Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution

Marcel Busch and Kalle Dirsch
{marcel.busch, kalle.dirsch}@fau.de
IT Security Infrastructures Lab
Department of Computer Science
Friedrich-Alexander University Erlangen-Nürnberg (FAU)

***Abstract*—Trusted Execution Environments (TEEs) constitute a major building block of modern mobile devices' security architectures. Yet, the analysis tools available to researchers seeking to examine these critical components are rudimentary compared to the vast range of sophisticated tools available for other execution contexts (*i.e.,* Linux or Windows userland). We see the primary reason for the lack of tools in the closed-source nature of TEEs. Trusted Applications (*i.e.,* userland applications executed in a TEE) account for the biggest attack surface, however we cannot investigate them during runtime on most devices because hardware primitives (*i.e.,* ARM TrustZone) prevent access to this higher privileged context.**

In this paper, we present our approach to investigate 1-day vulnerabilities of real-world Trusted Applications (TAs). Our system, SimTA, is based on angr and simulates the TA's execution environment. We build SimTA based on insights of static analysis and runtime parameters of a TEE which we revealed using an exploit on a physical device. In our evaluation we elaborate on how SimTA facilitates the binary-diff-guided analysis by replicating the analysis of a known critical vulnerability. Moreover, we reveal two additional issues, an authentication bypass and a heap-based buffer overflow, that have quietly been introduced by the vendor.

I. Introduction

In 2016, at an event called "GeekPwn", Stephens [20] presented a chain of exploits that led to code execution within the TEE of Huawei. Using these exploits, he could unlock the targeted device using the fingerprint sensor with a finger of any person or even a nose. His entry into the TEE is connected to CVE-2016-8764, which is an input validation vulnerability that an attacker could use to get an arbitrary code execution in the TEE context.

Commonly, a way to investigate vulnerabilities similar to this is binary-diffing in combination with meticulous manual analysis. To extract the patch for the vulnerability in question, we refer to CVE-2016-8764's summary [17], and identify the latest affected version in order to compare it with the following version. One problem that can arise while extracting the patch is that not only the vulnerable sequence of instructions appears in the binary-diff, but many others. For example, new features could have been introduced, or compiler flags might have changed, resulting in irrelevant sequences. To distinguish relevant sequences, indicators such as additional code accessing attacker provided input, could be used. Unfortunately, it is not possible to use dynamic analysis inside of the TEE, because access is usually locked down by vendors, in order to investigate which patches actually handle attacker controlled input. Another problem is that after the vulnerability has been found, an analyst needs many parameters from the address space to replicate the exploit. However, the layout of the address space (*i.e.,* the location where code and data are mapped to), which is necessary for the replication, is not publicly disclosed.

In this work, we present our insights and techniques to face these challenges. We studied CVE-2016-8764 using manual analysis guided by binary-diffing and performed a dynamic analysis on the device, treating the TEE as a black-box. We were successful in replicating Stephens' exploit, and gained insights into Huawei's TEE Trusted Core (TC). Using this exploit, we acquired the address space layout of the targeted TA. Next, leveraging the runtime parameters observed from the device, we implemented an angr-based [19] prototype, SimTA, capable of simulating the execution environment, and achieved a runtime behavior that is close to the expected execution of the TA.

In addition to having an execution environment for the targeted TA, SimTA allows us to annotate the attacker controlled input, thus, permitting us to filter patches dealing with attacker controlled input from the binary-diff. Furthermore, we can even selectively introduce symbolic inputs in order to better understand the constraints introduced by a patch. As a result, we found a previously unknown 1-day heap-overflow vulnerability, an authentication bypass, and, apparently, the already known type-confusion vulnerability of CVE-2016-8764. We elaborate on the analysis preceding these findings in our evaluation.

In summary, our contributions are the following:

- We share our insights on the interfaces, abstraction layers, and address space layout of one TA for the TC TEE. In order to get access to TEE internals and examine the runtime parameters of the TA, we implement and use an exploit for CVE-2016-8764 to collect the information from a real device.
- Based on these insights we implement SimTA, an analysis tool capable of executing the target TA with

NDSS Workshop on Binary Analysis Research (BAR) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

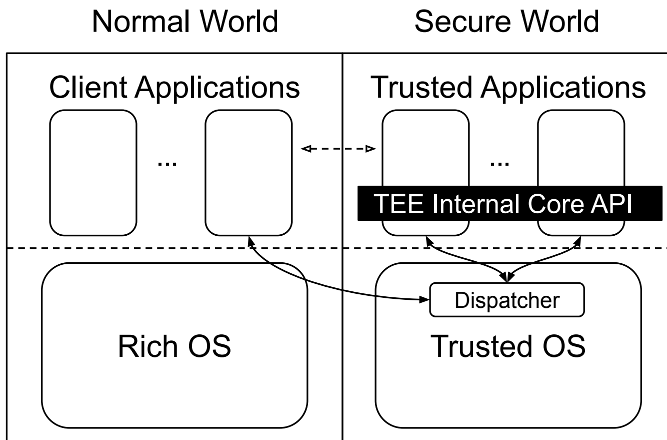

Fig. 1: Architecture and communication channels of modern TZ-based TEEs. The logical channel of a CA and TA interaction (dashed line) is actually carried out by both OSes that cooperatively forward and dispatch requests (solid lines).

selectively chosen symbolic input. We open-source SimTA to further TEE-related research.

- Using SimTA, we present a methodology to investigate binary-diff-guided analyses of security critical patches. We show the effectiveness of our methodology by replicating the analysis of CVE-2016-8764 and revealing two yet unknown critical issues affecting the same version.

Closed systems, like TEEs, pose a big obstacle for security research, because access is limited and information is scarce. With this research, we present an approach to enable the replication of CVEs and security fixes quietly introduced by vendors. By making our prototype publicly accessible, we hope to advance security research on TEEs. Our code and instructions on how to get the binaries, can be found on https://github.com/teesec/simta.

II. Background and Challenges

ARM TrustZone (TZ) [1] is common on modern mobile devices. It allows to partition the System-on-Chip (SoC) into two execution contexts – the Secure World (SW) and the Normal World (NW) – where code and data from the SW cannot be accessed by the NW. The idea is to run a feature-rich operating system and its userland in the NW and only execute trusted code in the SW. Recent TZ-based TEEs split the SW into a kernel, the Trusted Operating System (Trusted OS), and a userland, which hosts TAs as well.

For data exchange, a logical communication channel exists between a CA and a TA (dashed line in Figure 1). Using this channel, CAs can request services from TAs. For example, requesting the generation of an asymmetric keypair, where the private key resides in the TEE and the public key can be used by the CA, is a common use case. In addition to the key generation, the TA would also provide an API to perform cryptographic operations using the safely stored private key (*e.g.,* sign or decrypt messages).

Technically, the CA cannot call a TA directly. It needs to send its request to the Rich Operating System (Rich OS) that takes care of using a NW-SW shared memory region for the provided request data and initiates the world switch using a privileged instruction (*e.g.,* `smc`). Then, the Trusted OS dispatches the request to the addressed TA. When the TA has processed the request, it writes its output to the shared memory region used for this session, returns to the Trusted OS which, in turn, initiates the world switch back to the NW. Finally, the Rich OS returns execution to the CA. This communication channel is depicted by the solid line in Figure 1.

Vendors of TEEs have an interest in providing a common interface for TAs in order to execute third-party TAs on their platforms. One set of standards that gains popularity in this regard is specified by GlobalPlatform (GP). GP is a non-profit industry association that strives to enable collaborative and open ecosystems by developing specifications especially in regard to trusted computing technologies and particularly for TEEs. The specification relevant for our work is the GP TEE Internal Core API [7], which defines a common interface that can be used by TAs (depicted in Figure 1).

In this work, we are interested in dynamically analyzing patches introduced into TAs. In contrast to software executed in the NW, an analyst faces some challenges when trying to analyze TAs.

On modern Android devices, the trusted code (*e.g.,* all code that runs in SW) is proprietary, meaning there is no source code available. Furthermore, this code is executed in the SW and, due to the TZ-provided isolation, we cannot make use of common tools utilized to study the runtime behavior of programs (*e.g.,* debuggers). Also, we cannot modify a TA's code (*e.g.,* binary instrumentation) because all code executed within the TEE is signed and gets verified before execution. Given these constraints, it seems unfeasable to perform advanced analyses to study patches of TAs on the device.

In order to study TAs during runtime, there are two apparent approaches. The first approach consists of taking the entire SW software stack in binary format (*e.g.,* Trusted OS and TAs) and execute it in a system emulator like QEMU. One of the problems with this approach is that SoCs used in production are usually not supported by full system emulators because their datasheets are not publicly available. An option to make this approach feasible is to reverse engineer the interaction between the Trusted OS and the hardware in order to implement a proper emulation. The interested reader may be referred to Harrison *et al.* [10] who studied this approach.

The second approach focuses on the execution of individual TAs without the Trusted OS. This approach requires the knowledge of the virtual address space structure of the TA, the simulation of entries into and exits from the TA as well as input and output mechanisms. For SimTA, we went with this approach and implemented a prototype for one of the TAs found on Huawei Android devices.

III. Related Work

Vulnerability research on TEEs, so far, primarily happens statically and there are only a few cases known where researchers shared their insights.

Our work in this paper stands on the shoulders of Stephens' research [20], [21] who presented a chain of exploits targeting Huawei's TEE implementation TC. Using this exploit chain he could elevate his privileges from an Android application (the CA in this case) to the trusted OS. Out of this work, particularly CVE-2016-8764 [17] is relevant for our research. Prior work on TC was carried out by Shen [18] who presented an escalation to the trusted OS at BlackHat US 2015.

Further research targeting the Qualcomm Secure Execution Environment (QSEE) was conducted by Beniamini [3]–[6]. Beniamini, besides other issues, also elaborates on a chain of exploits leading to trusted OS code execution. In addition, Quarkslab [9] conducted research on an OTP TA running on QSEE and shared their insights.

Besides TC and QSEE, Samsung devices used to ship with a TEE which was first called MobiCore and later Kinibi. Atamali-Reihneh *et al.* [2] analyzed Samsung KNOX and its usage of the MobiCore TEE on a rather conceptual level. Later, Kinibi has been researched by Komaromy [12]–[14] and Beniamini [6] as well, where both analyses provide many details with technical depths.

As far as we know, all of the above mentioned research was carried out using manual static analysis and trial-and-error testing against a black-box on the phone as the only option for dynamic analysis. From our own work, we can tell that especially the last stage of exploit development (*e.g.,* getting the exploit stable without the target crashing) targeting a TEE is laborious.

Recently, more attempts to emulate and/or re-host TEE components surfaced. For instance, CheckPoint Security [16] describe their work on a QEMU-based prototype capable of executing QSEE TAs. Their prototype forwards syscalls of TA's to a manipulated TA inside of the TEE that acts as a proxy. The modifications of the proxy-TA are carried out using the flaws documented by Beniamini [4], [5]. Unfortunately, no source code of this prototype is publicly available yet.

An approach not requiring a physical device targeting TEEGris is pursued by Tarasikov [22]. TEEGris is a TEE implementation recently introduced on Samsung phones that presumably replaced Kinibi. Tarasikov is working on a QEMU-usermode implementation to execute TEEGris TAs.

A re-hosting solution capable of performing full-system emulation of TEEs named `PartEmu` will be presented at Usenix 2020 by Harrison *et al.* [10]. TC, the TEE in scope for our research, is not part of `PartEmu`'s evaluation.

Lastly, Hua *et al.* [11] propose a system called *vTZ* capable of virtualizing TEEs in ARM TZ. This work does not have the execution of closed source TEEs in scope and extending its scope would require significant reverse engineering effort in order to support the hardware requirements of different proprietary TEEs available on the market.

In our approach we build on top of the gained vulnerability research and connect our insights with an angr-based approach to execute TAs for TC.

IV. Understanding Trusted Applications for TrustedCore

In this section, we elaborate on the structure of TC-based TAs and share insights gained from an exploit on a physical device. The insights from this section will later allow us to implement SimTA, an angr-based prototype capable of simulating the execution environment of the target TA. We limit our scope to one TA (`storageTA`) that can be found on Huawei devices, although many insights apply generically to TC TAs (*i.e.,* they all use GP standards and, presumably, the same loader).

A. TA Lifecycle

Our first observation by looking at `storageTA` is that it makes use of the GP TEE Internal Core API specification [7]. This API defines the lifecycle of TAs. A simplified version of TC's implementation of this lifecycle is given in Listing 1. The `MsgRcv()` and `MsgSend()` functions represent the integration of the TA with the dispatcher (see Figure 1). The lifecycle functions have the following purpose:

- `TA_CreateEntryPoint`. Constructor, called once during initialization of TA.
- `TA_OpenSessionEntryPoint`. Opens client session, the provided session object is used to maintain state for the session.
- `TA_InvokeCommandEntryPoint`. Invocation of trusted application commands.
- `TA_CloseSessionEntryPoint`. Closes client session, frees space for the session object.
- `TA_DestroyEntryPoint`. Destructor, called once during tear down of TA.

The most interesting data structures processed by this API are passed to `TA_OpenSessionEntryPoint` and `TA_InvokeCommandEntryPoint`. The arguments `cmdId`, `paramTypes`, and `params` originate from a CA in NW and, therefore, are user controlled.

B. TA Commands

Inside of the `TA_InvokeCommandEntryPoint` lifecycle call, the actual interface of the TA is exposed. As depicted in Listing 2 (a simplified version of `storageTA`), the TA-specific command is chosen via the `cmdId` argument.

From this listing we can see that `params` is either a memory reference or a value. The corresponding union definition is given in Listing 3. The params argument of `TA_InvokeCommandEntryPoint` contains up to four pointers to a struct like this.

Now it also becomes clear that the `paramTypes` argument is necessary for the TA to get to know if the CA sent a buffer or values. For each command, the TA has to check the `paramTypes` to prevent wrong assumptions about how the provided data can be interpreted.

From Listing 2 we can also see that the TA itself can have a stateful API. The state for `storageTA` is stored within the

```
while ( 1 ) {
  LifecycleData* data = MsgRcv();

  switch ( data->lifecycle_cmd ) {
    case 1:
      void* sessCtx = NULL;
      // constructor
      if (data->init) {
        TA_CreateEntryPoint();
      }
      TA_OpenSessionEntryPoint(
          data->paramTypes,
          data->params,
          &sessCtx);
      data->sessCtx = sessCtx;
      break;
    case 2:
      TA_InvokeCommandEntryPoint(
          data->sessCtx,
          data->cmdId,
          data->paramTypes,
          data->params);
      break;
    case 3:
      TA_CloseSessionEntryPoint(
          data->sessCtx);
      // destructor
      if (data->deinit) {
        TA_DestroyEntryPoint();
      }
      break;
    default:
      break;
  }
  MsgSnd(data);
}
```

Listing 1: TC TAs follow the lifecycle specified in the GP Internal Core API.

`sessCtx` argument so that subsequent calls from the same CA can operate on this state (*e.g.,* read from an open file handle).

Looking at the function used inside the different commands (*e.g.,* `TEE_OpenPersistentObject()`), we can see that most of them map to the GP Internal Core API [7]. `storageTA` itself is just a thin layer of code that connects the TEE lifecycle with the storage API defined in the GP Internal Core API.

C. TA Address Space Layout

From the previous two sections we can learn about the integration of the TA into the TEE (*e.g.,* how the dispatcher provides data) and how the GP Internal Core API encapsulates most of the functions used by a TA. In order to simulate the execution environment on the device, however, we need the TA's address space layout. This layout will later be used as an input for SimTA.

To retrieve these runtime parameters, we build on the research of Stephens [20]. He found a type-confusion bug in `storageTA` that he could use to gain code execution in this TA's execution context. Since no proof of concept was available for this exploit, we needed to replicate his research and develop the exploit ourselves. Stephen's slides were invaluable for this process. Using the exploit, we were able to leak arbitrary memory from `storageTA`'s execution context and reconstruct its address space. The reconstructed address space is depicted in Figure 2.

The first element mapped is `globaltask` which is a binary without ELF header that can also be ex-

```
TA_InvokeCommandEntryPoint(sessCtx, cmdId,
  paramTypes, params) {
  switch ( cmdId ) {
    case FOPEN:
      if (paramTypes != FOPEN_PTYPES)
        goto ptype_error;

      char* path; size_t pathsz;
      uint32_t flags;
      TEE_ObjectHandle obj;

      path = params[0]->memref.buffer;
      pathsz = params[0]->memref.size;
      flags = params[1]->value.a;

      TEE_OpenPersistentObject(
          TEE_STORAGE_PRIVATE,
          path,
          pathsz,
          flags,
          &obj);
      ...
      break;
    case FCLOSE:
      if (paramTypes != FCLOSE_PTYPES)
        goto error;
      ...
      TEE_CloseObject(...);
      break
    case FREAD:
      if (paramTypes != FREAD_PTYPES)
        goto error;
      ...
      TEE_ReadObjectData(...);
      break;
    case FWRITE:
      if (paramTypes != FWRITE_PTYPES)
        goto error;
      ...
      TEE_WriteObjectData(...);
      break;
    ...
  }
  return;
ptype_error:
  log("bad param types");
  return;
}
```

Listing 2: Each TC TA has a `cmdId`-handler that implements different commands. This is a simplified handler of `storageTA`.

```
typedef union {
    struct {
        unsigned int buffer;
        unsigned int size;
    } memref;
    struct {
        unsigned int a;
        unsigned int b;
    } value;
} TC_NS_Parameter;
```

Listing 3: An array of four `TC_NS_Parameter` unions is the primary data structure for input and output for TC TAs.

tracted from the device's firmware image. Fortunately, we found a string table and a symbol table at the end of the binary. Using symbols like `TEE_TEXT_START`, `TEE_BSS_START`, and `TEE_GOT_START`, we were able to reconstruct `globaltask`'s sections. `globaltask`'s code and data sections are mapped according to the assigned virtual addresses from the ELF section headers. Directly above `globaltask`, we can find the `storageTA` binary itself. `globaltask` turns out to contain the implementation of all GP Internal Core API functions that are called by `storageTA`

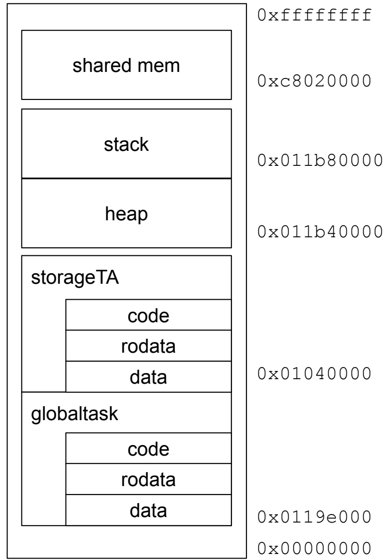

Fig. 2: Virtual address space of `storageTA` as extracted from a real device using an exploit based on CVE-2016-8764.

and it is the only mapped library. Consequently, `globaltask` should only be dependent on the syscalls of the Trusted OS. By leaking the content of `storageTA`'s `.bss` segment that contained references to the heap, we were able to locate the heap. Using a leaked stack pointer (`r13`) we could also reveal the stack location. Additionally, a shared memory region is mapped into the address space which is used for data exchange between SW and NW.

D. Generalizing SimTA

The insights gathered regarding TC TAs are based on the static and dynamic analysis of one TA (*e.g.,* `storageTA`). The firmware we investigated contains multiple TAs, namely[1]:

- `task_keyboard`
- `task_storage`
- `task_gatekeeper`
- `task_keymaster`
- `task_reet`
- `task_secboot`

By looking at these TAs, we could verify that all of them make use of the TA lifecycle and TA command processing as described above. Since we do not have an exploit to study the address spaces of these TAs, we were not able to verify if our gained insights from `storageTA`'s address space also hold for these TAs. A reasonable assumption is though, that all TAs are loaded by the same loader.

Besides the TAs found in the firmware image, we identified encrypted TAs that reside in the Android filesystem (*e.g.,* all `/vendor/bin/*.sec` files). Since we could not analyze these TAs statically nor dynamically, we cannot verify our gained insights for these TAs as well.

In order to get an idea of our insights' general applicability, we looked into OP-TEE [15] TAs. OP-TEE is an open source reference implementation for ARM TZ-based TEEs maintained by Linaro. We found that OP-TEE TAs use the same lifecycle functions as TC (see Section IV-A). One of the major differences is the lifecycle's integration with the dispatcher. Instead of the `while(1)-loop` structure communicating with the dispatcher using `MsgRcv()` and `MsgSnd()` (as described in Listing 1), OP-TEE uses two functions called `utee_entry` and `utee_exit` to enter and exit TAs. The `cmdId`-handler of OP-TEE TAs is similar to the one used by TC TAs. Apart from this, since OP-TEE's TA loader and libraries used by TA's are open source, it should be straightforward to adapt SimTA for OP-TEE TAs.

Other platforms potentially in scope for SimTA are Samsung's TEEGris, Trustonic's Kinibi, and Qualcomm's QSEE. An indicator for these TEEs also using GP standards can be found in the publicly available list of members [8]. Besides Huawei, also Samsung, Trustonic, and Qualcomm are full members of GP and, according to GP's webiste, share a common goal of developing GP's specifications. Additionally, we could verify that concepts from the GP specification are being introduced into the Linux kernel for Qualcomm chipsets[2].

In summary, a broader technical study is necessary to evaluate if a system like SimTA can be extended to other TEEs used in production. But, it seems that the ecosystem is converging towards a common specification which is a good sign for the reusability of SimTA's components.

V. Implementation

Our SimTA prototype is implemented based on angr version 8.19.4.5 and Python 3.6.7. TC on the ARM-SoC-based target device runs in 32bit mode, resulting in the TAs also being 32 bit. We could extract the `storageTA` ELF directly from the firmware image. TC in this version does not make use of program headers and just uses the virtual address of a section provided in the ELF file to load it into memory. In order to get correct relocations for the `storageTA` binaries, we needed to slightly modify the way angr performs relocations.

As mentioned in Section IV-A, TAs follow a lifecycle that is usually implemented using a `while(1)`-loop which, at its start, receives data from the execution environment, and, at its end, returns data to the execution environment. Since we want to simulate this environment, we hook the functions interacting with it using angr's SimProcedures. We chose the first instruction of the lifecycle-loop to be the entry point into the binary. Before execution begins, we initialize the state angr is working on using the address space parameters derived from our analysis described in Section IV-C. This includes loading `storageTA` and globaltask as well as initializing stack, heap, and the shared memory region. In regard to registers, we observed that only two registers needed to be initialized with references to the current stack frame. These two references could easily be identified from the disassembly by looking at frame-pointer relative address assignments to those registers.

Moreover, we implemented a subset of the features provided by the GP TEE Internal Core API functions

[1]The names are taken from header fields within the firmware image. `task_storage` corresponds to `storageTA`.

[2]https://android.googlesource.com/kernel/msm/+/refs/heads/android-msm-marlin-3.18-pie/drivers/misc/qseecom.c

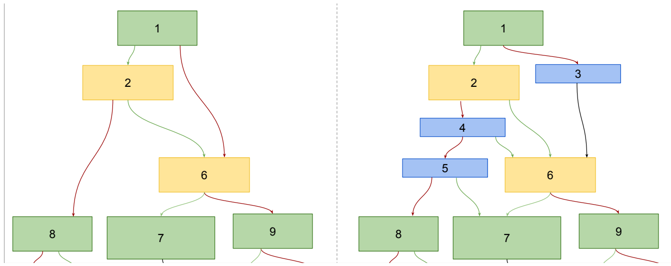

Fig. 3: This figure shows an extract of storageTA's control flow graph – unpatched version left and patched version right. The red, green, and black arrows indicate conditional and unconditional control flow transitions. Unchanged, changed, and new basic blocks are color-coded in green, yellow, and blue, respectively.

storageTA makes use of as SimProcedures. The functions implemented as SimProcedures are part of the Internal Core API's "Persistent Object Functions" group, "Data Stream Access Functions" group, and "Generic Object Functions" group. Additionally, we implemented a rudimentary heap implementation to support TEE_Malloc and TEE_Free. From reading the GP Internal Core API, we know what a particular function is supposed to do and for all functions used by storageTA we were able to implement SimProcedures that keep the TA in a useful state for the binary-diff analysis.

SimTA allows us to pass data consumed by the lifecycle functions from the MsgRcv()-function (Listing 1) into the storageTA process. This way we can pass data into the process from the perspective of a CA. By setting input data to symbolic values, we are able to perform advanced analyses, as we will elaborate on in the next section.

VI. Evaluation

For our evaluation, we generate a binary-diff of storageTA from firmware version VNS-L21C432B130 to version VNS-L21C432B160 and analyze the diff using SimTA. For ease of reference, we refer to the earlier version as sta and the later version as sta'.

First, we manually generate a binary-diff using Zynamic's BinDiff and filter for relevant patches. From BinDiff's "Matched functions" view we can identify matched functions containing changes. Using the *flow graphs* feature, we can view a color-coded control flow graph of both versions of a matched function side-by-side, as depicted in Figure 3. As a heuristic to filter relevant patches, we concentrate on blue-colored basic blocks that access attacker controlled input. These basic blocks represent new basic blocks in the patched version (*e.g.,* sta'). In order to do the filtering, we manually extract the blue basic blocks from BinDiff and perform an analysis with SimTA on sta' reveiling if attacker controlled input (*i.e.,* paramTypes and params as passed to the lifecycle functions in Listing 1) is accessed. If no user controlled input is accessed, we do not consider the change for further analysis.

Second, we run a further analysis using SimTA to examine the constraints introduced by the patches. From each filtered blue basic block (*i.e.,* basic blocks 4 and 5 remain), we look for the next subsequent basic blocks that have equivalent basic blocks in sta (*i.e.,* for basic block 5, these would be basic blocks 7 and 8). Since we know from the previous analysis which part of the attacker controlled input is accessed in basic block 5, we can selectively set this input to a symbolic value, and execute both versions of storageTA to a common basic block (*e.g.,* basic block 8). Lastly, we can view and compare the constraints in both versions using SimTA, and reason about the security impact of the introduced patches.

In the following paragraphs we elaborate on four analyses performed using SimTA.

One of the changed functions, strToByte(), is a hex-decode function and contains changes according to BinDiff. By passing the new basic blocks to SimTA, we can see that these basic blocks do not access user provided input and, therefore, we exclude it from further analyses.

A further function containing changes is the lifecycle function TA_InvokeCommandEntryPoint known from Section IV-A. Within the set of introduced basic blocks for this function, SimTA identifies basic blocks operating on the paramTypes argument. By selectively setting paramTypes symbolic and examining the constraints in both versions of storageTA, we unveil missing checks for all commands in this TA. This forgotten check is particularly critical for the FREAD and FWRITE commands, since these commands operate on user provided buffers to write to or read from. If we can control the location of these buffers, we have an arbitrary read or write primitive. It is not a surprise that we find a bug like this in storageTA, since these are exactly the primitives Stephens [20] used in his work. Using SimTA, we rediscovered the type-confusion underlying CVE-2016-8764.

Another interesting finding revealed by SimTA is an introduced length check for a buffer in the FOPEN command. This 1-day bug was discovered because of an additional length check of a buffer contained in the params argument. SimTA enabled us to identify this check and investigate the consequences of its absence in sta. For this bug it was necessary to investigate sta at a much later location than the actual check was introduced. Its severity becomes clear by looking at an unconstrained attacker controlled size and attacker controlled source buffer that get passed into a memory copy function having a fixed sized heap-based destination buffer. An attacker can corrupt the heap using this vulnerability and potentially gain code execution within sta. As should be clear from the context, this bug is fixed, but it is a perfect example for how SimTA supports binary-diff-guided analyses for TAs.

Our last discovery covers the authentication logic present in the TA_OpenSessionEntryPoint lifecycle function. Within this logic, an identifier of the calling CA and a constant client signature is checked. Aside of the fact that we can easily fake these inputs, since they are controlled by the user, we can see diverging constraints using SimTA in this logic. There are two allowed identifiers and from the introduced constraints it becomes clear that the signature for the identifier /system/bin/tee_test_store is not checked at all in sta. Presumably, this identifier is used for testing purposes and should not be part of code used in production. Using this identifier, we can get access to the core logic of sta without providing any client signature.

VII. Limitations

SimTA was not developed having an holistic emulation of the TEE in mind. It is rather an approach to have an effective and fast way to retrieve runtime information from two TAs – one being a patched version of the other – and facilitate the analysis of introduced patches.

The SimProcedures covering the GP TEE Internal Core API have been implemented using a least effort approach to have the minimal initialization done that is needed for the pursued analysis. However, we are confident that SimTA can be extended to other TC TAs or even other TEE implemenations that comply with GP specifications.

In summary, as can be seen from our evaluation, our method turned out to be effective and revealed two unknown 1-day vulnerabilities.

VIII. Outlook and Future Work

Our future work will extend SimTA to more TAs. Besides TC, there are TEEGris and OP-TEE that are known to follow GP specifications. From recent Linux kernel sources for Qualcomm SoCs, we can also see that GP concepts have been introduced. Consequently, the approach chosen by SimTA might be viable for all major TEEs.

We envision a collection of all security-critical bugs known in TAs and a cross-vendor large-scale study of security-related patches in order to distill the nature of TA security holes and provide effective mitigations.

One of the problems arising though, is vendors locking down their systems. This leaves security researchers with an interest in TEEs with many hoops to jump through before they can actually analyze their target.

IX. Conclusions

In this paper, we presented our ongoing research in finding 1-day vulnerabilities in TAs. We developed SimTA, an angr-based solution to mimic the TEE runtime for TAs found on Huawei devices. Using SimTA, we replicated a known bug in Huawei's `storageTA` and revealed two additional critical issues that have been silently patched by the vendor.

We hope to spark interest for TEE research and provide our tools to take some hurdles for other security researchers to get started.

Acknowledgment

This research was supported by the German Federal Ministry of Education and Research as part of the Software Campus project.

References

[1] ARM, "Arm security technology: Building a secure system using trustzone technology," http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf, 2008, accessed: 2019-08-28.

[2] A. Atamli-Reineh, R. Borgaonkar, R. A. Balisane, G. Petracca, and A. Martin, "Analysis of trusted execution environment usage in samsung knox," in *Proceedings of the 1st Workshop on System Software for Trusted Execution*, 2016, pp. 1–6.

[3] G. Beniamini, "Cve-2015-6639 exploit," https://github.com/laginimaineb/cve-2015-6639, 2016, accessed: 2019-08-28.

[4] G. Beniamini, "Exploring qualcomm's secure execution environment," https://bits-please.blogspot.com/2016/04/exploring-qualcomms-secure-execution.html, 2016, accessed: 2019-08-28.

[5] G. Beniamini, "Qsee privilege escalation vulnerability and exploit (cve-2015-6639)," https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, 2016, accessed: 2019-08-28.

[6] G. Beniamini, "Trust issues: Exploiting trustzone tees," https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html, 2017, accessed: 2019-08-28.

[7] GlobalPlatform, "Tee internal core api specification," https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/, 2019, accessed: 2019-12-05.

[8] GlobalPlatform, "Globalplatform current members," https://globalplatform.org/current-members/, 2020, accessed: 2019-01-15.

[9] J. Guilbon, "Attacking the arm's trustzone," https://blog.quarkslab.com/attacking-the-arms-trustzone.html, 2018, accessed: 2019-08-28.

[10] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, and M. Grace, "Partemu: Enabling dynamic analysis of real-world trustzone software using emulation," in *Proceedings of the 29th USENIX Security Symposium (USENIX Security 2020) (To Appear)*, August 2020.

[11] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vtz: Virtualizing ARM trustzone," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 541–556.

[12] D. Komaromy, "Unbox your phone - part i." https://medium.com/taszksec/unbox-your-phone-part-i-331bbf44c30c, 2018, accessed: 2019-08-28.

[13] D. Komaromy, "Unbox your phone - part ii." https://medium.com/taszksec/unbox-your-phone-part-ii-ae66e779b1d6, 2018, accessed: 2019-08-28.

[14] D. Komaromy, "Unbox your phone - part iii." https://medium.com/taszksec/unbox-your-phone-part-iii-7436ffaff7c7, 2018, accessed: 2019-08-28.

[15] L. Limited, "Open portable trusted execution environment," https://www.op-tee.org/, 2020, accessed: 2020-01-15.

[16] S. Makkaveev, "The road to qualcomm trustzone apps fuzzing," https://research.checkpoint.com/2019/the-road-to-qualcomm-trustzone-apps-fuzzing/, 2019, accessed: 2019-11-30.

[17] NIST, "Cve-2016-8764," https://nvd.nist.gov/vuln/detail/CVE-2016-8764, 2017, accessed: 2019-08-28.

[18] D. Shen, "Attacking your Trusted Core," https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android.pdf, 2015, accessed: 2019-11-28.

[19] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[20] N. Stephens, "Behind the pwn of a trustzone," https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose, 2017, accessed: 2019-08-28.

[21] N. Stephens, "Bypassing huawei's fingerprint authentication by exploiting the trustzone," https://www.youtube.com/watch?v=MdoGCXGHGnY, 2018, accessed: 2019-08-28.

[22] A. Tarasikov, "Qemu teegris usermode," https://github.com/astarasikov/qemu/tree/teegris_usermode, 2019, accessed: 2019-11-30.