École Polytechnique Fédérale de Lausanne

Fuzzing Logitech proprietary protocols for embedded systems and Web APIs with LogiFuzz

by Alessandro Bianchi

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Nabil Hamzi
In-Company Advisor

Damian Vizár
External Expert

Prof. Dr. Daniele Antonioli
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

August 20, 2021

Dedicated to my parents, my sister, and my friends.

Acknowledgments

As for many other tasks in my life, it would not have been possible for me to complete this Master Thesis alone. All of the following people have given their contribution to this work, and I am deeply grateful.

I would like to express my gratitude to my academic supervisor, Prof. Mathias Payer, for giving me the great opportunity to work on this project. I would also like to thank Nabil Hamzi, my supervisor at Logitech, for constantly guiding me throughout this thesis, and helping me choose its direction. His feedback and ideas have been vital to shape this project. I am deeply grateful to my advisor Prof. Daniele Antonioli, who I have had the pleasure to work with for one year by now. His constant and patient support has been vital to bringing this dissertation to its current level.

I would like to thank my parents, who have been supporting me and encouraging me throughout my studies. They have raised to me hard working and strong-willed, and taught me the importance of not giving up against adversities. My sister, Floriana, who I have been looking up to ever since I was a child, also deserves my most sincere gratitude. Her advice and emotional support has been of constant help for me, and it makes me strive to be as good of a brother as possible for her. I would also like to thank my new and fluffier sister Luna, who brought much happiness to my parents and, as a consequence, to me, and my aunt Silvia, who carries her mother's love and affection for me.

Finally, I would like to thank all my friends. Federico, who I have grown up with, and I can proudly call my brother. Giuseppe, for always being able to make me laugh and look at the bright side, and for all the time spent on matters we are so passionate about. Giuliana, for her unconditional love and support that she patiently shows me every day, and that no amount of free meals could ever repay. Mario and Alberto, for all the days and nights spent together joking, playing, and forgetting about all our worries: in the hope that many more of these times will come in the future. Clara, Giulia, Rocco, Gianluca, João, Fulvio, Giacomo, Donatella, Davide, Giorgio, and Francesco, for supporting me in one way or another. I feel incredibly grateful to all of you.

Lausanne, August 20, 2021 Alessandro Bianchi

Abstract

Fuzzing is an automated testing technique, that consists in sending unexpected data to a target program with the goal of finding security vulnerabilities. Fuzz testing has proven to be crucial, as it allows programmers to efficiently find a large amount of bugs and, as a consequence, vulnerabilities, that would otherwise remain undetected. A wide variety of different tools have been developed for this purpose, and they can be categorized as black box, white box, and grey box fuzzers. White and grey box fuzzers require access to the target's source code in order to instrument it, thus allowing for a deeper and more efficient testing. In some cases, however, such as those of embedded device firmwares and Web APIs, it may be complicated or impractical to employ them. Black box fuzzing, on the other hand, requires no knowledge of the target; nonetheless, even when using state-of-the-art tools, resorting to black box fuzzing cannot always be considered a worthwhile choice. Because of the very large input space, the fuzzer needs to take into account the protocol's syntax, as well as the feedback received during the process. The goal of this project is to design and implement a stateful, flexible, and multi-purpose black box fuzzer, which can perform efficient black box fuzzing a selected domains. The tool, named LogiFuzz, offers input validation and fine tuning, runtime feedback evaluation, and options for stateful fuzzing. Compared to the state-of-the-arts tools, LogiFuzz also leverages expert users' knowledge of the targeted protocols and interesting test cases to further reduce the input space. The fuzzer targets USB-related protocols, such as HIDPP, HID, and USB itself, as well as Web APIs. Its design and implementation, however, are heavily focused on modularity and flexibility, with most of the complexity laying in the implementation of the fuzzing logic and the parsing of the configuration file. This way, LogiFuzz allows programmers to add support for other protocols' syntax by writing few C++ methods.

Contents

Chapter 1

Introduction

1.1 Setting

Security of the deployed software is nowadays a crucial aspect for companies operating in various industries. Independently from the kind of software deployed, the presence of vulnerabilities can lead to the discovery of high-impact attacks: when performed, these can seriously impact the target's finances and reputation. For this reason, many large players in the software industry, such as Google [12] and Microsoft [11, 27], have lately joined the research on automated testing techniques, the most popular one being fuzzing. Google has proposed AFL (American Fuzzy Lop) [12], a very popular tool which will be discussed in subsection 6.1.1; Microsoft developed many state-of-the-art fuzzers, including Sage [11] and RESTler [27]. More information on RESTler is available in subsection 6.3.1.

Fuzzing consists in sending a large amount of unexpected data of various kinds to a target program, in the hope of triggering a faulty part in the code. Because of the attention it has received and still receives from the research community [48], fuzzing is a technique in constant evolution, which can adapt to the different contexts where it needs to be used. For these reasons, Logitech has decided to research a way to efficiently fuzz:

1. the firmware of its most popular embedded devices, such as mice and keyboards. Firmware is a class of software that is permanently installed on a hardware device, in order to provide low-level control [49].
2. Web APIs, and in particular the Logi ID API. API stands for Application Programming Interface, and represents a software used to allow communication between two or more software applications. A Web API is an API that grants communication with Web platforms, such as Web servers.

Fuzzers can roughly be divided in three categories: black box, white box, and grey box. Black

box fuzzers are usually simple and easy to use, as they operate with no information concerning the target's source code. White and grey box fuzzers, on the other hand, are more complex and need access to the tested program's source, but usually provide a higher efficiency. More in depth information will be provided in subsection 2.1.1. Regardless, a high level perspective offers enough insight to understand why it is not always feasible to employ white box and grey box fuzzers: the source code might be unavailable or hard to instrument, and each fuzzer would need to be built from scratch with a specific target in mind. For this reason, we have concluded that a black box fuzzer would be the most appropriate solution to the problem. The main challenge consists therefore in designing a modern, efficient, and flexible tool, that can fuzz the given targets in a protocol-aware fashion, leverage the given output, and ultimately perform better than any random "dumb" fuzzer for the given protocols.

Indeed, years of research have made available a very large amount fuzzers, including many open source ones. As a consequence, these could be potentially used to test the targeted protocols. Nonetheless, despite the quantity and variety of published tools, there is no guarantee on the existence of a fuzzer that can perform *efficient* testing in a given context. For example, let us take into account programs that accept a heavily formatted input: in this case, a fuzzer that is not aware of its grammar will perform poorly, as a large majority of the provided test cases will not be able to pass the input validation phase, and will be simply discarded early on. In addition, most state of the art, generic fuzzers do not combine input validation with leveraging runtime information [21]. Finally, many black box fuzzing tools offer some way to improve the efficiency of the process for the specific protocols they are targeting [5, 27], but none of them is flexible enough to cover all of our targets.

1.2 Contribution

We are therefore looking for a fuzzer that can target multiple protocols, including Web APIs and USB-related ones, and perform smart fuzzing on them. Indeed, none of the existing fuzzers fits the niche we are operating in. USB device fuzzing is covered by few "dumb" tools such as Frisbee Lite [13]. Web API fuzzing is certainly more popular, but state-of-the-art tools such as the aforementioned RESTler are not general-purpose. In addition, no fuzzing framework was found able to perform smart and customizable fuzzing of different platforms in a user-friendly manner. As a consequence, our goal is to design, implement, and test a fuzzing framework that allows to perform efficient black box fuzz testing for our targets, but can be easily extended to others.

The proposed solution, which will be discussed in the rest of this thesis, is *LogiFuzz*. LogiFuzz is a smart, flexible, and modular black box fuzzing tool, which currently supports fuzzing of USB-related protocols and Web APIs with the aid of a user-defined configuration file, through which its behavior can be fully customized.

LogiFuzz aims at performing protocol-aware fuzzing on all of its supported targets, mixing

feedback evaluation with user-defined syntax for the input. The smartness of the fuzzer is complemented by the user's knowledge of the fuzzed protocol, as the range and format of the fuzzed variables is chosen through the configuration file.

The following main contributions are offered in this thesis:

1. Design of a generic fuzzing framework, named LogiFuzz. LogiFuzz's design is heavily focused on modularity and customizability. A modular design allows programmers to easily extend the fuzzer to support more protocols. High customizability allows users to define the syntax of the generated test cases and define thresholds and formats for their values;
2. Implementation of LogiFuzz using widespread and modern technologies. The tool is fully programmed in C++14, and uses TOML in order to define configuration files, which are then used to customize its behavior. LogiFuzz currently supports USB-related protocols and Web APIs, with custom features added for both targets;
3. Evaluation of LogiFuzz in two use cases: (i) fuzzing of firmware of USB devices, and (ii) fuzzing of a Logitech Web API. Through this evaluation, we found a vulnerability that allows malicious users to perform ghost device attacks using a Logitech Unifying Receiver and a Logitech K800 keyboard.

In the following sections, we will provide an in-depth analysis on LogiFuzz. In chapter 2 we will discuss relevant background knowledge, that is essential to the full understanding of the contribution. In chapter 3 we will detail all of LogiFuzz's design choices, and explain why they fit with our objectives. In chapter 4 we will dive deeper into the most important implementation details, justify them, and relate them to our design choices. In chapter 5 we will describe the process we have employed to evaluate LogiFuzz, and give insight on the results we have obtained. Finally, in chapter 6 we will discuss how relevant state of the art fuzzers compare to LogiFuzz.

Chapter 2

Background

Before properly introducing LogiFuzz, it is vital to make sure that the reader is familiar with the core concepts behind the tool's design, as well as with both the employed and targeted technologies. By providing details on all the relevant background information, we mainly aim at improving the reader's understanding of chapter 3 and chapter 4, where we will thoroughly detail the design philosophy and implementation details of LogiFuzz. In this chapter, we will therefore give an overview of fuzzing, as well as briefly touch on the targeted protocols and some of the technologies used.

2.1 Fuzzing

Fuzzing, also known as fuzz testing, is an automated software testing technique, which consists in providing a target software with invalid, malformed, unexpected, or simply random input, and observe the output, or lack thereof, to find software bugs [38, 50]. The core intuition behind the technique is quite straightforward: through the generation of a large number of test cases, the fuzzer will likely trigger some unwanted or unexpected behavior from the targeted program, such as a crashes or hangs. The response obtained can also be used to improve the quality of the generated test cases. The process is shown through a block diagram in Figure 2.1.

The idea of fuzzing was first proposed in 1988, while the first paper on the matter was published in 1990 [31]. In the past 30 years, fuzzing has been consistently used across many operating systems to find bugs and vulnerabilities in a large number of real life applications, such as web browsers (Internet Explorer, Mozilla Firefox, Apple Safari), kernels (iOS, OpenBSD), and more (MySQL, Linux memory management etc.) [7, 29, 30, 54]. Since exhausting the whole input space for most of the modern software is nearly impossible, the research on the topic focuses on various approaches to improve fuzzing *efficiency*: this translates to increasing the number of unique errors found per unit of time [15, 52], or improving the coverage of the explored code

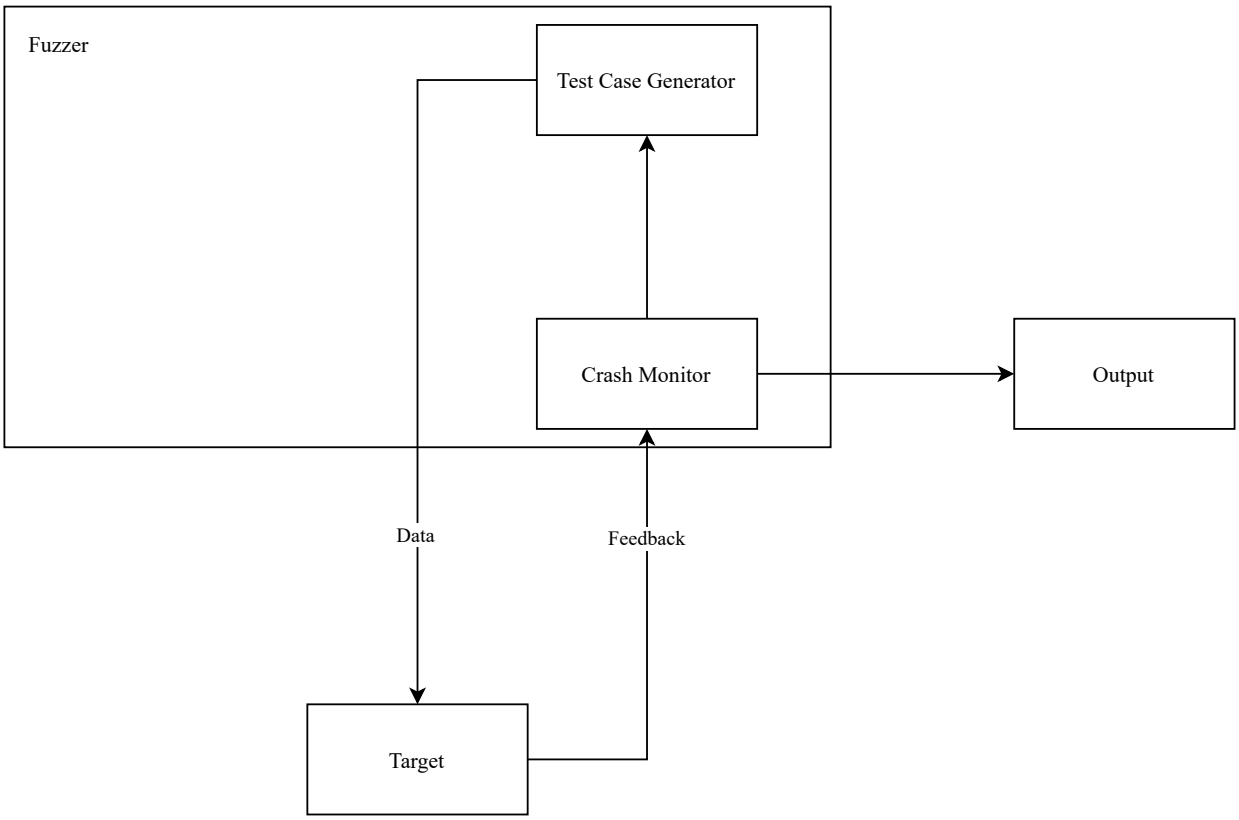

Figure 2.1 – High-level view of a generic fuzzing process.

base [37].

As expected with a pragmatic and widespread technique such as fuzzing, many different approaches have been devised, with the goal of satisfying varying requirements and adapting to a large amount of situations. Depending on the amount of knowledge that the fuzzer possesses on the target, fuzzing can be divided into three categories: *black box*, *white box*, and *grey box*. Black box fuzzers operate with no knowledge of the target's source code; white box fuzzers heavily rely on the target's source code to generate test cases that can maximize code coverage; grey box fuzzers may need either the source code or the binary, and are simpler than their white box counterpart. In addition, fuzzers can either generate the input they provide to their target from scratch, or mutate one or more test cases given by the programmer. In subsection 2.1.1 and subsection 2.1.2 we will provide more information concerning these different fuzzing techniques, detail their use cases, and briefly touch on their trade-offs.

2.1.1 Black Box Fuzzing vs White Box Fuzzing vs Grey Box Fuzzing

As stated earlier, fuzzers can be categorized as white, grey, and black box based on their reliance on the source code of the fuzzing target. We will now take apart the three techniques, explain

their requirements, detail their functioning, and analyze their benefits and drawbacks.

Black Box Fuzzing

Black box fuzzing is the most lightweight and simple application of the technique: as *no knowledge* of the target's source code is assumed, black box fuzzers only need to generate test cases in a fast manner and evaluate the received result. Black box fuzzers operate by either mutating one or more user-provided well-formed inputs, or generating test cases based on some well-known grammar [21].

The main advantage of black box fuzzers is their ease of use, as well as their usual state as general-purpose tools. Despite its simplicity, this technique turns out to be very effective at finding otherwise undetected bugs, and has commonly been used in industry with good results: Microsoft, for example, actively employs black box fuzzing for all of its products [7, 11]. Of course, the limitations of a black box approach are immediately apparent when analyzing the behavior of a program such as the following:

```
int foo(int x, int y)
{
    if (x == 42 && y == 42)
    {
        return y / (y - x);
    }

    return y + x;
}
```

In this case, supposing integers are 32-bits long, the probability of triggering a division by 0 (an undefined behavior that can result in a crash) when using purely random fuzzing is $1/2^{64}$. As a consequence, different approaches have been devised, along with improvements for the black box method [21].

White Box Fuzzing

Compared to its black box counterpart, white box fuzzing adopts an opposite approach, as it assumes *complete knowledge* and availability of the fuzzed software's source code, which is fully instrumented and analyzed. First and foremost, code instrumentation happens at compile time; the fuzzer adds its own code instrumentation to the program, in order to track the execution flow of the target. Subsequently, the fuzzer symbolically executes the program: this consists in executing the program in an abstract way, replacing the input data with symbolic expressions and dynamically tracking the effect of the provided input on target's conditional statements. This way, the fuzzer gathers constraints on said input: for instance, the main path of the code in Code 2.1.1 would impose $x \neq 42 \vee y \neq 42$. Those constraints are then negated, in order to

explore different code paths: in Code 2.1.1, this would impose $x = 42 \wedge y = 42$. Once this is done, the resulting constraints are fed to a constraint solver, which finds valid values that satisfy all of the given constraints. Finally, the results are mapped to new actual test cases, that will explore new code paths. This is usually repeated many times, using heuristics to attempt to cover all of the code paths, although this is not feasible in most cases [10].

Thanks to its strong focus on code coverage and heavyweight instrumentation, white box fuzzing is a very powerful technique: when testing the code in section 2.1.1, for instance, triggering the bug would be trivial. The provided example clearly shows the main benefits that white box fuzzers offer compared to black box ones; in practice, the former are often used in order to find deep-rooted and complex bugs, that the latter cannot trigger [10, 11]. Nonetheless, white box fuzzing still has drawbacks, and cannot be considered as a replacement for black box. First of all, full availability of the code is required for white box fuzzing. In addition, the code is sometimes dynamically instrumented at run-time: this is not always possible, particularly in the case of embedded devices. Finally, heavyweight instrumentation causes significant overhead.

Grey Box Fuzzing

Grey Box fuzzing is a middle ground between black box and white box fuzzing. Once again, this fuzzing technique requires access to the source code, as the fuzzer needs to perform a lightweight instrumentation: this usually aims at tracking code branches, in order to find out which input mutations manage to trigger unexplored paths, and further elaborate them [41, 55].

Despite sharing with the white box approach the need for dynamic, run-time information about the target's source code [21], grey box fuzzing is by design much more lightweight, as it mainly uses a feedback loop to evaluate which test cases manage to reach new code branches. In practice, this approach has proven to be a very effective technique [54].

2.1.2 Mutation-Based Fuzzing vs Generation-Based Fuzzing

In subsection 2.1.1 we have categorized fuzzing tools based on their knowledge of the target, as well as their reliance on its source code. In this section, we will instead compare the two different methods that fuzzers use to *generate inputs*. We will therefore detail two input generation techniques: (i) mutation-based, that creates new cases by modifying one or more user-provided ones, and (ii) generation-based, which relies on a well-defined grammar to define the structure of the test cases.

Mutation-Based Fuzzing

With a mutation-based approach, the fuzzer is provided with one or more good test cases, usually consisting of well formed, expected input. Subsequently, at each iteration, the fuzzer performs random or heuristic modification on those basic test cases: common ones include flipping bits, adding or removing bytes, and replacing part of the content [21, 32].

The main advantage behind a mutation-based approach is its plug-and-play nature: little to no knowledge is required on the actual structure of the test cases, as a single valid one is sufficient to kickstart the process [20, 32]. The strategy's ease of use, however, comes with clear drawbacks. On the one hand, the quality of the inputs, and their potential to trigger bugs, heavily depends on the nature of the provided test cases. On the other hand, software that requires a rigid input structure will most of the times simply reject a mutation-based fuzzer's input, as it will not pass the initial validation phase [20].

Because of their simplicity, mutation-based fuzzers are extremely popular general purpose tools. Such an example is AFL, which we will provide more details about in subsection 6.1.1 [12].

Generation-Based Fuzzing

With a generation-based approach, the fuzzer's input generation engine is usually more complex, as it relies on grammar defined by the programmer or the user to create new test cases. At each iteration, the fuzzer will attempt to create diverse inputs, that will however have to match with the defined grammar [21, 32].

Compared to mutation-based tools, generation-based fuzzers are undoubtedly more complex, as they require knowledge of the target's input structure [20]. Indeed, this increase in complexity is met with much greater effectiveness, as generation-based fuzzers manage to cover both of the drawbacks that have been mentioned for mutation-based ones. On the one hand, the quality of the input only depends on the generation algorithm used: this might either be random, or rely on heuristics. Regardless, this is often preferable to user-generated inputs, as they often lack the randomness that can help improve fuzzing effectiveness [32]. On the other hand, generation-based fuzzing is perfect for protocol-aware fuzz testing, as the structure of the input can be specified by the programmer, thus preventing input validation techniques from slowing down progress [20].

The power of generation-based fuzzing is exemplified by tools such as RESTler, that manage to provide smart and efficient black box fuzzing starting from well defined specifications of the expected input format [1]. We will provide additional information about RESTler in subsection 6.3.1.

2.2 Targeted Protocols and Technologies

In section 2.1 we have explained in detail what fuzzing is, discussed some of its different forms, and analyzed their advantages, drawbacks, and use cases. Before exploring LogiFuzz's design, however, we will offer some insight on its main supported protocols: Universal Serial Bus (USB), Human Interface Device (HID), and HID++, a proprietary extension of HID. When doing this, we will not go into full detail for each target, but rather just provide short and high level descriptions, to make the reader more familiar with the topic.

2.2.1 USB

USB (Universal Serial Bus) is an industry standard, that aimed at establishing uniform connectors, cables, and protocols that could be used across different devices [51]. In USB, communication occurs through a packet-based protocol, and always involves a USB host, which acts as the initiator, and a USB device. When a USB host sends data to an USB device, it does so through unidirectional endpoints [18]. USB offers four different transfer types: Control Transfer, Interrupt Transfer, Isochronous Transfer, and Bulk Transfer.

In this dissertation, we will only take into account communication through USB Control Transfers. This choice stems from two main reasons. On the one hand, control transfers are the only way to send data to Unifying Receivers. These are small USB wireless receivers, that can connect to multiple HID devices (such as mice, keyboards, and touchpads) at the same time [24], and represent one of the main targets for USB testing. On the other hand, in terms of inbound communication for HID devices, which we will detail in subsection 2.2.2, only support for Control Transfers is mandatory, while other types of transfers may or may not be allowed depending on the device.

In this thesis, we will not be focusing on any specific version of USB. This is possible thanks to the employed library, namely libusb [22], which offers the same programming interface regardless of the underlying USB version.

2.2.2 HID

HID (Human Interface Device) is a class of USB devices, including for example mice and keyboards [46]. Communication between a HID host and a HID device occurs through the exchange of HID reports, which can be seen as a Control Transfer at the lower USB level. As HID simply defines sets of usable USB commands for HID devices [17, 46], fuzzing of the USB and HID protocol is complementary.

2.2.3 HIDPP

HIDPP, also known as HID++, is a proprietary protocol developed by Logitech on top of the existing HID [23]. In HIDPP, communication is based on fixed-length vendor specific HID reports, which are exchanged between host and device. Two versions of HIDPP exist: HIDPP 1.0, which is implemented in Unifying Receivers, supports directly writing to registers; HIDPP 2.0 and higher, which is implemented in mice, keyboards, etc., has on the other hand a more complex set of commands, where it is possible to call appropriate functions belonging to supported features.

Chapter 3

Design

Among other things, in the previous sections we have discussed what fuzzing is, detailed its different forms, and highlighted its power and versatility. In this section, we will propose LogiFuzz, our own generic black box fuzzing framework, currently supporting USB, HID, HID++, and Web APIs. LogiFuzz makes use of a configuration file that allows programmers who possess in-depth knowledge to customize its behavior: this way, the tool aims at performing efficient and protocol-aware black box fuzzing, that provides the opportunity to focus on the specific features and test what the user deems most interesting.

3.1 Motivation

Indeed, intensive testing prior to release is particularly crucial for widespread commercial products, which are often built upon large code bases. With an industry average of 1-25 errors per 1000 lines of code [26], it is natural for commercially released software to potentially present a large number of bugs. As a consequence, it is obviously of great interest for enterprises to minimize the amount of fallacies in their code, thus presenting less vulnerabilities that may be exploited to perform very powerful attacks. For example, NSO Group's Pegasus spyware recently exploited a vulnerability in iMessage to perform a 0-click RCE (Remote Control Execution) attack on iPhones [25].

This effort is actually a crucial one: it has in fact happened several times in the past that products were released while still presenting undetected vulnerabilities, leading to several kinds of security breaches. A well known example of this is the MouseJack vulnerability, which allows to perform a RCE attack through a vulnerability in Unifying Receivers [2]. Another common attack vector lies in vulnerable Web APIs, often leading to data breaches [34]. These attacks can often have disastrous consequences, causing big players in the industry financial, legal, and reputational damage.

Given all of the aforementioned reasons, it is easy justify Logitech's efforts to further improve the security of its products. In addition, it must be of no surprise that a fuzzer has been identified as an effective tool for this purpose, given the overall effectiveness of the technique [54]. Therefore, we propose LogiFuzz. LogiFuzz is a generic and modular fuzzing framework, that allows to perform protocol-aware black box fuzzing of its supported protocols thanks to a user-defined configuration file. Currently, USB, HID, HIDPP and Web APIs are supported targets. In this section, we will detail the reasoning behind the core design decisions for LogiFuzz: we will highlight the existing problems and needs, and explain why the solution we propose is a good fit to solve them.

3.2 Built-In Targets

Logitech offers a broad range of products: much of the lineup consists of HID devices, such as mice, keyboards, headsets, webcams, and more. Indeed, the security of these products is directly impacted by possible vulnerabilities present in their firmware. The latter is traditionally seen as a secure black box; even when this idea has been challenged, much of the focus has been on identifying vulnerabilities in the USB host [4, 39], rather than on the way the firmware has been tampered with to make USB devices malicious. A vulnerable firmware, however, is a potential target for wide variety of attacks: in addition to the aforementioned MouseJack, some examples are DoS (Denial of Service) attacks, and ghost device attacks (where a device such as a mouse or a keyboard can be used, although it is not advertised as connected). The first goal for LogiFuzz is therefore to effectively and thoroughly fuzz the firmware of HID devices.

In addition to the integrity of its commercially released products' firmware, Logitech's second priority was the security of the Logi ID Web API. As a consequence, the fuzzer's scope was expanded, with the goal of fuzzing exposed endpoints in a generic Web API, in order to search for vulnerabilities such as privilege escalation and denial of service. Compared to USB fuzzing targeting firmware, Web fuzzing is much more developed, with some tools available for this purpose [9, 27]. However, LogiFuzz aims at covering a slightly different niche; the trade-offs with existing state of the art web fuzzers will be detailed in chapter 6.

3.3 Design Choices

3.3.1 Core Principles

We have therefore detailed the two core targets of LogiFuzz: firmwares of HID devices and Web APIs. Based on this, we will now expand on the reasoning behind all of the core design choices. We will start by addressing those that are common for both targets, and constitute the core of LogiFuzz's design; we will then proceed to explain elements that are specific to either of the two.

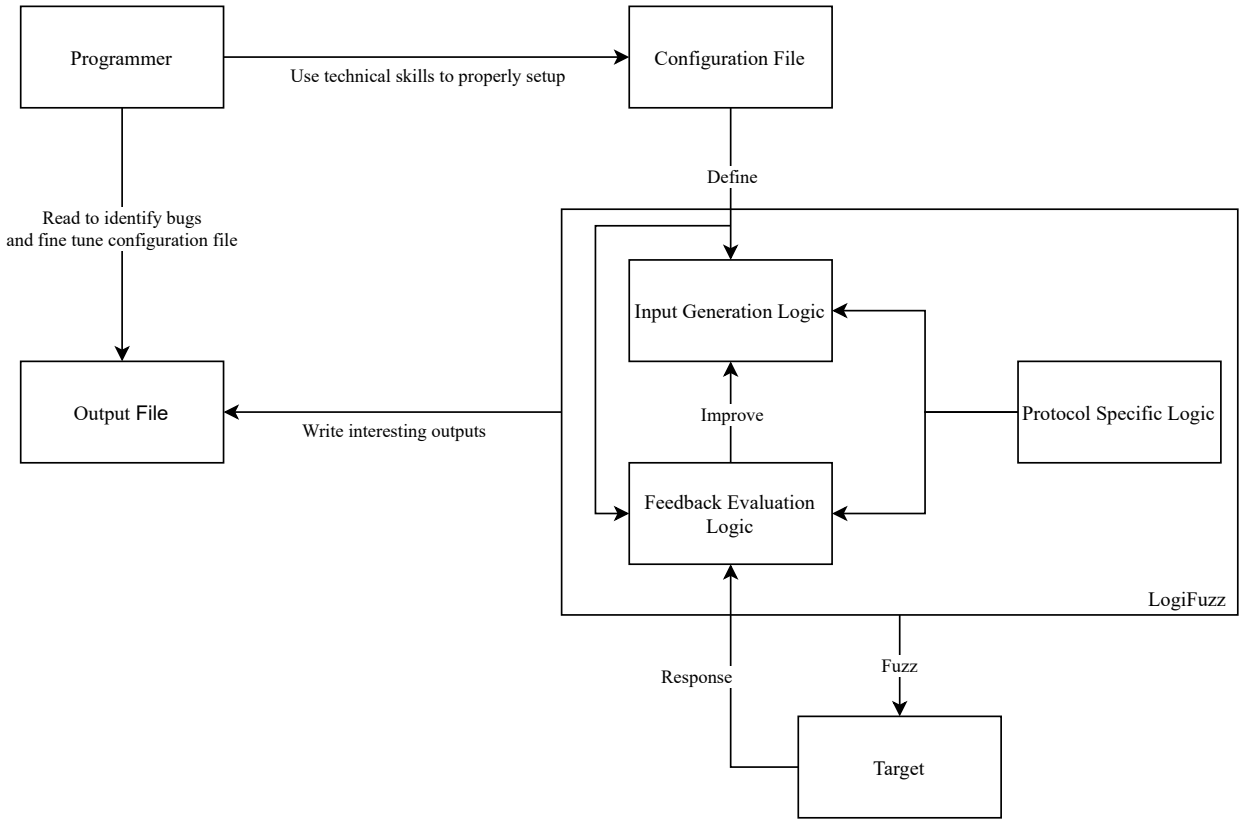

Figure 3.1 – High-level view of LogiFuzz's design.

If common elements present differences, we will detail them immediately, rather than calling back to them later on. It is worth noting that the use of a single fuzzing framework is itself a design choice, with two equally compelling reasons supporting it. On the one hand, there is little difference between the general structure and data generation logic for the two fuzzers. A modular design therefore allows to reuse most of the available features across all of the supported targets. On the other hand, providing the same input format and syntax for both fuzzers is an important added benefit, as it offers the programmers an experience that is much more user friendly, and, consequently, less error prone. A high-level view of the idea behind LogiFuzz's design can be observed in Figure 3.1.

3.3.2 Fuzzed Variables Specification

In order to support the aforementioned goal of being a general purpose fuzzer, LogiFuzz has been designed as a command line tool that is configured through a TOML (Tom's Obvious Minimal Language) configuration file. TOML is a format for configuration file, and it has been chosen because of its high readability and clarity [45]: these are in fact crucial properties, as they well integrate with the core flexibility requirement of the tool, and allow programmers to more easily tweak it to fit their needs. We will now give a high level view of what the configuration file

contains, as well as what can be customized through it; more details on how this is specified in practice will be given in subsection 4.3.1.

First and foremost, the configuration file defines all of the variables in the session: this includes both mandatory constants (such as the Vendor ID and Device ID of the device to fuzz, or the targeted website), and the actual variables to fuzz. This feature is designed to maximize flexibility in the fuzzing of platforms where there is no fixed number of parameters, but rather a varying amount of them. A good example of this are Web APIs, where requests to endpoint contain an arbitrary number of key-value pairs that constitute the query parameters and the request body. In more rigid protocols, such as USB and HID, on the other hand, the number of expected parameters is fixed, and inserting additional parameters is pointless. On a similar note, the configuration file can be used to freely choose the data type of each fuzzed variable: this includes integers of 1, 2, 4, and 8 bytes, booleans, strings, and arrays of any of the aforementioned integers. Once again, this feature aims at granting absolute flexibility when fuzzing platforms that support a wide range and variety of parameters; in the case of protocols with fixed data types, the data type specification is still present for consistency, as well as to remind the programmer about the specifications, thus minimizing the possibility of errors.

3.3.3 Fuzzing Parameters Specification

Now that we have established what programmers can control in terms of *what* is being fuzzed, we will detail the agency they have on *how* it is being fuzzed. As we have already established in chapter 2, fuzzers mainly come in two forms: generation-based and mutation-based. On this matter, LogiFuzz offers a per-variable choice to the programmer: with a generation-based approach, the fuzzer will generate a new random value in each loop, according to a customizable set of rules; with a mutation-based approach, the fuzzer will instead randomly flip bits starting by an initial value given by the programmer, once again taking into account defined limitations. With such an approach, LogiFuzz allows domain experts who are aware of potentially interesting (i.e.: dangerous) inputs to center fuzz testing sessions around those. On the other hand, if no such input is known, a generation-based approach allows for true randomized tests, which usually yield a higher code coverage [20, 32].

In addition to the two aforementioned fuzzing modes, LogiFuzz also offers two additional per-variable options. The first and most straight forward one simply allows to keep the value of a variable constant throughout the fuzzing process; the second one, on the other hand, lets the programmer specify the path of an input file, from which eligible values for the variable are read and adopted in a loop. The purpose of these additional options is to address the need to combine fuzzing with a more focused style of testing, that implies inference or at least partial knowledge of the source code. For example, a programmer might want to arbitrarily trigger some code paths, by letting a variable assume specific values that are known *a priori*.

As specified earlier, LogiFuzz allows programmers to fully customize the range of the fuzzed

	Min/Max Value	Min/Max Length	Format
Integer	✓	X (only 1, 2, 4, 8 bytes supported)	X
Array	✓ (for all elements)	✓	X
String	X	✓	✓ (predefined or through regex)

Table 3.1 – Customizable features for each supported data type

variables, both in the generation-based and mutation-based mode. We will now briefly explain which options are offered for each data type, and the reason why they have been made available. For a more concise view, refer to Table 3.1. Further implementation details will be provided in section 4.3.1.

For integers, LogiFuzz allows to limit adoptable values to custom non-disjoint intervals defined by the programmer. In addition, size of the integers in memory is chosen when declaring the type, as already stated earlier. In order to keep the syntax easy and understandable, it has been decided not to add any support for disjoint intervals, and multiple fuzzing instances are needed for this purpose. In a similar fashion, it is possible to define these limitations for arrays of integers: in this case, the provided interval is not per-index, but is rather applied to all of the elements. Once again, this choice aims at improving the readability and reducing the complexity of the configuration file. Furthermore, it is possible to impose upper and lower limits for the arrays' length, as well as define an exact one. This same property of length customization is shared by strings, due to their similarities to arrays. In addition, the programmer can define each string's format with absolute freedom. LogiFuzz primarily allows this by employing regular expressions: when one is provided as the string format, each of the randomly generated values for the variable will have to match the regex. Indeed, this approach offers the most control and accuracy; however, regular expressions are at times unnecessarily complex, and may hurt the readability of the configuration file. To make up for this, a second alternative is offered: the programmer may simply specify the set of characters allowed, by selecting options such as "alphanumeric" or "printable" in the format field. Finally, boolean variables are by definition not subject to any variation, and are subsequently not customizable.

3.3.4 Bug Detection and Reproduction

At this point, we have thoroughly explained how the programmer can manipulate the configuration file to control the nature of the input generated by the fuzzer. Indeed, this is the most crucial part of the process, as the efficiency of the tool heavily depends not only on its protocol-awareness, but also on smart fine tuning of its input. However, the overall usefulness of the fuzzer is also influenced by two additional factors: first, its capability to correctly identify and report a bug, when one is encountered; second, the support it can provide to the programmer in

the process of identifying and reproducing its findings. As a consequence, we will now detail how LogiFuzz attempts to address these challenges, staying true to its purposes of flexibility and simplicity. We will start by explaining how bugs are detected, what their nature is, and how the programmer can support the fuzzer in identifying them. We will then explore how LogiFuzz supports the reproducibility of the bugs found.

The most obvious classes of errors that LogiFuzz can encounter, as well as the easiest to identify, are crashes and timeouts. Indeed, these are automatically identified by the tool, without any need for external instruction coming from the programmer. Only focusing on errors of this nature, however, would be detrimental: on the one hand, it would severely limit the effectiveness of our solution; on the other hand, it would not line up with its design philosophy, nor fully take advantage of it. As a consequence, LogiFuzz allows programmers to define any number of error codes that must be considered suspicious or unexpected: when this feature is used, any input that triggers the given error is treated in the same way as a crash or a timeout, and flagged as a potential bug. For example, a $5xx$ response (implying a server error) would generally be considered unexpected and worth investigating when fuzzing a web server. Similarly, a 200 (success) response to an unauthorized request of a resource protected by access control should trigger an alarm. Indeed, similarly to the case of manipulation of variable values, specifying relevant return codes implies knowledge of the fuzzed target by the programmer who is using LogiFuzz.

Any output of the fuzzer is written to a text file defined by the programmer in the configuration file. This does not include basic logs, that are printed on the standard output. First and foremost, LogiFuzz prints to its output file the seed of the ongoing session, which is by default random and is used to initialize all of the pseudo-random generators internal to the tool. By editing the configuration file, it is then possible to set the seed to a user-defined value: indeed, this leads to complete reproducibility of any fuzzing session, as all of the generated values will be deterministic given the same seed. In many cases, however, running long fuzzing sessions multiple times is uselessly time-consuming. In the occasion that a potential bug is found, LogiFuzz outputs the entirety of the provided input, as well as the response received. This way, programmers may copy the information from the output file, use it set all of the relevant parameters to the desired (constant) values through the configuration file, and finally use the fuzzer itself in the attempt to trigger the bug once again. Without a doubt, this course of action is always worth pursuing, as it can help identifying the source of the error with relatively little effort. At times, however, the attempt to reproduce the unexpected response with this method will fail. This is not surprising, as some errors are a direct result of the state of the fuzzed target. In this case, repeating the fuzzing session greatly enhances the probability of triggering the same issue once again.

3.3.5 Support for Stateful Fuzzing

Because of the reasons we have just discussed, the state of the fuzzed platform is crucial: it may allow a certain input to uncover a crucial bug in the code, and make the difference between a successful fuzzing session and a useless one. In some cases, programmers possess no *a priori* knowledge on the relationships between different inputs, and must rely on the randomness of the fuzzer to uncover them. In others, however, some kind of information about the target's source code, structure, or behavior either is known, or can be safely assumed. As a consequence, LogiFuzz implements a feature that allows programmers to explicitly specify preconditions for each command. Because of the syntax of the tool, this can only be implemented for targets that can be swiftly split in "sub-targets": such an example are Web APIs. Without loss of generality, we will detail the core idea behind this feature by referring to the aforementioned case of Web APIs; the same principles may however be applied to many other kinds of applications.

For each fuzzed endpoint, the programmer may specify its preconditions by including a sorted list of endpoints that are documented in the configuration file. Whenever an endpoint is fuzzed, the list of specified preconditions is checked. By default, each of the specified targets is called one single time before each call to the actual target. In addition, for each of the preconditions, it is possible to specify whether it must be resolved before each call, or just once in the whole fuzzing session. This option provides programmers with more flexibility, allowing for example to avoid useless calls to endpoints that are only used for the purpose of setting up the ones that must be fuzzed. More details on this are provided in section 4.3.1.

3.3.6 Smart Fuzzing and Advanced Protocol Awareness

As mentioned in subsection 3.3.1, LogiFuzz aims at being a highly flexible and general purpose tool, which however relies on the programmer's knowledge of the target to improve its efficiency. Indeed, the user's actions are guided up to a certain degree by the syntax embedded in the fuzzer, as mentioned in subsection 3.3.2. This, however, only covers the *types* and the *names* of the expected parameters. One may therefore wonder how LogiFuzz faces the problem of determining suitable *values* for the fuzzed parameters, especially when dealing with protocols where most combinations of values cannot be considered legal according to the protocol's syntax. It is important to consider that, from this perspective, no support is given to Web API fuzzing. This is largely due to the absence of strict protocol specifications. In this subsection we will therefore refer to fuzzing of protocols such as USB, HID, and HIDPP.

Once again, the solution aims at giving the most amount of flexibility to the programmers: on the one hand, it provides them with the possibility of performing strictly protocol-aware fuzzing; on the other hand, it ultimately offers them the choice, also accepting completely random values for each parameter. In order to attain this, LogiFuzz provides prefilled auxiliary configuration files, containing data on supported commands for USB, HID, and HIDPP. The use of these files

can be easy enabled through the main configuration file, thus providing an easy option for a fully protocol-aware approach. On the other hand, should a programmer be interested in testing the behavior of the target when truly random data is sent, this would be feasible and easy to setup.

Chapter 4

Implementation

In this section, we will give insight on how the design choices that we have detailed in chapter 3 have been implemented in practice. In section 4.1, we will provide an overview of the tools and libraries used, as well as try to explain what has driven us to choose them. In section 4.2 we will dive deeper into the code, detailing the architecture of LogiFuzz. Finally, in section 4.3 we will cover various implementation details that we deem valuable to know. It is important to note that, throughout this chapter, we will mostly aim at giving the reader a high level view of LogiFuzz, explaining and justifying choices that are somewhat relevant to what has been described in chapter 3.

4.1 Technologies and Libraries

When planning the implementation of LogiFuzz, the first and most obvious issue was to decide which programming language to code it in. The choice aimed at finding a language that could support the multi-purpose nature of the tool, by offering a good level of abstraction and providing a large number of utility libraries, especially for relatively uncommon protocols such as HIDPP. In the end, we found C++ to be the programming language that fits these requirements the most: in particular, LogiFuzz is entirely implemented in C++14, because of its compatibility with all of the employed tools.

As mentioned, supporting libraries were one of the most important factors when choosing C++ as the implementation language. Indeed, many external open source libraries have been integrated with the LogiFuzz, in order to support its various features and allow communication with USB devices and web applications. In particular, libhidpp [47] and libusb [22] have been used in order to communicate with USB devices through HIDPP and HID/USB, respectively. To query Web APIs, libcurl [44] was chosen; jsoncpp [35] has been used to prepare the request body of POST requests. Additionally, toml11 [33] has proven to be vital to facilitate the parsing of the

TOML configuration files. Finally, regxstring [3] has been used to generate random strings that could match a specific regex, a feature not present in the native C++ regex library.

4.2 Structure of the Code

LogiFuzz is designed employing an *object-oriented* programming paradigm, with a big focus on *modularity*. Implementation-wise, each of the various fuzzers is represented by a class; first of all, a general-purpose fuzzer is implemented in the class BaseFuzzer, which we will explore in more detail in subsection 4.2.1. Subsequently, all of the protocol-specific fuzzers inherit from BaseFuzzer; we will discuss their features and structure in subsection 4.2.2. Finally, each of the fuzzers is called by a different main program, thus leading to the creation of multiple executables, one per supported protocol. We could therefore say that LogiFuzz is actually the aggregation of various different fuzzers that share the same input structure and mutation logic, as defined in BaseFuzzer. A visual representation of the structure that we just described is available in Figure 4.1.

4.2.1 Base Fuzzer

As the parent to all of the protocol-specific fuzzer classes, BaseFuzzer is meant to represent a general-purpose fuzzer, with the potential to be used for many different applications. Because of its nature, the class focuses on two main tasks: parsing information from a TOML configuration file, and generating or mutating values based on the data acquired in the previous step. As a consequence, BaseFuzzer is the largest and most important class of the project, as it implements the logic behind the generation of test cases. Indeed, the class does not go beyond this scope: once the data has been acquired and fuzzed, BaseFuzzer can only yield it through getter functions.

Am additional design choice for the BaseFuzzer class relates to its implementation as an instantiatable class, rather than an abstract one. Indeed, it would have been an option to introduce a contract by defining a pure virtual method, such as sendData(); each child class would have then been forced to implemented its own version of this method, based on the way the data is actually sent to the target. Choosing this course of action, however, would not have allowed programmers to create an instance of BaseFuzzer, use it to parse and fuzz data, and then retrieve said data to further process it. As a consequence, the current implementation allows the instantiation of BaseFuzzer objects, although they are never actually used in LogiFuzz.

4.2.2 Protocol-Specific Fuzzers

Three additional protocol-specific fuzzers have been implemented as children of BaseFuzzer: HIDPPFuzzer, USBFuzzer, and WebFuzzer. Although their targets are apparently evident by their

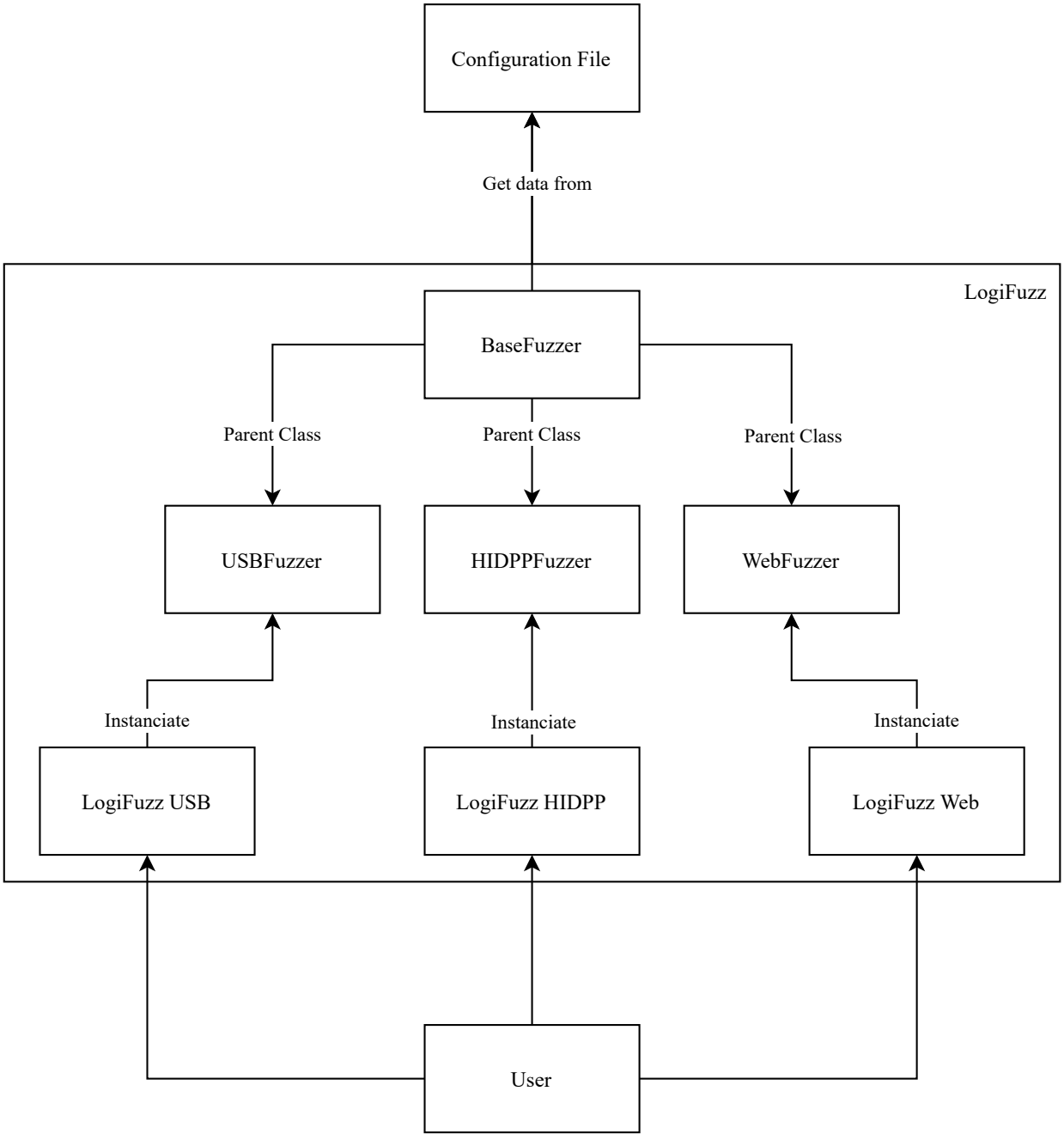

Figure 4.1 – High-level view of LogiFuzz's implementation.

names, it is important to note that USBFuzzer also implements HID fuzzing. The main goal of these fuzzers is to implement target-specific actions, such as opening devices, formatting the output generated through their parent class, and sending it.

4.3 Implementation Details

In this section, we will provide some lower level implementation details concerning LogiFuzz. In subsection 4.3.1 we will discuss the syntax of the configuration file, describe how it relates to the internal representation of the variables, and provide information on other aspects related to the behavior of the fuzzer in relation to the content of the file. In subsection 4.3.2 we will briefly touch on the strategy used in practice to generate new test cases. Finally, in subsection 4.3.3 we will provide some more details on features implemented by the various protocol-specific fuzzers.

4.3.1 Configuration File Structure

Parameters Specification

LogiFuzz uses internally defined conventions to tell apart static parameters and fuzzing parameters, as well as to parse metadata such as data types, restrictions on length for strings, and minimum and maximum values for integers. In the configuration file, simple key-value pairs are used to represent any piece of data that is *not* subject to being fuzzed: for example, the name of the output file and the list of error codes that should be reported fit in this category. On the contrary, variables that *might* be fuzzed are always described in a table named after the variable itself. Then, restrictions and details about the variables are expressed as key-value pairs inside the table.

Type is the only attribute that is mandatory for each fuzzable variable. As mentioned in subsection 3.3.2, integers of fixed size (1, 2, 4, and 8 bytes), strings, arrays, and booleans are the supported types. In the configuration file, integers are named using the WinAPI naming convention [28], and are always implemented by LogiFuzz using the corresponding uintN_t C++ type. This choice allows for . If they represent unsigned in machine, such as in the case of usb fuzzing, signed values are still accepted (though toml accepts also hex, which isp robably mbetter for this). In the case of a signed value, the values will simply loop around zero. For each integer value, it is possible to impose an upper limit u and lower limit l. Note that no condition such that $l < u$ is imposed: as a consequence, choosing $u < l$ allows to create intervals of the type $[l, max] \cup [min, u]$.

An example of a valid configuration file, that dictates the behavior of the HIDPP fuzzer, can be found at Code 4.1.

Code 4.1 – Configuration File for HIDPP fuzzing

```
device_path = "/dev/hidraw1"          # Path to the device
# device_index = 2       # Device index, for unifying
output = "hidpp_out.txt"
highlight_result = [0]
smart = true

# Hidpp 2.0
# feature  = {type = "string", fuzz = "random", value = "Root"}
# function = {type = "byte", min = 0, max = 15, fuzz = "random"}

# Hidpp 1.0
address = {type = "byte", fuzz = "random", value = 255}

# Both versions
parameters = {type = "vector", content_type = "byte", len = 3, fuzz = "random"}

# Path to file that defines available hidpp features (2.0)
features_config = "/home/abianchi/Documents/LogiFuzz/src/libhidpp/hidpp20/features.toml"

# Path to file that defines available hidpp functions for each feature (2.0)
functions_config = "/home/abianchi/Documents/LogiFuzz/src/libhidpp/hidpp20/functions.toml"
```

Statefulness

In subsection 3.3.5 we provided some basic information on the design and motivation behind the introduction of stateful elements in LogiFuzz. In practice, these are currently used to allow the specification of preconditions for the call of endpoints when fuzzing a Web API. Suppose that a list $[e_1, ..., e_n]$ is defined as the precondition for an endpoint e_0. Also, suppose that endpoints e_i, e_j, e_k only have to be called once before each call to e_0. As each endpoint e is defined as a table inside the configuration file, the preconditions for a call to endpoint e_0 are defined in a subtable named $e_0.pre$. Such subtable will then include at most two key-value pairs. One, with key named $endpoints$, will include the sorted list $[e_1, ..., e_n]$, where elements will be called in the order they are defined. The other, with key named $once$, will include the list $[e_i, e_j, e_k]$. Whenever e_0 is fuzzed, the list of preconditions $[e_1, ..., e_n]$ is checked, and all of the specified endpoints are called a single time. Subsequently, the elements present in $once$ are removed from $endpoints$ at runtime. Note that, if an endpoint present in $once$ is called for any other reason before the call of e_0, the precondition will already count as fulfilled.

4.3.2 Test Cases Generation

As we already mentioned in subsection 3.3.3, LogiFuzz may operate in generation-based or mutation-based mode, in a variable by variable basis. In any case, the responsibility of generating

random data is given to the class RandomGenerator, which is defined as an inner class of BaseFuzzer. One RandomGenerator is instantiated by every BaseFuzzer object, and initialized using a seed that is either random (using C++'s random_device) or specified by the user. The object is then used both for generating new test cases and mutating existing ones. In the first case, the implementation is quite straightforward: values are generated randomly, according to the restrictions provided in the configuration file. In the second one, each bit is flipped with a certain $flipping\_probability$ which is either provided by the user, or fixed. The same logic is applied for addition or removal of elements in vectors and strings. It is worth noting that no particular heuristic has been used to determine the standard value of the flipping probability.

4.3.3 Protocol-Specific Fuzzers Peculiarities

While in subsection 4.2.2 we have provided a high level view of the structure of the code for the protocol-specific fuzzers, in this section we will instead provide some low level details concerning their implementation.

USB and HIDPP Fuzzers

In terms of internal structure, the fuzzers targeting USB and HIDPP obviously share a lot of common traits, due to the similar nature of their targeted protocols. First of all, their related configuration files only contain one single main table, with one subtable for each fuzzed variable. For this reason, no preconditions can be established. In addition, they are responsible for checking that the inputs given are in line with the protocols' specifications; as mentioned in subsection 3.3.6, this is implemented, at least in part, through the use of additional configuration files.

Web API Fuzzer

As already mentioned, the Web API fuzzer is the only one that has multiple targets, as well as the only stateful one. At the current state, each endpoint has an equal probability of being chosen at every step of the fuzzing process. Indeed, a user-specified probability for each endpoint would be feasible, and could be implemented. In addition, fuzzing of different endpoints is currently sequential, and does not make use of any parallelism. This mainly has two purposes: one the one hand, it allows for an easier evaluation and execution of preconditions; on the other hand, it avoids overloading Web APIs with a huge number of requests.

Chapter 5

Evaluation

In this section, we will describe the testing process for LogiFuzz. In section 5.1 we will detail the setup we have chosen for the experiments, and attempt to explain the reasoning behind our choice. In particular, we will provide some background information concerning the tested targets, and go over the preliminary steps we have performed to ensure that the testing process could be effective. In section 5.2 we will go into detail on the experiments that have been preformed, and justify them by explaining their purpose. In section 5.3 we will describe the interesting findings that stemmed from the experiments. Finally, in section 5.4 we will discuss some of the limitations of the evaluation process, and offer some insight on how it could have been improved.

5.1 Setup

As mentioned in the previous sections, Logitech's interest was primarily focused on fuzzing the firmware of USB devices, in particular those belonging to the HID class, and Web APIs. In subsection 5.1.1 we will give some insight on which targets have been fuzzed in practice; in subsection 5.1.2 we will go over the process that preceded the experimentation phase.

5.1.1 Tested Devices and Platforms

The testing process of LogiFuzz was fully performed on a local laptop running the operating system Ubuntu 20.04. Indeed, because of the nature of the fuzzed protocols, external devices have been used throughout the testing process.

Fuzzing of the firmware of Logitech devices through USB, HID, and HIDPP has been evaluated using the devices in Figure 5.1. For this purpose, two different setups have been devised:

(a) Logitech Unifying Receiver (b) Logitech K800

Figure 5.1 – Targeted Devices

1. A Logitech Unifying Receiver is plugged to a USB port in the laptop. The receiver is paired with one or more devices. In particular, we will focus on the behavior of the Logitech K800 keyboard. In this setup, fuzzing can either target the receiver or the paired devices;
2. A HID device, such as a mouse or a keyboard, is directly plugged to a USB port and fuzzed. This allows to transfer packets directly to the device, without going through a Unifying Receiver's wireless protocol.

Concerning Web API fuzzing, LogiFuzz has been tested on the Logi ID API. In particular, some interesting and particularly sensitive endpoints have been selected.

5.1.2 Preliminary Steps

Fuzzing of USB-related protocols has proven to be a straightforward process. As the only prerequisite on a Linux machine, we installed the software Solaar [42]. Solaar is a device manager for Linux, specifically designed to introduce support Logitech Unifying Receivers. The tool is not officially supported by Logitech, but it is necessary in order to use Unifying Receivers on Linux. In addition, through Solaar it is possible to check which devices are paired to a connected receiver, as well as which devices are connected.

Web API fuzzing, on the other hand, required a more complex set up. First of all, a private instance of the Logi ID server had to be deployed, in order to allow testing on a dedicated environment. Secondly, the IP address used to perform the fuzzing had to be whitelisted, in order to prevent firewalls from denying suspicious requests. Lastly, endpoint-specific rate limiting mechanisms had to be removed. Despite these precautions, issues still manifested in the fuzzing process. More information will be given in section 5.4.

5.2 Testing

5.2.1 Testing Input Validation

Our first goal when testing LogiFuzz was to quantify our fuzzer's capability of bypassing the input validation of the targets, by comparing the results obtained with LogiFuzz with those yielded by another popular general-purpose fuzzer. For this test, we decided to use AFL as a benchmark tool. AFL is a mutation-based fuzzer, that can operate in either grey box or black box mode. Because of the lack of availability of a source code that could be instrumented, we have decided to employ the black box mode. More information on AFL is available in subsection 6.1.1.

In order to fuzz each of the different targets using AFL, we wrote three different C++ scripts, that send data using the same libraries and functions employed in LogiFuzz. Starting from valid inputs, AFL creates mutations and generates new test cases, that the programs then send to the target. On the other hand, LogiFuzz operates in generation-based mode, fuzzing randomly each of the given variables. We have decided to perform this test for protocol that define specific commands, such as USB and HIDPP. Two modes of LogiFuzz are evaluated: a *smart* one, that is aware of those, and a *dumb* one, which uses purely random fuzzing. We will now discuss the results that we have obtained with each of the targeted protocols.

USB and HID

In our first experiment, we have performed fuzzing of a Logitech Unifying Receiver using HID and USB commands. As a list of defined commands was given for this specific device, we have been able to employ LogiFuzz in both a dumb mode, and a protocol-aware mode. Through dumb fuzzing, we have noticed a peculiar behavior from the targeted device. After correctly accepting commands for a short time, the Unifying Receiver starts ignoring the given input, as each given command results in an *ERROR_IO*. The device must be detached and reattached in order to work properly again. The source of this behavior is unknown, but it is not triggered by any specific command, nor it manifests when smart fuzzing is employed. Regardless, in order to keep the testing process consistent for the same protocol, we have decided to only perform short fuzzing sessions, lasting one minute.

Table 5.1 shows a summary of the results of the tests. Clearly, dumb black box fuzzing is very time consuming, as even a large number of requests usually does not manage to trigger any valid command, and therefore bypass the input validation. In fact, most of the errors generated through this method are of the *ERROR_PIPE* category, which reports that the control transfer could not be interpreted by the target. When the commands defined for the targeted device are known, some transfers are successful.

USB Fuzzing	#Test Cases/Second	Success Rate
AFL	98	0%
LogiFuzz (Protocol-Aware)	117	18%
LogiFuzz (Dumb)	2200	0%

Table 5.1 – AFL vs LogiFuzz Performance in USB and HID Fuzzing

HIDPP

The same experiment described in section 5.2.1 has been repeated for HIDPP. Due to the differences between HIDPP 1.0 and HIDPP 2.0, two different experiments have been run, with the purpose of properly evaluating how the behavior of the fuzzer changes with the two versions of the protocol. Fuzzing of HIDPP 1.0 has been performed on the Logitech Unifying Receiver; fuzzing of HIDPP 2.0 has been performed on the Logitech K800 keyboard. Once again, we have been able to employ LogiFuzz in both a dumb mode, and a protocol-aware mode. For HIDPP 1.0, all of the available information on the target can be directly acquired through the libhidpp library. For HIDPP 2.0, a set of valid commands has been provided; this, however, describes all of the defined command, including those not implemented by the specific fuzzed device. As opposed to the USB case, HIDPP commands did not cause unexpected responses from the device, and longer testing sessions could be performed.

The results of the testing process for both protocols can be observed in Table 5.2. In this case, smart fuzzing proves to be even more effective, with a success rate of 100% in HIDPP 1.0. The lower success rate of HIDPP 2.0 can be associated with the presence of unsupported features in the configuration file; a longer fuzzing session, paired with the learning capabilities from LogiFuzz, would lead to an increase in the number of successful calls.

5.2.2 Fuzzing the Targets

As described in subsection 5.2.1, our first purpose was to evaluate whether LogiFuzz could outperform a black box fuzzer in the generation of valid test cases. Indeed, the yielded results show that it is much easier to overcome the input validation phase using LogiFuzz. The subsequent step in the testing process consisted therefore in trying to uncover vulnerabilities in our targets through dedicated fuzzing sessions.

The fuzzing process was divided in three main steps, that are summarized in Table 5.3. In a first phase, all the targets were tested without leveraging on any known information on the target, except for the format of the expected input, and the type of the fuzzed variable. This part of the process aimed at finding surface-level bugs, concerning for example the targets' parser. In a second step, we configured the fuzzer to take into account additional information

HIDPP 1.0 Fuzzing	#Test Cases/Second	Success Rate
AFL	64	0%
LogiFuzz (Protocol-Aware)	41	100%
LogiFuzz (Dumb)	885	∼0%

HIDPP 2.0 Fuzzing	#Test Cases/Second	Success Rate
AFL	27	6%
LogiFuzz (Protocol-Aware)	32	64%
LogiFuzz (Dumb)	31	11%

Table 5.2 – AFL vs LogiFuzz Performance in HIDPP 1.0 and HIDPP 2.0 Fuzzing

about the targets. For USB, HID, and HIDPP, this was done automatically, providing an auxiliary configuration file containing valid commands; for Web API fuzzing, the parameters of LogiFuzz were tuned manually, according to the information provided in the documentation. Finally, we attempted to exploit the statefulness of the Web API fuzzing in order to target vulnerabilities that could not be detected through crashes or timeouts. In particular, we attempted to perform privilege escalation attacks, trying to gain access to locked resources.

5.3 Findings

Throughout its testing process, LogiFuzz has managed to find a significant bug concerning the Logitech Unifying Receiver when paired with the Logitech K800 keyboard. In particular, the issue has been found through HIDPP fuzzing, and allows malicious users to mount ghost device attacks.

The attack is set up by plugging a Logitech Unifying Receiver, and pairing it with a Logitech K800 keyboard through the Logitech Unifying protocol. By issuing a specific command to the receiver, it is possible to reset it; when this happens, the receiver is in practice disconnected, and must be detached and reattached in order to use the paired devices. As soon as the receiver is reattached, all of the connected devices start working again. At this point, however, the behavior differs depending on the platform:

- On Linux systems, Solaar will behave in one of the following two ways: (i) display the K800 keyboard as a paired but disconnected device, even though the keyboard is connected and used; or (ii) not display the K800 keyboard at all among the paired devices, even though it is connected and used. It is still not clear what causes the difference in behavior.
- On Windows systems, Logitech Options (Logitech's Device Manager) will not display the keyboard until it is used. After this happens, the keyboard is correctly displayed as both paired and connected.

Indeed, the attack turns out to be much more powerful on a Linux machine, as it allows to write with a device that is seemingly not paired.

	Dumb Fuzzing	Smart Fuzzing	Targeted Specific Vulnerability
USB/HID	✓	✓	X
HIDPP	✓	✓	X
Web API	✓	✓	✓

Table 5.3 – Employed Fuzzing Strategies

The origin of the vulnerability is currently unknown; the cause of the issue may lie in Solaar, which is however not officially supported by Logitech. Because of its lack of official Linux support, Logitech has decided not to further investigate the issue.

5.4 Limitations

While the testing process has allowed us to find an interesting attack vector, we were surprisingly unable to identify other kinds of errors. In this section, we will try to explain some of the possible reasons.

First and foremost, LogiFuzz is designed to be used by experts, who possess in-depth knowledge of the tools that are being fuzzed. Throughout the testing process, however, no insight was received by external expert, and fuzzing was performed without any knowledge of the source code from the programmer. As a consequence, LogiFuzz's main benefit, that is allowing targeted, customizable fuzzing, was not at all taken advantage of.

In addition, the targeted products are all mature, and have already been tested and deployed a long time ago. As a consequence, many bugs were already found and fixed. Thanks to its flexibility, LogiFuzz could be used to exclusively fuzz newly deployed endpoints and features, thus focusing on untested parts of the code.

Finally, fuzzing of Web APIs encountered issues stemming from rate-limiting, up until the very end. As a consequence, certain endpoints could not be effectively fuzzed, as all the traffic would be rejected after a threshold of requests was met.

Chapter 6

Related Work

Fuzz testing is a hot topic in academic research, and new state-of-the-art fuzzers are constantly developed. In this section, we will discuss some of the fuzzing tools that closely relate to LogiFuzz, analyse their scope, and point out the differences in design choices and available features. When possible, we will also attempt to suggest possible improvements to LogiFuzz, based on ideas and intuitions found in related work. In section 6.1, we will briefly touch on other general purpose fuzzers; in section 6.2 we will discuss solutions to the USB fuzzing problem; in **??** we will analyse modern and efficient fuzzers for Web APIs.

6.1 General Purpose Fuzzers

6.1.1 AFL

AFL (American Fuzzy Lop) is an extremely popular plug-and-play grey box fuzzer, that quickly generates test cases in a mutation-based fashion. The tool can also work in binary mode, instrumenting black box binaries on-the-fly [12]. Because of its design, AFL has proven to be very effective both on its own [54] and when extended by other fuzzing tools such as AFL++ and USBFuzz [6, 39],

As mentioned in subsection 5.2.1, AFL has been used as a baseline for the testing of LogiFuzz against a dumb, but fast and easy to setup fuzzer. Indeed, it is worth pointing out that the dumbness depends on the impossibility to instrument the source code, thus nullifying AFL's coverage-guided fuzzing capabilities. Regardless, AFL only offers basic support for definition of protocol-specific grammar [53], and an integration would be worth considering only if a source code was available to instrument.

6.1.2 Peach, MozPeach, and GitLab Protocol Fuzzer

MozPeach is a fork maintained by Mozilla of the generic fuzzing framework Peach (v2.7) [43]. Peach's behavior is programmed through the use of XML configuration files, named "pits" in the context. Users can use pits in order to define the format of the generated test cases, as well as to establish how it should monitor for errors. Compared to LogiFuzz, the purely general purpose nature of MozPeach allows to target a wider variety of programs. However, this comes at the cost of a reduced readability for the configuration file, as well as lack of feedback evaluation.

GitLab Protocol Fuzzer Community Edition is a recently open-sourced generic fuzzing framework, based on the pre-existing Peach Fuzzer Professional v4 [8]. The project is still at a very early stage at the time of writing. Compared to MozPeach, the compilation process is much longer and more complex, and compilation issues have been reported on Linux systems.

6.2 USB Fuzzers

6.2.1 Frisbee Lite

Frisbee Lite is a GUI-based black box fuzzer that targets USB-devices. The tool is designed to run on Windows platforms, and performs dumb fuzzing over any kind of USB device by sending a large number of USB requests [13]. Similarly to LogiFuzz, Frisbee Lite allows users to manually set allowed intervals for the fuzzed variables, and does so through an intuitive graphic interface. Apart from the GUI, LogiFuzz offers the same functionalities as Frisbee Lite, and it slightly improves on these by adding a mutation-based fuzzing option, as well as allowing users to load values from an external file. In addition, Frisbee Lite does not offer any possibility to highlight specific results and error codes.

6.2.2 USBFuzz

USBFuzz is a fuzzer for USB drivers in various operating systems, including Linux, Windows, Mac OS, and FreeBSD [39]. Built as an open-source tool that extends the popular AFL [16], USBFuzz is a flexible fuzzing framework, capable of both dumb and (in the case of the Linux kernel) coverage-guided fuzzing, and has proven to be very effective, finding tens of bugs in the kernels of the aforementioned operating systems.

From the aforementioned description, it is easy to see how the fundamental scope of USBFuzz differs from LogiFuzz's. USBFuzz targets bugs in the USB *Drivers* in the *host*, while LogiFuzz's USB component focuses on the *firmware* of USB *devices*. For this reason, the two fuzzers somewhat complement one another, as they target two different components that may open to similar

classes of bugs through their vulnerabilities. Nonetheless, there are elements of USBFuzz's approach that may be used in order to improve the efficiency of LogiFuzz. In particular, USBFuzz uses software-based emulation to emulate USB devices, that generate the fuzzing input fed to the kernel. This strategy could potentially be employed by LogiFuzz, with the advantage of being able to access the source code of the firmware of the emulated device, thus allowing white box or grey box fuzzing.

6.2.3 Other USB drivers fuzzers

USBFuzz is not the only fuzzer that targets USB drivers in operating systems; many other tools, such as umap2fuzz [14] and vUSBf [**vUSBf**], share the same target. Regardless, the aforementioned fuzzers are proven not to be as efficient as USBFuzz. For instance, umap2fuzz does not offer any option for coverage-guided fuzzing; vUSBf can only perform dumb fuzzing.

6.3 Web API Fuzzers

6.3.1 RESTler

RESTler is a restful Web API black box fuzzer, that leverages on well maintained OpenAPI/Swagger specifications to generate test cases its targets. In its input-generation process, RESTler heavily relies on its statefulness to evaluate the relationship between different requests, and pairs this feature with a feedback evaluation process in order to greatly reduce the input space and improve code coverage [1, 27]. In practice, the fuzzer has proven to be very effective, as it has managed to uncover several bugs in platform such as GitLab, Office365, and Azure.

RESTler can undoubtedly be considered the state-of-the-art for fuzzing of Web APIs. Compared to LogiFuzz, its approach relies more heavily on a well-written documentation in a specific format. Indeed, if this is present, RESTler manages to smartly identify preconditions without the user's intervention, and can make use of a smarter logic to infer dependencies and evaluate feedback. In addition, RESTler allows to define *checkers*: these are simply additional user-defined conditions, that are used to specify additional errors that the fuzzer should look for, other than the obvious $5xx$ responses. This is a less straightforward but more precise system, compared LogiFuzz's definition of relevant response codes.

Given RESTler's high efficiency and automation for well-documented APIs, an integration with LogiFuzz might be an idea worth considering. In such a case, users would be able to select the mode of use based on the type and quality of their documentation, and could fully appreciate both fuzzers' advantages. In alternative, LogiFuzz could simply implement a feature that allows users to automatically compile a configuration file, based on Swagger/OpenAPI specifications. On the one hand, this would keep the input format consistent, thus allowing users to customize

input ranges and formats in a way that RESTler does not allow. On the other hand, doing this would prevent the tool from taking advantage of RESTler's powerful feedback evaluation and checkers-definition mechanisms.

6.3.2 Other Web API fuzzers

Compared to USB-related protocols, Web APIs are much more popular targets for fuzzing. As a consequence, a large amount of tools exist for such purpose. Similarly to RESTler, ApiFuzzer allows to parse Swagger and OpenAPI specifications and fuzz the documented APIs [19]. BooFuzz [40], an actively maintained fork of the no longer supported Sulley [36], is a well-documented tool that employs user-defined rules to perform fuzzing of Web APIs.

Chapter 7

Conclusion

Fuzzing is an ever-evolving technique, that is consistently used to uncover security vulnerabilities and prevent dangerous attacks from being designed and executed. For this reason, Logitech has decided to employ this technique in order to test its USB devices, as well as its web applications. While many widespread fuzzers could be used for this purpose, a need for increased flexibility and control from the programmer's part led us to design a new solution. We propose LogiFuzz, a generation-based black box fuzzer, that currently supports protocol-aware fuzzing of USB, HID, HID++, and Web APIs. Thanks to the use of a configuration file, LogiFuzz allows programmers to define constraints for the fuzzed variables, and target very specific features and functions that need to be tested.

Due to its reliance on user expertise to circumvent the natural limitations of black box fuzzing, LogiFuzz is meant to become a central tool in Logitech's production pipeline; thanks to the simple and well documented interface, LogiFuzz will be used by engineers to perform testing focused on new features. Finally, thanks to its modular architecture, LogiFuzz can be easily extended to support additional protocol: this way, the tool could be used to detect surface-level vulnerabilities and test newly implemented or reworked elements over most of the software developed by Logitech.

Bibliography

[1] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. “RESTler: Stateful REST API Fuzzing”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 748–758.

[2] Bastille. *MouseJack*. URL: https://www.mousejack.com/mousejack (visited on 07/13/2021).

[3] daidodo. *regxstring*. https://github.com/daidodo/regxstring.

[4] Timon Eßlinger, Annika Maier, and Frank Niemann. “Fuzzing USB Hosts”. In: Apr. 2020.

[5] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minghui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. *Snipuzz: Black-box Fuzzing of IoT Firmware via Message Snippet Inference*. May 2021.

[6] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. “AFL++ : Combining Incremental Steps of Fuzzing Research”. In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[7] Justin Forrester and Barton Miller. “An Empirical Study of the Robustness of Windows NT Applications Using Random Testing”. In: *4th USENIX Windows Systems Symposium (4th USENIX Windows Systems Symposium)*. Seattle, WA: USENIX Association, Aug. 2000. URL: https://www.usenix.org/conference/4th-usenix-windows-systems-symposium/empirical-study-robustness-windows-nt-applications.

[8] GitLab. *GitLab Protocol Fuzzer Community Edition*. https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[9] GitLab. *Web API Fuzz Testing*. URL: https://docs.gitlab.com/ee/user/application_security/api_fuzzing/ (visited on 07/19/2021).

[10] Patrice Godefroid, Michael Levin, and David Molnar. “Automated Whitebox Fuzz Testing”. In: Jan. 2008.

[11] Patrice Godefroid, Michael Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing”. In: 2012.

[12] Google. *AFL*. https://github.com/google/AFL.

[13] NCC Group. *Frisbee Lite*. https://github.com/nccgroup/FrisbeeLite.

[14] NCC Group. *Umap2*. https://github.com/nccgroup/umap2.

[15] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. "Seed Selection for Successful Fuzzing". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 230–243. ISBN: 9781450384599. URL: https://doi.org/10.1145/3460319.3464795.

[16] HexHive. *USBFuzz*. https://github.com/HexHive/USBFuzz.

[17] Mark Hopkins, David Coulter, Bill Lattimer, and Elliot Graff. *Introduction to Human Interface Devices (HID)*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/hid/ (visited on 08/12/2021).

[18] Keil. *USB Communication*. URL: https://www.keil.com/pack/doc/mw/USB/html/_u_s_b__endpoints.html (visited on 07/29/2021).

[19] KissPeter. *APIFuzzer*. https://github.com/KissPeter/APIFuzzer.

[20] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: vol. 1. Dec. 2018.

[21] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* (2018).

[22] libusb. *libusb*. https://github.com/libusb/libusb.

[23] Logitech. *Logitech HIDPP 1.0 specification for Unifying Receivers*. URL: https://lekensteyn.nl/files/logitech/logitech_hidpp10_specification_for_Unifying_Receivers.pdf (visited on 08/08/2021).

[24] Logitech. *USB Unifying Receiver*. URL: https://www.logitech.com/en-ch/products/mice/unifying-receiver-usb.910-005236.html (visited on 08/12/2021).

[25] Bill Marczak, John Scott-Railton, Noura Al-Jizawi, Siena Anstis, and Ron Deibert. *The Great iPwn: Journalists Hacked with Suspected NSO Group iMessage 'Zero-Click' Exploit*. URL: https://citizenlab.ca/2020/12/the-great-ipwn-journalists-hacked-with-suspected-nso-group-imessage-zero-click-exploit/ (visited on 08/12/2021).

[26] Steve McConnell. *Code Complete*. Microsoft Press, 2004 (2nd ed.)

[27] Microsoft. *RESTler*. https://github.com/microsoft/restler-fuzzer.

[28] Microsoft. *Windows Data Types*. URL: https://docs.microsoft.com/en-us/windows/win32/winprog/windows-data-types (visited on 07/22/2021).

[29] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. "An Empirical Study of the Robustness of MacOS Applications Using Random Testing". In: *Proceedings of the 1st International Workshop on Random Testing*. RT '06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 46–54. ISBN: 159593457X. URL: https://doi.org/10.1145/1145735.1145743.

[30] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. “An Empirical Study of the Robustness of MacOS Applications Using Random Testing”. In: *Proceedings of the 1st International Workshop on Random Testing*. RT ’06. New York, NY, USA: Association for Computing Machinery, 2006, pp. 46–54. ISBN: 159593457X. URL: https://doi.org/10.1145/1145735.1145743.

[31] Barton P. Miller, Louis Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: 1990.

[32] Charlie Miller and Zachary N. J. Peterson. “Analysis of Mutation and Generation-Based Fuzzing”. In: Independent Security Evaluators, 2007.

[33] Toru Niina. *toml11*. https://github.com/ToruNiina/toml11.

[34] CSO Online. *The 15 biggest data breaches of the 21st century*. URL: https://www.csoonline.com/article/2130877/the-biggest-data-breaches-of-the-21st-century.html (visited on 07/13/2021).

[35] open-source-parsers. *jsoncpp*. https://github.com/open-source-parsers/jsoncpp.

[36] OpenRCE. *Sulley*. https://github.com/OpenRCE/sulley.

[37] Lars Opstad. *Using Code Coverage to Improve Fuzzing Results*. URL: https://msrc-blog.microsoft.com/2010/02/24/using-code-coverage-to-improve-fuzzing-results/ (visited on 08/08/2021).

[38] OWASP. *Fuzzing*. URL: https://owasp.org/www-community/Fuzzing# (visited on 07/01/2021).

[39] Hui Peng and Mathias Payer. “USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation”. In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2559–2575. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/peng.

[40] Joshua Pereyda. *boofuzz*. https://github.com/jtpereyda/boofuzz.

[41] V. Pham, M. Boehme, A. Santosa, A. Caciulescu, and A. Roychoudhury. “Smart Greybox Fuzzing”. In: 01. Los Alamitos, CA, USA: IEEE Computer Society, Sept. 5555, pp. 1–1.

[42] pwr-Solaar. *Solaar*. https://github.com/pwr-Solaar/Solaar.

[43] Mozilla Security. *MozPeach*. https://github.com/MozillaSecurity/peach.

[44] Daniel Stenberg. *libcurl*. URL: https://curl.se/libcurl/ (visited on 08/02/2021).

[45] toml-lang. *TOML*. https://github.com/toml-lang/toml.

[46] USB. *Defined Class Codes*. URL: https://www.usb.org/defined-class-codes (visited on 08/12/2021).

[47] Clément Vuchener. *HIDPP*. https://github.com/cvuchener/hidpp.

[48] wcventure. *FuzzingPaper*. URL: https://wcventure.github.io/FuzzingPaper/ (visited on 08/04/2021).

[49] Wikipedia. *Firmware*. URL: https://en.wikipedia.org/wiki/Firmware (visited on 08/12/2021).

[50] Wikipedia. *Fuzzing*. URL: https://en.wikipedia.org/wiki/Fuzzing (visited on 07/28/2021).

[51] Wikipedia. *USB*. URL: https://en.wikipedia.org/wiki/USB (visited on 07/29/2021).

[52] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Designing New Operating Primitives to Improve Fuzzing Performance". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 2313–2328. ISBN: 9781450349468. URL: https://doi.org/10.1145/3133956.3134046.

[53] Michał Zalewski. *afl-fuzz: making up grammar with a dictionary in hand*. URL: http://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html (visited on 08/12/2021).

[54] Michał Zalewski. *American Fuzzy Lop*. URL: https://lcamtuf.coredump.cx/afl/ (visited on 07/13/2021).

[55] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. "Greybox Fuzzing". In: *The Fuzzing Book*. Retrieved 2020-11-23 12:48:59+01:00. CISPA Helmholtz Center for Information Security, 2020. URL: https://www.fuzzingbook.org/html/GreyboxFuzzer.html.