École Polytechnique Fédérale de Lausanne

Semester project on Fuzzing Trusted Execution Environments on COTS Android Devices

by Leonardo Pennino

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Project Advisor

Marcel Busch
Project Supervisor

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

January 6, 2023

Acknowledgments

I would like to thank the whole HEXHIVE group for supporting and guiding me for this project. In particular, I would like to thank Dr. Marcel Busch, for which I have the utmost admiral, for sharing his research and deep knowledge of the field with me. I would also like to thank Prof. Dr. sc. ETH Mathias Payer for his Software Security class at EPFL which has been of tremendous help for my improvement in the field. This report was written using HexHive Thesis Template.

Lausanne, January 6, 2023 Leonardo Pennino

Abstract

The TEEzz tool designed by Dr. Marcel Busch enables effective fuzzing of Trusted Environment by an automatic infer of the data and value dependency obtained by looking at the interactions of the TA. The goal of the project is to continue the implementation by building and all the required parts for the Fuzzer to work. The main work was focused on fixing and improving the existing Client Application Library Identification (CAID) tool by enabling multithreaded computation, creating drivers for Trusted Application in order to trigger interactions and setting up a working recorder for a test device able to generate seeds for.

Contents

Chapter 1

Introduction

This project's aim is to continue the development of the TEEzz tool via rewriting a part of the source code to give it a better structure a software engineering point of view, adding more test cases and supported libraries, improving performance and making the infrastructure testable and reusable by other researchers.

TEEzz-CAID The tool TEEzz-CAID which originally was working only on Android Devices of some specific vendors was rewritten to support any vendor by building and injecting missing binaries on the phone which are essential for the tool, furthermore the program was sped up by dividing its computation in multiple threads, and restructured for easier understanding from other researchers.

Writing drivers for closed sourced client applications The repository contains an example driver for Mlipay, which is a Xiaomi library that handles payments on smartphones. The driver is currently able to call one function of the TA.

Writing drivers using Java Reflection The project contains drivers for "Gatekeeper" and "Keystore" which using android binder mechanisms, allow us to call into the Client Application by using Java Methods. All functionalities of Keystore and Gatekeeper were implemented in the driver.

Seed Recording The seed recording was made into a docker container able to automatically download the android code, compile required libraries, and set up all the required objects for the recorder to work with. In particular, it was automated the generation of header files from Android's HIDL files and the js code required by frida for the test device *Hikey620*.

Chapter 2

Background

Explain how android Binder works, how trusted applications communicate and the goal of teezz Communications with the main ta library are usually handled with a central library (as libteec.so).

Chapter 3

Library Identification

The first goal of the project is to provide a mean to detect which Android Applications or libraries are accessing the Trusted Execution Enviroment. To achieve this goal the tool "teezz-caid" is able to scan an android device and look for every app or library that is directly or indirectly calling communicating with the TA. The program requires as input the *main* CA library for that specific device e.g.:*libteec.so*. The program will then build a graph where on its vertexes we find all the libraries and consumers that eventually call into the ta which is the root node. The process is divided in three parts: Downloading binaries, Disassembling and Finding Dependencies.

3.1 Downloading binaries

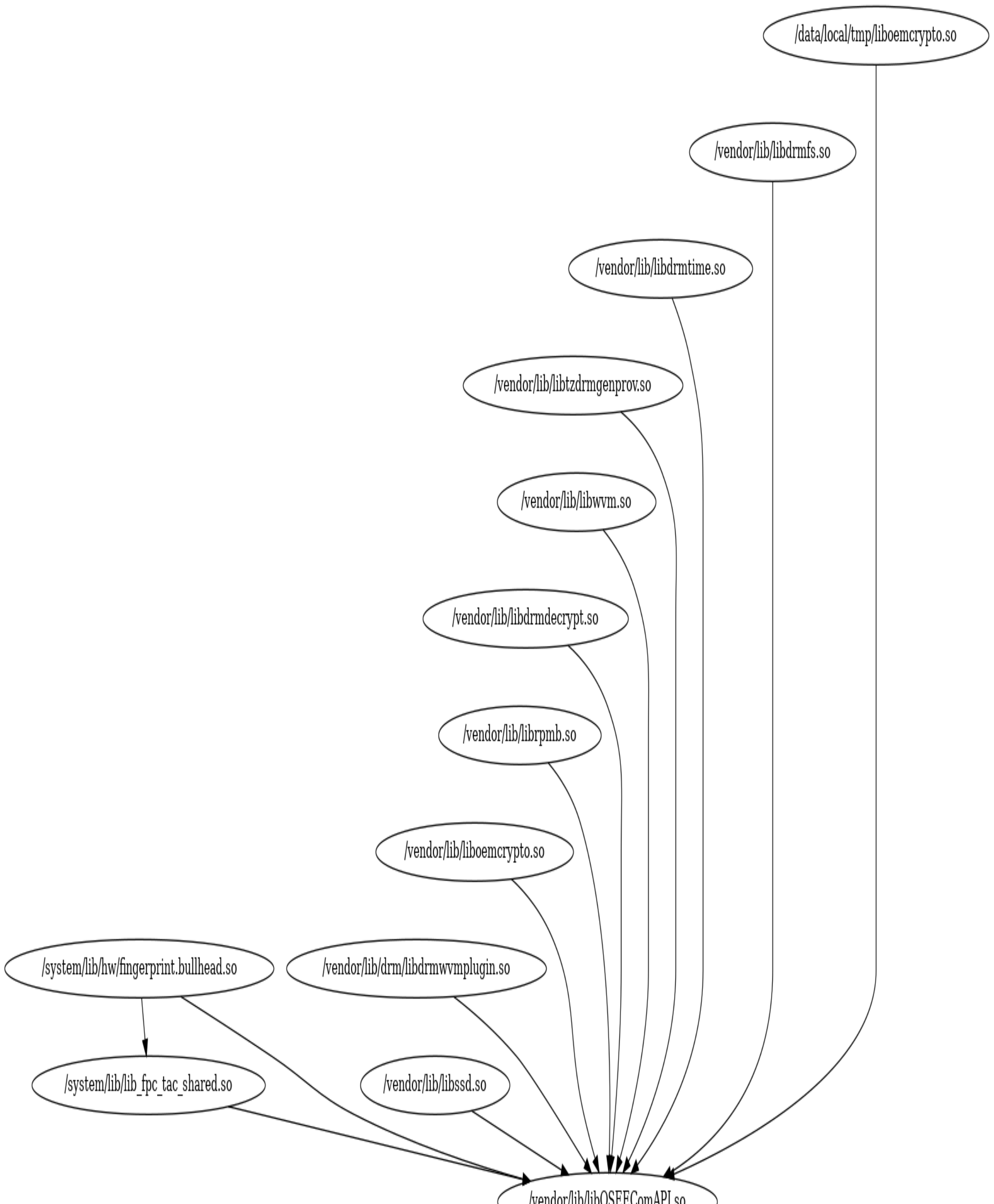
The first part of the process relies in getting all executables, libraries and VDexs from the device. In order to do so we spawn a shell and we execute *file* for each one of them. The first problem faced was that many vendors do not include all required *unix binaries* which are needed by the program such as *file* or *find*. To solve this problem we have statically compiled those utilities that will be injected into the device where needed. Another issue is that pulling large amounts of files from android phones takes time, to solve this we divide the pulling in more threads to speed up the process.

3.2 Disassembling

Once the program downloads the required files, we have two different categories of files and each will be treated differently: **ELF files**, and **VDexs**. For ELF files *readelf* is used to get all the libraries needed by the executable. It will also list libraries loaded with *dlopen* by specifically looking for the *dlopen* or *hw _get_module* symbols. *VDexs* files instead are firstly extracted using vdexExtractor, then decompiled using jadx. Once that is done a process similar to ELFs is followed, in particular we look for *System.LoadLibrary* to check which libraries it is using.

3.3 Finding Dependencies

At the end of the before mentioned process, a list of Vdexs and Elfs is obtained, each one with their own dependencies. The program starts by putting the main CA library given as input on a stack, then it will scan the built list for every file that has that library as a dependency. Every match is put on the stack and the previous item is popped, then recursively repeats the process until no items remain on the stack. At the end we obtain a graph with all the found libraries and applications. Below is an image of a graph obtained for the Nexus 5X.


/data/local/tmp/liboemcrypto.so
/vendor/lib/libdrmfs.so
/vendor/lib/libdrmtime.so
/vendor/lib/libtzdrmgenprov.so
/vendor/lib/libwvm.so
/vendor/lib/libdrmdecrypt.so
/vendor/lib/librpmb.so
/vendor/lib/liboemcrypto.so
/system/lib/hw/fingerprint.bullhead.so
/vendor/lib/drm/libdrmwvmplugin.so
/system/lib/lib_fpc_tac_shared.so
/vendor/lib/libssd.so
/vendor/lib/libQSEEComAPI.so

Chapter 4

Triggering interactions with the TA: Drivers

After the discovery of the targets, before talking about the fuzzing we have to first find a way for triggering TA interactions that TEEzz relies on for the data and state infering. Interaction with the TAs can be achieved in two ways: By manual interaction with the device, or automatically via calling a function of a Client Application.

4.1 Manual Interaction

Manual interactions with the Trusted Execution Environment happen when we change things that should be handled by the TEE such as a lockscreen code on an android phone, gatekeeper will trigger a request to the ta in order to submit and approve the changes. Doing this effort manually each time requires a lot of time and makes our fuzzing very inefficient. The first approach was to emulate touches on the android phone using a program that I designed to trigger the touches as fast as possible. This solution was fast to implement and correctly working however it took around 3 minutes for some specific interactions. Certain android devices such as Huawei in order to remove the lockscreen code, needed to trigger a particular interaction in the *Gatekeeper*, require 5 wrong tries, with 30 seconds delay between each other, thus the whole process took 3 minutes each time.

4.2 Interactions using Java Reflection

Seeing these downfalls, we opted for another path. We found a way to trigger interactions using Java Reflection. By exploiting the way Android Binder Mechanism and Android HIDL Java, we were able to call native libraries code via Java. We first build a java program which hooks into Android Java classes such as *Gatekeeper* via **Class.forName**, then we can trigger methods or create objects. After having built our java file we **dex** it and inject into our test device and run it via *app_process*. This approach does not have the drawbacks of the one discussed before, however not every Client Application has a java callable interface, hence why this approach cannot be used in all scenarios.

Listing 4.1: "Example driver for Keystore"

```
public KeystoreClient() {
        Class IKeystoreService = Class.forName(
        "android.security.IKeystoreService");
        Class stub = IKeystoreService.getDeclaredClasses()[0];
        Method mAsInterface = stub.getDeclaredMethods()[0];
        //Final object able to call binder
        oKeystoreService = mAsInterface.invoke(null, getKeystoreBinder());
        //now we can access the methods
        mReset = oKeystoreService.getClass().getDeclaredMethod("reset");
        mGet = oKeystoreService.getClass().getDeclaredMethod("get", String.class, i
        // we can call methods like this
        mGet.invoke(oKeystoreService,...params);
    }
```

4.3 Interaction with custom C/C++ drivers

For example Xiaomi's *mlipay* library which is used for payments, due to it not having a Java HIDL interface, requires the developing of a C++ driver which is able to construct the MLIPAY object and call into its functions. In order to build such a driver, first we have to reverse engineer the library to understand its functionalities and how it works, then we create an handle and attach it using *dlopen*. We have only managed to write a driver for one of MLIPAY's functions as it is very time consuming to reverse ARM64 android C++ applications.

Listing 4.2: Example driver for mlipay

```
int main(int argc, char **argv) {
  void *handle = dlopen("libmlipay.so", RTLD_NOW | RTLD_GLOBAL);
  if (handle == NULL) {
    printf("Error␣opening␣handle␣to␣libmlipay.so");
    return -1;
  }
  void *(*fn)(void) = NULL;
  void *(*constructor)(void) = NULL;
  void *(*get_key_version)(void) = NULL;
```

```
    printf("Created␣handle␣\n");
    *(void **)(&fn) = dlsym(handle, "HIDL_FETCH_IMlipayService");
    if (fn != NULL) {
      printf("Calling␣fn\n");
      void **obj = (void **)(*fn)();
      void **vtable = (void **)*obj;
      *(void **)(&constructor) = (void **)*(vtable);
      *(void **)(&get_key_version) = (void **)*(vtable + 0xe);
      //The address above was obtained via reverse engineering
      (*get_key_version)(); // Calling our target function
    }
  }
```

Chapter 5

Seed Recording

The last part of the project focused on improving and making the seed recorder portable and easily reproducible and adaptable to various devices. The seed recorder is the part of the project that is responsible to create type aware seeds for the fuzzer. The recording works at different abstraction levels: **IOCTL** and the **Hardware Abstraction Layer**. It uses Frida to dinamycally instrument code on the target device to execute arbitrary code allowing the inspection and dumping of the memory. IOCTL is the lowest abstraction layer, it contains no type definitions, only raw bytes are sent. For this layer we just need to hook into the *ioctl* system call and dump the data sent. The HAL layer is just above the IOCTL and this contains struct definitions, types and value dependencies. The seeds that need to be generated from HAL and IOCTL are very different. More work is needed for the HAL to correctly identify the data that the application passes to the TEE. The seed recorder consists of the following parts: the **Interceptor**, **Generator**, **FridaDumper**, **HalDumper** and **DualRecorder**.

5.1 Interceptor

The role of the interceptor is to attach the frida debugger to the function calls we want to look into. The interceptor takes as parameter a json file describing the interface's functions and how to locate them, either via symbols if the binary is non stripped or by offset. It can either attach by symbol name or function offset. After attaching to a function it automatically dumps the incoming and outgoing parameters.

5.2 Gendumper

The gendumper takes as input a C/C++ file and a struct or class and dumps the js code which will be then used by frida, with type and state awareness. In order to correctly parse the input file it makes use of clang to build an AST, which it explores to find the definition for each type and function of the input symbol. To work as expected, it requires all C / C++ headers that the input file uses. To make the software portable and readaptable, the download and compilation of required libraries has to be automated. The docker container included in the seed recorder contains *Makefiles* for different devices that automatically download and compile the needed libraries from the AOSP and generate the header files required by the gendumper.

5.3 FridaDumper

The FridaDumper instruments the Client Application to hook into *ioctl* function calls which then dumps the passed bytes.

5.4 HalDumper

The HalDumper takes as input the javascript files generated by the GenDumper and instruments the HAL interface to generate seeds at runtime.

5.5 Dual Recording

The dual record is the final step of our project, it combines both the haldumper and the ioctl dumper into one unique program which is able to generate both seeds that will then be used by the fuzzer.

Chapter 6

Challenges

Chapter 7

Conclusion

In the conclusion you repeat the main result and finalize the discussion of your project. Mention the core results and why as well as how your system advances the status quo.