time. Therefore, the scripting language OSL was created to facilitate Orthrus' accessibility requirement, which will be discussed in section 3.4. Following the theme of being accessible to most people, if a user would want to make an improvement to the code, either for oneself or the community, a programming language that was friendly to beginners would have to be chosen. Hence, a platform that is relatively easy to use, a CircuitPython device using the CircuitPython language, was chosen to build this project, which will be discussed in greater detail in section 3.2.1.

1.1.2 Maintainable

The use of CircuitPython also has a lot to say for the maintainability of the program, which is another goal of Orthrus. Many projects with aims similar to Orthrus, as will be discussed in section 2.3, have chosen to write their projects in a low-level language in a very imperative approach. CircuitPython, on the other hand, is, in comparison, more declarative and enables users to accomplish their tasks with fewer lines of code.

1.1.3 Low investment; high yield

For Orthrus to be a viable platform to adopt into one's toolkit, the investment into the system, both in terms of finances and time, cannot be too high compared to the yield one expects to get out of the project. A device cannot be too complicated to set up, for example, having a very complicated firmware flashing process. Therefore only relatively simple devices were considered for the design, something that is discussed in detail in section 3.1 and 3.2.

One also has to consider that the yield of utilising Orthrus in one's toolkit must be realistic and defined, which the further chapters will attempt to convey.

1.1.4 Open-source

Having the code open source means that it is available to everyone to improve upon, scrutinise as well as for educational purposes.

Researchers also argue that there is a case to be made for digital forensics tools to be open-source as they may more comprehensively and more clearly meet the guideline requirements than proprietary tools (Carrier, 2002). The author notes that for open source tools to find acceptance in a legal setting, more comprehensive tests, which designs have to be open to the public, have to be researched.

> *"Digital forensic tools are used to fire employees, convict criminals, and demonstrate innocence. All are serious issues, and the digital forensic application market should not be approached in the same way that other software*

markets are. The goal of a digital forensic tool should not be market domination by keeping procedural techniques secret."

Brian Carrier (2002)

1.1.5 Modular

Orthrus also aim to be built in such a way that it allows for modularity, as will be discussed in section 3.4.8, with the ultimate goal that this modularity enables the user not to have to extensively look at and understand the underlying code to use the device. This ties into Orthrus' open-source nature as everyone may contribute modules to it.

1.2 Scope and limitations

This research aimed to examine the feasibility of a platform such as Orthrus existing in a digital forensic environment and working towards that with a prototype. Hence, getting the baseline for what such a system could be was the focal point of this research. A crucial part of the scope, which is itself defined the scope, was to investigate solutions that were adjacent to Orthrus' aims and contrast the final result of Orthrus to these previous works.

Topics that were out of scope in this research was the production of scripts for Orthrus, as it would dilute the essence of this project, of which is the prototype towards being a viable platform. Some scripts have, however, been produced to demonstrate the capabilities of the platform. Extensive testing into how a user's use of Orthrus could impact the system (for example through forensic artefacts) was deemed out of scope, as will be discussed in section 4.3.1, because of the highly variable nature unique scripts will have on a system. Only the baseline configuration of Orthrus was evaluated.

1.3 Methodology

The methodology of this research is highly baked into the formation of the prototype, its application in practice as well as testing. Therefore, a lot of the methodology will be

discussed in chapters 3 and 4. This is especially true for the iterative design process, as software frameworks and environments have been investigated and changed several times during the initial design process, as will be explained in chapter 3.

Throughout this project, open-source software and hardware have been used whenever possible so that there is a high likelihood that all the results can be replicated by other researchers.

1.4 Document conventions

This document, written in LaTeX, uses extensively clickable hyper-references to facilitate navigation. It adheres to the Harvard referencing standard. Code repositories will be referenced instead of a project's "home page" where suitable. Throughout the document, code snippets in the form of listings will be presented. Instructions on how to obtain and run the code that this project produced may be found in appendix A. Code fragments, such as individual commands will be pointed out using a `monospaced font`.

1.5 Document structure

This document is divided into chapters which represent the major sections in the iterative research process. In chapter 2, previous projects and research will be discussed and evaluated, in particular how they came to shape this project's design choices, the result of which will be shown in chapter 3. The design and results will subsequently be evaluated and contrasted to previous works in chapter 4 in addition to which improvements can be done and a critical evaluation of the results concerning its intended purposes. Finally, a closing summary will be provided in chapter 5.

The heart has its reasons
of which reason knows nothing.

Blaise Pascal

2

Literature review

This chapter will explore the underlying technologies and techniques that Orthrus relies on to function. In section 2.1 concepts around HID and HID injection will be introduced. In section 2.2, practices and conventions in digital forensics will be presented and how they relate to this project. Subsequently, past projects and research and the established techniques which were presented are discussed in sections 2.3 and 2.4.

As stated in chapter 1, most of the corporate devices was shown to not block USB devices from connecting by default. One may rely upon this statistic to one's advantage and therefore assume that the following techniques will work with most devices. As one may see in this chapter, the techniques this allows for are quite numerous and diverse.

2.1 Human interface device (HID)

A Human interface device (HID) is the term used for devices that humans may interact with to produce some output or action. A keyboard is an example of an HID device, but so is a web camera. In the family of HID devices, there are USB HID devices, which is a specification by the USB Implementers' Forum USB Implementers' Forum (2004). It is up to device manufacturers and developers to support these standards as they are officially specified.

Because of the ubiquity and versatility of USB HID devices, there is a general trust that these devices are what they say they are. This unconditional trust has been put to the test by devices like "BadUSB" in the past (Anthony, 2014). Because of this very reason, at the time where BadUSB was shown to the public, one recommended way of circumventing the inherent security vulnerability was to instead use the older PS/2 standard to connect one's mice and keyboards. As one might imagine, such advice does not really apply today as many modern devices, especially laptops, do not have a PS/2 connector. Hence, one may, therefore, deduce that any vulnerability that USB have are going to affect a great number of devices, and this is where the idea of HID injections come into play.

2.1.1 HID injection

HID (Human Interface Device) injection is the act of having a device act as an HID device, for example, a keyboard or a mouse, and then injecting HID commands, such as keystrokes and clicks, onto the target device. Hence, the term *keystroke injection* is a common term that often is interchangeably used with HID injection, even though keystroke injection is technically a part of the greater category that is HID injection. Another term that is associated with HID injection is the *rapid-keystroke injection attack* (Tey, 2013a), which is, as it sounds, a technique which revolves around injecting a great number of keystrokes within a short amount of time. One may consider this aforementioned technique the cornerstone of HID injection as its versatility and speed open up for many other techniques and attacks. Nevertheless, it is important to acknowledge that the techniques underlying the greater category of HID injection are broad and that the potential applications are diverse and numerous. While keystroke injection might be the most common technique, using

HID injection to, for example, prevent the computer from going to sleep without any substantial modification to the system is equally valuable. An example of this could be a mouse wiggler, as shown in section 3.7.3.

A common theme of projects (examples discussed in section 2.3) that have similar methods of injecting commands and extracting data is that many of them are arguably not suited for a forensic environment where data confidentiality is of the highest priority. One recurring feature is wireless connectivity which can potentially compromise the confidentiality of data. Similarly, devices that allow connections with remote web servers can also violate this principle, as well as introducing other variables into the equation as the data has to take a greater distance, both physically and virtually, to its intended destination. In other words, the ideal device for a forensic scenario is one with tightly controlled factors—behaviours that the base platform exhibits should be defined, predictable and well-documented.

There is a great number of open-source projects using HID injection, as will be discussed in 2.3, but the amount of formal research published about HID injection is rather meagre. However, the underlying technologies are present and widely used (as stated in section 2.1) for different purposes other than forensics. Hence, it is an excellent candidate for further research.

While one may argue that devices similar to Orthrus exist (for example, Bash Bunny discussed in section 2.3.3), many are neither free nor open-source or have unwanted behaviour such as wireless capabilities. Some proprietary devices are also often priced so high that for many, this can be a barrier to the field of digital forensics. Hence, efforts towards strengthening and diversifying the foundation of open-source digital forensics are not only a matter of academic interest but also of the public good.

2.1.2 Injection categories

It is essential to make the distinction between the various techniques under the umbrella of HID injection. In itself, it is a part of the much broader field of USB based attacks. In an attempt to formalise and categorise the differences, one may first subdivide it into two main categories, namely *external and internal*, depending on where the attack originates from. The former describes a source of attack where the main events causing the attack

are happening on external devices and the latter being where the main events take place on the host machine itself. This definition only refers to the origin of the injection (an attack is most likely to be caused by external factors) and not the behaviour of the attack after the first contact, as that would move into the territory of lateral movement (MITRE Corporation, 2020), not the injection itself.

What is important to note is that an attack might not be restricted to only one category. For example, one may have a device that listens to keystroke from a victims computer (therefore, MITM based) to then use HID injection to either save those keystrokes to a different place, such as a webserver. Working with these different categories require their own set of skills to either execute or to protect against.

Hence, the categories of injection can be summed up to the following:

- External injections
  - HID emulation injection
  - Man-in-the-middle (MITM) based injection
- Internal injections
  - User-caused injection
  - Independent injection

These categories are mostly arbitrary set out of the convenience so that they may be grouped, as such would aid in the identification and description of USB injections. As USB devices can be emulated in software (Liao *et al.*, 2011), the category "internal injection" is surely an interesting attack vector. This category will, however, not be discussed in this paper as it is widely out of scope.

2.2 Forensic process

With the evolution of computers, the evolution of crime adapted to its new digital environment. Moreover, with this, a forensic process removed from the traditional techniques of DNA analysis has emerged, which rather cares about log files and digital forensic artefacts.

Hence, a very particular set of skills and practices are therefore needed.

To adhere to these principles, good practice must be followed, and thorough testing of one's methods must be done before the actual operation takes place.

2.2.1 Good practices

Solid principles to follow are the principles of confidentiality, integrity and availability, also called CIA triad (Howard, 2002). It describes the three principles which should be at the core when thinking about information security as a whole. In relation to Orthrus, adhering to these three principles can be done through the following steps:

Confidentiality

As Orthrus gives a user the means to do many different types of modification to the system through its scripting capabilities, there are several ways of approaching the issue of confidentiality. First and foremost, a common scenario would be for the user to get data from a target system unto a separate storage medium. Just like with any other storage medium containing confidential and potentially obscene material, these must be handled with great care as in any other forensic situation.

However, a user can also potentially use the scripting capabilities to interface with online resources, in which the matter of confidentiality can be highly variable and unpredictable. Hence, interfacing with either online resources or through other networks in which the data could have its confidentiality compromised should be handled with great care.

Integrity

Integrity is important as not to spoil any evidence's validity. The changes that the baseline configuration of Orthrus does to the system is discussed in section 4.3.1. However, a user's scripts may also severely impact the validity of evidence; hence great care must be taken, as well as thorough testing, so that this may not happen. Furthermore, if it so happens that the data is somehow modified, these changes must be properly documented.

Availability

Finally, availability is something that is impacted by Orthrus quite a bit. During Orthrus' operation, the nature of HID injection means that the target device cannot be used in the meantime as doing so would interfere with the HID injection.

Because Orthrus has the potential to delete files through its scripting capabilities, availability may also be impacted in this way. Once again, the responsibility of care is put in the hands of the user, as the baseline Orthrus configuration arguably does nothing in terms of impacting availability on a target device.

Further discussions on these issues may be found in section 4.3.1.

2.3 Similar efforts

Many projects of various sizes have attempted to incorporate HID injection into their payload delivery strategy. This section will discuss some of these devices that employ methods similar to that of Orthrus.

2.3.1 MalDuino

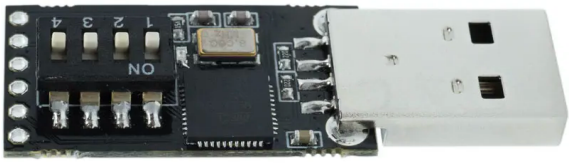

Figure 2.1: Malduino device (Maltronics, 2020)

MalDuino is an open-source platform that enables a user to write HID injection scripts (Seytonic, 2017). Its rather simplistic set of commands allows for an easy to use method for injecting scripts into a computer.

The simplicity of MalDuino does not come with its caveats. Its command set, shown in

table 2.1, is rather limited. It should be noted that the command set is actually an extension of "Ducky Script", described in section 2.3.2. While it has some interesting commands such as returning a random integer or replaying the last command, it has no way of looping certain commands. This makes continuous functions, such as a mouse wiggler (Orthrus example can be found in section 3.7.3), an impossible affair. It does, however, have a very convenient feature which is the collection of 17 keyboard layouts allowing it to work well on computers with different localisations.

Command	**Description**
REM	Comment
DEFAULTDELAY	Time in ms between every command
DEFAULTCHARDELAY	Time in ms between every character
DELAY	Delay in ms
RANDOM	Returns a random integer
RANDOMMIN	Set min random value (default 0)
RANDOMMAX	Set max random value (default 100)
STRING	Types the following string
REPLAY	Repeats the last command n times

Table 2.1: Table showing MalDuino's command set (Seytonic, 2017)

At the time of writing, this project has also not been updated since September 29 2017. There are, however, some forks of the project, but they have not added all too much in terms of functionality.

2.3.2 USB Rubber Ducky

As the MalDuino in section 2.3.1 was based upon the USB Rubber Ducky (hak5darren, 2016) from HAK5, it is no wonder that these two devices are very similar. The shortcomings of the USB Rubber Ducky are also similar to that of the MalDuino. In addition, the process of flashing the firmware can be a bit convoluted, especially in comparison to Orthrus and other devices.

Figure 2.2: USB Rubber Ducky device (Hak5, 2020b)

2.3.3 Bash Bunny

Figure 2.3: Bash Bunny device (Hak5, 2020a)

Going with the theme of HID injection devices, HAK5 also released a much more capable device called the Bash Bunny (Hak5, 2020a). Featuring a quad-core ARM processor and 512 megabytes of RAM, it should come as no surprise that the device is simply a Linux computer in a USB pluggable format.

It has its scripting language named "Bunny Script" which is very similar to Bash scripting

(which is Turing complete). It allows for a versatile and, because of its similarity with Bash; also a familiar way of automating one's HID injections.

However, despite all its features the project is, unfortunately, proprietary closed-source software; an issue which will be discussed in section 4.4.3. This poses a problem whenever the tools need to be vetted for use in legal processes, for example, a court case. While a larger corporation might have the legal power to persuade a court that their software follows the utmost perfect standard when it comes to information security, such cannot be done with a close-sourced project such as this. If the source code was open-source it could be vetted by professionals and therefore have a much higher likelihood of being a device that could be considered for applications such as digital forensics.

2.3.4 P4wnP1

The P4wnP1 (Dawes, 2018) and the newer P4wnP1 A.L.O.A. (Dawes, 2020) projects are frameworks which makes a Raspberry Pi Zero W work as a platform for pentesting and related tasks in the information security field. The frameworks are very flexible with no static workflow, hence enabling the user to utilise the device for a diverse range of scenarios.

Some of its features include USB device emulation, HID injection through its scripting language HIDScript, Bluetooth connectivity, WiFi connectivity, act as different types of network interfaces as well as remote control via either a command-line interface or a web client. Since it is running on what is a full-fledged Linux computer, one may also run several different programs along with P4wnP1 to aid in one's tasks.

To control a device using P4wnP1, the Raspberry Pi Zero W creates a wireless access point which the user can connect to either through a different computer or a smartphone. This functionality, in comparison to many other solutions, makes it so that the user does not have to do any hardware modifications to the Raspberry Pi itself, such as buttons. Also, given its wireless capabilities, the user is not required to have direct physical access apart from the initial connection of the target.

This solution is generally quite good as its versatility, and functionality can support

many scenarios, both for investigators acting as first-responders and for deeper and more thorough analyses in a controlled environment. However, from a forensic standpoint, as a P4wnP1 device is meant to be controlled wirelessly as well as transmit data wirelessly, one may argue that such creates problems with adhering to the CIA principles. Indeed, it is stated on the project's repository that security is not a focus for this project.

> *"The whole project isn't built with security in mind (and it is unlikely that this will ever get a requirement)."*
>
> Rogan Dawes (2020)

Transmitting data wirelessly when proper security measures are not in place, such as encryption, can be a great security risk. Even more so, in a forensic environment where the data might be of a highly confidential nature as well as potentially being illegal. Hence, it is unlikely that such a device can be used for forensically sound applications, despite its numerous features.

2.4 Past research

Because the possibility for keystroke injection has existed for a very long time, there is a myriad of resources available on this topic. However, when it comes to formal research, the number of research papers are rather meagre compared to what one would expect, given how known the technique is. Even more meagre (id est practically nonexistent) are research papers with applications that are not in the realm of offensive security or penetration testing, for example, digital forensics, especially in connection to work done by first-responders.

Work has been done towards solutions that prevent keystroke injection from being possible (Denney *et al.*, 2019). Some researchers described a hardware-based device that analyses the delay between keystrokes, otherwise known as the key-transition time, to create an estimation of its likelihood to be malicious using a classification algorithm. After publishing the aforementioned research, the same researchers published a continuation of their work.

This new research paper describes a detection framework called USB-Watch, which uses, as with their original research, a hardware-based approach and dynamic threat detection (Tey, 2013a).

In the realm of MITM based injection, there are also work being done. The device Malboard impersonates a user's keystroke patterns, gathered from a USB MITIM attack (Farhi *et al.*, 2019). In the case of anti-virus solutions that have the capability of identifying malicious keystroke injection, Malboard might be a serious anti-forensic challenge for the creators of said anti-virus solutions. If the keystrokes look like inconspicuous keystrokes being entered, one can imagine that the next step to combat such techniques would have to be rather advanced. Luckily, the same research paper also describes a proposed detection module which was able to use information such as the power consumption of the keyboard, the sound of keys being pressed on the keyboard and the user's behaviour concerning their ability to react to typographical errors on the screen. This detection module had a 100% success rate of detecting the impersonated keystroke made by Malboard, with neither false positives nor false negatives being made. There is also research done in the field of MITM based injection attacks, and how to mitigate them, with a focus on keystroke pattern, similar to the Malboard, but with a focus on biometrics (Tey, 2013b).

The utilisation of a wired connection and USB protocols are not, however, the only possibility. Wireless HID injection has been identified to be possible due to wireless HID devices not encrypting the wireless stream between the HID device and the target computer (Newlin, 2016). Researchers used the lack of encryption to spoof wireless data packets that are sent to the target computer. In the research, it is noted that wireless keyboards often have their wireless communication encrypted. However, in the case of wireless mice, encryption is much rarer. Keystroke injection using a Secure Digital Input Output (SDIO) connection has also been proposed (Does *et al.*, 2016) as an alternative to using USB.

2.5 Summary

In summary, many of the methods discussed in this chapter are building on the inherent trust and versatility that HID USB devices are privileged to. Keystroke injection is one of the most common techniques in the umbrella category of HID injection, but it is far from the only technique that can be employed. While many of the devices similar to Orthrus have a diverse feature set, none of them offer an easy but still extensible platform for use in digital forensics, either because of their closed-sourced nature, or features that are not compatible with standard forensic practice. Past research, albeit meagre, shows that there are many novel ways of abusing USB HID to do one's bidding. Further research into this field is sorely needed.

One of my most productive days was
throwing away 1000 *lines of code.*

Ken Thompson

3

Design

During the design process, a great deal of time was spent trying to find the perfect device. This was done by attempting to create prototypes that would satisfy the aims and constraints set for Orthrus, as discussed in chapter 1. This iterative design process became essential for when it came to prototyping the final device, as knowing what methods worked with various devices and where these methods may fall short was valuable knowledge. Also, as one of the aims was for Orthrus to be accessible and easy to use, experiences from working with the failed prototypes became a valuable asset.

In section 3.1 the initial design proposal will be discussed along with how these revelations impacted the final design, of which are discussed in section 3.2. After a breif discussion on templating in section 3.3, the design process and functions of Orthrus' scripting language, OSL, is subsequently discussed in section 3.4. Following this, a short description of the two control modes of Orthrus is presented in section 3.5. Platform specifics will

be discussed in section 3.6 with a focus on what one can do if administrator privileges cannot be obtained in a Linux environment. Finally, some examples written in OSL will be presented in section 3.7.

3.1 Initial design proposal

Several devices were considered for the prototype that was to become Orthrus, including ATMEL 32u4 based boards and the Raspberry Pi Zero W. With a focus on simplicity and maintainability, it was soon discovered that the solutions using the aforementioned devices were not the most elegant ones.

Most of the initial designs were oriented around embedded devices that used the ATMEL 32u4 microchip (Technology, 2020), specifically Arduboy (Arduboy, 2020) and Adafruit Feather Adalogger (Adafruit Industries, 2020a), which can be seen in figures 3.2 and 3.3 respectively.

This chip was a central part of the initial design process as not only are most devices using this chip generally quite cheap, but they also have a small size and have a remarkably low power consumption.

On the question of utilising ATMEL 32u4 based boards, which many existing projects that also focus on HID injection are using, questions arose around the lack of simplicity around using such low-level methods to communicate with the hardware. As most existing solutions used Arduino, such as MalDuino as discussed in section 2.3.1, it is a given that these solutions also employ the use of C++ to program their hardware. One of the main aims of Orthrus is to enable users to have a platform that is easy to approach and make amendments to. Hence the issue with C++ is that it can be fairly unforgiving to users that are not accustomed to lower-level languages.

3.1.1 Raspberry Pi Zero W

The Raspberry Pi Zero W has the ability to emulate several different devices, such as a keyboard, mouse, internet adapter as well as a mass storage device (Barnes, 2020). This makes it a very capable and affordable way of doing a wide range of functions that

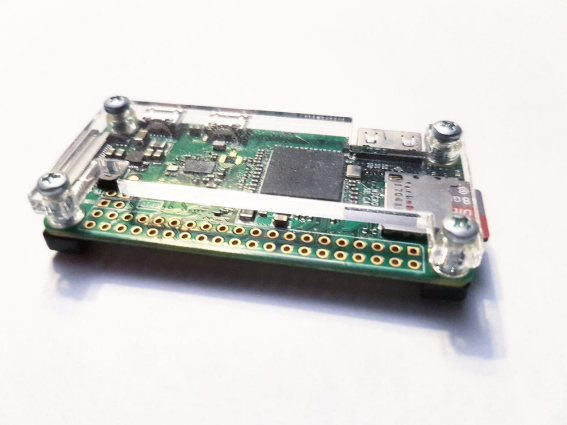

Figure 3.1: Raspberry Pi Zero W

the Orthrus project aims to do. This is not a novel idea either, as the P4wnP1 projects, discussed in section 2.3.4, use the Raspberry Pi Zero and the Raspberry Pi Zero W to run the P4wnP1 software.

While there are several guides on how to modify the software and the firmware to do the necessary functions that Orthrus would require, many of these are outdated. In addition, they can be rather convoluted to a user that is unfamiliar with modifying Linux or Linux in general for that matter. Hence, it breaks the principle of being accessible and simple to use. The setup process must be simple, as briefly discussed in section 1.1.3. This was also reflected in the numerous difficulties of getting the Raspberry Pi Zero W to emulate said devices during the initial design phase. While some sought after functionality was achieved, having all the functionality working at the same time was deemed a convoluted matter. In broad strokes, one may say that such a setup process, even if documented properly, would present many challenges for users who do not have experience of tinkering with the inner workings of Linux devices. Therefore, using the Raspberry Pi Zero W was deemed unsuitable for the Orthrus project.

3.1.2 Arduboy

These devices were evaluated as they could serve two very different purposes. The Arduboy, as can be seen in figure 3.2, has an onboard display. Despite this, it has a very slim and convenient form factor. Its original purpose is to act as a small handheld game console on which hobbyists can develop their own games and programs. One proposal

Figure 3.2: Arduboy device

for the initial design was to create a rudimentary graphical user interface (GUI) using the Arduboy and its libraries to facilitate easy HID injection by selecting it from a menu on the screen.

Some prototypes were made, and they both executed their primary goal of making HID injection easier. However, the as maintainability and modularity was found to be something that could be quite cumbersome with this setup. While it was possible to change scripts on the fly, after a certain point, the GUI became more a hindrance than being an aid to the user.

After some investigation and additional iterations to improve upon the design, it was eventually superseded in favour of the Adafruit Feather Adalogger device.

3.1.3 Adafruit Feather Adalogger

While the microchip is the same as with the Arduboy, the Adafruit Feather Adalogger has a very convenient SD card slot which can be used to store substantial amounts of information.

Many of the problems that were identified with this design were similar to the issues that were discovered during the viability of Arduboy, described in section 3.1.2, as the device of choice. While the inclusion of vast storage space could be very beneficial, that alone did not warrant the use of this device as the device of choice. Hence, instead of using the go-to solution that is Arduino with C++, a more elegant and simplistic solution was

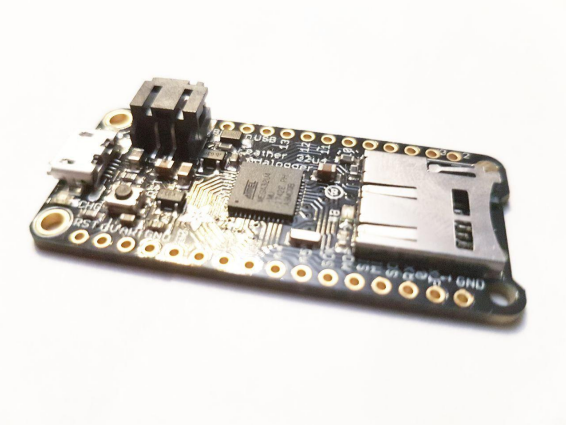

Figure 3.3: Adafruit Feather Adalogger

proposed using CircuitPython.

3.2 Final design proposal

With the knowledge of how and where the proposed devices in section 3.1 fell short concerning the aims of the Orthrus project, the work around the final design proposal could be initiated. While not all the development boards that were proposed for the final design ended up functioning as intended, there are plans at making them work as will be outlined in section 5.3.1.

3.2.1 CircuitPython

In stark comparison to many other projects that are written in lower-level languages such as C++, Orthrus is written in Python. Specifically, it is written in a version of Python called CircuitPython meant for low-powered microcontrollers. CircuitPython is a fork of the MicroPython project by Adafruit to empower people that are very new to the field of programming.

3.2.2 Development boards

Three devices were evaluated to be used in this project, namely Metro M4 Airlift Lite, Feather M0 Express and Trinket M0, all made by Adafruit. They were chosen mainly for the different form factor that they had in comparison with each other.

Adafruit Trinket M0

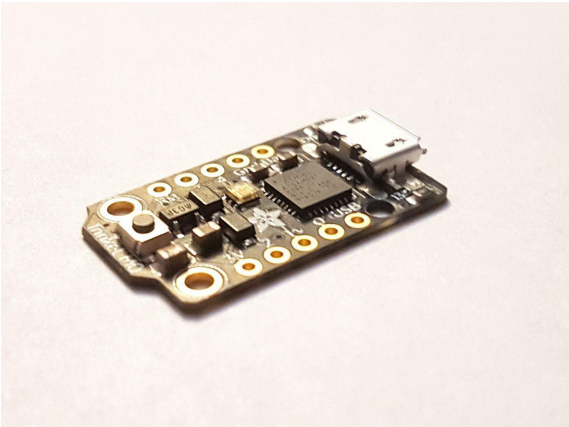

Figure 3.4: Adafruit Trinket M0

Trinket M0 has had the smallest form factor out of these three devices. The small form factor would enable it to be used in offensive security applications, for example, in security audits. Its small form factor could also allow it to be embedded into other devices, such as a decoy USB drive or a USB cable. It's a relatively cheap price, approximately 9 USD according to Adafruit's store at the time of writing (Adafruit Industries, 2020b), means that many can be deployed in an operation.

Unfortunately, given the low storage capabilities of the device, along with some incompatibility of the HID CirtuitPython library, this device was not possible to work with for Orthrus.

Adafruit Feather M0 Express

The Feather M0 Express has a much greater storage capacity than the Trinket M0. The storage capacity of it enables it to store Orthrus as well as a whole set of additional payloads, which can be very useful if no other additional device is desired to be used.

Similar to the Trinket M0 is the problem that some libraries are not supported. In the case of the Feather M0 Express, the JSON library is not supported in this device. In the case of Orthrus, the JSON library is essential to enable the customisability that such a tool should have to provide the user with as much aid in their scenarios as possible. While the JSON library can be swapped out for a customised configuration file, it was deemed out of

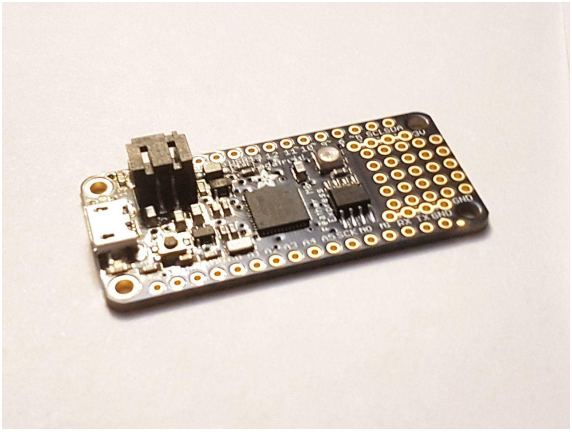

Figure 3.5: Adafruit Feather M0 Express

scope for this project, as this project merely intend to serve as a proof of concept towards automated forensics, not the final solution.

Adafruit Metro M4 Airlift Lite

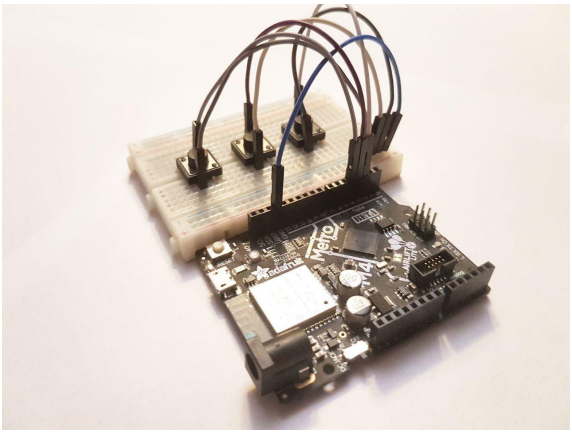

Figure 3.6: Adafruit Metro M4 Airlift Lite connected to buttons on a breadboard

The Adafruit Metro M4 Airlift Lite, seen in figure 3.6, was the CircuitPython device of choice as it allowed for the maximum amount of customisability, given that is powered by an ATSAMD51J19A micro-controller and an ESP32 co-processor in comparison with the other CircuitPython devices that were evaluated to be used in this project.

While the price is much higher than the previously mentioned devices, there are also alternative devices that are very similar to the Adafruit Metro M4 Airlift Lite, either Adafruit themselves or from other manufacturers that also supply devices that run CircuitPython.

These devices have not been tested to work with Orthrus so results may vary.

3.3 Templating

Hard-coding every command into the program may not only prove to be a messy affair but also highly unscaleable. For example, if extra command injections are to be added to the program, in the case that commands are hard-coded into the program, the user would have to do so the same way. This would mean that the user would have to have a deeper understanding of how the program works, and hence a novice programmer may not find Orthrus approachable or useful at first glance.

3.3.1 Configuration files

Instead of hard-coded values and instructions, one may utilise configuration files or template files that tell the program what to execute. In this scenario, the program would only act as a mediator between the command injections specified in the template file and the target computer. This method equates to a much more approachable tool that could be picked up by users that may not have a deep understanding of programming. Only the template structure needs to be understood.

There are many ways one could go about doing this. The simplest solution, especially in terms of implementation, is to take a widely used format such as JSON or YAML. At the time of writing, CircuitPython does not have a library that enables YAML parsing. However, parsing JSON is still possible and therefore, will be used to structure the configuration file.

The configuration file that Orthrus uses has some basic parameters that that complements the functionality to its scripting language. In this configuration file, which can be seen in listing 3.1, parameters can be set that alter how Orthrus is run.

Listing 3.1: config.json

```
1 {
2     "automatic": false,
3     "default_script": "routines/automatic.osl",
4     "clear": false,
```

```
5     "logging": true,
6     "prewait": 0
7 }
```

If a user requires that no input is needed for the script to execute, one may switch the `automatic` parameter to `true`. This causes Orthrus to pick the default script, supplied as a string to the configuration parameter `default_script`, to be executed.

Other useful features that the configuration file brings includes rudimentary logging by giving the option to have lines along with their line numbers printed to the serial console. The configuration file can also enable the clearing of the serial console for an easier viewing experience.

To be sure that Orthrus does not run before it has had time to properly connect, one may delay Orthrus by using the `prewait` parameter. Such a function is essential if automatic execution is toggled. The `prewait` parameter takes a floating-point number as an argument.

3.4 OSL

For Orthrus, implementing a rudimentary scripting language was reasoned to being a good solution. Not only does it improve the user experience of never having to touch the underlying code, but it also enables rapid prototyping of solutions. The scripting language designed for the Orthrus platform is called Orthrus Scripting Language (OSL). OSL aims to present a user with a simple but solid method of writing out HID injections. The highly declarative nature of OSL paves the way for promoting the overarching process of an HID injection rather than the actual code and commands that are taken to achieve it.

Orthrus still requires the user to have a basic understanding of scripting in (depending on one's operative system) either Powershell or Bash as Orthrus does not concern itself with what is being injected as long as the injection is facilitated.

Through using Adafruits various HID libraries, OSL can be parsed instead of manually writing the equivalent in CircuitPython. Consider the example written in OSL in listing 3.6, In this example, one can see the key commands taking place to get a Powershell

window in Windows with administrator privileges.

The full command set of OSL may be found in 3.1

3.4.1 Parser

Calling Orthrus a real scripting language is not entirely accurate. It would be more accurate to consider it a form of shorthand that enables a user to access functionality that one may need when doing HID injections.

The parser works by pattern match with the commands, which are lexical tokens or keywords. When the parser identifies a token that is in its command set, the specific function assigned to the keyword is carried out. These tokens are only evaluated when they are at the beginning of a line; hence one cannot chain multiple commands on the same line. While limiting, this also forces the user to write their scripts in a procedural and structured manner.

When an OSL file is evaluated by the program, it first goes through each line of the program looking for variable definitions. These definitions are stored in a dictionary. After each line has been inspected for definitions, variable substitution can take place. Once again, each line is inspected, but this time, Orthrus is looking for if any variables have been called using the `{{<VAR NAME>}}` keyword. If a matching variable name has been found within the dictionary that hold variable definitions, these variables will be replaced with their values. While this may indeed sound like a tedious way of doing variables, it allows the user to define variables anywhere in the OSL file, and they would still work.

After variable substitution, the code is evaluated and executed line by line. That is, unless either a loop occurs or OSL fragment is called, which will be explained in sections 3.4.6 and 3.4.8 respectively.

3.4.2 OSL commandset

Command	Description
K: <KEYCODE>	Keycode command.
C: <CCC>	Consumer control code command.
S: <STRING>	HID injects string.
SL: <STRING>	HID injects string and presses ENTER.
LOOP: <INT>	Demarcates the beginning of the loop and loops n times.
ELOOP	Demarcates the end of the loop.
VAR: <NAME>, <VALUE>	Sets variable.
<VAR NAME>	Use variable.
OSL: <PATH (as string)>	Run external OSL fragments within an OSL script.
M[move]: X(<INT>), Y(<INT>)	Moves the mouse (X, Y coordinates).
M[click]	Clicks the mouse (right or left mouse button).
EP: <PATH (as string)>	Injects external payload.
SLEEP: <FLOAT>	Sleeps for n seconds until executing next line.
SER: <STRING>	Print string to serial console.
QUIT	Quits the routine/script.
//	Comment command. Line will not be evaluated.

Table 3.1: Table showing OSL's command set

3.4.3 Keycode command

Keycode commands enable the user to inject keystrokes such as `GUI` which is also known as the "Windows button". In the example above, the `GUI` key is paired with the `R` key, forming the key-chord which opens up a dialogue box on Windows which allows the user to run either programs or commands. In this scenario, the user intends to get a Powershell with administrator privileges.

A full list of available keycode commands are available from Adafruit in their source code (Adafruit, 2020) which is based upon the USB HID usage tables document (USB Implementers' Forum, 2004).

3.4.4 Consumer control code command

As a Windows machine might make a notifying sound when UAC is activated, to minimise the possibility of getting noticed, the user uses a consumer control code to mute the computer, namely `C: MUTE`. Consumer control codes ofter represent functions many laptop and consumer keyboards have, such as volume up and down, play and pause and other buttons with quick-access control to various functions. In short, one may regard them as hard-coded macros that are often utilised by a wide range of users. This can be used to OSL's advantage, as complex functions supported by consumer control codes can be boiled down to a single line of OSL.

3.4.5 String command

The string command, `S:`, takes whatever the user supplies to it and enters the string to whichever application that is in focus. This, of course, enables the user to enter simple text strings and combine this with other OSL commands to do whatever one needs to do. However, an arguably more useful way of using the string command is to inject one-liner scripts. As a convenient shorthand for `S:` and `K: ENTER` on the next line, a user can utilise the `SL:` command (short for "send line"). This will send the string and subsequently press enter, which can be very useful in a terminal environment.

3.4.6 Looping command

Introducing looping to a rudimentary scripting language posed some challenges. In a script with looping, not only does the loop have to be parsed in a way that the code runs in its proper order, but the code parsing the OSL commands also have to consider that loops may be present within loops. To solve this problem, many iterations of code was attempted. Earlier attempts only managed to parse simple loops with no recursion. To support loop recursion, the code that took care of the looping behaviour had to be taken out from the parser itself so that whether a block of code was in a loop, and hence should be repeated, was evaluated not along with the rest of the code.

Listing 3.2: Python function responsible for checking loops and execution of lines of code

```
    def code_runner(current_code):
    if not current_code:
        return  # if not empty
    for i, line in enumerate(current_code):
        if config()["logging"]:
            print(f"{i} | {line}")
        if is_loop(line):
            count = sanitise(line, "LOOP:", True)
            try:
                arg = argument(line)
                if arg == "inf":
                    while(True):
                        code_runner(current_code[i+1:])
            except AttributeError:
                pass  # no argument, hence no inf loop
            rest = None
            for _ in range(int(count)):
                rest = code_runner(current_code[i+1:])
            code_runner(rest)
            break
        if is_loop_end(line):
            return current_code[i+1:]
        run_code(line)
```

In the code from the `code_runner` function that can be seen in listing 3.2, the code is ingested into the function and then evaluated whether not it is empty, signalling that either the code or loop has ended. Subsequently, the lines of the ingested code are enumerated, and then each line is checked if its either a standard line of OSL or the beginning of a loop. If the line is declaring a loop, the subsequent lines will be run for as many times as specified by the user until the end of the loop. As the same function that runs the code is called when running code within loops, loops can exist within loops. That is if maximum recursion depth is not exceeded.

Unlike when executing external OSL fragments, as will be explained in section 3.4.8, variables are preserved within recursive loops, regardless of their depth.

3.4.7 Variables

Variables can be set with the `VAR:` command, as can be seen in listing 3.7. Then they can be referenced in the code by using double curly braces (`{{variable}}`). Internally in Orthrus, the definition of variables are set before any code execution takes place. Hence, the placement of the variable declaration does not impact the execution of any OSL routine.

3.4.8 OSL fragments

External OSL scripts can be called within any OSL file by using the command:

```
OSL: <path/to/OSLfile>.
```

This allows the user to call snippets of code that execute a predetermined set of actions, such as bypassing UAC in Windows. These snippets of code are referred to as fragments, and some prewritten examples are located in the `./routines/fragments/` folder in Orthrus' repository. Consider the scenario where a user has several scripts that all have a section which does the same action, but apart from that, they are different in function. By taking the common sections and refactor them to a single OSL fragment, the OSL code can be simplified and also be much simpler to read and debug. In environments where storage space on a device is premium, refactoring to OSL fragments can also be an effective method in maximising one's available storage.

Important to note is that fragments are treated as different OSL files. Hence, variable definitions are not transferred to the OSL fragment that is called, as demonstrated in listing 3.3.

Listing 3.3: OSL example code: Shows nonfunctioning code

```
1 VAR: print_this, this will be printed
2 OSL: routines/fragments/printer.osl
3 <Below is the printer.osl fragment>
4 SER: {{print_this}}
5 <This will print the string "{{print_this}}" as the variable ↩
     is not defined>
```

OSL fragments also allow for increased modularity as fragments made by the community can be added to Orthrus' repository and implemented into one's scripts.

3.4.9 Mouse commands

Mouse commands can be issued through the `M` command. It requires an argument, namely which action is to be performed. The argument that can be supplied are `move` and `click`. Consider the following example:

```
M[move]: X(10), Y(10)
```

In this example, the mouse is moved ten positive units both in the X and Y axes. The mouse command also takes negative values as well as click arguments so that the user may also use the right and left click of the mouse.

The user can also click the mouse. If the user requires many subsequent clicks, for example, click every five seconds (example in 3.4), one may put the click command in a loop with an infinity argument.

Listing 3.4: OSL example code: Infinite loop where left mouse button is clicked every five seconds.

```
1 LOOP[inf]
2     M[click]: left
3     SLEEP: 5
4 ELOOP
```

3.4.10 External payload command

The external payload command, `EP:` is very similar to the string command, explained in section 3.4.5. However, instead of taking the input as a string in OSL, the input is defined by the name of the path (for example `../payloads/ping.ps1`) which then is rapidly injected into the window in focus. This command is very useful in the scenario that one has either a premade exploitation script that is already contained in a file, or that a script is very long.

3.4.11 Sleep command

While simple, the sleep command `SLEEP:` is vital to ensure that scripts are not executed before the target system has had the time to respond to the previous command issued. Because the Orthrus only has one-way communication to the system, the user will have to experiment with how much one's script is supposed to sleep. The necessary delay between commands is highly variable between systems depending on their specification as well as which command injections and GUI manipulation the user is trying to achieve. Hence, the user needs to appreciate this fact and adapt one's OSL scripts accordingly.

3.4.12 Printing to serial

Debugging a novel scripting language with no available linters for code correction can be a chore, especially due to the unidirectional nature of communication that HID injection has. In addition, when one adds other ways of scripting though other languages such as Powershell, some unforeseen bugs may arise. To mitigate this, Orthrus prints the number of each line when the user is connected to its serial port along with complementary logging information.

Additionally, a user may also print directly to serial themselves with the `SER:` command so that the logging information is relatively more complete compared to just the output from the python debugger.

3.5 Control

Two methods of controlling Orthrus was implemented, namely, automatic and simple. These should cover most use-case scenarios, but additional methods of controlling Orthrus have been planned for future expansion of the project, which will be discussed in section 5.3.1.

3.5.1 Automatic

As discussed in section 3.3.1, an automatic injection can be enabled by turning it on in the configuration file, as seen in 3.1. Such functionality can be beneficial if a device simply does not have a button, for example, a small device such as the Adafruit Trinket M0. It can also be important in cases where only one script is needed and should be deployed upon insertion of the device.

3.5.2 Simple

The "simple control" mode is the default mode. It assumes that the user has hooked up a couple of buttons to the CircuitPython device so that three different OSL scripts may be executed.

3.6 Platform specifics

In chapter 1 it was stated that the Orthrus project is aiming to work cross-platform. To honour this aim, this section will explain the differences between the supported platforms (Windows and Linux) and how one has to change one's approach when dealing with these.

3.6.1 Windows

While one may write one's payloads, there are many pre-written payloads such as PowerSploit (PowerShellMafia, 2016) and the open-source payloads written for projects such as BashBunny, as discussed in section 2.3.3.

A problem with pre-written payloads and exploits is that anti-virus suites detect them (if they are indeed known to the suite) and either quarantine the potential threats or removes them entirely. An example of this happening can be seen in figure 3.7. This occurs both in the case of keystroke injection and the script merely being present on the disk. Hence, to make the function of the payload as intended, they have to be obfuscated. PowerSploit comes with its own set of tools to obfuscate the payloads. Other tools explicitly for this purpose, also exist, for example, Invoke-Obfuscation (Bohannon, 2019), which is a Powershell utility to obfuscating one's code.

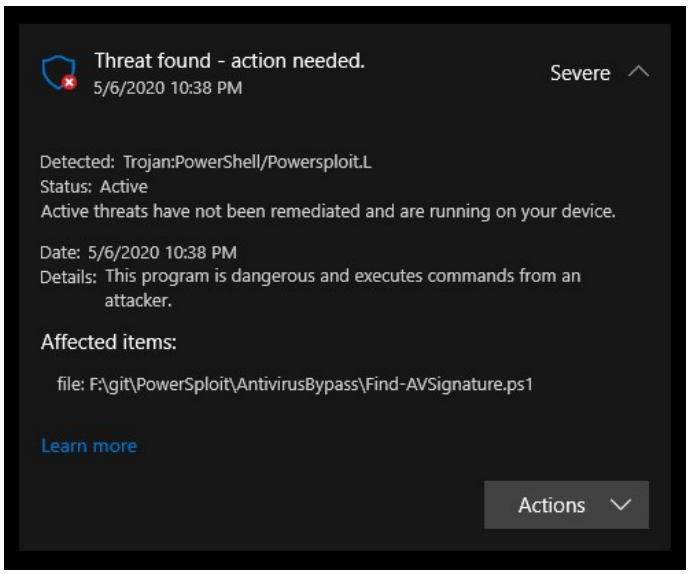

Figure 3.7: Windows Defender showing a that a PowerSploit script has been detected.

Invoke-Obfuscation may use many methods to obfuscate the code so that the automatic detection is done by for example Windows Defender (Microsoft, 2020) is unable to detect the injection attempt or the mere existence of the file for that matter. A simple method that Invoke-Obfuscation uses is the *compression* function. This method compresses the Powershell commands into a compressed and base64 encoded payload (Bohannon, 2019). Through this function alone, most, if not all, anti-virus suites may be defeated as they often rely on pattern matching algorithms (Zhou *et al.*, 2008). Many other ways of obfuscating one's injection attempt also exist such as the anti-virus bypass utility already built into PowerSploit (PowerShellMafia, 2016). Whichever method one ends up using, one should note that not all anti-virus suites work in the same way. Some might incorporate more advanced methods of detection, such as machine learning (Al-Kasassbeh *et al.*, 2020), which could make the task of obfuscating one's scripts less trivial. However, in most cases, simply obfuscating one's scripts using relatively basic techniques should be enough to bypass most anti-virus suites (Haffejee and Irwin, 2014).

3.6.2 Linux

Different to Windows, the diaspora of various distributions of Linux are numerous. The same goes for the various shell environments (such as bash, fish and zhs). Hence, one must first think of which device one will be using Orthrus on. Failing to do so could cause scripts to cause either unintended functionality or simply to not work at all. As there is no way for Orthrus to know what kind of system it is connected to (discussed in section 4.2.3), such information has to be provided to the script ahead of time.

Unlike in the case of Windows, as discussed in section 3.6.1, getting "root" or administrator access to a shell is not as easy. This is mainly because Linux distributions often ask for a password every time a task that requires administrator privileges is attempted to be run, usually through the command `sudo`. Hence, a solution as simple as the one discussed in section 3.6.1 to gain root access to a shell (given that a user already has signed in to the target computer) is not feasible.

Even installing new software, for example through `apt install <package>` requires the input of the current user's password. Hence one may conclude that to get the same level of

functionality with Orthrus with a Linux distribution as with Windows the password must, therefore, be acquired. There exist many proposed methods to gather the password from a target running Linux. One may attempt doing memory forensics to find the password residing in clear text in memory (Davidoff, 2008). One may also try to gather the hashed password and then crack the password with a different device (Hatch *et al.*, 2001) to then use this password in one's scripts, a place where OSL's variables functionality can be convenient. There is also a vast range of ways to use common privilege escalation techniques to bypass local security restrictions, such as gaining administrator privileges. The project GTFOBins GTFOBins (2020) has a list of a wide range of Unix binaries with details of how they can be used to exploited to gain root access to Unix distro. However, this technique requires a lot of background knowledge into the system and may not apply to many scenarios, for example, for forensic investigators acting as first-responders.

The common theme with these solutions is that they restrict the "plug-and-play" aspect and automated nature that Orthrus is trying to achieve. In scenarios where a large amount of additional work with other devices is required to make Orthrus a viable method to extract data from the target system, strong and valid arguments against Orthrus' role can be made.

3.6.3 A world without root

One may also attempt at going at this restriction from a different angle if one set the requirement for one's script not to require any authentication and then build one's script from this presumption. Consider the example in listing 3.5, with the assumption that this computer is a presenting only a CLI environment, a user is already signed in and that the USB-drive is both mounted and its path is known. In this example, the script attempts to copy all files on the target recursively whilst preserving permissions until either the command finishes or the USB-drive runs out of storage.

Listing 3.5: OSL example code: Copying all files from Linux recursively and preserving permissions.

```
1    SL: cp -a /* /mnt/h/extracted/
```

On the removable storage that the command in listing 3.5 writes the information, the result could help a first responder in getting a general overview of the system, even though not all files are accessible. For example, one may use the `tree` Linux command to get a hierarchically structured overview of all the files and folders.

While many Linux distributions, especially those who are attempting to be user friendly, may mount USB drives by default, a great number of distributions simply do not. One might beg the question, and rightfully so, why one cannot simply mount the file system from the terminal, to which the answer is, again, "one requires root". Hence, as Orthrus lives in a world currently without root, such a possibility is unfeasible.

3.7 Example scripts

In order to provide some context on how OSL can be used with Orthrus in practice, this section will provide some example scripts in an attempt to demonstrate Orthrus' functionality. Further areas of applications will be discussed in section 4.3.

3.7.1 Gain root shell in Windows

One of the primary factors that enable full access to a system is to have administrator or "root" privileges in a command prompt or "shell". Depending on what shell one wants, there are a number of ways to acquire access to them. In listing 3.6, a Powershell window is opened with administrator rights. This is done by using the "Run prompt" (which can be accessed by the key combination `GUI + R`) and running PowerShell with a certain set of flags. These flags cause the User Access Control (UAC) window to appear, in which the script selects the "Yes" button and clicks it. Subsequently, PowerShell command prompt appears, as seen in figure 3.8.

Listing 3.6: OSL example code: Method to gain Powershell with administrator rights.

```
1    K: GUI, R
2    SLEEP: 1
3    SL: powershell -Command "Start-Process powershell -Verb ↩
        RunAs"
4    SLEEP: 3
5    // Bypass UAC
6    K: LEFT_ARROW
```

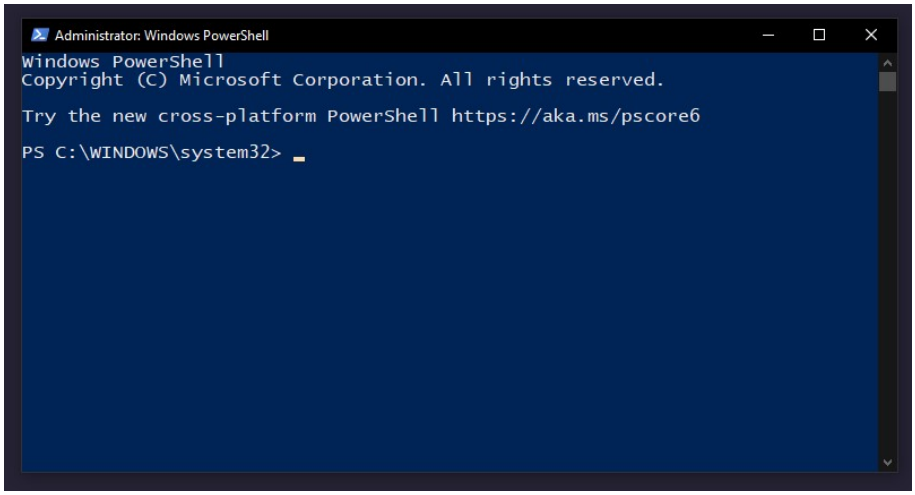

Figure 3.8: A PowerShell command prompt with administrator privileges gained after the script in listing 3.6 had been run.

```
7       K: ENTER
```

In listing 3.7, a CMD window is opened with administrator rights. This is done by running the program `msconfig` with the parameter `-5` from the Run prompt, meaning that the fifth tab in that program will be opened. Subsequently, the OSL script will go fourteen items down to the "Command Prompt" item and this item is selected by using the key combination `LEFT_ALT + L`. Now the user is in possession of a command prompt with administrator privileges.

Listing 3.7: OSL example code: Method to gain CMD with administrator rights.

```
1       VAR: sleep_time, 1
2       K: GUI, R
3       SLEEP: {{sleep_time}}
4       SL: msconfig -5
5       SLEEP: {{sleep_time}}
6       LOOP: 14
7           K: DOWN_ARROW
8       ELOOP
9       K: LEFT_ALT, L
10      SLEEP: {{sleep_time}}
```

These example techniques can be used as an OSL fragment, as discussed in section 3.4.8, which would allow for rapid access.

3.7.2 Fast gather

There already exist several solutions that exfiltrate data, such as PowerSploit as discussed in section 3.6.1.

The small catch with PowerSploit however, is that Powershell recognises the code and subsequently refuses to run it. This can be mitigated in several ways, but a common technique is to obfuscate the code, as discussed in section 3.6.1. If this is not done, even the mere act of plugging a device can cause a script to be detected as malicious. Not only is this behaviour destructive towards a forensic examiner's workflow, but it can also leave behind residual forensic artefacts that can weaken one's case. Hence, to make sure that one's operation runs smoothly, it is crucial to ensure that the scripts are not only going to be undetected by anti-virus software but also that they don't leave additional artefacts in the process. If, however, one writes one own's scripts, this is far less likely to happen.

Many security professionals also write their so-called "one-liners", meaning a short script that could, in theory, fit on one line. These are perfect in combination with Orthrus as they demonstrate the speed and versatility that one might demand in an HID injection platform. These one-liners are also easy to implement, as they often require little to no changes made to them to work with Orthrus.

Consider the example in listing 3.8. The PowerShell one-liner presented in this example is used to gather the currently active TCP connections of active processes. In this example, firstly a PowerShell with administrator privileges is gotten by running an OSL fragment with this functionality; a functionality described in section 3.4.8. Then the target drive is set with a PowerShell one-liner, and finally, the forensic one-liner gathering the netstat output is run, and the output is saved to the remote storage device. With this script, a forensic examiner can quickly gather information about which connections the target computer is currently making and then make decisions based on that knowledge, such as if the computer ought to be disconnected from the internet.

Listing 3.8: OSL example code: Forensic one-liner gathering netstat data.

```
1 // Get PowerShell with administrator privileges
2 OSL: routines/fragments/adm_pwsh.osl
3
```

```
4 // Set target drive
5 SL: $target_drive = (Get-WmiObject Win32_Volume -Filter "↩
     DriveType='2'" | ?{$_.Label -eq 'TARGET'}).DriveLetter
6
7 // Run netstat and save to remote storage medium
8 SL: netstat -anb -p TCP | Out-File -FilePath "${target_drive}↩
     extracted\netstat.txt"
```

Scripts such as these can enable first responders to get a rapid overview of the situation and gather valuable data with minimal input. It can even act as a standard protocol in which several machines can be gathering information from at the same time without needing an operator.

3.7.3 Mouse wiggler

A common tool in many forensic toolboxes is a mouse wiggler. The tool, often in the form factor of a small USB device, is plugged into a device, for example, a laptop, that needs to not go into sleep mode. Consider the following hypothetical scenario where a computer has been seized from a suspect. The password of the computer is not known, and without continuous activity, it is not unlikely that the computer will enable sleep mode, which will upon waking the computer back from sleep mode, require the investigator to know the password or any way around it. By wiggling the mouse, this scenario can be prevented for long enough until a way around the password can be found.

As this function is rather trivial to implement with the functionality that OSL embodies, one of the example routines of Orthrus is a simple mouse wiggler, as seen in listing 3.9. The module also demonstrates how the scripting language makes it easy to run mouse commands.

Listing 3.9: OSL example code: Mouse wiggler

```
1 LOOP[inf]
2     M[move]: X(-10), Y(-10)
3     SLEEP: 5
4     M[move]: X(10), Y(10)
5     SLEEP: 5
6 ELOOP
```

This example uses the `LOOP[inf]` command to initiate infinite looping. This allows for

the mouse wiggler to work indefinitely until stopped by resetting the device using the reset button which all CircuitPython devices are supposed to have. Alternatively, the device may be unplugged.

3.8 Summary

Throughout the design process, many potential solutions that were evaluated but did not come to fruition served as a valuable lesson in designing a platform such as Orthrus. The specific aims called for a design process that was considerate towards the possibility that non-technical users interested in its functionality. This alone warranted the existence of a scripting language such as OSL, which is both simple and rather capable compared to many other devices reviewed in section 2.3. With Orthrus' aim of supporting multiple platforms, the sensibilities of each platform were discussed, and these will have implications for chapter 4, especially in section 4.3.1.

The magnitude of inadequacy you can feel working in computer security is almost unparalleled. You're exposed to the work of seemingly superhuman subversion. This is not your lack of sufficiency. It is the immaturity of our profession, expounded; not reason to existentially fret.

SwiftOnSecurity

4

Discussion

In this chapter, the methodology of testing the final design of Orthrus will be presented. The forensic artefacts that Orthrus may create, even from just being plugged in, not running any scripts and having the baseline configuration will be discussed in 4.3.1. Design choices will be reflected upon as well as how well those design choices stay true to the original aims, as will be discussed in 4.4.

4.1 Testing

During the development of Orthrus, testing was an integral part of the process. For example, will be mentioned in the section on debugging 4.2.5, finding out what went wrong in an OSL script is not always a straight forward task to solve.

Edge case testing to find out the absolute limits of OSL were carried out after the syntax

was formalised. No noticeable difference between Windows and Linux was found to severely impact the baseline Orthrus configuration, although as will be seen in section 4.3.1, precautions in relation to forensic artefacts that must be considered are different between the operative systems.

4.2 Design

In hindsight, a lot of the decisions made during the design phase could have been simplified and planned better. Some of these decisions were attributed to factors such as lack of documentation and a limited amount of research in the area of HID injection. The general diverse methods of carrying out HID injections were both an aid and an obstacle, as finding a narrow set of methods that are certain to succeed posed to be a challenge.

4.2.1 Hardware

The CircuitPython devices that were evaluated for the project was, as discussed in section 3.2.2, found to be highly variable in what functions they supported that were needed for this project. Out of the three that were evaluated, only one device had all the functionality required.

Going towards future expansions of the Orthrus software, the lower tier CircuitPython devices, in general, might not be capable of running Orthrus as there is simply far too little storage and not enough support for libraries such as the HID library and the JSON library. One might expect that similar limitations with further expansions to the Orthrus software can occur; hence it may further break the ideal solution that user would be able to run the software on the most CircuitPython devices, thus breaking compatibility. Going towards the future, this problem must be addressed, and solutions which can expand compatibility have to be investigated.

4.2.2 Platform

The complexity of the existing solutions was known to be rather unscaleable and unsustainable. The initial research was mainly centred around if the method behind these

devices could be simplified and improved. It was shown, as demonstrated in section 3.1, that the methods that previous attempts were centred around were viable (in terms of maintainability) until a certain point where the codebase became merely too complicated for most people to contribute to.

The P4wnP1 project discussed in section 2.3.4 has an extensive range of application because of its many functions and versatility. With this functionality, however, there is a massive codebase behind it which makes it tick. While the codebase is written in a higher-level language, bing JavaScript, with more efficient modules written in Go, it is a rather daunting task for volunteers to take on and contribute code to. Orthrus means to counter-act this by offering the entirety of the code in one simple language, being CircuitPython.

4.2.3 Unidirectionality of communication

An inherent weakness of devices that are solely relying on HID injection to get the work done is the intrinsic nature of the unidirectional communication that such entails.

While serial connections to and from the target device are possible, it is not only highly dependent on the target device of choice, but its reliance upon many uncertain factors makes it a rather unforgiving method of communication. Imagine the scenario where one would want to communicate information through a serial connection between Orthrus and a target computer. First of all, the serial connections have to be enabled on the target device. Then, the device that is attacking must have the capability and authorisation to access the serial device to communicate to it. Finally, the device attacking must also have the ability to store the received information and potentially processing it in some way. Such a scenario would call for a much more advanced and complex pipeline than what Orthrus can offer while still trying to be a simple platform that can be approachable. Many devices, such as the Adafruit Metro M4 Airlift Lite used for this project, has a rather low internal memory. There are of course different devices that offer far greater expandable storage, for example in the form of an insertable memory card, but the added complexity can be argued to defeat the purpose of having the platform is simple and approachable.

There exist, of course, a possibility of using methods of communication that transcends

the limitations that serial communication might facilitate, but that also adds complexity. A really interesting project related to this is the exfiltration of data using LED status indicators (Zhou *et al.*, 2017). The current solution where the communication is still unidirectional, and where one device is in charge of the execution of commands whilst the other is in charge of storage, is a straightforward solution that allows for customisability and maintainability.

4.2.4 Defined behaviour

As stated as one of the original aims, the behaviour has to be defined, as discussed in sections 1.1.3 and 2.1.1. Having a clearly defined and documented behaviour is essential for Orthrus to facilitate the automation of a forensic process. A large part of this is to have the forensic artefacts that the baseline Orthrus configuration creates be documented, which will be discussed in section 4.3.1.

4.2.5 Software

In comparison to using Arduino and C++ as described in section 3.1, CircuitPython was proven to be easy to work within many more ways than just the process of writing the code.

However, it is essential to note that it is not as simple as one might think. There are differences between Python, MicroPython and CircuitPython. Python, and its documentation, is, of course, the baseline of what the language can do, but not all Python functions are possible to use in the two latter languages. While the syntax may be close to identical, one is working with a completely different set of hardware targets where one's code is supposed to run. Hence, efficiency, low memory footprint and low requirements for storage are all factors that are vital for MicroPython and CircuitPython to be a valid alternative to lower-level languages such as C++.

The programmer must also take the situation in mind, being that the target is not a decently powerful desktop computer. This paradigm shift in thinking can be difficult when using a language as permissible and forgiving as Python often is regarded as. One example that was encountered during the testing phase of the different CircuitPython devices concerning

this different way of thinking was the small storage capabilities and memory limitations discussed in 3.2.1. One might think that this is mostly a hardware problem, and one would be correct, but the implications for the software side of things are the interesting part of the story. For example, on a desktop computer, one may be used to use as many external pre-written libraries as possible to minimise the time spent writing code and making one's program. Given that neither memory limitations nor storage limitations are usually frequent problems on a desktop computer, this way of thinking makes sense. On a low-power device such as embedded devices, however, one might find oneself writing more custom code to fit one's purpose rather than adapting one's solution to prewritten libraries. This is, of course, a more time consuming and labour-intensive process, but it is indeed necessary for Python to work in environments such as these. A side effect of not being able to import all kinds of libraries that might contain code that is not strictly necessary for one's program to run is that the code base is usually less bloated and is, therefore, easier to debug.

The vision of universal compatibility between CircuitPython devices that was envisioned in the stages towards the final design was unfortunately not achieved. Further research is needed to enable Orthrus not only to run but be certified to work on multiple devices fully. This issue will be further explained in section 5.3.1.

Debugging

Debugging on an embedded device, even though the syntax looks like Python, can prove more challenging than with one's more standard workflow on just a desktop machine. Also, since Python is an interpreted language, the debugging process is unique compared to development on other embedded platforms such as Arduino. From the research done, it was discovered that debug and error messages were not completely similar to that of what one would expect from Python running on a desktop computer. One might say, rather fittingly, that the debug messages were more minimal and not as clear compared to what a Python programmer might be used to.

There is also the difference that debugging with the device is done through a serial connection. Hooking up one's debugger of choice is, therefore, a difficult affair, which

means that different (and often more simplistic) methods have to be utilised instead. In the case of Orthrus', having a way of printing to serial, as discussed in section 3.4.12, in an easy manner was therefore deemed vital to the creation process. Also, being that Orthrus is attempting to stand as a viable platform where you should not have to touch the source code, this function is also highly useful for users in general. The serial printing command is, of course much more powerful in conjunction with the opportunity for variables in OSL.

4.2.6 OSL

While the original idea and design behind Orthrus did not include its own scripting language, the iterative research process pointed in the direction that it would allow for a much broader utility spectrum to a wider range of diverse fields in technology where HID injection-based automation could be utilised.

4.3 Applications

While Orthrus is originally intended to be used in digital forensics operations, it was also designed to be as universal as possible; hence the possibility to be used in different fields are definitely present. In broad strokes, one may say that in places where automation in software is not possible on the device itself, and one also has the possibility to connect an external device through the USB port, Orthrus may be utilised.

4.3.1 Digital forensics

One of the main aims of the initial design of Orthrus was to enable the automation of tasks frequently done in the field of digital forensics. However, it evolved into something more general that could facilitate automation through HID injections in several information security fields. Despite this, the focus was constantly geared towards its potential applications in the field of digital forensics.

Therefore, a focus point of the project after the initial code was finalised was to reveal what the changes to a system were after interfacing with an Orthrus device. While highly

dependant on the system that is connected to, some forensic artefacts were produced by Orthrus. Also, do note that changes done to the system as a direct effect of scripts being run will not be evaluated, as the forensic artefacts of such actions would be highly variable to each unique script. While one might anticipate certain actions to be logged, for example, unsuccessful attempts at using sudo, it would mostly be guesswork, and as such, this is left to the individual examiner to evaluate.

Windows

For Windows devices, the registry (accessible through for example the Registry Editor) recorded Orthrus connecting through USB. For example, as can be seen in listing 4.1, several places in the registry changed and recorded that an Adafruit Metro M4 Airlift Lite device had connected. Information such as model number, product name and manufacturer may, of course, be found.

Listing 4.1: List of locations in the registry in which the Orthrus can leave forensic artefacts.

```
1 Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\↩
     DeviceClasses
2 Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\↩
     STORAGE
3 Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\USB
4 Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\↩
     USBSTOR
5 Computer\HKEY_LOCAL_MACHINE\SYSTEM\MountedDevices
```

Similar evidence may be found in the Event Viewer, especially upon the first connection with an Orthrus device where evidence of driver installation may be found. An example of this can be seen in listing 4.2.

Listing 4.2: Entry from Event Viewer showing drivers being installed upon first connection of an Orthrus device.

```
1 A driver package which uses user-mode driver framework version↩
     2.31.0 is being installed on device SWD\WPDBUSENUM\_??↩
    _USBSTOR#DISK&VEN_ADAFRUIT&PROD_METRO_M4_AIRLIFT&REV_1.0#A↩
    &1941627&0&7242B80723457335020202 93A093B0FF&0#{53F56307-↩
    B6BF-11D0-94F2-00A0C91EFB8B}.
```

While only baseline Orthrus configurations are evaluated forensically in this research, it is

important to note that other forensic artefacts that can manifest from running pre-written scripts that anti-virus suites deem malicious, as discussed in 3.6.1. Forensic artefacts can also be produced by scripts that bypass User Account Management (UAC) which are viewable in the Event Viewer. Also, if PowerShell is run with flags, such as in listing 3.6, these events are also logged in the Event Viewer; an example of which can be seen in listing 4.3.

Listing 4.3: Entry from Event Viewer showing that PowerShell has been run with a specific set of flags.

```
1 NewEngineState=Available
2 PreviousEngineState=None
3 SequenceNumber=13
4 HostName=ConsoleHost
5 HostVersion=5.1.19041.1
6 HostId=c092800d-a30b-4b7c-a9e5-71639daec419
7 HostApplication=C:\WINDOWS\system32\WindowsPowerShell\v1.0\↩
     PowerShell.exe -Command Start-Process cmd -Verb RunAs
8 EngineVersion=5.1.19041.1
9 RunspaceId=88c5625a-676e-4645-8ecb-83c95fe03eaf
```

Linux

The evidence of a device having connected is also evident on Linux. One of the most common places to look for such information is the log files, most commonly found in `/var/logs/`. Both Orthrus itself and a USB-drive were investigated for such forensic artefacts. In listing C.1, one may see the output from `/var/log/dmesg` which describes the recorded events that took place after the USB-drive was connected. One may clearly see that it is a mass storage device is connecting to the Raspberry Pi 3 (one of the devices used for testing Linux functionality) and going through its setup procedures. From the information in dmesg, one may see its storage capacity (being 120 megabytes), its serial number and whether or not it is write-protected. In listing C.2, the syslog describes a similar story, in addition to a few new pieces of information such as its mount point (`/media/pi/TARGET`). The syslog also shows us that this system it is connected to (the Raspberry Pi 3) has an auto-mounting feature, as stated by one of the entries, being `(...) Got automount request for (...)` on the very last line of the aforementioned listing.

When connecting the Orthrus device to a Raspberry Pi 4 running Ubuntu 19.10, one could also see a syslog being populated with entries, as can be seen in listing C.3. Information such as the product name (being Metro M4 Airlift Lite) and manufacturer (Adafruit Industries LLC). One may also see that Orthrus has established a communication port from the entry stating `(...) cdc_acm 1-1.1:1.0: ttyACM0: USB ACM device (...)`. The `ttyACM0` device is a USB Abstract Control Model device which is a standard of communication that microcontrollers often use (Tardieu, 2018). In short, one may say this is a way for serial devices to use USB to communicate with a host (Microchip Technology, 2004).

One may also see that a `usbhid` device has been registered. In a scenario where such information is presented, and one has to conclude whether or not a malicious device has been used, one only has to look at the clear evidence. Having the knowledge that a USB HID device has been connected, along with the fact that a device that is clearly not a keyboard (as per the product name), one may with high confidence say that this device is indeed likely to be impersonating an HID device. Thus, this revelation warrants further research into the matter. Indeed, just a few lines down, even further evidence in the syslog may be found that this device is certainly impersonating not only a keyboard but also a mouse device and a consumer control device. It is also revealed that the device that was connected, id est the Orthrus device, also has a storage capacity of two megabytes.

The syslog files have every entry in it marked with both a date and time stamp, which could be very useful in a forensic investigation. That is if one is certain that the information has not been tampered with. This is admittedly something a device such as Orthrus could do *only if* administrator rights had been obtained, as sudo is needed to both copy, remove or edit the syslog file.

4.3.2 Penetration testing

Penetration testing is an obvious field where Orthrus may shine. Obvious, because HID/keystroke injection is a common technique in many on-premise security audits. This can be attributed to its proficiency of rapid injection, for example, to enable a backdoor to the system without arousing much suspicion by using the keyboard.

One of the scenarios in mind when creating the many iterations of Orthrus was using it in combination with a separate storage device. This way, scripts could be used to instruct the target device to send information to the said storage device and not be limited by the small storage space that Orthrus would have (approximately 2 megabytes of storage). This method of extracting data from a system is also where Orthrus got its name, as its "two-headedness" could remind one of the two-headed dog Orthrus from Greek mythology.

It would indeed be possible only to use a CircuitPython device if such a device had enough storage. However, no such device is widely available at the time of writing. Even if such a device currently existed, one may questions its suitability in this project, as the inclusion of higher storage space would very likely also increase its cost.

4.3.3 Setup automation

A very likely and useful use-case for Orthrus is the automation of setup processes. In the scenario that a forensic examiner needs to boot to a specific distribution of Linux from for example a memory stick on a target device, Orthrus can be used to set the Linux distribution to be initialised with a specific set of parameters and settings. This would be an excellent use case for live forensics as booting from a memory stick and doing analysis this way is a rather common task. While there are alternatives to setting up devices, for example through a Bash script, Orthrus could serve as an alternative way of doing so where the aforementioned method with Bash is somehow not feasible.

In theory, one could not only use Orthrus for setup automation in the case of desktop computers but also mobile devices that are mainly touch-oriented. Similar as in the previous scenario, one could automate the setup process of a device to have a certain set of features enabled. It should be noted, however, that systems that have a strong reliance on touch input might be difficult to control and will probably prove to pose extra challenges to the user in comparison to, for example, a desktop computer.

4.3.4 Playtesting

Orthrus' versatility also enables users in other fields than those directly involved with digital forensics or other fields in close proximity In listing 4.4 one can imagine the scenario where a playtester wants to investigate the possibility of recognising automated behaviour from a player which uses these kinds of methods to progress in a game, hence giving them an unfair advantage. As raw input into a video game console can be harder to detect than if a malicious user was using only software (that often anti-cheat solutions can identify), this gives such a playtester an easy to use the platform to test their hypotheses with much less work and much higher accuracy.

Listing 4.4: OSL example code: Advanced OSL example with nested looping, fragments, vairables

```
1
2  VAR: print_this, this will be printed to serial
3
4  LOOP[inf]
5      OSL: routines/fragments/target_enemy.osl
6      LOOP: 2
7          M[click]: LEFT_BUTTON
8      ELOOP
9      SER: {{print_this}}
10     OSL: routines/fragments/move_character.osl
11     M[click]: RIGHT_BUTTON
12     K: ENTER
13 ELOOP
```

While gamepad support is not yet implemented in Orthrus, such is entirely possible, and Adafruit has, at the time of writing, recently included rudimentary support for gampad controls in their CircuitPython libraries. Including gampad support can further diversify Orthrus' use cases in the different fields of automated testing.

4.4 Reaching aims

While making an HID injection device has been completed, the project cannot be regarded as successful unless the overarching goals also have been completed. This section will discuss this aspect of the research.

4.4.1 The user aspect

There have been surveys done on the user-friendliness of digital forensics tools. In the survey, consisting of 115 people, it was found that the tools that they tested were not considered user-friendly and that they were unintuitive (Hibshi *et al.*, 2011). While the tools tested are very different in both goal and function from Orthrus, one may argue that there is a trend among popular proprietary solutions that the software tends not to be either intuitive or user-friendly. One can attribute this to the vast range of functionality that is needed in such software, as information can be hidden in various ways. Still, there is a case to be made for tools that are simple to use and offer a smaller set of functions, which was discussed in the aforementioned survey.

4.4.2 Maintainability

Compared to the initial design where convoluted C++ code was produced to facilitate simplistic HID injections, the current code is compact and mostly self-explanatory in comparison. One often talks about pythonic code being self-documenting, and in the case of Orthrus, one may argue that such is the case. For a beginner, the balance between abstractions and imperativeness should provide a good middle ground for not only understanding the software but also encourage participation in the further development of Orthrus.

Maintainability also depends on the further development and support of the open-source community into the MicroPython and CircuitPython ecosystem. As engagement is currently quite high, this factor should not be a major hindrance towards expanding upon the Orthrus software.

4.4.3 Open-source

While having open-source tools is excellent for it being more accessible to more people (although it should be noted that open-source does not always mean free), one may argue that the biggest benefit is that the examiner or other experts in the forensic field being able to review the code. Alfred Korzybski, a renowned philosopher, once said, *"the map is not the territory"*. In other words, the map is not the ground truth, and neither is a

compiled program. That the source code is able to be reviewed and scrutinised is essential in understanding its actual functions. Also, being able to take that code, compile it and have a version that one can without a doubt say is as close to the ground truth as possible, is invaluable especially so when one thinks about the life-changing decisions that can be taken depending on what evidence that is found (Altheide and Carvey, 2011).

There is also an issue with obsolescence. Proprietary solutions, both tools and formats, may have their support dropped by the company that distribute said software, formats or licences to these. Hence, these solutions will not be updated and therefore, become obsolete. In a field such as digital forensics where the technological development is very fast-paced, using obsolete software is close to useless as it could potentially mean that much data is not gathered, hence giving an incomplete picture of a case. The issue of obsolescence also ties into working with older cases which used the now obsolete software. Imagine the scenario in which an old case have to be reopened and that the evidence has to be reexamined. In the case of the material is using software that is obsolete, or even worse, possibly not distributed anymore, one might find oneself in a hopeless situation. Of course, one may use the current and more modern tools to reexamine the evidence from scratch, but that is if one still has the evidence, which depending on the jurisdiction, might not be a viable alternative.

If the tool was open-source, however, the community could continue to update this software so that the aforementioned scenarios would never an issue in the first place. Even if the software were only minimally supported, such compatibility updates, by the open-source community, this would be enough for at least most worst-case scenario to never happen, being that the software cannot even be run.

There was a tendency for a long time that the field of digital forensics was dominated by closed-source tools (Altheide and Carvey, 2011). Given the aforementioned reasons why this can be detrimental to, for example, an investigation, it is therefore vital that the digital forensics community, therefore, start investing their expertise into open-source solutions.

Because of all this, Orthrus is open-source and licenced under the GNU General Public License v3.0.

4.5 Summary

This chapter has underlined the importance of understanding one's aims, how to conform to them and to identify areas where one might be lacking. For Orthrus specifically, most of the aims are perceived to be successfully achieved, although some areas might warrant more research. While the forensic impact in terms of artefacts was discussed in section 4.3.1, this area could be expanded upon to include research into how a user might introduce forensic artefacts as well; a point which will be discussed more in section 5.3.1. The importance and values of havings one's source open to the public were also discussed and how that especially relates to the field of digital forensics.

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

5

Conclusion

This research aimed to develop a platform which was accessible to users of various levels of technical skills. Its focus was especially on those who work in the field of digital forensics. This chapter summarises the work that was produced by this research, contrasts the work with the initial aims and attempts to conclude from said results.

5.1 Key aspects

The main aims of this research, as described in section 1.1, can be summarised as the following:

- Accessible: Orthrus must be easy to use for a diverse range of professionals having various technical skills.
- Maintainable: Orthrus must be reasonable to maintain for contributors.

- Low investment; high yield: Investments, both in terms of finances and time, cannot be too high compared to the yield one would expect from a tool such as Orthrus.
- Open-source: Orthrus must be open-source so that everyone may improve upon it and scrutinise it. It can, therefore, not be a black box.
- Modular: Orthrus has to be built in a way that scripts written for it can be used in a modular manner.

The rest of this section will briefly summarise the findings, draw conclusions from them and contrast them to the original goals.

5.1.1 Development

The initial phase of the development was rather convoluted as previous methods were very diverse. Many existing solutions had features that went against the main aims of this project, such as not being accessible or maintainable enough. It was therefore concluded that the final design had to take a rather different approach than previous works whilst still appreciating this effort and acknowledging it.

The final design, as described in section 3.2, was made possible after a suitable platform was found; both in terms of hardware and software. While not all the hardware devices proposed were found to be suitable, one of the devices, namely the Adafruit Metro M4 Airlift Lite, suited the project well and could allow for future expansions (such as wireless control described in 5.3.1).

The production and design of the parser was a major part of the design process as it is indeed a major part of Orthrus itself. It allows for quick and easy prototyping of automation scripts without necessarily needing a lot of technical skills. The parser itself was designed in such a way that it would not be a problem for future expansion, hence honouring the goal of maintainability. The command set was designed to be as simple as possible while still allowing for a wide range of functionality.

To enable Orthrus to be truly cross-platform, extensive testing and documentation of the differences between the platform had to be done. While the syntax and how Orthrus internally was not heavily impacted by testing on the different platforms, it was essential

to have at least tested that Orthrus perform identically on both platforms.

5.1.2 Forensics

As the focus of this research was Orthrus' potential applications in the field of digital forensics, a review of forensic artefacts had to be done. It was found that the baseline system did indeed leave a trace. On Windows, changes could be seen in places such as the registry and Event Viewer. This information included the product name of the Orthrus device, namely Adafruit Metro M4 Airlift Lite, as well as manufacturer details and model number. It was also possible to view the exact time when these forensic artefacts were created.

Similarly, on Linux, it was possible to find the same kind of information, although with much greater granularity. By looking through both `/var/log/syslog` and `/var/log/dmesg`, it was possible to get a wide range of information. This information included very similar entries as with Windows, but also additional information such as which specific communication ports were used. Timestamps were, as a result of this increased granularity, much more numerous and detailed.

5.1.3 Applications

The Orthrus device was originally intended for applications in the field of digital forensics. While this aim remained throughout the project, other applications were identified such as penetration testing (or offensive security in general), setup automation and playtesting. This came as a result of the design process focusing on making the Orthrus platform a solid step towards automated data collection in digital forensics, and in the process of doing so, all the basics for Orthrus to be used in alternative applications thus became possible. Some examples of this can be seen in section 4.3. Expansion into further applications can also result from the future work on Orthrus, of which is discussed in section 5.3.

5.2 Closing statements

In summary, this project has achieved its aims described in section 1.1. In addition to creating a device, namely Orthrus, that can serve as an example of an open-source device used for digital forensics, this project has also provided a platform capable of HID injection that can be used in several fields; not only digital forensics. The research has also provided a theoretical overview of future work and applications.

5.3 Future work

A number of features and improvements are in the works for Orthrus as well as further investigations into the forensic side of Orthrus to make sure that it is forensically sound.

5.3.1 Features

Some features that have been proposed to the further development of Orthrus, with a focus on features that can make Orthrus even easier for non-technical users such as first responders that may not be trained in the field of digital forensics.

Automatic obfuscation

As stated in section 3.6.1, obfuscation is necessary for a script not to be marked as malicious by the anti-virus. However, having, for example, pre-obfuscated Powershell scripts can prove to be not very user friendly, especially if such scripts have to be customised for a particular system in order to run as intended.

Hence, some automatic obfuscation done by Orthrus that can make this part of one's operation less labour intensive is planned. Rudimentary research and implementation have been tested, but so far, nothing that would make the process less labour intensive has been formalised.

Screen control

Being a prototype, Orthrus can be improved by adding a light GUI by using a small screen. This can allow a user to select scripts from a list rather than choosing from a set of buttons, which is the current case. While this can bring additional setup requirements for the user, there is a strong case to be made that it would enhance the user-friendliness of such a device considerably.

Wifi enabled control

While having the possibility of controlling an Orthrus device wirelessly might sound useful, the security aspect of having a device that may be used for forensic operations would argue for that doing so is a bad idea. As discussed in section 2.3.4, one of the reasons why this device cannot be deployed to a forensic environment is its reliance on wireless connectivity, both for control and data transfer. Especially the latter can prove to

However, by only enabling wireless control, many of the issues of having access to the actual information that Orthrus may have been less problematic. Hence, implementing wireless control predictably and securely may allow Orthrus to become even more user friendly.

Multiline comments and strings

Most scripting languages have either multiline comments, multiline strings or both, while Orthrus does not. The reason for this is an inherent weakness of parsing a script line by line using a very simplified parser. To enable features such as these, however useful, may prove to be a challenge. Indeed, a complete reevaluation of how Orthrus is parsing the OSL scripts may be needed for such features to work reliably.

There is an argument to be made that further complicating the code defeats the purpose of Orthrus being approachable as a project to contribute to, so decisions such as rewriting the parser must be considered carefully.

Ducky Script to OSL conversion

Ducky Script, as mentioned in sections 2.3.2 and 2.3.1, is an established and widely known way of facilitating HID injection. Therefore, to have a way of converting the numerous scripts that are available to OSL would be very beneficial. Seeing the different command sets of Ducky Script (table 2.1) and OSL (table 3.1), such a feature is not at all an impossibility. This is because all of the commands that Ducky Script supports are also supported by OSL. Hence, it should be only a matter of translating the commands between the different scripts.

Gamepad support

HID devices are not just comprised of keyboards and mice. Gamepads are also a part of the family of HID devices, and adding support for them in Orthrus is a planned feature. This would allow a playtester, a scenario discussed in section 4.3.4, to further test their game with hardware instead of software.

Device support

This project only considered three CircuitPython devices, as discussed in section 3.2.2, but future work might entail getting Orthrus certified to work on more devices than just the Metro M4 Airlift. Ideally, a device that is even cheaper than the aforementioned device would be tested as it can enable more users to benefit from this project, as discussed in section 1.1.3.

Additional keyboard layouts

A problem with Adafruit's CircuitPython HID library is that it only supports the standard US keyboard layout. This can obviously pose as a problem in situations where devices are using not this keyboard layout as keystrokes are interpreted differently than from what the user intended. On Adafruit's CircuitPython HID library repository (Adafruit, 2020) there is currently a pull request for adding the German keyboard. Also, the Norwegian keyboard layout is expected to be added as a continuation of this project and to give back to the open-source community.

5.3.2 Forensics

Further investigation into the forensic impact that Orthrus may have is needed. While some forensic artefacts have already been discovered in section refssec:digital-forensics, there are still methods that can be attempted to make sure that the forensic impact that Orthrus has is properly documented and defined.

Research into how a user's scripts can leave a forensic impact, such as the examples shown in section 3.7, would be a valuable addition going forwards. While not every scenario may be covered by such research, to give users an indication of how some actions might impact the forensic soundness of a system may prove to be a great candidate for further research.

References

Adafruit. Adafruit_CircuitPython_HID. Apr 2020. [Online; accessed 3. May 2020].
URL https://github.com/adafruit/Adafruit_CircuitPython_HID

Adafruit Industries. Adafruit Feather 32u4 Adalogger. Apr 2020a. [Online; accessed 27. Apr. 2020].
URL https://www.adafruit.com/product/2795

Adafruit Industries. Adafruit Trinket M0 - for use with CircuitPython & Arduino IDE. Apr 2020b. [Online; accessed 28. Apr. 2020].
URL https://www.adafruit.com/product/3500l

Al-Kasassbeh, M., Mohammed, S., Alauthman, M., and Almomani, A. Feature selection using a machine learning to classify a malware. In *Handbook of Computer Networks and Cyber Security*, pages 889–904. Springer, 2020.

Altheide, C. and Carvey, H. Digital forensics with open source tools. Elsevier, 2011.

Anthony, S. Massive, undetectable security flaw found in USB: It's time to get your PS/2 keyboard out of the cupboard - ExtremeTech. Jul 2014. [Online; accessed 4. May 2020].
URL https://www.extremetech.com/computing/187279-undetectable-indefensible-security-flaw-found-in-usb-its-time-to-get-your-ps2-keyboard-out-of-the-cupboard

Arduboy. Arduboy. Apr 2020. [Online; accessed 27. Apr. 2020].
URL https://arduboy.com

Barnes, R. Make a Pi Zero W Smart USB flash drive — The MagPi magazine. May 2020. [Online; accessed 6. May 2020].
URL https://magpi.raspberrypi.org/articles/pi-zero-w-smart-usb-flash-drive

Bohannon, D. Invoke-Obfuscation. Mar 2019. [Online; accessed 3. May 2020].
URL https://github.com/danielbohannon/Invoke-Obfuscation

Carrier, B. Open source digital forensics tools: The legal argument. Technical report, stake, 2002.

Davidoff, S. Cleartext passwords in linux memory. *Massachusetts institute of technology*, pages 1–13, 2008.

Dawes, R. P4wnP1. Dec 2018. [Online; accessed 2. May 2020].
URL https://github.com/RoganDawes/P4wnP1

Dawes, R. P4wnP1_aloa. Feb 2020. [Online; accessed 2. May 2020].
URL https://github.com/RoganDawes/P4wnP1{$_$}aloa

Denney, K., Erdin, E., Babun, L., and Uluagac, A. S. Dynamically detecting usb attacks in hardware: poster. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, pages 328–329. 2019.

Does, T., Geist, D., and Van Bockhaven, C. Sdio as a new peripheral attack vector. 2016.

Fabian, M. Endpoint security: managing usb-based removable devices with the advent of portable applications. In *Proceedings of the 4th annual conference on Information security curriculum development*, pages 1–5. 2007.

Farhi, N., Nissim, N., and Elovici, Y. Malboard: A novel user keystroke impersonation attack and trusted detection framework based on side-channel analysis. *Computers & Security*, 85:240–269, 2019.

GTFOBins. GTFOBins. Apr 2020. [Online; accessed 24. Apr. 2020]. URL https://gtfobins.github.io

Haffejee, J. and Irwin, B. Testing antivirus engines to determine their effectiveness as a security layer. In *2014 Information Security for South Africa*, pages 1–6. IEEE, 2014.

Hak5. Bash Bunny. May 2020a. [Online; accessed 1. May 2020]. URL https://shop.hak5.org/products/bash-bunny

Hak5. USB Rubber Ducky. May 2020b. [Online; accessed 4. May 2020]. URL https://shop.hak5.org/products/usb-rubber-ducky-deluxe

hak5darren. USB-Rubber-Ducky. Dec 2016. [Online; accessed 1. May 2020]. URL https://github.com/hak5darren/USB-Rubber-Ducky

Hatch, B., Lee, J., and Kurtz, G. Hacking Linux exposed: Linux security secrets & solutions. Osborne/McGraw-Hill New York, 2001.

Hibshi, H., Vidas, T., and Cranor, L. Usability of forensics tools: a user study. In *2011 Sixth International Conference on IT Security Incident Management and IT Forensics*, pages 81–91. IEEE, 2011.

Howard, C. Systems maintenance programs-the forgotten foundation and support of the cia triad. *SANS Institute GSEC*, 1(3), 2002.

Liao, X., Xie, X., and Jin, H. Sharing virtual usb device in virtualized desktop. In *2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming*, pages 156–160. IEEE, 2011.

Maltronics. Maltronics. May 2020. [Online; accessed 4. May 2020]. URL https://maltronics.com/collections/malduinos

Microchip Technology. Migrating from rs-232 to usb bridge specification. 2004. URL http://ww1.microchip.com/downloads/en/appnotes/doc4322.pdf

Microsoft. Antimalware Scan Interface (AMSI) - Win32 apps. May 2020. [Online; accessed 3. May 2020].

URL https://docs.microsoft.com/en-us/windows/win32/amsi/antimalware-scan-interface-portal

MITRE Corporation. Lateral Movement, Tactic TA0008 - Enterprise | MITRE ATT&CK®. Apr 2020. [Online; accessed 6. May 2020].
URL https://attack.mitre.org/tactics/TA0008

Newlin, M. Injecting keystrokes into wireless mice. 2016.

PowerShellMafia. PowerSploit. Dec 2016. [Online; accessed 3. May 2020].
URL https://github.com/PowerShellMafia/PowerSploit

Seytonic. malduino. Sep 2017. [Online; accessed 1. May 2020].
URL https://github.com/Seytonic/malduino

Tardieu, S. What is the difference between /dev/ttyUSB and /dev/ttyACM? Mar 2018. [Online; accessed 6. May 2020].
URL https://rfc1149.net/blog/2013/03/05/what-is-the-difference-between-devttyusbx-and-devttyacmx

Technology, M. ATmega32U4 - 8-bit AVR Microcontrollers. Apr 2020. [Online; accessed 27. Apr. 2020].
URL https://www.microchip.com/wwwproducts/en/ATmega32U4

Tey, C. M. A study of the imitation, collection and usability issues of keystroke biometrics. 2013a.

Tey, C. M. A study of the imitation, collection and usability issues of keystroke biometrics [poster]. 2013b.

USB Implementers' Forum. Universial serial bus (usb) - hid usage tables (v1.12). 2004.
URL https://www.usb.org/sites/default/files/documents/hut1_12v2.pdf#page=53

Zhou, X., Xu, B., Qi, Y., and Li, J. Mrsi: A fast pattern matching algorithm for anti-virus applications. In *Seventh International Conference on Networking (icn 2008)*, pages 256–261. IEEE, 2008.

Zhou, Z., Zhang, W., Yang, Z., and Yu, N. Exfiltration of data from air-gapped networks via unmodulated led status indicators. *arXiv preprint arXiv:1711.03235*, 2017.

Appendices

A

Code instructions

The code repository, along with instructions on how to set up Orthrus as well as run it, may be found at the following URL:

https://github.com/vagnes/orthrus

B

Orthrus logo

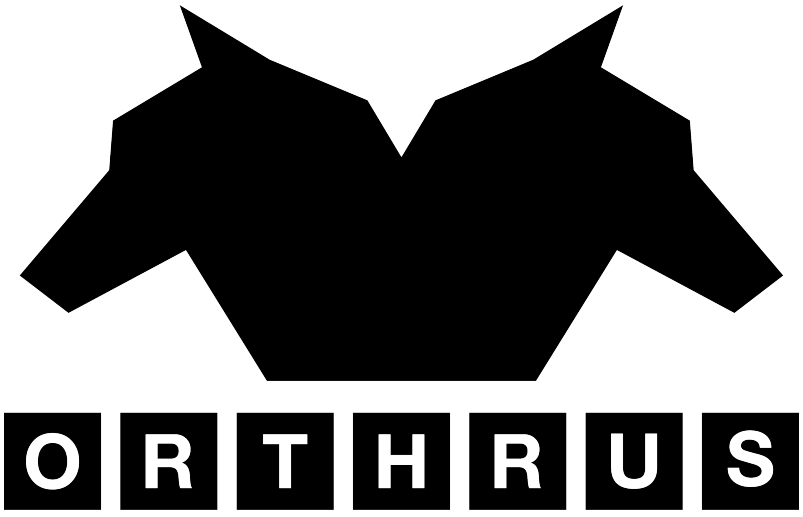

Figure B.1: Orthrus logo made by Johannes Tomren Røsvik

C

Logs

Listing C.1: Indications in /var/log/dmesg for connecting an USB-drive to a Raspberry Pi 3 Raspbian 10 (buster).

```
1         [109113.988478] usb 1-1.4: new high-speed USB device ↩
             number 6 using dwc_otg
2         [109114.119379] usb 1-1.4: New USB device found, ↩
             idVendor=1908, idProduct=1320, bcdDevice= 0.00
3         [109114.119389] usb 1-1.4: New USB device strings: Mfr↩
             =1, Product=2, SerialNumber=3
4         [109114.119394] usb 1-1.4: Product: Disk
5         [109114.119399] usb 1-1.4: Manufacturer: USB
6         [109114.119403] usb 1-1.4: SerialNumber: 372748↩
             DE43515174
7         [109114.129404] usb-storage 1-1.4:1.0: USB Mass ↩
             Storage device detected
8         [109114.136193] usb-storage 1-1.4:1.0: Quirks match ↩
             for vid 1908 pid 1320: 20000
9         [109114.136348] scsi host0: usb-storage 1-1.4:1.0
10        [109114.213720] usbcore: registered new interface ↩
             driver uas
11        [109115.199052] scsi 0:0:0:0: Direct-Access     USB ↩
                  Disk             2.60 PQ: 0 ANSI: 2
```

```
12          [109115.204209] sd 0:0:0:0: [sda] 245760 512-byte ↩
               logical blocks: (126 MB/120 MiB)
13          [109115.204429] sd 0:0:0:0: [sda] Write Protect is off
14          [109115.204441] sd 0:0:0:0: [sda] Mode Sense: 0b 00 00↩
               08
15          [109115.204604] sd 0:0:0:0: [sda] No Caching mode page↩
               found
16          [109115.204611] sd 0:0:0:0: [sda] Assuming drive cache↩
               : write through
17          [109115.206709]  sda:
18          [109115.208056] sd 0:0:0:0: [sda] Attached SCSI ↩
               removable disk
19          [109115.222564] sd 0:0:0:0: Attached scsi generic sg0 ↩
               type 0
```

Listing C.2: Indications in /var/log/syslog for connecting an USB-drive to a Raspberry Pi 3 running Raspbian 10 (buster).

```
1           May  5 18:18:04 raspberrypi rngd[332]: stats: Time ↩
               spent starving for entropy: (min=0; avg=0.000; max↩
               =0)us
2           May  5 18:36:30 raspberrypi kernel: [109113.988478] ↩
               usb 1-1.4: new high-speed USB device number 6 using↩
                dwc_otg
3           May  5 18:36:30 raspberrypi kernel: [109114.119379] ↩
               usb 1-1.4: New USB device found, idVendor=1908, ↩
               idProduct=1320, bcdDevice= 0.00
4           May  5 18:36:30 raspberrypi kernel: [109114.119389] ↩
               usb 1-1.4: New USB device strings: Mfr=1, Product↩
               =2, SerialNumber=3
5           May  5 18:36:30 raspberrypi kernel: [109114.119394] ↩
               usb 1-1.4: Product: Disk
6           May  5 18:36:30 raspberrypi kernel: [109114.119399] ↩
               usb 1-1.4: Manufacturer: USB
7           May  5 18:36:30 raspberrypi kernel: [109114.119403] ↩
               usb 1-1.4: SerialNumber: 372748DE43515174
8           May  5 18:36:30 raspberrypi kernel: [109114.129404] ↩
               usb-storage 1-1.4:1.0: USB Mass Storage device ↩
               detected
9           May  5 18:36:30 raspberrypi kernel: [109114.136193] ↩
               usb-storage 1-1.4:1.0: Quirks match for vid 1908 ↩
               pid 1320: 20000May  5 18:36:30 raspberrypi kernel: ↩
               [109114.136348] scsi host0: usb-storage 1-1.4:1.0
10          May  5 18:36:30 raspberrypi mtp-probe: checking bus 1,↩
                device 6: "/sys/devices/platform/soc/3f980000.usb/↩
               usb1/1-1/1-1.4"
11          May  5 18:36:30 raspberrypi mtp-probe: bus: 1, device:↩
                6 was not an MTP device
12          May  5 18:36:30 raspberrypi kernel: [109114.213720] ↩
               usbcore: registered new interface driver uas
13          May  5 18:36:30 raspberrypi mtp-probe: checking bus 1,↩
                device 6: "/sys/devices/platform/soc/3f980000.usb/↩
               usb1/1-1/1-1.4"
14          May  5 18:36:30 raspberrypi mtp-probe: bus: 1, device:↩
                6 was not an MTP device
15          May  5 18:36:31 raspberrypi kernel: [109115.199052] ↩
               scsi 0:0:0:0: Direct-Access     USB      Disk ↩
                           2.60 PQ: 0 ANSI: 2
```

```
16          May  5 18:36:31 raspberrypi kernel: [109115.204209] sd↩
                0:0:0:0: [sda] 245760 512-byte logical blocks: ↩
               (126 MB/120 MiB)
17          May  5 18:36:31 raspberrypi kernel: [109115.204429] sd↩
                0:0:0:0: [sda] Write Protect is off
18          May  5 18:36:31 raspberrypi kernel: [109115.204441] sd↩
                0:0:0:0: [sda] Mode Sense: 0b 00 00 08
19          May  5 18:36:31 raspberrypi kernel: [109115.204604] sd↩
                0:0:0:0: [sda] No Caching mode page found
20          May  5 18:36:31 raspberrypi kernel: [109115.204611] sd↩
                0:0:0:0: [sda] Assuming drive cache: write through
21          May  5 18:36:31 raspberrypi kernel: [109115.206709]  ↩
               sda:
22          May  5 18:36:31 raspberrypi kernel: [109115.208056] sd↩
                0:0:0:0: [sda] Attached SCSI removable disk
23          May  5 18:36:31 raspberrypi kernel: [109115.222564] sd↩
                0:0:0:0: Attached scsi generic sg0 type 0
24          May  5 18:36:32 raspberrypi systemd[1]: Created slice ↩
               system-clean\x2dmount\x2dpoint.slice.
25          May  5 18:36:32 raspberrypi systemd[1]: Started Clean ↩
               the /media/pi/TARGET mount point.
26          May  5 18:36:32 raspberrypi udisksd[357]: Mounted /dev↩
               /sda at /media/pi/TARGET on behalf of uid 1000
27          May  5 18:40:27 raspberrypi systemd[1]: proc-sys-fs-↩
               binfmt_misc.automount: Got automount request for /↩
               proc/sys/fs/binfmt_misc, triggered by 7143 (findmnt↩
               )
```

Listing C.3: Indications in /var/log/syslog for connecting an USB-drive to a Raspberry Pi 4 running Ubuntu 19.10.

```
1           May  5 19:14:31 rpi4 kernel: [12978.941242] usb 1-1.1:↩
                new full-speed USB device number 4 using xhci_hcd
2           May  5 19:14:31 rpi4 kernel: [12979.107379] usb 1-1.1:↩
                device descriptor read/all, error -32
3           May  5 19:14:31 rpi4 kernel: [12979.205243] usb 1-1.1:↩
                new full-speed USB device number 5 using xhci_hcd
4           May  5 19:14:31 rpi4 kernel: [12979.383472] usb 1-1.1:↩
                New USB device found, idVendor=239a, idProduct↩
               =8038, bcdDevice= 1.00
5           May  5 19:14:31 rpi4 kernel: [12979.383486] usb 1-1.1:↩
                New USB device strings: Mfr=2, Product=3, ↩
               SerialNumber=1
6           May  5 19:14:31 rpi4 kernel: [12979.383497] usb 1-1.1:↩
                Product: Metro M4 Airlift Lite
7           May  5 19:14:31 rpi4 kernel: [12979.383507] usb 1-1.1:↩
                Manufacturer: Adafruit Industries LLC
8           May  5 19:14:31 rpi4 kernel: [12979.383517] usb 1-1.1:↩
                SerialNumber: 7242B80723457335020202 93A093B0FF
9           May  5 19:14:31 rpi4 kernel: [12979.406632] usb-↩
               storage 1-1.1:1.2: USB Mass Storage device detected
10          May  5 19:14:31 rpi4 kernel: [12979.407139] scsi host0↩
               : usb-storage 1-1.1:1.2
11          May  5 19:14:31 rpi4 kernel: [12979.442028] cdc_acm ↩
               1-1.1:1.0: ttyACM0: USB ACM device
12          May  5 19:14:31 rpi4 kernel: [12979.445003] usbcore: ↩
               registered new interface driver cdc_acm
13          May  5 19:14:31 rpi4 kernel: [12979.445009] cdc_acm: ↩
```

```
      USB Abstract Control Model driver for USB modems ↩
      and ISDN adapters
14    May  5 19:14:31 rpi4 kernel: [12979.464587] usbcore: ↩
      registered new interface driver usbhid
15    May  5 19:14:31 rpi4 kernel: [12979.464593] usbhid: ↩
      USB HID core driver
16    May  5 19:14:31 rpi4 kernel: [12979.536344] usbcore: ↩
      registered new interface driver snd-usb-audio
17    May  5 19:14:31 rpi4 kernel: [12979.552596] input: ↩
      Adafruit Industries LLC Metro M4 Airlift Lite ↩
      Keyboard as /devices/platform/scb/fd500000.pcie/↩
      pci0000:00/0000:00:00.0/0000:01:00.0/usb1↩
      /1-1/1-1.1/1-1.1:1.3/0003:239A:8038.0001/input/↩
      input0
18    May  5 19:14:31 rpi4 snapd[1289]: hotplug.go:199: ↩
      hotplug device add event ignored, enable ↩
      experimental.hotplug
19    May  5 19:14:32 rpi4 kernel: [12979.609707] input: ↩
      Adafruit Industries LLC Metro M4 Airlift Lite Mouse↩
       as /devices/platform/scb/fd500000.pcie/pci0000↩
      :00/0000:00:00.0/0000:01:00.0/usb1↩
      /1-1/1-1.1/1-1.1:1.3/0003:239A:8038.0001/input/↩
      input1
20    May  5 19:14:32 rpi4 kernel: [12979.610400] input: ↩
      Adafruit Industries LLC Metro M4 Airlift Lite ↩
      Consumer Control as /devices/platform/scb/fd500000.↩
      pcie/pci0000:00/0000:00:00.0/0000:01:00.0/usb1↩
      /1-1/1-1.1/1-1.1:1.3/0003:239A:8038.0001/input/↩
      input2
21    May  5 19:14:32 rpi4 kernel: [12979.610547] input: ↩
      Adafruit Industries LLC Metro M4 Airlift Lite as /↩
      devices/platform/scb/fd500000.pcie/pci0000↩
      :00/0000:00:00.0/0000:01:00.0/usb1↩
      /1-1/1-1.1/1-1.1:1.3/0003:239A:8038.0001/input/↩
      input3
22    May  5 19:14:32 rpi4 kernel: [12979.610696] hid-↩
      generic 0003:239A:8038.0001: input,hidraw0: USB HID↩
       v1.11 Keyboard [Adafruit Industries LLC Metro M4 ↩
      Airlift Lite] on usb-0000:01:00.0-1.1/input3
23    May  5 19:14:32 rpi4 systemd[1]: Reached target Sound ↩
      Card.
24    May  5 19:14:32 rpi4 kernel: [12980.414650] scsi host0↩
      : scsi scan: INQUIRY result too short (5), using 36
25    May  5 19:14:32 rpi4 kernel: [12980.414676] scsi ↩
      0:0:0:0: Direct-Access     Adafruit Metro M4 ↩
      Airlift Lite 1.0  PQ: 0 ANSI: 2
26    May  5 19:14:32 rpi4 kernel: [12980.415797] sd ↩
      0:0:0:0: Attached scsi generic sg0 type 0
27    May  5 19:14:32 rpi4 kernel: [12980.421907] sd ↩
      0:0:0:0: [sda] 4089 512-byte logical blocks: (2.09 ↩
      MB/2.00 MiB)
28    May  5 19:14:32 rpi4 kernel: [12980.425989] sd ↩
      0:0:0:0: [sda] Write Protect is off
29    May  5 19:14:32 rpi4 kernel: [12980.425997] sd ↩
      0:0:0:0: [sda] Mode Sense: 03 00 00 00
30    May  5 19:14:32 rpi4 kernel: [12980.430567] sd ↩
      0:0:0:0: [sda] No Caching mode page found
31    May  5 19:14:32 rpi4 kernel: [12980.435998] sd ↩
      0:0:0:0: [sda] Assuming drive cache: write through
32    May  5 19:14:32 rpi4 kernel: [12980.479398]  sda: sda1
33    May  5 19:14:32 rpi4 kernel: [12980.508470] sd ↩
```

```
0:0:0:0: [sda] Attached SCSI removable disk
```