Executive summary

The Trusted Execution Environment (TEE) is a technology, which enables developers to delegate security functions to a separate secure environment, apart from the normal execution environment. The main advantage of delegating such security functions to an isolated environment such as TEE is its logical and physical separation from the Rich Execution Environment (REE) that can be prone to insecure software. TEE has gained significant interest and is widely adopted by the payment industry, media and entertainment as well as the Internet of Things (IoT). Most modern devices including general-purpose computers, smartphones, and TVs are equipped with TEE.

Developing secure TEEs is paramount for the secure application of the TEE technology within the automotive industry. This paper is written to support development teams, including product owners, design architects, product engineers, and security experts. Within the automotive industry, the TEE is used for applications such as In-Vehicle-Infotainment (IFI) and Advanced Driver Assistance Systems (ADAS). The automotive manufacturers are offering vehicle-related services through back-ends and mobile applications and have recognized the need to understand the technology that they rely on.

As the leading expert in TEE security assessments since 2011, Riscure describes the most frequent security pitfalls for TEE developers and integrators. This paper includes real-world examples for the issues we describe, based on our experience with the evaluation of over 50 different TEE solutions in the past years. Riscure also provides expert advice and best practices enabling the automotive industry to develop secure TEE based solutions, meeting the state of the art security requirements for TEE.

A lot of research has been performed in the field of TEE security. This paper is not intended to provide an exhaustive overview of such research and the cases we provide are used to illustrate the examples in order to enable the reader to quickly explore additional resources. For more generic software mistakes, we have provided a few examples that go beyond TEE. We finalize the paper with an outlook of the most important future TEE security topics.

1 The Arm Ecosystem in Automotive, Rusling, David 2017
Introduction into TEE

In general terms, the TEE offers a hardware-backed secure execution environment that provides a higher level of security than a rich operating system (e.g. Android, iOS, Linux), but offers more functionality than a fully-isolated hardware-only solution such as a 'secure element' (SE). Furthermore, a TEE can share resources with the rest of the chipset, achieving higher performance compared to hardware-only solutions. To provide sufficient security, a TEE must be isolated from the REE by the hardware and software. The TEE provides a secure environment for multiple trusted applications (TA). These TAs should be isolated from the rest of the TEE so that the compromise of a single TA does not allow the compromise of other TAs or the TEE kernel itself.

TEEs provide three main isolation mechanisms, all of which need hardware support from the platform:

1. **Separation between the TEE and REE**: This separation guarantees that the TEE’s data, code, and resources cannot be accessed by the REE. This isolation includes process execution isolation, memory isolation, I/O and peripherals (e.g., control of hardware IP).
2. **Separation between the TEE and TAs**: This separation is required to make sure that a vulnerability in a TA, or a malicious TA, is not leveraged to attack the whole system.
3. **Separation between different TAs**: This separation is similar to the previous one, but it is used to separate TAs from each other, to guarantee that a vulnerability in a TA cannot be exploited to attack other TAs.

These isolation mechanisms rely on the correct configuration of the security features, and critically, on the correct initialization of these features.

**Assets**

An asset is any data, device, or other components that should be protected and secured against an adversary. The assets protected by a TEE depends on the market-specific use cases. In mobile banking applications, the asset includes customer data such as PIN as well as secret keys. In the case of a mobile car key application, the
Decryption and encryption keys are the main asset. Due to its safety concerns, the automotive industry has execution and program flow as an asset.

The whole chain of components involved in securing these assets must also be protected by the platform. This means that the confidentiality and integrity of the TEE software and the integrity of the separations are always considered as an asset.

**Attacker model**

The attacker type and model can differ per market vertical, but due to the nature of the TEE as a secure environment, the attacker is considered to be an entity outside of the TEE. This includes remote attackers or applications in REE or even a potential threat of a malicious or compromised TA. During the first phase when an attacker is exploring the target to identify vulnerabilities, physical access to the device can act as a stepping-stone, but the identified vulnerabilities could potentially be abused remotely.

The TEE can suffer from the same vulnerabilities as any other piece of software with additional architectural and logical vulnerabilities native for the TEE architecture. However, the TEE is intended to have a smaller codebase and restricted interfaces compared to the REE and hence have a smaller attacker surface.
6 most frequent TEE implementation security pitfalls and how to address them:

For vendors that are developing or relying on TEE to protect assets, it is important to understand the security risks and what can be done about them. In this section, we list the most relevant and common security pitfalls, how they are abused by attackers and how this can be prevented.

1. General software vulnerabilities - memory corruption and thread safety

What is it?

These are the types of security vulnerabilities, with which developers and security specialists are quite familiar. Memory corruption vulnerabilities include issues such as “buffer overflows”, “off by ones”, “use after free”. Buffer overflows are one of the most common issues found in the code. Further thread-safety issues are linked to the mutexes semaphores, (spin)locks, flow-control, etc.

There is a wide variety of ways how these vulnerabilities can occur and most software engineers are aware of the issues from the functional point of view. It is necessary to keep in mind that not all programming errors of these types can be exploited by an attacker; those cases are not considered as a security issue. The coding errors can also be exploited next to or together with logical security issues and we will mention this later in the text.

These types of bugs are not based on a lack of understanding of security concepts such as cryptography or secure boot or based on vulnerable system architecture. These types of security vulnerabilities are commonly introduced, accidentally or due to lack of understanding of how and why memory can be corrupted.

How can an attacker use it?

An attacker can approach these vulnerabilities in the same way, as with any other software component. What an attacker can gain with an exploit of these types of vulnerabilities ranges from (1) secret data leakage to (2) run time control of the TEE. The attack path often depends on the number of steps that are to be taken before there
is a successful attack. But the attacker has time and generally doesn’t mind backtracking and taking a different route. Sometimes issues that are currently not exploitable may become exploitable due to an unrelated update or fix, enabling the attacker to complete a full attack path.

Public Example:

What to do about it?
The solution to this general, but a continuing problem in software security is discipline, code hygiene, training, and tools. The use of static and dynamic code analysis and software testing techniques incorporated with internal and external reviews is helpful. Additionally, it is highly recommended to introduce software exploitation countermeasures for additional security. Some examples of software exploitation countermeasures include Address Space Layout Randomization (ASLR), stack canaries, control-flow integrity, non-executable stack and heap (NX), stack buffers and so on.

2. Input validation errors and TOCTOU issues

What is it?
Input validation errors and Time-Of-Check-Time-Of-Use (TOCTOU) issues are two types of security vulnerabilities that are caused by a lack of input validation. The functions implemented for the handling of external data are in the center of these critical security issues. The external data could come over a security boundary between two entities such as TAs, TEEs or REEs through a mailbox or shared memory, but also could come from an external memory such as flash or over the internet. Some examples of root causes related to this type of security vulnerability include:

- Lack of verification of buffer sizes
- Addresses pointing to secure space
- Any data/code an attacker might be able to change due to a lack of verification. Additionally, if there is the time between verification and data usage, it is possible for an attacker to modify data after it is verified but before it is used in a critical application.

For all the data consumed by the TEE, the time of use should not differ from the time of check, in order to avoid data modifications that could occur between the verification and usage.

How can an attacker use it?
In case there is a lack of data verification, e.g. buffer sizes or memory addresses, an attacker could cause buffer overflows or load code in a memory space that should be protected and secure.

TOCTOU exploitation relies on modern multi-core architectures, after the core running the TEE checks the data, another core in parallel can modify it.

Public Examples:
BIOS examples with SMM http://www.c7zero.info/stuff/AttackingAndDefendingBIOS-RECon2015.pdf

What to do about it?
The system architect should be aware of all the data that is consumed by the system and enforce data verification. The implementation should also be reviewed to make sure there are no parts that an attacker could
use in combination with general vulnerabilities. TOCTOU issues are more difficult to identify as they depend on how the verification is done and where the data is. Both types of logical security vulnerabilities can be resolved by copying data into secure memory before checking and using it, as well as linking the checking and usage time so that it cannot be modified by untrusted components.

3. Interfacing between REE and TEE, (TA and TEE) and peripheral input validation

What is it?
The security vulnerabilities in this group are all the security issues that occur at the boundary between different security entities and sometimes incorporate the first two types of vulnerabilities. In addition to the two types, this group also includes vulnerabilities introduced at the architectural level.

The REE, TEE, and TAs are separated and protected from each other. However, the structure of the communication and lack of adherence to the design security goals might introduce fundamental vulnerabilities as shown in the examples below. Some of the structural vulnerabilities include:

1. Too open and permissive APIs between security components
   a. Too permissive syscalls
   b. Too permissive calls between REE and TEE, TA–TA, TEE – TAs
   c. Too permissive drivers’ APIs

2. Structural memory protection issues
   a. Incorrect pointer validation of memory addresses passed between REE, TEE and TAs that results in prohibited memory access
   b. Shared memory, mailboxes, and register protection and separation issues

3. Peripheral control and separation when input is consumed by the trusted environment (for example, fingerprint sensor)

4. Drivers privileges and access control for different security IPs in the system such as other CPUs in the system, GPU, accelerators.

Additionally, previously mentioned issues such as lack of input validation such as buffer size, parameter (variable) type confusion and TOCTOU issues are also relevant in this context.

How can an attacker use it?
The attacker could use such common vulnerabilities to leak secret data between TEE – REE, TEE – TA, TA – TA. Additionally, it could be possible to influence the code flow by changing parameters or even enable an attacker to execute code in a more privileged mode.

Public Example:

What to do about it?
Being aware of the attacker’s capabilities and security responsibilities of the different components in the architecture is the first step in implementing secure interfaces between the TEE components. Additionally, the best practice is to perform a review of the implementation for security vulnerabilities and to verify whether the implementation corresponds to the architecture. Such a security review should be conducted by a team of security experts.
4. Separation configuration

What is it?

As discussed in the introduction, the separation of resources, code, and data is critical for the security of the TEE as well as for the security of the assets, which the TEE protects. While memory separation into different regions available exclusively to the TEE and special TAs are the first thing that comes to mind, any other IP or peripheral in the system has to be addressed as well. For example, if a Direct Memory Access (DMA) engine has access to all parts of the memory, while it can be configured by the REE code for data transfer, it could lead to exposure or modifications of sensitive code and data. As memory access is critical, any peripheral having full access to the TEE space with potential control from the REE side could be abused. Some examples include GPU, PCIe and similar. For example, if TEE/TA is accessing sensors, it is necessary to restrict full control to the TEE to prevent sensor data from being modified by an attacker.

Apart from SoC IPs, debug access to the trusted environment should not be left open to an adversary. Open secure debug access provides full control over the code and data in the secure domain. If the separation configuration is not stored securely and protected from modifications, it could be modified by an adversary and reduce the overall security.

How can an attacker use it?

By abusing misconfigured DMAs or code running on the GPU, an attacker might be able to read and write secure memory regions and thereby influence TEE execution or collect secret data. While these attacks could be quite complex, some of them are easier and rely on public domain knowledge.

Using open JTAG, an attack could access TEE code and data of different privilege levels, leading to a modification of secure data, influencing execution flow and even uploading of attacker code into the TEE. Finally, the configuration data is stored in the system memory and registers. Providing access to such configuration data to untrusted software could lead to modification of configuration data and result in access to the TEE execution flow and secret data.

What do to about it?

The system security architect(s) should be aware of both the hardware and software components in the system and have a good overview of how everything is configured at different stages of boot and runtime. In particular, the system architect should ensure that:

- The secure JTAG is locked
- DMA masters that can access secure memory are not accessible to the REE,
- Registers that configure security-relevant features are not accessible to the REE, etc.

Configuration verification by code as well as encryption/verifications of the configuration could also be implemented. This prevents modifications and confidentiality of the configuration. Finally, the implementation should be reviewed to confirm that the security architecture is implemented correctly.

Public Example:
SVE-2018-12881 [https://labs.bluefrostsecurity.de/files/TEE.pdf](https://labs.bluefrostsecurity.de/files/TEE.pdf)
5. Initialization - Secure boot mistakes

What is it?

This class of security vulnerabilities, as well as the previous one, includes several different issues on the logical level. Logical vulnerabilities are not easily detectable by automated code analysis techniques. For this reason, attackers frequently target logical vulnerabilities. Some examples of initialization security vulnerabilities include:

- Missing/partial verification of parts of the boot flow or data used in the boot flow
- Incorrect/too permissive security configuration during different stages of the boot flow
- Lack of proper initialization of the next stage (memory and IPs)
- Lack of version checking of the next stage of the boot (anti-rollback)
- Incorrect use of cryptography for security verification
- Weak or unprotected security configuration storage and root of trust storage (one time programmable/ROM)

The vendor verifies the security principles of the secure boot such as verification of different boot stages with keys of proper length as a part of the testing process. However, what frequently stays overlooked are smaller data segments that are loaded during secure boot and can influence the execution flow. When the length and loading address are not verified they become a target for the attacker to modify.

Software upgrade functionality is commonly integrated into secure boot processes and as such could lead to any of the previously mentioned problems. The upgrade mechanism is an important feature to mitigate flaws in the field, however, the upgrade mechanism itself must be hardened against attacks. In principle, any firmware on the device should follow the same rules as installed firmware, which includes updated firmware or any recovery firmware.

How can an attacker use it?

Interpretation of data and code before verification could lead to changes in the code flow and recovery of secret data. Lack of proper initialization can enable attackers to achieve privilege escalation. Furthermore, missing of version checks leads to roll-back attacks to previously vulnerable software versions.

Public Examples:

http://theiphonewiki.com/wiki/0x24000_Segment_Overflow
http://blog.azimuthsecurity.com/2013/05/exploiting-samsung-galaxy-s4-secure-boot.html
https://alephsecurity.com/vulns/aleph-2017026
https://github.com/Qyriad/fusee-launcher/blob/master/report/fusee_gelee.md


What do to about it?

System security architect(s) should be aware of the security of all stages and configuration options in the system to be able to make decisions that lead to a secure system. In more detail, a securely implemented boot process should:

- Make sure that all stages are verified
- Always copy images to RAM before verifying them
- All fields in software images should be verified
- Have a way to implement anti-rollback, possibly for all stages
- Lock root keys and any settings that can influence secure boot
- Demote running software as soon as possible
• Erase any key information that could be in memory/registers after they are used

The implementation should be carefully reviewed to confirm that the security architecture is implemented. Limiting the functionality of the firmware upgrade mechanism could provide the first layer of defence. Smaller functions with dedicated purpose contribute to the reduced number of vulnerabilities. Furthermore, partial updates should be avoided and all individual blocks should be signed.

6. State errors – standbys

What is it?

For power management reasons sometimes the device is storing the state, so the resume procedure can start faster and with the same conditions it ended up with. In such situations, it is wise to consider whether:

• An attacker has access to the state data
• If the resume could be influenced
• If some security-sensitive registers might be deleted once sleep mode is enabled

The security of the stand by functionality depends on where the state is stored and how the state sequence could be influenced by an attacker.

How can an attacker use it?

In case a resume state of the TEE is stored in memory without any protection it could be possible to subvert the proper boot process and modify the code or data of the TEE, which could lead to full privilege escalation.

What to do about it?

The mitigation of such an attack needs to be focused on verification of the state before usage, as well as validation of parameters before use and in some cases initialization of all security-relevant registers. Some options given to the developer to protect their solution during warm boot include cryptographically signing and verifying the state from which the device resumes, as well as analysis of all state variables before usage, such as exception handling, suspend/resume storage and integrity.
Security software vulnerabilities of the future

Software is getting more secure every day. More experienced teams are making smaller and more secure TEE cores, the code and designs are rigorously evaluated and tested. In the foreseeable future, the number of vulnerabilities discussed in the previous sections is expected to be reduced. At that moment, the focus of attackers and developers will turn to more exotic attacks on the edge between hardware and software. These attacks require more expertise and knowledge, but are equally scalable and can be executed remotely. In this section, we provide a preview of the future to come.

1. Micro-architectural side channel attacks

What is it?
The Micro-architectural side channel attacks such as Spectre and Meltdown can reveal secret data such as keys. They use speculative execution together with the timing properties of the cache and the processor to reveal data:

- Meltdown uses a short time window during which the speculatively processed data is in the cache to export it externally.
- Spectre entails speculative execution by the processor based on unverified data before the verification is performed.

By very carefully constructing the user data and setting up the cache in advance, it is possible to force the victim program to leak data. Without nanosecond-precision timing, these attacks would be impossible.

How can an attacker use it?
Attackers can use these approaches to extract secrets such as keys or PINS protected by the TEE, from the REE side.

Public Examples:
Spectre and Meltdown
https://spectreattack.com/

What to do about it?
Spectre and Meltdown attacks are based on hardware optimization and therefore only limited protection is possible in firmware. It is advisable to use the ARM mitigation techniques and hardware that is not susceptible to these types of attacks.

2. Software induced perturbation attacks (CLKScrew)

What is it?
Introducing faults in the program operation could lead to unpredicted and insecure program behaviour. These faults are typically introduced by physically modifying the voltage, clock or EM field in which the chip operates. However, changing the configuration in which the chip operates can lead to faults and security issues as demonstrated in the CLKScrew paper referenced below. Similarly, hardware and software interplay is present with the RowHammer attack, also referenced below.
How can an attacker use it?

An attacker can use a well-crafted code to control the device power, clock or physical interfaces, which could lead to errors in the execution and change the code flow. These type of vulnerabilities are highly dependent on the hardware support of the secure system and requires a lot of knowledge and expertise to execute.

Public examples:
“CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management” by Adrian Tang, Simha Sethumadhavan and Salvatore Stolfo
“Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors” by Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, Onur Mutlu

What to do about it?

All the resources that can change the system conditions should be controlled by the secure domain. Additionally, attacks such as row hammer rely on the physical environment of the system and cannot be always prevented.

Conclusion and next steps

Various industry verticals are relying more and more on the flexibility and security offered by TEE implementations, to protect the assets of their customers. A notable example of disruptive innovation based on the TEE is visible in the payment industry with software-based mobile payment solutions and a strong movement away from traditional smartcard based solutions. Data right management solutions in media and entertainment are following a similar trend. The automotive industry is integrating payment and media in their systems but also has high requirements for safety and car security that could rely on TEE.

When an industry adopts new technology, many risks arise due to the fact that the technology was available to attackers before it was introduced to a specific vertical. Securing solutions in all the markets require complex software-based countermeasures and proper design integration and configuration. For most solution developers and businesses interested in developing and adopting TEEs to protect their solutions, the best way to ensure the security of the system and data is to utilize the expertise of independent third-party security experts who specializes in this innovative technology.

How Riscure Can Help

Riscure has been a front-runner in assessing the security of TEE products and is recognized by many certification schemes and suppliers. Riscure has performed over 100 security evaluations covering TEE OSes, TEE hardware implementations, and final products implementing secure solutions using TEEs (Mobile Payment solutions, Smart TVs, Set-top-boxes, IoT devices, etc). Additionally, Riscure has performed over 250 security evaluations of TEE based OEM Pay Solutions, TEE based Biometric Authentication solutions (also referred to as CDCVM), and software security tools including e.g. Obfuscators, Self-defence mechanisms, White-box-Cryptography.

Riscure is also actively involved in TEE standardization bodies and organizations relying on TEE based solutions, including GlobalPlatform, Linaro, FIDO, EMVCo, and others.

Riscure provides a broad, efficient and flexible offering for solution developers aiming to secure and certify their solutions. With our services and expertise, we actively support our customers on each stage of their solution development process in order to reduce security risks, prevent delayed time-to-market and security certification costs. Riscure is perfectly positioned to support its clients and partners in their secure development process by providing tools, training, evaluations, testing, and certifications.
Interested to learn more about our offering and how to secure your own solution?
Visit our website at www.riscure.com or contact us via inforequest@riscure.com.