Memory Corruption Attacks within Android TEEs: A Case Study Based on OP-TEE

Fabian Fleischer
Friedrich-Alexander University Erlangen-Nürnberg
fabian.fleischer@fau.de

Marcel Busch
Friedrich-Alexander University Erlangen-Nürnberg
marcel.busch@fau.de

Phillip Kuhrt
Friedrich-Alexander University Erlangen-Nürnberg
phillip.kuhrt@fau.de

ABSTRACT

Many security-critical services on mobile devices rely on Trusted Execution Environments (TEEs). However, due to the proprietary and locked-down nature of TEEs, the available information about these systems is scarce. In recent years, we have witnessed several exploits targeting all major commercially used TEEs, which raises questions about the capabilities of TEEs to provide the expected integrity and confidentiality guarantees. In this paper, we evaluate the exploitability of TEEs by analyzing common flaws from the perspective of an adversary. We provide multiple vulnerable TEE applications for OP-TEE, a reference implementation for TEEs, and elaborate on the steps necessary for their exploitation on an Android system. Our vulnerable examples are inspired by real-world exploits seen in-the-wild on commercially used TEEs. With this work, we provide developers and researchers with introductory knowledge to realistically assess the capabilities of TEEs. For these purposes, we also make our examples publicly available.

CCS CONCEPTS

• **Security and privacy** → **Trusted computing**; **Mobile platform security**; *Software security engineering*.

KEYWORDS

TEE, Exploitation, Trusted Applications, ARM TrustZone

ACM Reference Format:
Fabian Fleischer, Marcel Busch, and Phillip Kuhrt. 2020. Memory Corruption Attacks within Android TEEs: A Case Study Based on OP-TEE. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3407023.3407072

1 INTRODUCTION

Trusted Execution Environments (TEEs) are the backbone of security architectures for a vast majority of modern mobile devices. Security-critical features like mobile payment, user authentication, or digital media protection, leverage the TEE to provide a level of security beyond the capabilities of the regular Operating System (OS) (*e.g.,* Android or iOS). Despite being critical for a device's security, knowledge is sparse about the systems that power the root-of-trust of the devices we use daily.

ARES 2020, August 25–28, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-8833-7/20/08…$15.00
https://doi.org/10.1145/3407023.3407072

Reflecting on recent history, we can see that all major commercially used TEEs failed. For instance, the TEE implementations of popular devices like the Google Nexus 6 [10], the Samsung Galaxy S7 Edge [11], or the Huawei P9 [30] were successfully compromised. Veritable incidents like these leave researchers and industry professionals in the dark: How is it even possible to break into a TEE, and if so, what would be the consequences? To fully understand and begin to answer these questions, these exploits need to be recapitulated. Unfortunately, for multiple reasons, it is exceedingly difficult to replicate real-world exploits targeting TEEs. We shall detail on the top three reasons, applicable for most cases. Firstly, reproduction requires the availability of the corresponding hardware platform. Secondly, a specific vulnerable firmware version must be installed on the target device. Thirdly, a deep level of expertise is necessary to reproduce a real-world exploit on a platform of a particular vendor.

In this paper, we fill this knowledge gap by providing multiple vulnerable TEE applications and elaborate on the steps an adversary would take to exploit these flaws. Moreover, we base this discussion on OP-TEE, an open-source reference implementation for ARM TrustZone (TZ)-based TEEs. Using OP-TEE allows us to overcome the three major problems of real-world exploits, mentioned above when encountering real-world exploits. Beginning firstly by using virtual hardware (*e.g.,* an emulator), we overcome the hardware dependencies. Next, we can compile and execute our own examples of vulnerabilities, which are not dependent on specific firmware versions. From this, we can focus on the most important aspects of the vulnerabilities and the root causes for their exploitation without the need to reverse-engineer vendor-specific proprietary firmware components.

In particular, our contributions are the following:

- We provide multiple real-world examples of vulnerable TEE applications and elaborate on their exploitability.
- Our discussion about the exploitability of TEEs is based on open-source software and our examples are publicly available as an educational tool[1].
- We put each step of the exploitation process into an understandable context and pinpoint how commercially used TEEs were affected by certain design choices.

2 BACKGROUND

The underlying hardware of a computing system dictates its trusted computing capabilities. Since modern mobile devices predominantly

[1] https://github.com/teesec-research/optee_examples

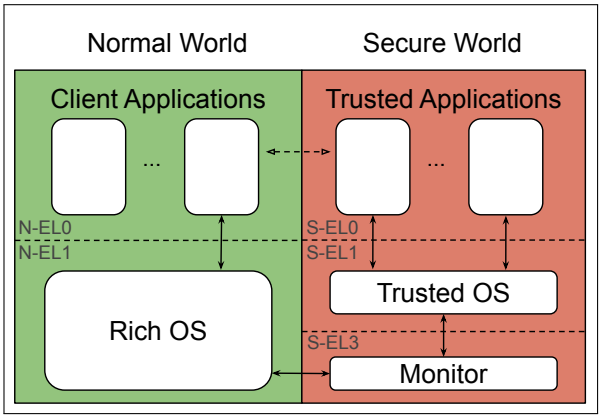

Figure 1: ARM TrustZone splits the software architecture of mobile devices into two states, the SW and the NW.

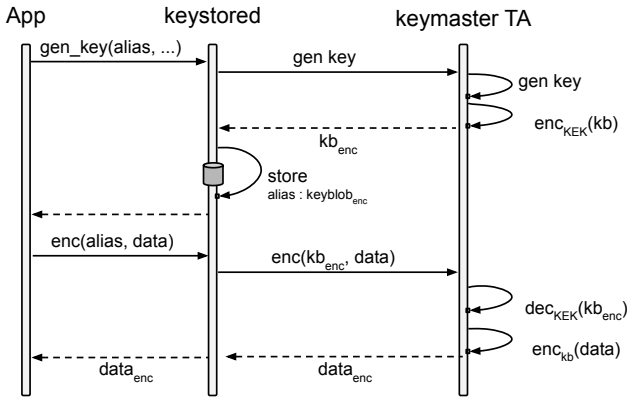

Figure 2: Generation and usage of cryptographic keys within the Android keystore system.

use ARM chipsets, our work concentrates on the capabilities provided on ARM-based systems. Thus, this section gives an overview of software architectures as used on modern mobile devices.

2.1 ARMv8-A and ARM TrustZone

ARM TZ on the ARMv8-A ISA partitions the software architecture of a mobile device into two states, the Secure World (SW) and the Normal World (NW) [6], as illustrated in Figure 1. Each of these states has up to four privilege levels, called Exception Levels (ELs). The NW is hosting a traditional Rich Operating System (RichOS) such as Android on N-EL1 and its userland applications on N-EL0, whereas the SW is hosting a small Trusted Operating System (TrustedOS) on S-EL1 and its userland on S-EL0. In this work, we refer to NW and SW userland applications as Client Applications (CAs) and Trusted Applications (TAs), respectively.

The current state of a CPU core is indicated by an additional Non-Secure (NS) State bit transmitted via the Secure Configuration Register (SCR) [4] over the AXI system bus to all connected hardware components [3]. This mechanism enables system designers to introduce TZ-aware peripherals that can exclusively be accessed by software running in the SW. A common use case on mobile devices is to make the fingerprint sensor only available to the SW to prevent leakage of fingerprint images to the untrusted NW.

To protect the SW from the NW, the memory controller is usually configured to deny all accesses to memory hosting SW code and data. This hardware-enforced isolation does not apply vice-versa, making the SW the higher privileged context. In order to transition from one world to the other, an OS has to call the Monitor (S-EL3) using a privileged instruction (*e.g.,* `smc`).

A common Android system service utilizing TZ capabilities is the keystore system [19]. It allows for the generation and storage of cryptographic keys within the TEE. Additionally, it provides an Application Programming Interface (API) to carry out cryptographic operations using these TEE-backed and export-protected keys. From an app's perspective, a typical interaction with the keymaster TA (*i.e.,* the TEE-based portion of the keystore system) looks as depicted in Figure 2. First, an app requests the generation of a key from the keystored, which is a system service running in the background. The keystored is the CA in this use case, responsible for forwarding the request to the keymaster TA. Then, the request is not directly passed to the TA but must be passed through the RichOS, Monitor, and TrustedOS before it reaches its destination. Next, the keymaster TA generates the key as requested and encrypts the resulting key blob (kb) using a key encryption key (KEK). Lastly, the keystored only receives the encrypted key blob and stores it using the alias initially provided by the app. Using this alias, the app can refer to this key in succeeding cryptographic operations. Figure 2 illustrates how an encryption operation using the freshly generated key would be carried out.

2.2 Attack Surface

Cerdeira *et al.* [15] point out that the Trusted Computing Bases (TCBs) of TEEs used on mobile devices are excessive. TAs account for the largest portion of the TCB in commercially used systems. Additionally, most of the logic provided by TAs is accessible from N-EL0 (*e.g.,* NW user-mode) using the communication infrastructure provided by the RichOS. The TrustedOS or the Monitor, in contrast, are not directly accessible from this context. Therefore, we concentrate on flaws in TAs in this research.

As mentioned in Section 2.1 and depicted in the keystore system example in Figure 2, third-party apps do not directly interact with TAs. Instead, on Android systems, it is usually a system service having the capabilities to access the RichOS interface to the TEE that carries out the interaction. Consequently, an adversary would at least need a two-stage privilege escalation to get into the TEE. First, a privilege escalation to a system service with proper capabilities (*i.e.,* as shown by Beniamini [9] and Stephens [30]) and, second, another exploit to gain code execution within any TA. In this research, we focus on the second of these two stages.

All manufacturers of commercially used TEEs use programming languages with explicit memory management (*e.g.,* C and C++) to develop TAs. Inherent to this choice of programming languages is the risk of memory safety violations. Lapid *et al.* [22] discussed the consequences of a stack-based buffer overflow originally found by Beniamini [11] in an OTP TA that was shipped with Samsung devices. This TA implements a mechanism to generate one-time

passwords on the device. As Lapid *et al.* point out, using the buffer overflow to gain code execution within the TA is fatal because it has the same capabilities as the keymaster TA in this implementation and, therefore, can access the key-encryption key (see Figure 2).

A further example of a memory-corruption bug in a secure storage TA shipped to Huawei devices is mentioned by Machiry *et al.* [24] and demonstrated by Stephens [30] (*e.g.,* CVE-2016-8764 [26]). The problem in this TA is a type-confusion bug that lets an attacker read from or write to arbitrary memory within the virtual address space of the TA. Using this flaw, Stephens demonstrates how to manipulate the fingerprint TA to accept any fingerprint. Since he has access to the fingerprint TA's memory, he could have also leaked sensitive fingerprint data of users.

As Busch and Dirsch [13] later found, this very same secure storage TA had a heap-based buffer overflow vulnerability, which was already fixed by the vendor at the time of the finding. Heap metadata attacks are one of the primary ways for attackers to exploit memory corruption vulnerabilities [16] and as dangerous as stack-based buffer overflows.

2.3 Peculiarities of AArch64

Since both x86 and x86-64 processors follow CISC principles with variable-length instruction opcodes, many instructions without memory addresses can be encoded in one byte. When using Return Oriented Programming (ROP) for exploitation, this can be an advantage for the attacker, since the one-byte opcodes of the instruction set include instructions like *ret*, popping values from the stack into a register, and jumping to the specified address [25]. Since instructions do not need to be aligned, it is, in general, easier to find suitable ROP gadgets. In contrast, AArch64 uses four-byte instructions that have to be aligned. Moreover, since the return instruction, *ret* on AArch64, reads the return pointer from a register, already eight bytes of correct instruction opcodes are required at the end of every ROP gadget. In general, this reduces the probability of finding suitable ROP gadgets compared to the x86 architecture.

Due to the function call return pointer being passed via the link register, its placement on the stack is optional, and not necessary for leaf functions, for functions only invoking other functions at the end of their own code, or for functions inlining all inferior functions' code. This causes only a few functions' epilogues to be suitable as a ROP gadget. For AArch64 the Procedure Call Standard for the ARM 64-bit Architecture (AAPCS64) specifies how data is arranged in the stack frame. In contrast to x86, AArch64 stores the link register last on the stack. Therefore, a stack-based buffer overflow requires at least two function returns before the overwritten return instruction pointer is popped from the stack. The stack layout on AArch64 is depicted in Figure 3 conforming AAPCS64.

3 METHODOLOGY

Our research focuses on the Open Portable Trusted Execution Environment (OP-TEE), an open-source reference implementation for TZ-based TEEs. We implemented multiple vulnerable TAs on OP-TEE that are inspired by flaws seen "in the wild" within commercially used TEEs [2, 10, 11, 13, 30]. These flaws include stack-based buffer overflows, type-confusion bugs, and heap corruption vulnerabilities.

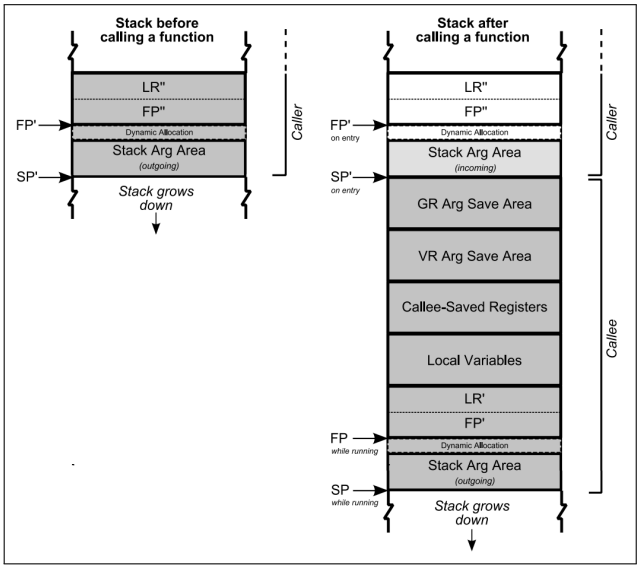

Figure 3: Visualization of the ordering of data within stack frames on AArch64, as shown on the AAPCS64 standard [7, Image "Example stack frame layout"]. Higher addresses are on top.

Having these vulnerable TAs, we take the perspective of an adversary and systematically assess OP-TEE to escalate a memory corruption to more powerful exploitation primitives. For this purpose, we selectively review OP-TEE's source code [23] and analyze the compiled binaries of our vulnerable examples. Section 4 provides relevant insights on OP-TEE that we distilled from reviewing OP-TEE's source code and build system. Section 5 and Section 6 contain our results from statically and dynamically analyzing binaries of TAs.

4 ARCHITECTURE OF OP-TEE

OP-TEE is the most favorable choice for TEE-related experiments on Android. It can run on an emulator as well as on comparatively inexpensive development boards. The latter option allows for a setup containing Android, OP-TEE, and the ARM Trusted-Firmware [20], which comes close to the software architectures found on commercially available devices. Furthermore, it is actively maintained and well documented.

In this section, we discuss several aspects relevant to an adversary trying to exploit memory corruption vulnerabilities in OP-TEE TAs.

4.1 TA Instance and Session Model

Depending on the TA's configuration, OP-TEE can run a TA multiple times in parallel, creating multiple *instances* of the same TA. These instances behave similarly to processes on other operating systems, *e.g.,* by assigning dedicated writable memory regions to each of the instances to avoid influence on each other. The communication of a client application running in the NW with a TA is implemented as a session model. This model is widespread among different TEE implementations and many vendors are compliant

with the APIs specified by GlobalPlatform [17, 18] (so is OP-TEE). A client application can open a session, request the execution of commands within an established session, and close the session. Differing from RichOSs like Linux or Windows, the lifetime of a TA instance within OP-TEE is controlled by OP-TEE according to the TA's configuration properties. These options determine whether multiple sessions are handled by the same or by different TA instances and whether resources are fully deallocated upon release of the last active session. Within an instance, only a single thread can run at any point in time, ensuring that there is no parallel execution requiring error-prone synchronization and locking mechanisms. An adversary needs to be aware of the instance and session model of TAs in order to set the target up to accept commands.

4.2 Communication Path from Android Application to TA

As already mentioned in Section 2, a CA makes use of the communication infrastructure exposed by the RichOS to interact with a TA. On OP-TEE the `libteec` library is used as an abstraction for the low-level `ioctl` interface of the TEE driver that implements the NW side of the communication infrastructure. Either the `libteec` interface or the `ioctl` interface of the TEE driver are the primary entry points to launch an attack against a TA.

Note, in order to pass buffer contents to a TA, a CA passes a pointer from within its virtual address space to the RichOS. Since NW and SW have different page tables, the RichOS and TrustedOS have to use a convention about how to translate addresses from their user-mode applications to each other. On OP-TEE, the RichOS copies the buffer contents to a dedicated shared-memory region, which is accessible from both worlds. It then passes the physical address of the copied buffer to the TrustedOS. The shared memory infrastructure is a difficult component in TEE-backed systems, and almost all vendors struggled with correct implementations in the beginning [24]. This aspect of TEEs will become relevant later in Section 5.2 where we elaborate on type-confusion bugs.

4.3 Exiting TAs

A memory corruption most likely results in a crash of the target. An adversary is usually interested in properly exiting the target to use subsequent interactions with the same TA instance in an exploit.

An interaction with a TA consists of a sequence of command invocations. During a command invocation, an OP-TEE TA is entered via its `__utee_entry()` function and left using a special system call that can signal the state of the TA to the TrustedOS. A proper exit indicates a *TEE_SCN_RETURN* to the OP-TEE kernel, which results in the target being able to receive further commands.

4.4 OP-TEE TA Libraries and Tools

OP-TEE enforces a write-never-execute policy within its TAs, meaning that a writable page can never be executed. This policy forbids the placement of shellcode in the target's address space. Consequently, an adversary has to launch code-reuse attacks (*e.g.,* Return Oriented Programming).

Code-reuse attacks are dependent on the executable code that is already mapped in the virtual address space of the TA. OP-TEE provides many libraries to facilitate the life of TA developers. These libraries include *libutils*, *libmpa*, *libmbedtls*, and *libutee*, and implement commonly used features of ISO C, multi-precision integer representations, and cryptographic functions. Since those libraries are mapped in the address space of the TAs, they are relevant for code-reuse attacks.

Furthermore, to load a TA OP-TEE uses a loader called `LDELF`. `LDELF` is a module executed within the virtual address space of the TA, which is responsible for loading the TA ELF file. One may note that the `LDELF` binary is not unmapped and can also serve as a target for code-reuse attacks.

4.5 Memory Layout and ASLR

An effective code-reuse attack requires knowledge about mapped memory contents and their locations. Therefore, we review the common memory layout of OP-TEE TAs.

Typical mitigation against code-reuse attacks is Address Space Layout Randomization (ASLR). The essential idea is to randomize the placement of code within the virtual address space to prevent an adversary from determining the location of reusable code fragments. Although OP-TEE provides ASLR on AArch32, it is not supported on AArch64 builds, which is our target. According to Cerdeira *et al.* [15], all commercially used TEEs lack of proper ASLR and, thus, OP-TEE serves as a realistic environment for experiments.

Generally, the Memory Management Unit (MMU) abstraction of OP-TEE always assigns the lowest-possible address to new memory mappings within an appropriate range and the padding configured, if not instructed with a specific address to be used, resulting in a predictable stacking behavior. A schematic of OP-TEE's memory mappings is depicted in Figure 4. Reviewing OP-TEE's source code reveals that no padding nor specific address for new memory regions is configured, which causes mappings to generally be placed next to each other without any guard pages.

The QEMUv8 target of OP-TEE uses an MMU abstraction layer that, by default, starts mapping memory at address 0x4000 0000. As depicted in Figure 4, the first mapping in the address space is the OP-TEE kernel itself, which is not accessible from a TA running in S-EL0. All further mappings are visible from a TA's perspective.

Following kernel memory and a guard page, OP-TEE successively maps the *LDELF* binary segments (stack, text, data, and heap) to 0x4000 4000. The loader then loads the TA segments in the address space directly after the loader itself without a guard page. Since all page sizes of LDELF and the TA are deterministic, the memory addresses of the pages in the virtual address space of the TA are deterministic as well. The loader also places the stack of a fixed size (defined at compile time) right after the TA segments, which makes those addresses deterministic too. Next, the loader maps additional pages after the stack, such as additional ELF files, empty pages for further use, and parameters of the current call to the TA instance. Around parameters, but not in between, the loader adds one guard page.

The heap of a TA is part of the *.bss* segment, its size is determined at compile-time. Although both the OP-TEE kernel and the heap allocator (BGET) offer options to allocate additional memory at runtime, this feature is currently not implemented.

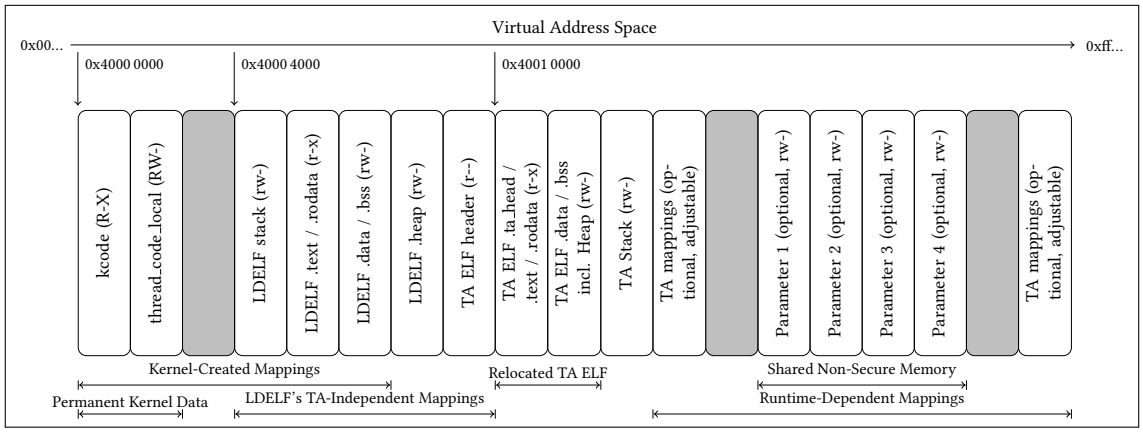

Figure 4: Visualization of the memory layout as seen from a TA. Each block represents a separate region, as managed by the MMU hardware abstraction layer. Gray blocks represent guard pages. Usual permissions of the regions are represented as read/write/execute flags, privileged access flags are annotated in uppercase, where user-mode access is denied.

The absence of ASLR and the deterministic placement of all memory mappings make addresses in the virtual address space of OP-TEE TAs predictable for an adversary. These circumstances facilitate exploit development tremendously because all locations required for code-reuse attacks are known.

5 CONTROL FLOW HIJACKING

In this section, we describe several ways to hijack the control flow of a TA using memory corruption vulnerabilities. The chosen vulnerabilities are inspired by memory corruptions from TAs that were shipped with commercially available mobile devices. For each of the vulnerabilities, we implemented example TAs for OP-TEE and studied their exploitability. In order for other researchers, vendors, and educators to use our insights, we open-source our vulnerable OP-TEE TAs, including their corresponding exploits on https://github.com/teesec-research/optee_examples.

5.1 Stack-based Buffer Overflows

A stack-based buffer overflow is a stereotypical example for control flow hijacking, given a missing length check or a suitable integer overflow. By default, the build system for OP-TEE TAs does not generate any protection against stack-based buffer overflows (*e.g.,* stack canaries). As mentioned in Section 2.3, overwriting a stack-based buffer will not overwrite the return instruction pointer of the declaring function, but of its caller. This situation is also depicted in the stack layout shown in Figure 5. Thus, depending on the context of the overflow, new stack contents must be carefully crafted to avoid restoring unexpected values into variables saved on the stack, risking a crash within the parent function before the control flow can be hijacked.

As shown in Figure 4, the stack is located above the heap, without any guard page separating these memory regions. Consequently, if a heap-based buffer overflow occurs, an adversary is also able to overwrite the stack. Note, the exploit must take care of the overwritten memory contents before reaching the stack to avoid crashes.

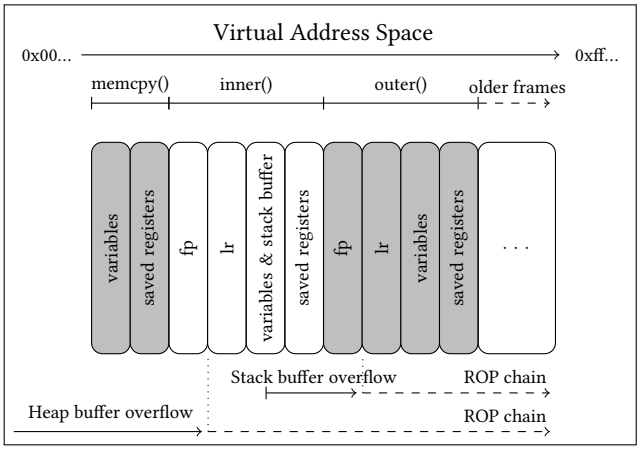

Figure 5: Using an arbitrarily long `memcpy()` either on the stack or the heap, an adversary can overwrite return instruction pointers on the stack (`lr`) within OP-TEE TAs. Leaf functions (`memcpy()` in this case) do not store the return instruction pointer on the stack.

A heap-based buffer overflow also has one advantage: Since the buffer overflow starts overwriting memory below the innermost stack frame, using the overflow might be able to overwrite the innermost return instruction pointer of the stack, instead of depending on the location of the overwritten buffer on the stack. This memory layout can reduce the risk of corrupting data on the stack by hijacking the control flow earlier, and overwriting less, if no ROP chain is necessary, as shown in Figure 5.

The stack-based and heap-based buffer overflows, as explained in this section, were present on TAs used on mobile devices purchased by millions of customers. For instance, a similar stack-based buffer overflow vulnerability was exploited within an OTP TA on the popular Samsung Galaxy S7 Edge, as already mentioned in Section 2.2.

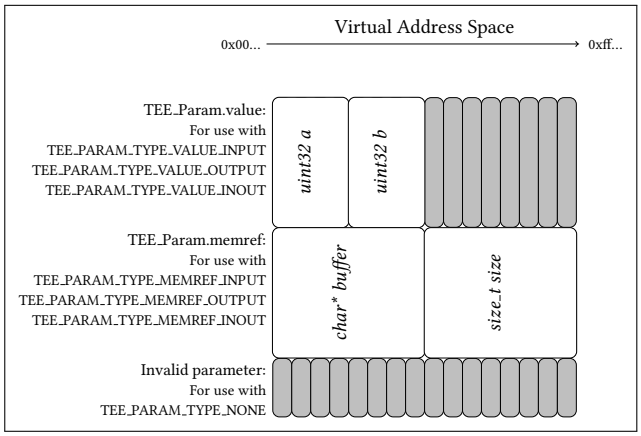

Figure 6: Comparison of the possible memory layouts for the content of the union type TEE_Param dependent on the parameter's type. Gray boxes are bytes of unused memory.

Furthermore, the memory layout that allows overflowing from the heap into the stack (because of missing guard pages) is similar to the heap-based buffer overflow found within the secure storage TA on the popular Huawei P9 Lite (also mentioned in Section 2.2).

In our examples available on https://github.com/teesec-research/optee_examples we provide the `vuln` TA, which has a stack-based buffer overflow vulnerability. It can be used for the reproduction of our analysis and further inspection. The `vuln` TA also contains a type confusion bug, which is evaluated in the following Section 5.2.

5.2 Type-Confusion Bugs

As pointed out by Machiry *et al.* [24], a lack of type validation enables an adversary to bypass pointer sanitization mechanisms if pointers can be passed as non-pointer types; which can also be observed in the wild. An example of an exploit using a type-confusion bug was demonstrated by Stephens [30]. Since the implemented GlobalPlatform API uses union types and associated type fields for its parameters, with either pointers or integers present in the same memory, OP-TEE TAs are potentially vulnerable to this type of attack.

Although the union types are similar, the sanitization during conversion, as described in Section 4.2, is not performed after copying of the data. Instead, OP-TEE treats the union types and less similar intermediate types as completely different data structures, and only selectively copies and passes appropriate contents. Figure 6 shows the memory layouts for the different parameter types based on our code review. With the union type's data layout, a type confusion allows us to pass two integer values treated as a pointer, which depending on whether an output or input buffer is expected can lead to both, data leaks and data manipulation.

In this situation, however, there is an additional size field. If the parameter contains a struct with constant size, a string, or binary data in OUTPUT or INOUT buffers, usually a size check can be expected, if it has not been omitted too by mistake. Otherwise, for binary data with varying length read from INPUT buffers, the size itself might determine the amount to be read. Thus, it is important to accomplish at best a controllable, at least a non-zero value to be present in the size field.

Favorably, during the last conversion step within libutee, for both command invocation and session initialization, the parameters are placed on the stack and then passed to the TA code by reference. Our dynamic analysis reveals that this stack position is constant, and the stack memory is not wiped before being selectively written.

As a result, the value of the size field of the last call passing a memory reference will still be present during later calls. This allows an attacker to invoke the TA with an appropriately sized buffer to set the size. Whether this call is successful or fails with an error code is irrelevant and does not change the outcome, as long as it does not crash the TA. Then, the attacker can invoke a command using a parameter as a buffer without checking its type, and pass a pointer declared as an integer value to the TA, circumventing pointer sanitization. Since there is no ASLR present, the pointer to be passed can be calculated based on the memory layout illustrated in Section 4.5.

The vulnerable command implementation will then either read and process input data expected to lie in shared memory, but actually reading its own private memory, potentially exposing its contents. Alternatively, the implementation might write at the specified location to return a result via an output buffer instead of modifying its own private memory.

The applicability of such an attack may depend on the complexity of the data processed by a TA. A simple vulnerable echo command copying data between two buffers could be an easily exploitable example, while commands with a single buffer, only exposing or injecting data via integer parameters, after multiple calls or after complex processing of the data read or to be written may increase the effort necessary for such an exploit.

Examination of the kernel code responsible for calling a Pseudo Trusted Application (PTA) exposes a similar situation. Again parameters are passed to the PTA without wiping unused parameter fields and only wiping completely unused parameters. This results in similar options for the exploitation of the kernel from a malicious TA, given the lack of type checks within PTA code.

5.3 Heap-Corruption Bugs

Corresponding to heap exploitation on Linux [29], TAs potentially contain similar vulnerabilities. Since a client can request the execution of multiple commands by the TA within the same session, we have a good precondition for heap exploits. This exposes the risk of double-free and use-after-free vulnerabilities. Of course, the exploitability highly depends on the heap implementation and exploit primitives provided by the individual TA. For OP-TEE, the fact that the heap is not subject to ASLR, is beneficial for exploitability.

As mentioned in Section 4.5, OP-TEE uses the BGET heap implementation [31]. Using a double-free vulnerability, it is possible to manipulate BGET's free list to gain an arbitrary read and write primitive. Furthermore, the merging process of different chunks can be used to overwrite the fields of other chunks. As heap exploits highly depend on the heap implementation, we do not discuss further technical details about BGET in this paper and refer the

interested reader to our open-sourced examples. We provide a detailed example TA with a double-free heap vulnerability (`heap` TA) and an example exploit open-sourced with this work.

6 ARBITRARY CODE EXECUTION

Once an attacker hijacked the control flow, usually the goal is arbitrary code execution. We evaluate two techniques for leveraging vulnerabilities to the execution of arbitrary code. First, we elaborate on the injection of shellcode into the target TA. We then focus on the applicability of Return Oriented Programming (ROP) in the context of TAs.

6.1 Shellcode Injection

Given control over the program counter and the stack, it is not possible to directly insert shellcode for arbitrary code execution. Similar to the Never Execute (NX) flag on x86-based systems, ARMv8-A AArch64 supports an Execute Never (XN) flag. OP-TEE uses this flag by default, thus, memory regions are either writable or executable but never both.

In theory, it is possible to have write-executable memory within OP-TEE TAs. This configuration can be accomplished by modifying the linker script shipped with the TA software development kit. However, additional mitigation for `wx` memory used by OP-TEE is the system-wide Write-Execute-Never (WXN) flag [5] within the CPU control register (*SCTLR_EL1*). This flag is set by OP-TEE (*e.g.,* configuration option *CFG_CORE_RWDATA_NOEXEC*) at an early stage and is never disabled again. When switching between SW and NW, the Monitor can be expected to switch and restore the appropriate control registers for EL1 for both the OP-TEE and the Linux environment. This will cause all memory regions with write permissions to be treated as XN [5], effectively preventing any code execution on writable memory, independent of the flags, which OP-TEE configured in the MMU page tables.

A scenario as described by Stephens [30], where `rwx` memory is part of each TA, is therefore only possible with heavy modifications of OP-TEE's default configuration. Instead, an adversary needs to leverage the existing code to launch a code-reuse attack such as ROP. Beniamini was the first to make a TA exploit based on the ROP technique publicly available [10]. His target was a Digital Rights Management (DRM) TA, called Widevine, which was distributed with the Google Nexus 6 mobile phone.

6.2 Return-oriented Programming (ROP)

A ROP chain that can call arbitrary functions and copy memory contents of a TA into the memory region shared with the NW can be considered an arbitrary code execution. This ROP chain necessarily needs ROP gadgets that can set function parameters (*e.g.,* `x0` – `x7`). Thus, to evaluate the viability of ROP, we analyze the availability of ROP gadgets and jump gadgets in OP-TEE TAs. Additionally, we study the primitives found in the TA and LDELF binaries.

6.2.1 ROP gadgets within TAs. To get an overview of available ROP gadgets in an OP-TEE TA, we examine the commonly mapped code of a TA using tools like ropper [28] and xrop [14]. The result shows that there are only little useful gadgets available. Many results returned by these tools are not usable as a gadget since they include all sequences of instructions ending with a `ret`. As discussed earlier in Section 2.3, leaf functions do not pop the return instruction pointer from the stack, rendering the gadget unusable for an attack. Further results contain conditional jumps, function calls, or memory accesses that depend on values determined during runtime, which often prevents their use in a ROP chain. In general, gadgets that set function parameters (*e.g.,* registers `x0` to `x7`) are rare, while gadgets that restore callee-saved registers (*e.g., x19* to *x28*) are more common.

Parameter and Result Register x0. A common gadget found allows us to set *x0*, which is used as the first parameter of a function or a function return value. The original purpose of the following gadget is to return a variable in the *x19* register by moving its value to the *x0* register:

AArch64-Assembly

```
mov x0, x19;
ldp x19, x20, [sp, #0x10]; # relative addressing
ldp x29, x30, [sp], #0x20; # manipulation of SP prior to use
ret;
```

By calling this gadget twice, we can control the value of the `x0` register. This or similar gadgets tend to be present in TAs since it is used to return the value of a variable stored in `x19`. However, for further 64-bit parameter registers, there are not necessarily suitable ROP gadgets available.

Further Parameter Registers x1 – x7. Thinking from a compiler's perspective, we can explain the lack of parameter-setting gadgets in the following way: the parameter registers *x1 – x7* are used as scratch registers except for subroutine calls. Consequently, the compiler is free to use them for the storage of variables in between subroutine calls, effectively mandating their use as caller-saved registers. Due to this circumstance, these registers will usually not be popped or otherwise intentionally set within the function epilogue. According to the AAPCS64, these registers are intended for result values, too, but this only manifests when returning result values larger than 64 bit, which is rare.

6.2.2 Jump-oriented Programming Gadgets. To increase the available amount of gadgets, we can use the so-called "functional gadgets" from the Jump-oriented Programming technique [12]. This technique allows us to regain control over the control flow by setting register values that will be jumped to.

An analysis of our example TAs reveals the presence of such gadgets. The following gadget moves data from callee-saved registers to parameter registers *x0 – x3* and then jumps to one of the callee-saved registers.

AArch64-Assembly

```
mov    x2, x19
mov    x1, x20
mov    x0, x21
blr    x25
```

In combination with another (more common) gadget that loads stack values into callee-saved registers, we have an effective ROP chain to set the first three parameters. That second gadget looks as follows.

AArch64-Assembly

```
ldp x19, x20, [sp, #0x10];
ldp x21, x22, [sp, #0x20];
ldp x23, x24, [sp, #0x30];
ldp x25, x26, [sp, #0x40];
ldp x29, x30, [sp], #0xe0;
ret;
```

6.2.3 GCC Register Usage. For more complex TAs, the occurrence of helper functions that unpack values from a referenced structure and pass them to a leaf function is likely. Instead of only relying on ROP gadgets, such a helper function could help to get a primitive for setting further parameter registers. Examples for such functions could be helper functions for arithmetic operations:

Exemplary arithmetic function to test for unchanged registers

```
uint64_t animate(struct animation_definition* animation)
{
    return calc_animation(animation->parameter1, /* seven more
        ↪ parameters */);
}

uint64_t calc_animation(uint64_t parameter1, /* seven more
    ↪ parameters */)
{
    // do calculation
    return /* some arithmetic result */;
}
```

Alternatively, debugging helper functions invoking functions such as `printf()` might be of use, since `printf()` shifts and then passes its parameters to `vsnprintf()`, and then only overwrites a few of these when calling `puts()` and the `syscall`. These function are mostly side-effect free and could look like this:

Exemplary print function to test for unchanged registers

```
void debug_print_animation(struct animation_definition* animation)
{
    printf("animation values:%llu, <repeat six times>", animation->
        ↪ parameter1, /* six more parameters */);
}
```

In our investigation, we found that such functions usually overwrite the parameter registers. The reason for the overwritten registers is that GCC's heuristic for register assignment prefers reusing these parameters for local variables if the parameters' values are not required again later in the function. OP-TEE is compiled using a GCC compiler toolchain.

An example of a suitable function could contain a conditional use of the parameters omitted at runtime. This has caused GCC to keep parameters in their original registers during the examination of different variants of such functions. However, the manual identification of such functions during reverse engineering can be expected to require a notable effort.

Further research might help to automatically identify whole functions that are suitable for integration in ROP chains using symbolic execution of binary code. Such techniques go beyond the current capabilities of traditional gadget-search tools like xrop or ropper.

In summary, leaf functions such as `vsnprintf()` and the exemplary `calc_animation()` function cannot be expected to leave the parameter registers unchanged.

6.2.4 Available Workarounds. The version 3.6.0 of OP-TEE features a command for TAs to change memory permissions. Unfortunately, the corresponding function requires five parameter registers to be set accordingly, which leaves us with a chicken-and-egg scenario.

However, the LDELF module uses this functionality, and we found that it is still mapped when the TA is running. LDELF contains a helper method `ta_elf_finalize_mappings()` which takes only one parameter that points to a descriptor that sets the permission flags on the specified memory regions. Since an attacker can expect gadgets to be present to set the first parameter, this function can be called from a ROP chain. The descriptor consists of multiple structs with pointers to each other. Thus, when constructing such a descriptor, its resulting address has to be known. The choice of the section to be remapped with new access permissions is restricted: only memory registered by OP-TEE as a region can be remapped, which excludes shared memory. Useful choices could be the whole segment consisting of the *.data* and the *.bss* sections as well as the stack and the heap. Once code execution is possible, the memory can be restored after any malicious operation to its previous state, avoiding permission faults that would crash the TA.

6.2.5 Conclusion. With our discussion of gaining arbitrary code execution within OP-TEE TAs, we conclude our research about the exploitability of TAs. As we have shown, depending on the compiler behavior and architectural characteristics, it can be challenging to gain arbitrary code execution within the context of a TA. Nevertheless, the exploits targeting commercially used TAs prove that it is possible to escalate memory corruptions to a full-fledged arbitrary code execution. Those exploits [10, 11, 30] use different techniques and multiple stages to achieve that goal, requiring in-depth knowledge of the TEE and the target TA. For further reference, our open-sourced code also provides an example exploit utilizing ROP to gain code execution in the `vuln` TA.

7 RELATED WORK

A recent study by Cerdeira *et al.* [15] gives an overview of issues found in TEEs in general. The vulnerabilities discussed in our work are validated bugs in their categorization of implementation issues. Complementing the work by Cerdeira *et al.*, we elaborate on the viability of exploits on TEEs taking OP-TEE as an example.

With their work on PARTEmu, Harrison *et al.* [21] introduce a rehosting solution to run proprietary TrustedOSs in an emulator. In their evaluation, they ran a coverage-guided fuzzer on production TAs and found several new bugs. These bugs are not disclosed yet, but could be memory corruptions that can be leveraged for exploits as discussed in our work.

Machiry *et al.* presented Boomerang [24], a new category of bugs within TEEs that results from the semantic gap between the SW and the NW. If not properly sanitized, a pointer sent from the NW to a TA can result in the corruption of NW memory according to Machiry *et al.*. The type-confusion bug discussed in our work is

related to this issue and results from improper validation of inputs that originated from the untrusted NW.

Furthermore, all commercially used TEEs have been investigated individually by researchers [1, 2, 8, 10, 11, 13, 22, 27, 30]. The bugs presented in our work are in particular inspired by Stephens [30], Beniamini [11], and our previous work on the Huawei's TEE [13].

8 SUMMARY

In this paper, we reviewed the exploitability of TEE vulnerabilities in TAs. In particular, we focused on the exploitation of TAs for OP-TEE, a reference implementation for TZ-based TEEs. Inspired by several memory-safety violations found in the wild, we looked at the options and challenges of an adversary. We evaluated different techniques to take control over control flow and gain arbitrary code execution within the context of a TA. To foster research on TEE security, we make our example applications and exploits publicly available.

ACKNOWLEDGMENTS

This research was supported by the German Federal Ministry of Education and Research as part of the Software Campus project (Förderkennzeichen: 01/S17045).

REFERENCES

[1] Tarasikov Alexander. 2019. *Reverse-engineering Samsung Exynos 9820 bootloader and TZ.* Retrieved June 27, 2020 from https://allsoftwaresucks.blogspot.com/2019/05/reverse-engineering-samsung-exynos-9820.html

[2] Adamski Alexandre, Guilbon Joffrey, and Peterlin Maxime. 2020. *A Deep Dive Into Samsung's TrustZone (Part 3).* Retrieved July 3, 2020 from https://blog.quarkslab.com/a-deep-dive-into-samsungs-trustzone-part-3.html

[3] ARM. 2008. *TrustZone System Design.* Retrieved January 3, 2020 from https://web.archive.org/web/20080505021706/http://www.arm.com/products/esd/trustzone_systemdesign.html

[4] ARM. 2010-2016. *Secure Configuration Register.* Retrieved January 3, 2020 from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0434c/CIHBJJCF.html

[5] ARM. 2013. *System Control Register, EL1.* Retrieved July 3, 2020 from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0488c/BABJAHDA.html

[6] ARM. 2017. *Security in ARMv8 systems, Version 1.0.* Retrieved July 3, 2020 from https://developer.arm.com/docs/100935/0100

[7] ARM. 2019. *Procedure Call Standard for the ARM 64-bit Architecture (AArch64).* Retrieved July 3, 2020 from https://github.com/ARM-software/software-standards/blob/master/abi/aapcs64/aapcs64.rst

[8] Ahmad Atamli-Reineh, Ravishankar Borgaonkar, Ranjbar A. Balisane, Giuseppe Petracca, and Andrew Martin. 2016. Analysis of Trusted Execution Environment Usage in Samsung KNOX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (Trento, Italy) *(SysTEX '16)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3007788.3007795

[9] Gal Beniamini. 2016. *Android privilege escalation to mediaserver from zero permissions (CVE-2014-7920 + CVE-2014-7921).* Retrieved July 2, 2020 from https://bits-please.blogspot.com/2016/01/android-privilege-escalation-to.html

[10] Gal Beniamini. 2016. *QSEE privilege escalation vulnerability and exploit (CVE-2015-6639).* Retrieved June 27, 2020 from https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html

[11] Gal Beniamini. 2017. *Trust Issues: Exploiting TrustZone TEEs.* Retrieved July 3, 2020 from https://googleprojectzero.blogspot.com/2017/07/trust-issues-exploiting-trustzone-tees.html

[12] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (Hong Kong, China) *(ASIACCS '11)*. Association for Computing Machinery, New York, NY, USA, 30–40. https://doi.org/10.1145/1966913.1966919

[13] Marcel Busch and Kalle Dirsch. 2020. Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution. In *Proceedings of the 3rd Workshop on Binary Analysis Research.*

[14] Amat Cama. 2014–2019. *xrop.* Retrieved July 3, 2020 from https://github.com/acama/xrop

[15] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P), San Francisco, CA, USA*. 18–20.

[16] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security. In *27th USENIX Security Symposium (USENIX Security 18).* USENIX Association, Baltimore, MD, 99–116. https://www.usenix.org/conference/usenixsecurity18/presentation/eckert

[17] GlobalPlatform. 2019. *TEE Client API Specification.* Retrieved July 3, 2020 from https://globalplatform.org/specs-library/tee-client-api-specification/

[18] GlobalPlatform. 2019. *TEE Internal Core API Specification.* Retrieved July 3, 2020 from https://globalplatform.org/specs-library/tee-internal-core-api-specification-v1-2/

[19] Google. 2018. *Android Keystore.* Retrieved July 3, 2020 from https://source.android.com/security/keystore

[20] Dan Handley, Charles Garcia-Tobin, and ARM. 2014. *Trusted Firmware Deep Dive.* Retrieved July 3, 2020 from https://www.slideshare.net/linaroorg/trusted-firmware-deepdivev10/7

[21] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020. PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/harrison

[22] Ben Lapid and Avishai Wool. 2018. Navigating the Samsung TrustZone and Cache-Attacks on the Keymaster Trustlet. In *Computer Security*, Javier Lopez, Jianying Zhou, and Miguel Soriano (Eds.). Springer International Publishing, Cham, 175–196.

[23] Linaro. 2019. *OP-TEE/optee_os.* Retrieved July 3, 2020 from https://github.com/OP-TEE/optee_os

[24] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. 2017. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *NDSS*.

[25] Dru Nelson. 2013. *Single Byte or Small x86 Opcodes.* Retrieved July 3, 2020 from http://xxeo.com/single-byte-or-small-x86-opcodes

[26] NIST. 2017. *CVE-2016-8764.* Retrieved July 3, 2020 from https://nvd.nist.gov/vuln/detail/CVE-2016-8764

[27] Eloi Sanfelix. 2019. *TEE Exploitation - Exploiting Trusted Apps on Samsung's TEE.* Retrieved July 3, 2020 from https://downloads.immunityinc.com/infiltrate2019-slidepacks/eloi-sanfelix-exploiting-trusted-apps-in-samsung-tee/TEE.pdf

[28] Sascha Schirra. 2014–2019. *ropper.* Retrieved July 3, 2020 from https://github.com/sashs/Ropper

[29] Shellphish. 2020. *shellphish/how2heap.* Retrieved July 3, 2020 from https://github.com/shellphish/how2heap

[30] Nick Stephens. 2017. *Behind the PWN of a TrustZone.* Retrieved July 3, 2020 from https://www.slideshare.net/GeekPwnKeen/nick-stephenshow-does-someone-unlock-your-phone-with-nose

[31] John Walker. 1996. *BGET.* Retrieved July 3, 2020 from https://www.fourmilab.ch/bget/