École Polytechnique Fédérale de Lausanne

Xiaomi TEE Beanpod Emulator

by Li Shi

Master Project Report

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

June 4, 2024

Abstract

The increasing reliance on Trusted Execution Environments (TEEs) and Trusted Applications (TAs) in modern computing necessitates comprehensive research into their security. TEEs, such as Intel SGX and ARM TrustZone, provide isolated environments intended to secure sensitive computations and data from potentially compromised operating systems and applications. However, the growing complexity and ubiquity of these technologies introduce novel attack vectors that can undermine their integrity. Similarly, TAs, which run within these TEEs, are critical components for ensuring secure operations, yet they also face significant threats from sophisticated attacks targeting their execution and data.

Due to the inaccessibility to the inner world of TEEs, current research works on emulating the TEE OS infrastructure so that TAs can run on the emulated system. We focus on the Xiaomi Beanpod TEE, rewrite all the library functions in TAs and develop a lightweight function-level TA emulator. With this emulator, we analyse and debug 18 TAs and find 19 unique bugs in them. Among these bugs, eight bugs were previously unknown. Also, we develop an exploit with one of the bugs and achieve arbitrary code execution in the TA context.

Contents

Chapter 1

Introduction

Nowadays, smartphone applications use Trusted Execution Environments (TEEs) to protect critical operations from malicious code. TEEs are designed with privileged access to the entire system and have their own dedicated regions that are inaccessible to the rest of the system. In the threat model of TEEs, they remain secure even if the entire system is compromised. This ensures the integrity and confidentiality of Trusted Applications (TAs) against various application- and kernel-level exploits [2], [10], [16]. As a result, smartphones can securely process private user data and perform sensitive tasks such as financial transactions [5], user authentication [14], and handling DRM-protected media [4].

The security of Trusted Applications is crucial for maintaining data integrity, system integrity, user trust, and compliance with regulations. Implementing robust security measures is essential to mitigate risks and safeguard the functionality and security of these applications within a system. Vulnerabilities in TAs could result in unexpected behaviour in TEEs, attackers can even compromise the whole TEEs with chains of bugs, leaking fingerprint information and forging online payment etc. [12].

Real-world TEEs enforce strict format and protocol interactions when communicating with TAs, which reduces the bugs and thwarts exploit to some degree. Also, TEEs are designed to expose little insight into their inner working, users can only interact with TAs over this well-defined interface. Due to the little exposure of how the system works in TEEs, both attackers and defenders research TEE security mostly by static analysis of TAs and speculating about inner behaviours by debug log. At the time of writing, tools that enable researchers to dynamically analyse TAs have great limitations, they either need hardware debugger-enabled phones or have not been released. This drives researchers to design emulators to gain introspection capabilities in TEEs.

We propose an approach to dynamically analyse TAs by implementing an easy-to-use user-level TA emulator with function-level hooking. TAs are designed according to the GlobalPlatform standards and these GP-compliant TAs mostly use the same APIs to interact with the system. Our emulator implements part of the GP API. During emulation, whenever a TA calls a GP API function, execution is redirected to our emulator's stub implementation. To demonstrate that our approach works for real targets we implement support for beanpod TAs in Xiaomi Redmi devices.

Chapter 2

Background

A Trusted Execution Environment (TEE) is a secure area within a main processor that provides an isolated environment for the execution of trusted code and secure data storage. It ensures that sensitive operations, such as cryptographic computations and secure data handling, are protected from interference or access by potentially malicious software running in the main operating system or other non-secure applications.

There are three main components in a TEE: Trusted Applications (TAs), TEE Kernel and Communication Interface. TAs are running within the TEE, they respond to functional requests from outside the TEE and perform sensitive operations like encryption, decryption, and authentication. The TEE kernel is the core component that manages the execution of trusted applications, enforces security policies, and handles communication between the TEE and the main operating system. The secure communication interface facilitates interaction between the main operating system and the TEE. This interface ensures that data exchanged between the two environments is protected against tampering and eavesdropping. The applications lay outside TEEs that interact with TAs within TEEs to perform functionalities are called Client Applications (CAs).

ARM TrustZone is a widely used TEE technology that leverages hardware-based isolation to create secure and normal worlds. The digital region is divided into two parts: the secure world and the normal world. The secure world handles security-sensitive tasks, such as cryptographic operations, secure boot, trusted applications, and access to sensitive data. It is isolated from the normal world, ensuring that secure operations are protected from potentially compromised software running in the normal world. The normal world runs standard applications and the main operating system. It operates without access to the secure world, ensuring that any compromise in this environment does not affect the secure operations. ARM TrustZone uses hardware mechanisms to enforce isolation between the secure and normal worlds. This includes separate memory spaces, CPU registers, and other resources. The CPU can switch between the two worlds using a special instruction, typically through a secure monitor that acts as a gatekeeper. The secure monitor is a small piece of software that runs in the secure world and controls the switching between the secure and normal worlds. It ensures that the transition is secure and that the normal world cannot access secure resources directly.

The standard for Trusted Execution Environments (TEEs) and Trusted Applications (TAs) is primarily governed by the GlobalPlatform specifications. GlobalPlatform is an industry consortium that develops and maintains these standards to ensure interoperability, security, and consistency across different implementations of TEEs and TAs. The defined standards include the fundamental architecture of a TEE, the design and deployment of TAs within the TEE, TEE Internal APIs [6] that provide a set of Application Programming Interfaces (APIs) for developing trusted applications, TEE Client APIs [15] that enable applications in the normal world (non-secure applications) to interact with TAs in the secure world. These standards largely reduce the work of implementing an emulator for different TEEs.

The beanpod TEE is the trusted environment on Xiaomi and Redmi smartphone devices. The beanpod ISEE (TEE OS) is based on the micro-kernel technology and trusted computing theory. It can effectively protect information such as biometrics, passwords, keys, files and location by using the chip layer of the intelligent device operating system. The ISEE is a GlobalPlatform-qualified product, passed the TEE security evaluation by China telecommunication technology labs and got the TEE security certification which is the first TEE manufacturer in China to obtain this award. In the Chinese market, Tens of millions of mobile phones have been equipped with ISEE 2.0 and supported many business fields such as mobile payment, biometrics, DRM etc.

Chapter 3

Design

We observe that TAs are strictly developed according to the TEE standard: the GlobalPlatfrom specifications, which means most library functions in TAs are TEE-independent. Based on this observation, we implement the function-level emulator by hooking every library calls with our own library functions. Note that even though most functions are reusable, there are still some custom functions we need to implement for each TA or TEE, which will be discussed later. So for the emulator to support more TAs, more functions need to be implemented. Currently, our emulator only supports the beanpod TAs.

3.0.1 Functions to Hook

Library function calls in all TAs can be divided into two categories:

- standard functions: the TEE internal functions that are defined in the GlobalPlatform API specification, e.g. `TEE_Malloc` and `TA_CreateEntryPoint`. The function declaration (parameters, return value etc.), exception handling, what the function should or should not do are clearly defined in the GlobalPlatform. So we implement our own library functions according to the GlobalPlatform and hook the emulated TAs with them.
- custom functions: the library functions not defined in the GlobalPlatform. These functions are vendor-specific and there is no standard documentation. In fact, most of these library functions are highly customized by the beanpod TEE producer. We dump libraries from the actual firmware and reverse engineering the functions so that we rewrite these custom functions one by one in our emulator. An example here is `ut_pf_km_enc_pw`, which reads a 32-byte password, encrypted with an internal RSA key and returns the cypher text.

We rewrite all the library functions, both standard and custom, to enable the emulator to apply function-level emulation for TAs. Standard functions can be used for all TAs, even for TAs in other TEEs. Custom functions, however, are only meaningful for the beanpod TEE. But it is good news that the number of custom functions in the beanpod TEE we need to rewrite are much less than the number of standard functions.

Every TA has a life cycle, by which the TEE OS controls the creation, functioning and destruction of a TA, based on requests from CAs. This is done by the life-cycle functions defined in the GlobalPlatform. For every TA, there are five life-cycle functions.

- `TA_CreateEntryPoint`: the Trusted Application's constructor, which the TEE OS calls when it creates a new instance of the Trusted Application.
- `TA_DestroyEntryPoint`: the Trusted Application's destructor, which the TEE OS calls when the instance is being destroyed.
- `TA_OpenSessionEntryPoint`: The TEE OS calls this function when a client (CA) requests to open a session with the created Trusted Application. This happens when CA calls `TEEC_OpenSession`.
- `TA_InvokeCommandEntryPoint`: The TEE OS calls this function when the client invokes a command within the given session. The Trusted Application can access the parameters sent by the client (CA) through the arguments. It can also use these arguments to transfer response data back to the client. This happens when CA calls `TEEC_InvokeCommand`.
- `TA_CloseSessionEntryPoint`: The TEE OS calls this function to close a client session. This happens when CA calls `TEEC_CloseSession`.

Additionally, since we implement the libraries ourselves, we also add some basic security checks to enhance the emulator, e.g. a simple heap Address Sanitizer to detect out-of-bound heap memory read/write, double free, use after free etc.

3.0.2 Interacting with the Emulator

To maximize the emulator's usability, we design it in the way that a CA's source code can be compiled either for the real device or for the emulator with some extra infrastructure header files included.

CA source code can be compiled with emulator infrastructure headers `repro.h` and `tee_client_api.h`. When building for the real device, the headers just reuse the functions in CA libraries so that the output binary can directly run on the real device to interact with TAs running in the real TEE. For emulator builds, all the CA library functions are patched to socket functions to interact with the emulator. When the user starts the emulator with a certain TA, the emulator hooks all the library functions in that TA with our own functions. The TA runs on the corresponding ISA emulator and

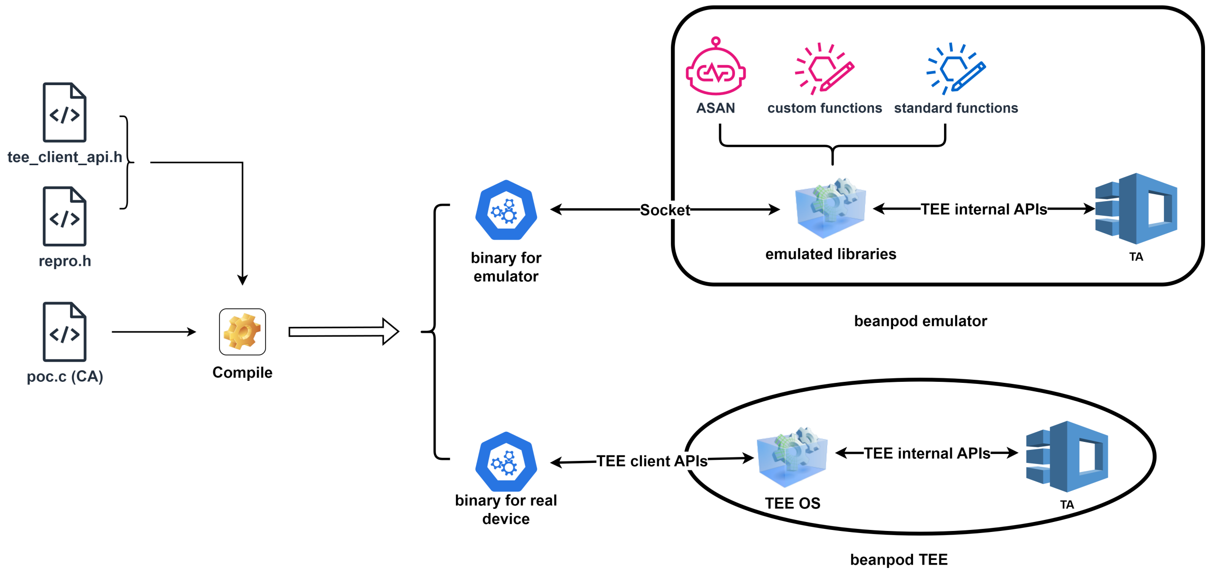


Figure 3.1: Emulator structure

waits for the output CA binary to connect. Then when the user executes the output CA binary on the local environment, the CA binary and TA are interacting with each other just like on the real device. The structure diagram is shown in 3.1.

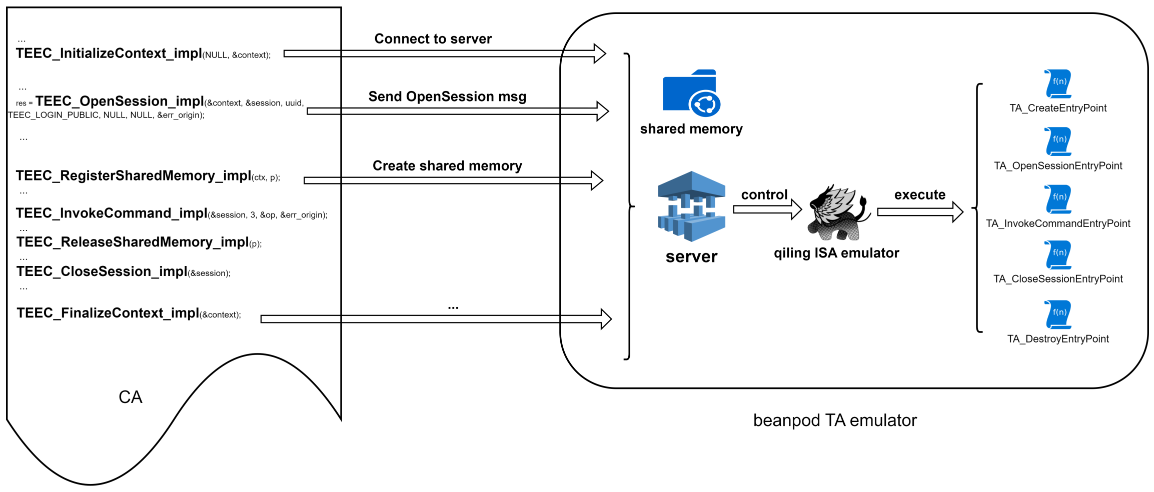
Chapter 4

Implementation

We choose the qiling framework [9] as the ISA emulator, which is an advanced binary emulation framework backed by the Unicorn engine. All the library functions are then rewritten in python code. It allows users to easily write the hook implementation at certain PC address. We take advantage of this and hook every library function call in the TA to execute our own functions. Also, python has great library support for cryptographic functionalities, making it easy for us to rewrite encryption- and signature-related TA library functions.

Normally, TAs create shared memory in secure world with CA to pass parameters during interaction. Our emulator also implements this by creating shared memory between the CA process and the emulator process. For the CA side, we use a pre-negotiated key to create a shared memory and send it to the server with keys in the payload. For the emulator side, we use `ctype` library to retrieve the created shared memory with the key just like in C.

The emulator is designed to be accurate in emulating the TAs lifecycle. When the emulator starts to emulate a TA, it first executes the creation functions of the life cycle: `TA_CreateEntryPoint` and then starts a socket server and waits for the CA to connect. The CA binary is integrated with a socket client, when it succeeds in connecting with the emulator, a message is sent to the server whenever CA calls a function to affect TA life-cycle so that the emulator can proceed to execute more. For example, when CA calls `TEEC_OpenSession`, the emulator gets the control message from the socket, executes `TA_OpenSessionEntryPoint` and waits again for further control message after `TA_OpenSessionEntryPoint` is executed. The structure is shown in



```
...
TEEC_InitializeContext_impl(NULL, &context);

...
res = TEEC_OpenSession_impl(&context, &session, uuid,
TEEC_LOGIN_PUBLIC, NULL, NULL, &err_origin);

...

TEEC_RegisterSharedMemory_impl(ctx, p);
...
TEEC_InvokeCommand_impl(&session, 3, &op, &err_origin);
...
TEEC_ReleaseSharedMemory_impl(p);
...
TEEC_CloseSession_impl(&session);
...

TEEC_FinalizeContext_impl(&context);
```

Figure 4.1: Emulation build infrastructure

Chapter 5

Evaluation

5.0.1 Utility and scalability

We support the emulation of 18 TAs which are deployed on two Xiaomi/RedMi devices. Among all the library functions that appear in these TAs, 60 are standard functions, and 16 are custom TAs. In this part, we showcase one of the Xiaomi devices we analysed: Xiaomi Redmi 9A (code name: dandelion). There are 13 TAs in total on this device, seven of them are accessible to normal users, which means either vendor or third-party applications talk to these TAs with certain functionalities. Two of them are not accessible to normal users, but we can still emulate the TA and interact with it them in our emulator. Three of them do not use any standard life-cycle APIs so likely they are not exposed to the normal world, as shown in 5.1. We emulated those accessible and inaccessible TAs, which give us nine.

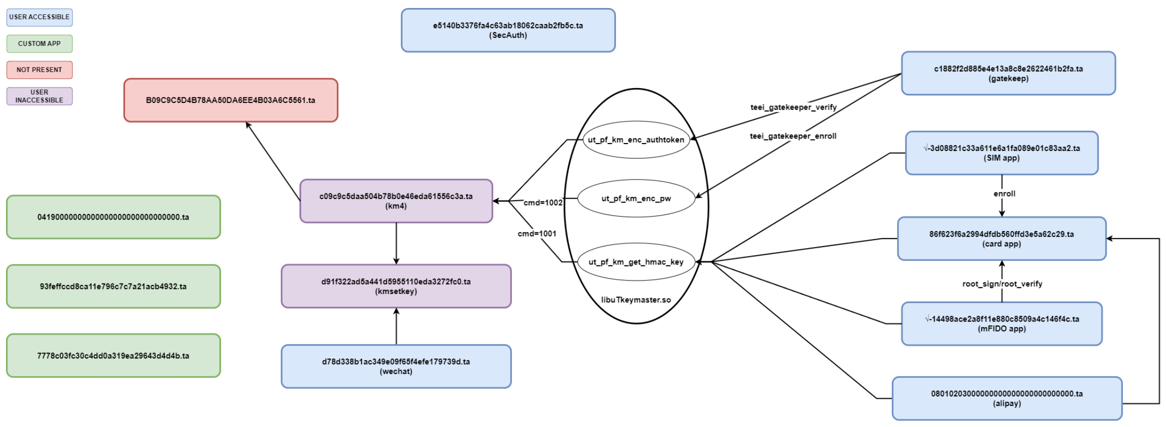

USER ACCESSIBLE
CUSTOM APP
NOT PRESENT
USER INACCESSIBLE
e5140b3376fa4c63ab18062caab2fb5c.ta (SecAuth)
B09C9C5D4B78AA50DA6EE4B03A6C5561.ta
c09c9c5daa504b78b0e46eda61556c3a.ta (km4)
d91f322ad5a441d5955110eda3272fc0.ta (kmsetkey)
d78d338b1ac349e09f65f4efe179739d.ta (wechat)
04190000000000000000000000000000.ta
93feffccd8ca11e796c7c7a21acb4932.ta
7778c03fc30c4dd0a319ea29643d4d4b.ta
cmd=1002
cmd=1001
ut_pf_km_enc_authtoken
ut_pf_km_enc_pw
ut_pf_km_get_hmac_key
libuTkeymaster.so
teei_gatekeeper_verify
teei_gatekeeper_enroll
c1882f2d885e4e13a8c8e2622461b2fa.ta (gatekeep)
√-3d08821c33a611e6a1fa089e01c83aa2.ta (SIM app)
enroll
86f623f6a2994dfdb560ffd3e5a62c29.ta (card app)
root_sign/root_verify
√-14498ace2a8f11e880c8509a4c146f4c.ta (mFIDO app)
08010203000000000000000000000000.ta (alipay)

Figure 5.1: Xiaomi Redmi 9A TA inetraction

To better understand the effort required when adding support for a TEE from scratch we present the number of python hooks that needed to be newly written for each TA, in chronological order, as shown in 5.2. Among all the library functions in these accessible and inaccessible TAs, a large majority of them are standard functions. Although there are many custom functions in some TAs, we found that many of them are called in other TAs too. As shown below, only the functions in the green part need to be rewritten. The average ratio of new custom functions to all library functions in each TA is 12.65%. On average to support a new TA, four new custom functions need to be implemented.

Note that TA no.4, 5 and 6 are highly customized because they are related to Alipay, WeChat and Xiaomi Pay and there is a whole custom library for each of them. Thus we took more efforts to emulate these application-related custom functionalities in these TAs.

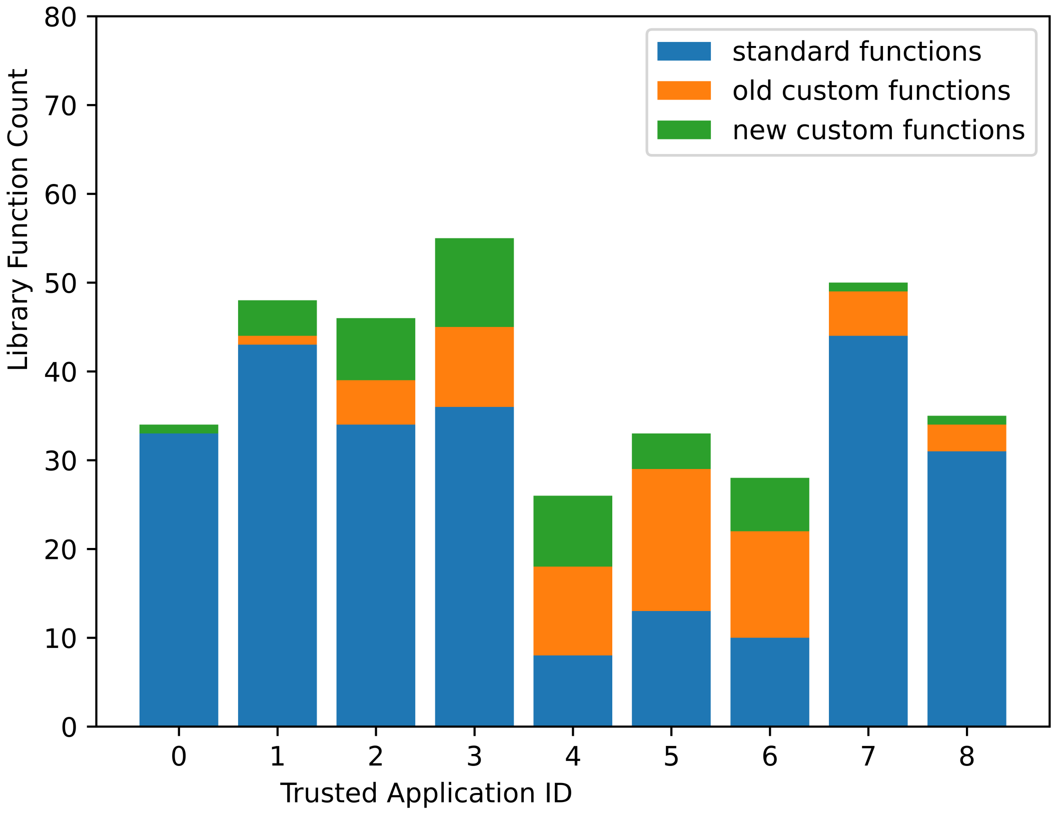


Figure 5.2: standard v.s. custom functions in analysed TAs

This experiment shows that the emulator manages to emulate every piece of code in many real-world Xiaomi/RedMi TAs. More custom functions need to be implemented for currently not supported TAs but the future cost is pretty low. This proves the usability and scalability of the emulator.

5.0.2 Fidelity and correctness

Using the emulator, we found 19 bugs in these TAs, as shown in 5.1. Due to the time limit, we did not look into Bug No.1, 14 and 15. All of the other non-reproducible bugs are shadowed by Bug No.8, as in, TAs crash there and can not reach the other vulnerable code. Bug No.16 happens in the TEE kernel thus beyond our research scope.

Bug No.8 is reproducible on real devices but not on our emulator. This is a bug in the internal TEE infrastructure, which is beyond our scope of emulation. We conclude that all the bugs reproducible on our emulator are also reproducible on real devices. Also, not only for these bugs, we did not spot any inconsistency between the log output of real devices and of the emulator throughout the emulation. This proves the fidelity and correctness of the emulation.

5.0.3 Case Study 1: Effort to Implement Function Hooks

In this part, we evaluate how hard it is for the emulator to support more TAs in the future.

Implement standard functions

We choose the `TEE_AsymmetricDecrypt` to show how we write the emulation stub and hook for those GlobalPlatform-defined functions. According to the standards, `TEE_AsymmetricDecrypt` function decrypts a message within an asymmetric operation. The decryption algorithm depends on the parameter `operation`, we showcase one of the valid operation `RSAES_PKCS1_V1_5_Operation` here.

```
TEE_Result TEE_AsymmetricDecrypt(
           TEE_OperationHandle operation,
[in]       TEE_Attribute* params, uint32_t paramCount,
[inbuf]    void* srcData, size_t srcLen,
[outbuf]   void* destData, size_t *destLen );
```

We hook the function and get all the parameters using qiling.

```
def TEE_AsymmetricDecrypt(ql:Qiling, func_name):
    global OPERATION_ID, id2opration
    params = ql.os.resolve_fcall_params({'operation': UINT, 'params': POINTER,
        'paramCount': UINT, 'srcData': POINTER, 'srcLen': UINT, "destData": POINTER,
        'destLen': UINT})
    param_operation = params['operation']
    param_params = params['params']
    param_paramCount = params['paramCount']
```

BUG No.	TA/component	description	reproducible	crashing
0	vSIM	oob read in `sim_auth`	✓✓	✗✗
1	vSIM	timing side channel in `sim_add`	-	-
2	mFIDO	invalid free in `TEE_RpmbOpenSession`	✗✗	✗✗
3	mFIDO	oob rw in `fido_read_pubkey_cert`	-	-
4	mFIDO	oob read in `add_authenticator`	✓✓	✗✗
5	mFIDO	missing size check in `key_load`	-	-
6	mFIDO	missing size check in `key_load`	-	-
7	mFIDO	Invalid free in `fido_write_data`	-	-
8	card	internal crash in `TEE_AsymmetricVerifyDigest`	✓✗	✓✗
9	card	no size check in `cmd_ecc_internal_sign`	✓✓	✗✗
10	mFIDO	param type confusion in `TA_InvokeCommandEntryPoint`	✓✓	✓✓
11	0801020300	param type confusion in `TA_InvokeCommandEntryPoint`	✓✓	✓✓
12	mFIDO	param type confusion in `TA_InvokeCommandEntryPoint`	✓✓	✓✓
13	mFIDO	functionality bug in `add_authenticator`	✓✓	✗✗
14	0801020300	missing size check in `import_key`	-	-
15	0801020300	Heap buf oob read in `import_key`	-	-
16	l4re kernel	Heap buf overflow in `base64_encode`	-	-
17	e97c270ea5	param type confusion in `TA_InvokeCommandEntryPoint`	✓✓	✓✓
18	0811000000	param type confusion in `TA_InvokeCommandEntryPoint`	✓✓	✓✓
19	vSIM	oob read in `auth_sim_v1`	✓✓	✓✓

Table 5.1: all bugs we found. ✓✓: both reproducible on/crashing the emulator and real devices. ✓✗: reproducible on/crashing real devices but not reproducible on/crashing the emulator. ✗✗: not reproducible on/crashing both the emulator and real devices.

```
    param_srcData = params['srcData']
    param_srcLen = params['srcLen']
    param_destData = params['destData']
    param_destLen = params['destLen']
```

`TEE_OperationHandle operation` should be opened before, so we use a global map to record all opened operations. Then we get the decrypted content and set the return value accordingly.

```
    op = id2opration[param_operation]
    ct = ql.mem.read(param_srcData, param_srcLen)
    pt = op.decrypt(ct, ql)
    if len(pt) == 0:
        ql.log.info("\tdecrypt error")
        ret = TEE_ERROR_CIPHERTEXT_INVALID
    else:
        ql.mem.write(param_destData, pt)
        ql.mem.write_ptr(param_destLen, len(pt))
        ret = TEE_SUCCESS

    ql.os.fcall.cc.setReturnValue(ret)
    ql.arch.regs.arch_pc = ql.arch.regs.lr
```

The operation is implemented with the PyCryptodome library, using out-of-the-box cypher APIs.

```
class RSAES_PKCS1_V1_5_Operation(Operation):
    def __init__(self, operationID, mode, keySize, ql) -> None:
        super().__init__(operationID, ql)
        self.keySize = keySize
        self.mode = mode
        self.key = None
        self.cypher = None
        self.initialized = False

    def initialize(self, params, ql):
        n = bytes_to_long(params['n'])
        d = bytes_to_long(params['d'])
        e = bytes_to_long(params['e'])
        self.key = RSA.construct((n, e, d))
        self.cypher = PKCS1_v1_5.new(self.key)
        self.initialized = True

    def decrypt(self, ct, ql:Qiling):
        ct = bytes(ct)
        ql.log.info(f"\tct: {ct}, len: {len(ct):#0x}")
        pt = self.cypher.decrypt(ct, None)
        ql.log.info(f"\tpt: {pt}, len: {len(pt):#0x}")
        return pt
```

It takes some work to implement these crypto-based standard functions, but they are reusable for all the TAs.

Implement custom functions

We choose `ut_pf_km_get_hmac_key` in one of the TAs to showcase how to implement custom functions. We need to find the binary and reverse engineer the function to figure out the functionality of it. The easiest way here would be to rewrite this function in Python according to the result of decompiler tools. Take part of the code in this function for example, the decompilation result is quite readable.

```
int __fastcall ut_pf_km_get_hmac_key(int a1, _DWORD *a2)
{
  v2 = a2 == 0;
  if ( a2 )
    v2 = a1 == 0;
  v3 = v2;
  if ( v2 )
  {
    ut_pf_log_msg(5, "[%s:%d/%s]<err>%sbad params\n", "ut_pf_km.cc", 30,
        "ut_pf_km_get_hmac_key", "");
    return -1;
  }
  else
  {
    *a2 = 32;
    v10 = 32;
    v8 = v3;
    v9[0] = v3;
    v9[1] = v3;
    v11 = v3;
    v12 = v3;
    v13 = v3;
    v14 = v3;
    v9[2] = a1;
    v5 = TEE_OpenTASession(&skm_ta_uuid, v3, 98, v9, &v8, v3);
    //...
```

To emulate this, we get the parameters and map necessary memory regions for global and local buffers in emulated functions.

```
def ut_pf_km_get_hmac_key(ql: Qiling, func_name):
    # will go into subroutine so lr needs to be recorded
    current_lr = ql.arch.regs.lr
    ql.arch.regs.arch_sp -= 0x38

    params = ql.os.resolve_fcall_params({"a1": UINT, "a2": POINTER})
    a1 = params["a1"]
    a2 = params["a2"]
    hmac_size = ql.mem.read_ptr(a2)

    ql.log.info(f"ut_pf_km_get_hmac_key {hex(a1)}, {hex(a2)}, {hex(hmac_size)}")

    ql.mem.write(a1, b'a'*hmac_size)
    ql.os.fcall.cc.setReturnValue(0)
    ql.arch.regs.arch_pc = current_lr

    temp_mem = ql.mem.map_anywhere(
        0x1000, minaddr=EMULATED_LIB_BSS_MEM, perms=3, info="emulated_libc_bss"
    )
```

Then we do exactly what we see in the decompilation result. One thing to mention here is before calling other library functions (both standard and custom), we modify the registers accordingly.

```
    v2 = a2 == 0
    if a2:
        v2 = a1 == 0
    v3 = v2
    if v2:
        ut_pf_log_msg_fake(
            ql,
            5,
            "[%s:%d/%s]<err>%sbad params\n"
            % ("ut_pf_km.cc", 30, "ut_pf_km_get_hmac_key", ""),
        )
        ql.mem.unmap(temp_mem, 0x1000)
        ql.os.fcall.cc.setReturnValue(-1)
        ql.arch.regs.arch_sp += 0x38
        ql.arch.regs.arch_pc = current_lr
    else:
        ql.mem.write_ptr(a2, 32)
        ql.mem.write_ptr(v9_addr, v3)
        ql.mem.write_ptr(v9_addr + 4, v3)
        ql.mem.write_ptr(v9_addr + 8, a1)
        v10 = 32
        ql.mem.write_ptr(v9_addr + 12, v10) # v10
        ql.mem.write_ptr(v9_addr + 16, v3) # v11
        ql.mem.write_ptr(v9_addr + 20, v3)
        ql.mem.write_ptr(v9_addr + 24, v3)
        ql.mem.write_ptr(v9_addr + 28, v3) # v14

        # call TEE_OpenTASession(&unk_9188, v3, 98, v9, &v8, v3)
        ql.arch.regs.r0 = unk_9188
        ql.arch.regs.r1 = v3
        ql.arch.regs.r2 = 98
        ql.arch.regs.r3 = v9_addr
        ql.mem.write(ql.arch.regs.arch_sp, p32(v8_addr))
        ql.mem.write(ql.arch.regs.arch_sp + 4, p32(v3))

        TEE_OpenTASession(ql, "TEE_OpenTASession", True)
        v5 = ql.arch.regs.r0
```

In this way, we can rewrite the custom functions without having a comprehensive understanding of how the function works, which reduces the effort of reverse engineering to the minimum.

Note that our emulate does not support multiple TAs at the same time so we would just fill the return buffer with some reasonable content if it interacts with other TAs in the emulated functions.

5.0.4 Case Study 2: Prototyping a TA exploit

We use our emulator to develop an exploit for TA 08110000000000000000000000000000 on RedMi note11s device, to achieve arbitrary code execution in the TA context. In this part, we analyse the root cause of this bug and showcase the proof of concept of the exploit.

In TA_InvokeCommandEntryPoint, the paramtype argument determines whether or not the params is a memory reference or value reference. However in switch case 0xA, it parses params[2] as a memory reference without any check for the paramtype argument. So we could pass a value reference there and cause type confusion. Later the output writes to the kernel log is a great arbitrary read primitive.

```
int __fastcall TA_InvokeCommandEntryPoint(int a1, unsigned int cmd, int paramtype,
    struct TEE_Param_Memref *params)
{
  switch ( cmd )
  {
    ...
    case 0xAu:
      msee_ta_printf_va("[KI_TA] INFO:");
      msee_ta_printf_va("TZCMD_DRMKEY_INSTALL_PLAINTEXT start");
      msee_ta_printf_va("\n");
      v26 = params[2].buffer; // let it be a value ref instead of a memory ref
      v27 = v26[3];
      v28 = v26[2];
      ...
      msee_ta_printf_va("[KI_TA] INFO:");
      msee_ta_printf_va("keyid      = %d", v27);    // arb read
      msee_ta_printf_va("\n");
      msee_ta_printf_va("[KI_TA] INFO:");
      msee_ta_printf_va("keytype    = %d", v28);    // arb read
      msee_ta_printf_va("\n");
  }
}
```

This is the same for switch case `0x1`, when it reaches `query_drmkey_impl`, `a1` is our input buffer and a3 is supposed to be the output buffer but due to type confusion, this could be a random value. After bypassing many checks for our input, we could construct an arbitrary write primitive here. Instead of statically analysing the code and reverse engineering, the emulator enables us to debug the TA to easily construct input that bypasses the checks and reaches this function. Although we can not develop our exploit against the emulator, it greatly eases the work of finding the bugs and triggering the bugs.

```
int __fastcall query_drmkey_impl(_BYTE *a1, int *a2, int a3, unsigned int a4)
{
  // a bunch of check
  if ( *a1 != 0x4B || a1[1] != 0x42 || a1[2] != 0x50 || a1[3] != 0x4D )
  {
    msee_ta_printf_va("[KI_TA] ERROR:");
    msee_ta_printf_va("Magic number error in Query DRM Key");
    goto LABEL_16;
  }
  ...
  v8 = a1 + 0x48;
  while ( 1 )
  {
    msee_ta_printf_va("[KI_TA] INFO:");
    msee_ta_printf_va("query_8, i=%d, size=%d", v9, 88);
    msee_ta_printf_va("\n");
    memcpy(dest, v8, sizeof(dest)); // ‘dest‘ is controllable
    ...
    v12 = dest[0];
    *(_DWORD *)(a3 + 4 * v9) = v12; // arb write
    ...
}
```

We use the write primitive to overwrite the return address of a random function to achieve code execution.

Chapter 6

Related Work

Much research has been done to emulate the TAs and apply dynamic analysis techniques. PARTEMU [7] enables dynamic analysis techniques such as feedback-driven fuzz testing, symbolic and concolic execution, taint analysis, and debugging for real-world TrustZone software. CROWBAR [11] performs native feedback-driven fuzzing based on ARM CoreSight infrastructure. However, Partemu is not open-source yet and CROWBAR requires a device with coresight enabled.

TEEzz [1], on the other hand, requires a physical phone to interact with. There is no emulation so the fuzzer only works specifically on a given TZ OS and it can only apply black box fuzzing on its physical setup. Our emulator, however, is mostly OS-independent and does not rely on a real device.

TEEMU [8] emulates the TZOS by manually re-implementing specific system calls. This limits TEEMU to testing TAs that use only those system calls, and does not allow testing the TZOS itself. Furthermore, reproducibility is dependent on the fidelity of re-implementation of the TZOS system calls. DTA [13] needs a custom TA in the secure world to relocate TAs outside of the secure world by implementing an alternative context switch mechanism and delegating secure world system calls to a proxy handler. Our emulator is a lightweight function-level emulator, it does not emulate the whole TEE OS infrastructure thus no further interface or work needs to be done at the TEE OS level.

The newest research about the beanpod TEE [3] comes with an exploit that only forges payment packets, indicating that beanpod TAs and TEEs are not fully researched yet.

Chapter 7

Conclusion

In this paper, we developed an easy-to-use, lightweight function-level TA emulator for XiaoMi/RedMi beanpod TEE. With this emulator, we found 16 unique bugs. Using the emulator we prototype an exploit and achieve code execution in TA context.

Bibliography

[1] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. “TEEzz: Fuzzing Trusted Applications on COTS Android Devices”. In: *IEEE International Symposium on Security and Privacy*. 2023.

[2] CVE. *Google android security vulnerabilities*. `https://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html`.

[3] *Digging into Xiaomis TEE to get Chinese money*. `https://research.checkpoint.com/2022/researching-xiaomis-tee/`.

[4] Google. *Drm*. `https://source.android.com/devices/drm`.

[5] Google. *Google play billing overvie*. `https://developer.android.com/google/play/billing/billingoverview`.

[6] *GPD TEE Internal Core API Specification*. `https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf`.

[7] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. “PartEmu: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation”. In: *Usenix Security Symposium*. 2020.

[8] D. Komaromy. *Unbox Your Phone - Exploring and Breaking Samsung's TrustZone Sandboxes*. `http://www.ekoparty.org/charla.php?id=756`.

[9] *Qiling framework*. `https://research.checkpoint.com/2022/researching-xiaomis-tee/`.

[10] SecWiki. *Android kernel exploits*. `https://github.com/SecWiki/android-kernel-exploits`.

[11] Haoqi Shan, Moyao Huang, Yujia Liu, Sravani Nissankararao, Yier Jin, Shuo Wang, and Dean Sullivan. *CROWBAR: Natively Fuzzing Trusted Applications Using ARM CoreSight*. `https://www.researchgate.net/publication/377720651_DTA_Run_TrustZone_TAs_Outside_the_Secure_World_for_Security_Testing`.

[12] shen and blackhat. *Attacking-Your-Trusted-Core*. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf.

[13] JUHYUN SONG, EUNJI JO, and JAEHYU KIM. *DTA: Run TrustZone TAs Outside the Secure World for Security Testing*. https://www.researchgate.net/publication/377720651_DTA_Run_TrustZone_TAs_Outside_the_Secure_World_for_Security_Testing.

[14] He Sun, Kun Sun, Yuewu Wang, and Jiwu Jing. "Trustotp: Transforming smartphones into secure one-time password token". In: *ACM Conference on Computer and Communication Security*. 2015.

[15] *TEE Client API Specification*. https://globalplatform.org/wp-content/uploads/2010/07/TEE_Client_API_Specification-V1.0.pdf.

[16] xairy. *kernel exploits*. https://github.com/xairy/kernel-exploits.