MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

TEE-based Remote Platform Attestation

Scientific Students' Association Report

Author:

Márton László Bak

Advisors:

Levente Buttyán, habil. PhD
Dorottya Futóné Papp

2021

Contents

Abstract

The Internet of Things (IoT) consists of network-connected embedded devices that enable a multitude of new applications in various domains, such as industrial automation, transportation, building automation, healthcare, and agriculture - just to mention a few of them. However, as usual, new technologies also create new risks. In particular, embedded IoT devices can be infected by malware. Operators of IoT systems not only need malware detection tools, but also scalable methods to reliably and remotely verify malware-free-state of their IoT devices. In this document, we address this problem by proposing a remote attestation protocol for IoT devices that takes advantage of the security guarantees provided by a Trusted Execution Environment (TEE) running on the device.

In this document, we provide the necessary information required to understand the attestation process, and various approaches to implementing remote attestation. We perform threat assessment in order to identify the possible weaknesses of remote attestation protocols. We present the overview of the TEE-based remote attestation scheme we designed. We define the two main participants, the Verifier and the Prover, in our protocol. We discuss how a secure channel is established between the two parties, and the messages used in order to carry out the attestation. We briefly describe the checks we use during remote attestation to verify the malware-free-state of the device. We provide a prototype implementation of the proposed protocol, and evaluate its features and performance. We conclude that our TEE-based remote attestation is capable of verifying the malware-free-state of IoT devices.

Kivonat

A dolgok internete (Internet of Things, IoT) hálózatba kötött beágyazott rendszerek sokaságát jelenti, ami renget új felhasználási lehetőséget biztosít, többek között az egészségügyben, tömegközlekedésben, iparban és mezőgaz- daságban, csak pár lehetőséget említve. Ezzel egyidőben mint általában minden új technológia, új veszélyeket is hordoz magában. Ezek az eszközök gyakran esnek kártékony kód alapú támadások áldozatává. Az IoT eszközök üzemeltetőinek nem csak kártékony kód detekcióra van szükségük, hanem egy olyan megbízható és skálázható módszerre, amely távolról képes meggyőződni arról, hogy egy eszköz fertőzetlen állapotban van. A dolgozatban erre a problémára próbálunk megoldást nyújtani egy olyan protokoll segítségével, melyet felhasználva az IoT eszközök üzemeltetői fizikai hozzáférés nélkül képesek meggyőződni az eszköz kártékony kód mentes állapotáról, melynek alapjául egy biztonságos futtatási környezet (Trusted Execution Environment, TEE) szolgál.

A dolgozatban biztosítjuk a távoli kártékony kód mentes állapot igazolására szolgáló protokollhoz szükséges alapismereteket, és tárgyaljuk a protokoll különböző megvalósítási lehetőségeit. Fenyegetettség felmérést végzünk annak érdekében, hogy megállapítsuk a protokoll gyengeségeit. Bemutatjuk az általunk tervezett protokollt, melynek alapjául egy biztonságos futtatási környezet (angolul Trusted Execution Environment, TEE) szolgál, és tisztázzuk a protokollban szereplő két entitás szerepét. Bemutatjuk a protokoll üzeneteinek továbbítására használt biztonságos csatornát, a protokoll által küldött üzeneteket, és azokat az integritás ellenőrző számításokat, melyeket a kártékony kód mentes állapot igazolására használunk fel. Bemutatjuk az általunk tervezett protokoll prototípus implementációját, és kiértékeljük azt. A kutatásunk konklúziója, hogy az általunk tervezett protokoll képes kártékony kód mentes állapot igazolására beágyazott eszközökön.

Chapter 1

Introduction

The Internet of Things (IoT) consists of network-connected embedded devices that enable a multitude of new applications in various domains, such as industrial automation, transportation, building automation, healthcare, and agriculture - just to mention a few of them. This connectedness enables the IoT devices to communicate with each other and share data, without the requirement to be interfered by humans. This feature enables solutions such as smart factories, automated transport systems and sensor controlled agriculture.

However, as usual, new technologies also create new risks. Due to the high demand for such IoT devices, manufacturers are in a constant rush of developing and adapting their devices, potentially sacrificing security measures on the way. This facts leads to devices with minimal, or even nonexistent, security features. Consequently, IoT devices are constantly exposed to cyberattacks. To address the situation, several solutions [10], standards [6, 8] and guidelines [5] are being proposed, and regulatory bodies are making steps[1] in order to ensure the following of such proposals.

One particular issue IoT devices face is malware infection [1]. Malicious actors can use such programs in order to infect millions of devices, compromise complete IoT ecosystems, exfiltrate data from sensory networks or complete large scale Distributed Denial of Service (DDoS) attacks. In order to avoid such problems, IoT ecosystem operators need malware detection tailored for IoT devices. Additionally, they need a scalable and reliable way of verifying the malware-free-state of IoT devices and their system. In this document, we address this problem by proposing a remote attestation scheme for embedded IoT devices, that take advantage of security guarantees provided by a Trusted Execution Environment (TEE) running on the device.

[1]The California IoT cybersecurity law SB-327 became effective Jan 1, 2020.

Remote attestation is a process where a trusted party, a Verifier, reliably checks the state of a device, the Prover, via open network. In our case, the Verifier is a trusted party, running on a server, operated by the system's administrator, and the Prover is an untursted IoT device, which may be compromised by malware. Remote attestation is used to verify the malware-free-state of such untrusted devices. Implementing this attestation process remotely means, that the administrator does not require physical access to the devices, allowing for a large scale and automated approach to verify all devices inside an IoT ecosystem.

The organization of the document is as follows: In Section 2, we provide the necessary information required to understand the attestation process, various approaches to implementing remote attestation and the basics of a Trusted Execution Environment. In Section 3, we perform threat assessment in order to analyze the system and identify its potential weaknesses. Next, we present the overview of the TEE-based remote attestation scheme we designed in Section 4. In Section 5, the attestation process is detailed: the role of the Verifier and Prover is established; the channel we use to communicate between the two parties, and the messages used in order to carry out the attestation are described. In Section 6, we briefly describe the checks we use during remote attestation to verify the malware-free-state of the device. A prototype implementation of the protocol is provided in Section 7. Finally we evaluate the proposed attestation scheme and conclude our work in Section 8 and 9, respectively.

Chapter 2

Background

Remote attestation is a process between two entities: the Verifier and the Prover. During remote attestation, a remote Verifier wants to ascertain the integrity of the Prover, which, in our case, is an IoT device. This is done with a challenge-response

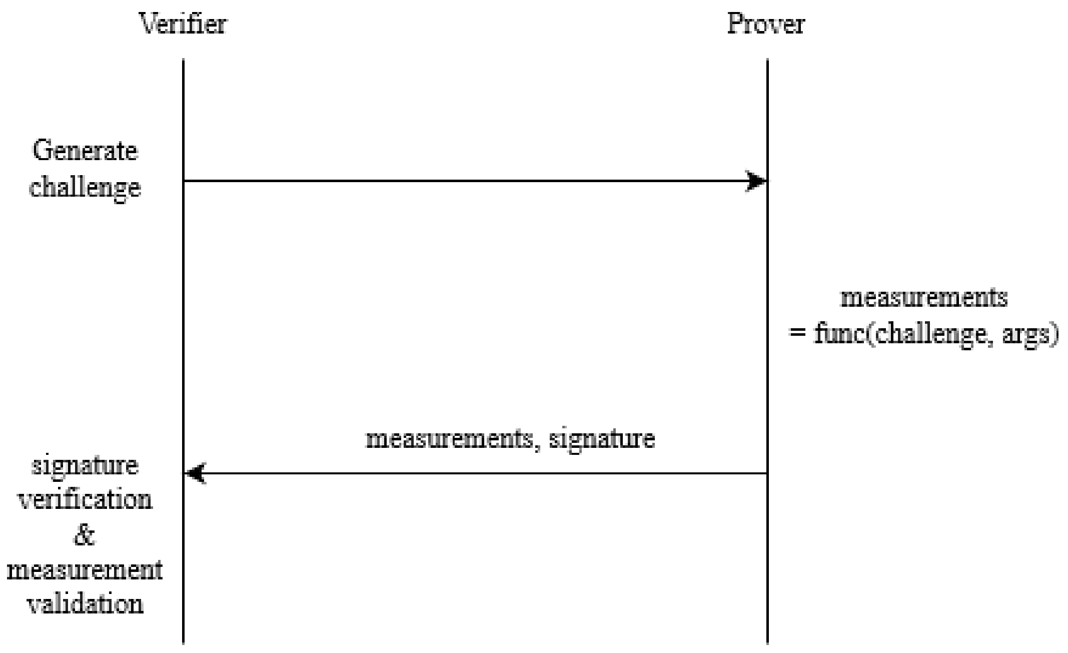

Verifier
Prover
Generate challenge
measurements = func(challenge, args)
measurements, signature
signature verification & measurement validation

Figure 2.1: Example run of remote attestation

protocol, which usually has three major steps as it is demonstrated in Fig. 2.1.:

1. The Verifier sends a challenge to the Prover,
2. the Prover calculates a response, which depends on the challenge and proves its integrity to the Verifier

3. and lastly the Verifier determines the correctness of the response sent by the Prover.

There are three general approaches to remote attestation: hardware-based, software-based, and hybrid remote attestation. Hardware-based attestation relies on a secure co-processor (a TPM chip[1] for example) which is capable of producing a digitally signed summary of the hardware- and software-state of the device under attestation. The summary mentioned is typically a hash, produced by combining the hashes of system components and software modules started during the boot process. The key used to sign the final hash is kept in the secure co-processor and is protected by its access control and physical tamper resistance features. Since this method of attestation requires a secure co-processor, it is typically deemed to be too expensive for use in embedded IoT devices.

In contrast to hardware-based approaches, software-based attestation does not require any additional hardware in the Prover device, making it a more favorable choice in IoT devices. Solutions following this approach (e.g., [16, 17]) typically rely on executing a protocol in which the Verifier probes the Prover, and the response of the Prover needs to convince the Verifier that the Prover is in a malware-free state. To generate the response, the Prover runs a checksum function, which traverses memory locations in a pseudo-random order, based on a seed generated by the Verifier. The Verifier then checks the correctness of the response by calculating the same checksum on an expected prerecorded state. There are two possible ways of failing the attestation, in case a malware is present on the device. Firstly, if the checksum calculated by the Verifier differs from the response sent by the Prover, hinting that the memory layout of the devices is modified. Secondly, if the malware tries to hide itself by checking and redirecting memory accesses referring to the location it is in memory, the attestation attempt will take longer than expected. In this latter case, the Verifier can detect, that the response took too long, and discover the presence of malware that way.

The main challenge of software-based approaches is network jitter, which makes it practically impossible to remotely measure the exact computation time of the checksum [13]. Another weakness of such approaches is that the adversary can delegate the checksum computation to another, much faster, device, which cannot be detected by the Verifier. Moreover, even if we do not take delegation attacks into consideration, this approach assumes that the required checksum computation cannot be performed faster than the speed of the actual implementation of the

[1]https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/ Last visited: Oct 7, 2021.

checksum function. Unfortunately, this assumption does not always hold, enabling attacks, where a sophisticated way of computing the checksum faster leaves time for the malware to check and redirect memory references that would uncover its presence [2]. Another tricky attack a malware can perform is that it compresses the original code of the device in order to fit the malicious code into memory. During attestation, the malware can decompress this memory segment and retrieve the original content, thus passing the attestation [2].

Due to the above weaknesses of software-based attestation, hybrid approaches were proposed (e.g., [4, 3]) that are largely software-based, but also require hardware support of some sort. For example in SMART [3], the authors propose a scheme, that uses ROM-based checksum calculation and the secret key required for authenticating the computed checksum is kept in memory only accessable by ROM-based code. This latter property is enforced by hardware-based memory access logic, which verifies that the instruction pointer is in the ROM region when the secret signing key is being accessed. This check ensures that the confidentiality of the key is preserved, even if the device is infected by a malware. This also means that the response presented by the Prover, authenticated with the secret key must be genuine. This fact ensures the Verifier that the response came from the actual devices and thus can be checked.

The next question naturally arises: What is the minimum hardware support required for a hybrid remote attestation scheme to be secure? This question is investigated by A. Francillon et al. in "A minimalist approach to remote attestation" [7], where they conclude that the following set of requirements is sufficient and necessary (hence minimal) for secure attestation of embedded devices:

- Custom hardware to enforce exclusive access to a secret key;
- Reliable and secure memory erasure;
- Attestation code must be executed from immutable memory;
- Execution of the attestation must be uninterruptible;
- Controlled execution of the attestation, meaning it can only be invoked from the original entry point.

As presented by K. Elderfrawy et al. in HYDRA [4], the requirements stated in [7] can be achieved in another way. The authors of HYDRA [4] propose a hybrid remote attestation scheme that relies on security features provided by a formally verified seL4[2] microkernel to obtain similar properties. In HYDRA:

[2]https://sel4.systems/About/seL4-whitepaper.pdf, Last visited: Oct 7, 2021.

- Exclusive access to the remote attestation key is handled by a privileged process;
- Reliable and secure memory erasure is required to ensure no information about the secret key is leaked. This is ensured by the strict memory separation of the seL4 microkernel;
- Immutable memory is also required to make sure the checksum routine cannot be modified. This is ensured by the microkernel which provides isolated process memory and code integrity checks;
- Prioritized interrupt handling of the microkernel ensures uninterruptible execution of the checksum code which is run with the highest priority possible;
- Controlled invocation is enforced by the operating system.

A. Francillon et al. state [7], that secure reset is required whenever an attempt of running the checksum function from the middle of its code detected. This is not required in case of HYDRA [4], since the seL4 microkernel enforces controlled invocation. The authors of HYDRA [4] claim that using seL4 imposes fewer hardware requirements on the underlying microprocessor and building upon a formally verified software component increases the confidence in security of the overall solution. Another approach to satisfy the minimal requirements [7] of a secure remote attestation could be to use a Trusted Execution Environment (TEE). TEE can be simultaneously run next to a Rich Execution Environment (REE), a normal operating system (e.g.: Linux). A TEE provides isolated environment for trusted processes where they can execute without being interfered by processes running in the REE, which might be compromised. Moreover, TEE also provides secure storage for keys, such as the one used to sign the attestation. TEEs usually rely on hardware support to provide the above mentioned security guarantees. For example, on ARM powered IoT devices, the TrustZone[3] technology provides such support. In case of TrustZone, a special register in the processor keeps track of whether the given memory is used by the Secure World (trusted OS and trusted applications of the TEE) or the Normal World (REE). This enables memory separation of the device, defending the Secure World applications from Normal World access. This property is enforced on a hardware-level by the memory bus fabric. TEE implementations usually take advantage of this low level support built into the processor itself. An example TEE is OP-TEE[4], an open-source implementation a TEE, which is based on the ARM TrustZone technology.

[3]https://developer.arm.com/ip-products/security-ip/trustzone Last visited: Oct 10, 2021.
[4]https://www.op-tee.org/, Last visited: Oct 12, 2021.

Chapter 3

Threat Assessment

Threat assessment is a widely accepted methodology for developing and maintaining a secure system. The main goal of this methodology is to derive the security requirements of the system under development. In this section, we derive the security requirements by applying the threat assessment methodology proposed by ETSI [12]. Our analysis also takes the functional requirements and use cases of remote attestation into consideration.

Remote Attestation is performed between two parties, the Verifier and the Prover. The Verifier role will be taken by a Server, while the Prover will be a remote IoT device reachable via open network. The architecture should support the following requirements:

- Server can initiate remote attestation on a device.
- A device can periodically request remote attestation from the server.

The structure of this section is as follows. We discuss the functional requirements of the system in Section 3.1, then list the assets of the system in Section 3.2. Using the assets and the functional requirements, we build an attacker model in Section 3.3. Finally, we derive the security requirements of the system in Section 3.4.

3.1 Functional Requirements

Based on the described functionality, our system should be automatic, operating without external user interaction. Attestations would be called periodically per device. If we want to allow the administrator(s) of the system to be able to initiate a

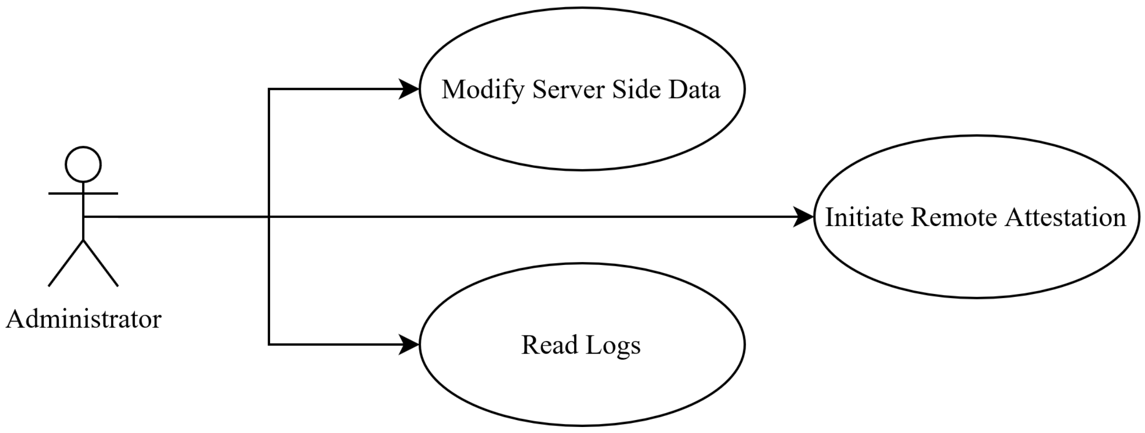


Figure 3.1: Use case diagram of the Server

remote attestation, a command line interface should be implemented. Administrators can modify server side data – which will be discussed later in Section 5.3.1–, as well as read the logs and the results of previous attestations. These are the only user interaction that can be made to the system, other interactions happen between system components. The use case diagram can be seen in Fig. 3.1.

3.2 Asset identification

In light of the functional requirements, we can create an overview of the system, as shown in Fig. 3.2. In order to identify our assets, we have to analyze the system's use cases. During Threat, Risk, Vulnerability Analysis, we identify the three following assets: human assets (e.g.: administrator), physical assets (e.g.: hardware components and network) and logical assets (e.g.: data storage).

The only human asset in our system, the administrator, appears in three of our use cases: reading the logs, modifying attestation data and initiating remote attestation. The system does not require authentication, since the attestation is automatic and the login to the server is out of scope for this architecture.

From what we previously described about the architecture, it is clear that there are physical assets, mainly an environment to run the server side application and store data on, and one or more embedded devices. Connection between the physical devices is achieved via open network, which needs to be taken into consideration as well. These together form our physical assets.

To determine logical assets, the data flow for achieving all use cases needs to be analyzed. It helps if we create a more detailed data flow diagram. The extended data flow diagram can be seen in Fig. 3.3. Our system consists of two main components,

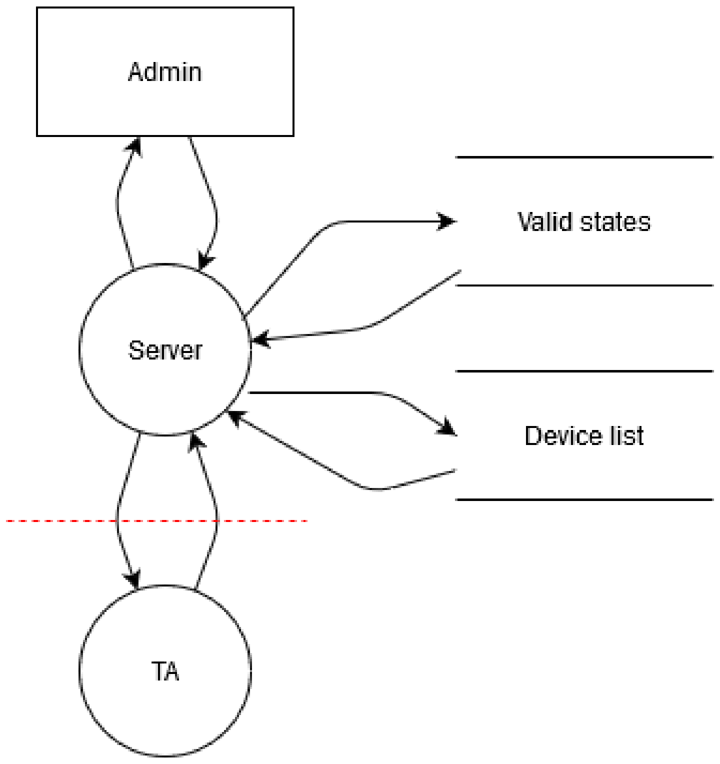


Figure 3.2: Overview of the system

the Server (Verifier) and the Trusted Application (Prover). These components communicate with each other via untrusted network. On the server, we store data in order to verify the correctness of the attestation response. We also store the list of devices to be attested by the Verifier. On each embedded device we have a Trusted Application (TA) that:

- handles attestation requests sent by the Server,
- calculates the challenges,
- then sends the response back to the Verifier, signed with an attestation key.

3.3 Attacker model

Constructing the attacker model, we have to take every component's potential weaknesses into consideration. To organize threats, we use STRIDE framework by Microsoft [14], which includes considering threats related to: Spoofing, Tampering,

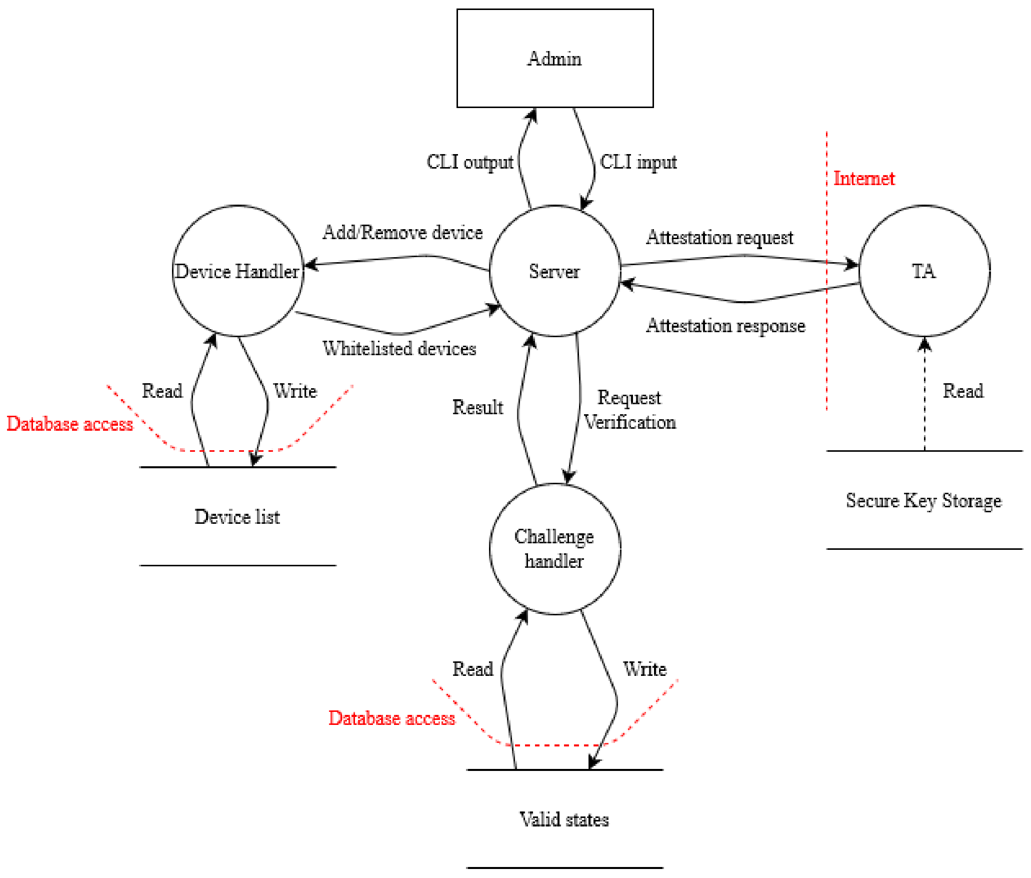


Figure 3.3: Extended data flow diagram of the system

Repudiation, Information disclosure, Denial of Service and Elevation of privilege. While constructing the attacker model, we identified the following attack vectors:

- *[Spoofing]* Impersonating either the server of a device during attestation
- *[Tampering]* Adversary can block/modify attestation requests and responses
- *[Tampering]* If adversary can remove a device from the servers' list, the device would not be checked, hence the adversary could hide the potentially compromised device
- *[Tampering]* Adversary can replay a challenge, which he/she already captured the response to, making a valid attestation
- *[Information disclosure]* Adversary can monitor the network thus gaining information on devices
- *[Information disclosure]* Adversary can detect the periodicity of the attestation
- *[Information disclosure]* Adversary can use network traffic analysis to acquire information about the system

- *[Denial of Service]* Both the server and devices can be overloaded with network requests
- *[Denial of Service]* Adversary can modify messages, potentially hanging the attestation process, if not handled correctly

We note that two categories, namely Repudiation and Elevation of privilege, are missing from the listing above. Elevation of privilege is not possible, since the system does not require any privileges to run.

3.4 Security Requirements

On the server, an operator can initiate a remote attestation and modify data corresponding to attestation devices. Since the security of the server is out of scope for this project, we can assume, that no malicious attempt is made in order to tamper with the attestation on the server side. However, trust becomes an issue between the server and the attested devices. Since these components communicate via open network, an adversary can read, replay, block and modify the communication between them. This critical section is marked with a dashed red line on Fig. 3.2 and 3.3. The systems' security requirements can be sorted into six categories: CIA[1] and AAA[2]. In the following list, we define the requirements for each category:

- *[Confidentiality]* Data sent between the TA and the server must be encrypted.
- *[Confidentiality]* Valid states of the Prover device should be securely stored.
- *[Integrity]* Integrity of the sent data must be protected on both sides.
- *[Integrity]* Time window should be implemented for accepting attestation.
- *[Availability]* System should use the most up-to-date libraries to combat CVE[3].
- *[Authentication]* The origin of the attestation challenge must be authenticated by the Prover.
- *[Authentication]* The source of the challenge response must be authenticated by the server.
- *[Auditing]* Every action should be logged on both sides.

[1]Confidentiality, Integrity, Availability
[2]Authentication, Authorization, Auditing
[3]Common Vulnerabilities and Exposures https://cve.mitre.org/, Last visited: Oct 12, 2021.

To satisfy security requirements, we have to plan, implement, and test various security functionalities. We can get the list of required security related functional requirements by analyzing security requirements and abuse cases[4]. Since remote attestation follows a request/response scheme, it is easy to detect if a challenge or it's attestation response got halted. To detect if any change has been made to any messages and protect against replay attacks, we have to implement:

- sequence numbering
- and message authentication.

Time limit for accepting attestation should be implemented, since an adversary could block the attestation response, or try to fabricate a valid response. Logs will be kept server side, of which defense is out of scope. If an adversary is present on the network, he cannot read/modify messages, if sufficiently protected, however there is no way to stop the adversary from getting know to the address of the device. Dummy messages can also be sent between the server and the devices to defend against Network Traffic Analysis (NTA)[5]. This way the attacker won't be able to detect the periodicity of the attestation. To ensure that the attestation key is not leaked, we can use the secure storage offered by the TEE on the embedded devices. Against Denial of Service (DoS) attacks, we can have our embedded devices whitelisted on server side, only accepting network traffic from whitelisted entities on the port of the application. The address of the server can also be stored on the devices themselves.

[4]Using a feature in a way that was not intended by the implementer

[5]https://www.cisco.com/c/en/us/products/security/what-is-network-traffic-analysis.html, Last visited: Oct 12, 2021.

Chapter 4

Overview

Our main idea is to follow the approach of HYDRA [4], but instead of a secure microkernel, we rely on a Trusted Execution Environment (TEE). Checking malware-free-state can be implemented in a trusted process running in the TEE, and the secret key used for authentication the result can be stored in the TEE-protected secure storage.

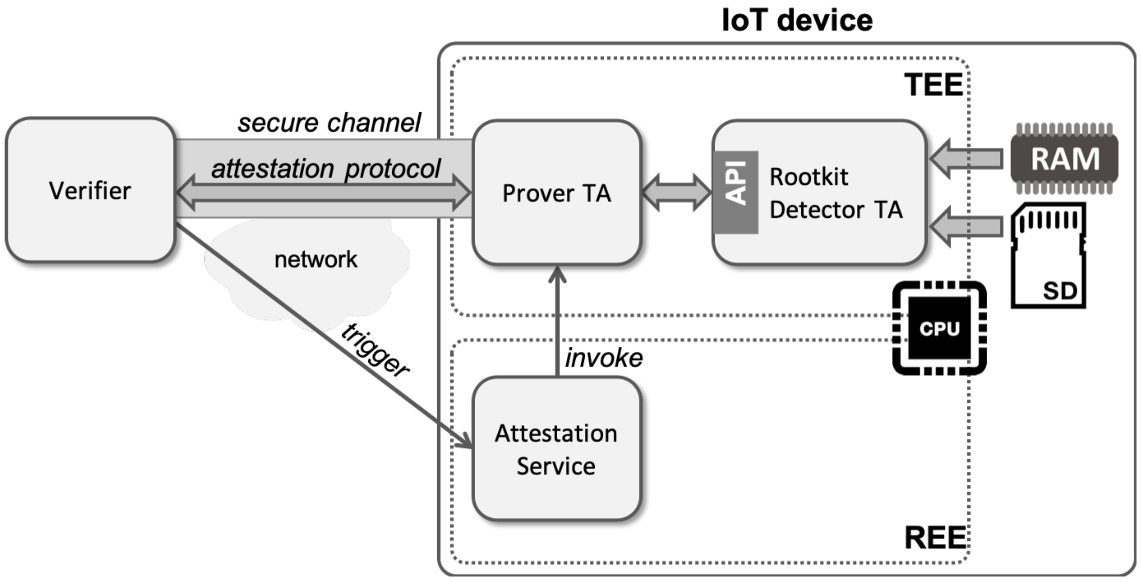


Figure 4.1: Overview of the TEE-based remote attestation architecture

We note that embedded devices equipped with ARM or similarly powerful processors are typically capable of hosting such a TEE. The architecture of our TEE-based remote attestation scheme designed for IoT devices is shown in Fig. 4.1. In our TEE-based attestation scheme, the Verifier is assumed to be a remote entity that communicates with the Prover via open network. The Prover is a Trusted Application (TA) running inside the TEE hosted by the processor of the IoT device. An attestation service is running in the Rich Execution Environment (REE) also hosted by the same processor. The term “rich” refers to the fact that the REE may be a full-blown operating system (OS) that provides only basic services to the trusted applications. Isolation between the two execution environment is supported by the processor and its Memory Management Unit (MMU).

There are two ways to initiate a remote attestation in our scheme. Either the Verifier directly challenges the Prover via the Attestation Service, or it is initiated automatically after a certain period of time. The Attestation Service is assumed to be always available. If it is not, then it already proves that the IoT device is not in a normal state and may already be compromised by malware. In both cases, the Attestation Service invokes the Prover TA via a controlled invocation mechanism provided by the TEE (and supported by the processor). After being initiated, the Prover TA establishes a secure connection to the Verifier that is used to execute the remote attestation protocol securely.

In the remote attestation protocol – described in detail in Section 5–, the Prover is challenged by the Verifier with a set of tasks to complete. These tasks consists of the execution of one or more integrity checks that are implemented by a Rootkit Detector TA, a component that actively searches for hidden malware, also known as rootkits, running in the TEE. The integrity checks requested by the Verifier are invoked by the Prover TA via a well-defined API provided by the Rootkit Detector TA. More information about the Rootkit Detector TA is provided in Section 6. The result of each call is returned back to the Prover TA. After all tasks are completed, the accumulated results are sent back to the Verifier via the secure channel.

The Rootkit Detector TA has access to the entire memory of the IoT device as well as its persistent storage. The memory includes the memory of the processes and the OS kernel running in the REE and the persistent storage includes the file system that they use. The integrity checks implemented by the Rootkit Detector TA analyze this memory and file system, collect relevant data (e.g., the hash of the text segment of the OS kernel in the REE), and try to detect anomalies that may signal the presence of a malware (e.g., hooked function pointers).

More information about the remote attestation protocol and the integrity checks of the TEE-based remote attestation scheme is provided in Section 5 and 6, respectively.

Chapter 5

Attestation process

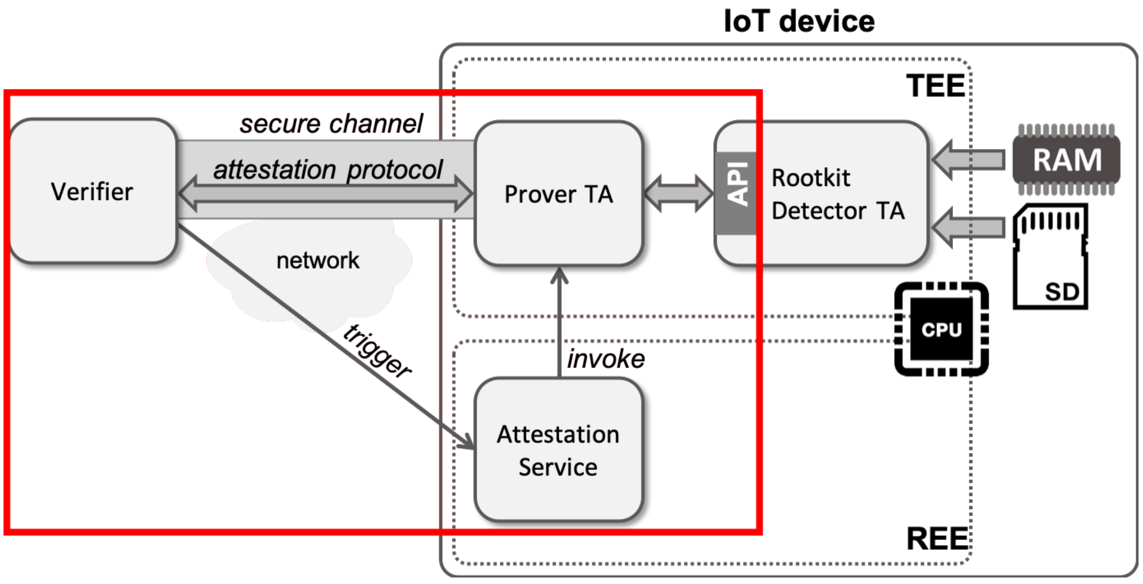


Figure 5.1: Components of the remote attestation protocol

In Section 4, an overview of the whole system is presented. In this section, the highlighted area in Fig. 5.1 is presented in more detail. The following subsections will present an in-depth design of the remote attestation components: the Verifier, the Attestation Service and the Prover TA; and the messages/protocols used to communicate between them.

The attestation process is an interaction between the Verifier and the Prover TA. The prerequisite of the attestation for the Prover TA, is to be loaded and have control over the execution. This can be achieved two ways. Either the device initiated the remote attestation process in a periodical manner, or the Verifier requested an attestation. In the latter case, the Verifier can trigger the Attestation Service, listening inside the REE, to invoke the Prover TA, passing the execution and activating it the same time.

We note, that the Attestation Service should be always available on the IoT device. If that is not the case, we can already suspect the presence of malware on the device. It is also a possibility that the adversary has control over the REE on the device, and blocks the invocation of the Prover TA. To counteract this, after triggering a remote attestation from the Verifier, a timer should be set. If an attestation request does not occur in the given time window, we can also suspect, that an adversary might have control over the device.

The structure of this section is as follows. We present the attestation protocol used between the Verifer and the Prover TA in Section 5.1, and the secure channel used to transfer these messages between the two participants in Section 5.2. Last, we present both the Verifier and the Prover TA in detail in Section 5.3 and 5.4, respectively.

5.1 Attestation protocol

In this section, we present the challenge descriptor language used to describe an attestation request. It is a high-level text-based language, used by the Verifier to describe the required calculations the Prover has to make during attestation. An attestation challenge can contain one or several tasks. Each task that can be called by the Verifier is mapped to a function call, provided by the Rootkit Detector TA's API. The Rootkit Detector TA provides many ways to check the integrity of the device under attestation. These checks include:

- reading the list of running processes on the device,
- checking the integrity of the file system
- and checking the integrity of the kernel.

Each task has a text-based identifier, followed by the required arguments divided by whitespaces. The tasks are divided by the `\n` character.

```
HASH_FILE /foo/bar
HASH_DIR --recursive /opt /opt/do/not/hash/this
HASH_KERNEL_TEXT
HASH_SYSCALL
PROCLIST
NETFILTER_HOOKS_CHECK
```

Listing 5.1: Example remote attestation description

An example attestation challenge can be seen in Listings 5.1. In Table 5.1 we describe each task that can be included in an attestation challenge.

Command ID	**Arguments**	**Functionality**
HASH_FILE	The one and only argument is the absolute path of the file to hash.	Hashes the file given as argument.
HASH_DIR	This task has an optional switch, as well as two parameters, where the second one is also optional. If the switch `--recursive` is provided the function is called recursively on all subdirectories.	Hashes the folder given as argument, with the exception of provided blacklisted elements.
HASH_KERNEL _TEXT	No argument needs to be provided.	Hashes the kernel's text segment.
HASH_SYSCALL	No argument needs to be provided.	Hashes the syscall table.
PROCLIST	No argument needs to be provided.	Returns the names of the running processes on the device.
NETFILTER_ HOOKS_CHECK	No argument needs to be provided.	Checks network hooks on the device.

Table 5.1: Available tasks for the Verifier

Command ID	**Output**	**Length**
HASH_FILE	SHA-256 hash of the file.	32-bytes
HASH_DIR	SHA-256 hash of all files under the given path, excluding blacklist elements.	32-bytes
HASH_KERNEL _TEXT	Hex encoded SHA-256 hash of the kernels text segment.	64-bytes
HASH_SYSCALL	Hex encoded SHA-256 hash of the syscall table.	64-bytes
PROCLIST	List of running process names separated by `'\0'`	Cannot be determined ahead of time.
NETFILTER_ HOOKS_CHECK	Binary output, 0 if all checks pass, 1 otherwise.	1-byte

Table 5.2: Expected responses for all tasks

In order to check the correctness of the attestation response, it is important that the expected outcome for each attestation task is defined. In Table 5.2, we define the expected output for each task.

5.2 Secure channel

The protocol discussed in Section 5.1 assumes a secure channel between the Verifier and the Prover TA. Thus, it is a prerequisite to have a secure network connection between the two participants. One could use TLS-based [18] socket handling in order to establish such secure connection, however it might not be supported by some TEE implementations. This is the reason, we opt for basic TCP-based socket handling, and propose a simple yet effective security protocol. In order to establish a secure connection we need:

- message encryption,
- message authentication,
- integrity and replay protection.

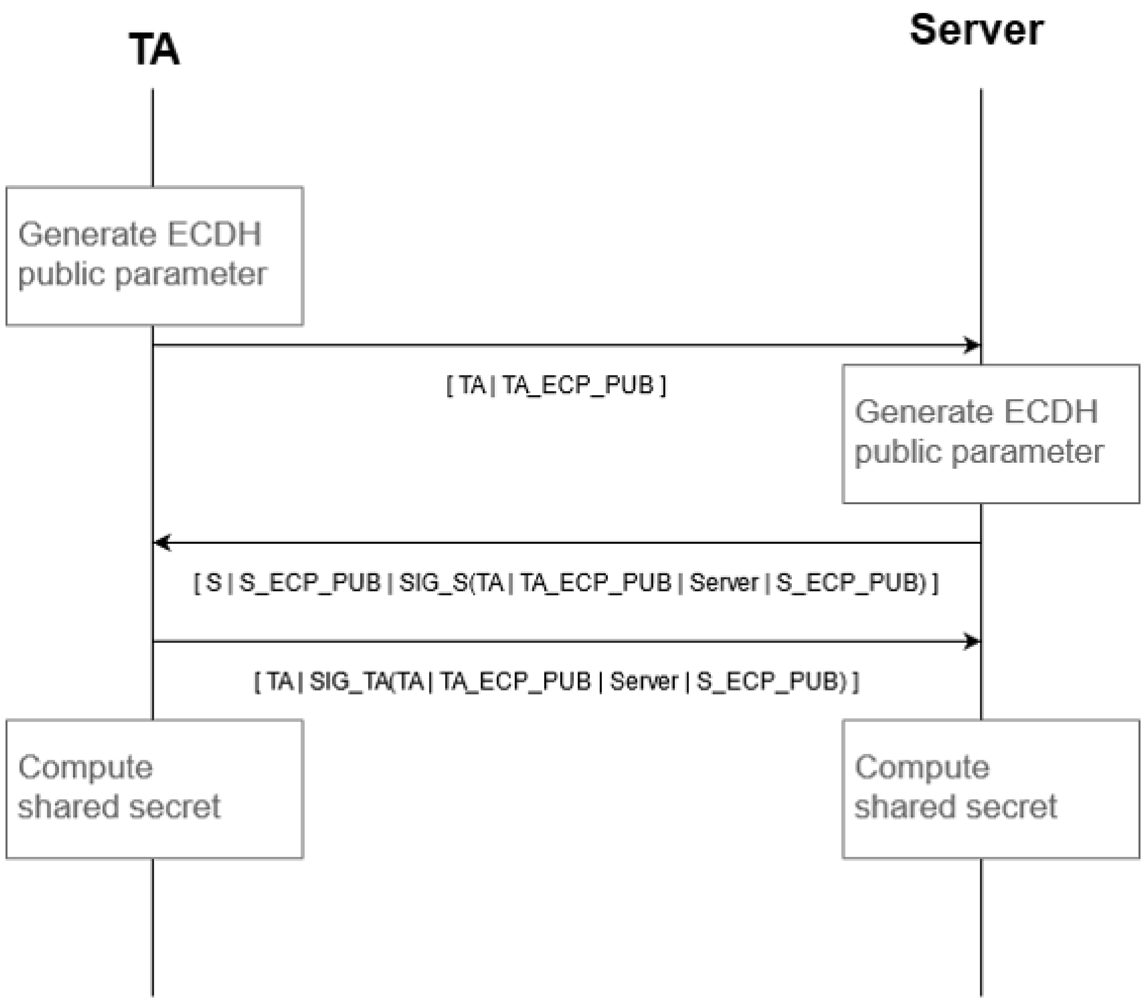

TA
Server
Generate ECDH public parameter
[TA | TA_ECP_PUB]
Generate ECDH public parameter
[S | S_ECP_PUB | SIG_S(TA | TA_ECP_PUB | Server | S_ECP_PUB)]
[TA | SIG_TA(TA | TA_ECP_PUB | Server | S_ECP_PUB)]
Compute shared secret
Compute shared secret

Figure 5.2: Authenticated Diffie-Hellman used for key exchange

The security protocol assumes, that both parties already exchanged their elliptic-curve signing key with each other. Elliptic-curve Diffie-Hellman (ECDH) is used in order to exchange a common secret between the Verifier and the Prover TA. The key exchange is authenticated with the previously exchanged keys, using Elliptic Curve Digital Signature Algorithm (ECDSA). The Trusted Application begins the exchange by generating its public ECDH parameter, and sending it to the Server with its identifier. After the Server recieved this message, it also generates a public ECDH parameter. The response sent to the the TA includes the identifier of the server and its public parameter, as well as the signature of all the previously exchanged data (identifier of the TA; public parameter of the TA; identifier of the Server; public parameter of the Server) signed with the Servers private key. In the last message, the TA sends it identifier as well as the signature calculated on the same parameters with its own private key. This process is displayed in Fig. 5.2.

After the common secret is established, both the Verifier and the Prover TA use the PBKDF2 key derivation function to derive two keys from the common secret: a 32-byte symmetric encryption key and a 32-byte message authentication key. The encryption key is used with the AES cipher in CBC mode. Since padding is required in CBC mode, messages are padded with PKCS#7 before being encrypted. The second key is used with HMAC with SHA-256 as the hash function in order to authenticate messages.

```
0 .. 1   ..    3   ..  7  ..  9   ..   13   ..  16
+----+---------+-------+------+--------+--------+
| ID | Version | Magic |  Seq | Length | Unused |
+----+---------+-------+------+--------+--------+
```

Listing 5.2: Header format used in the secure channel

A header format is also in use, in order to carry information such as sequence numbering. The exact header format is shown in Listings 5.2. The fields of this header include:

- *ID*: Identifies the message type.
- *Version*: Version of the attestation protocol. 1-byte for major and 1-byte for minor versioning.
- *Magic*: 4-byte magic value.
- *Seq*: 2-byte integer for sequence numbering.
- *Length*: 4-byte integer that contains the length of the whole message.
- *Unused*: Currently unused, leaves space for other fields in case a future version of this header is designed.

This header is used with every message sent over the secure network, during ECDH key exchange, as well as in the attestation request and response messages.

5.3 Verifier

The remote attestation is a protocol performed between two entities: the Verifier and the Prover. We have already established that the Prover will be a Trusted Application running on an IoT device supporting TEE, however there were not many details said about the Verifier yet. In our case the Verifier will be a central component, a server, in the architecture capable of handling the attestation of multiple IoT devices.

The Verifier can provide attestation to multiple IoT devices, hence the Verifier needs to keep track of each devices' progress in the protocol. This can easily be achieved with a stateful approach. The distinct states that we identified can be seen in Fig. 5.3.

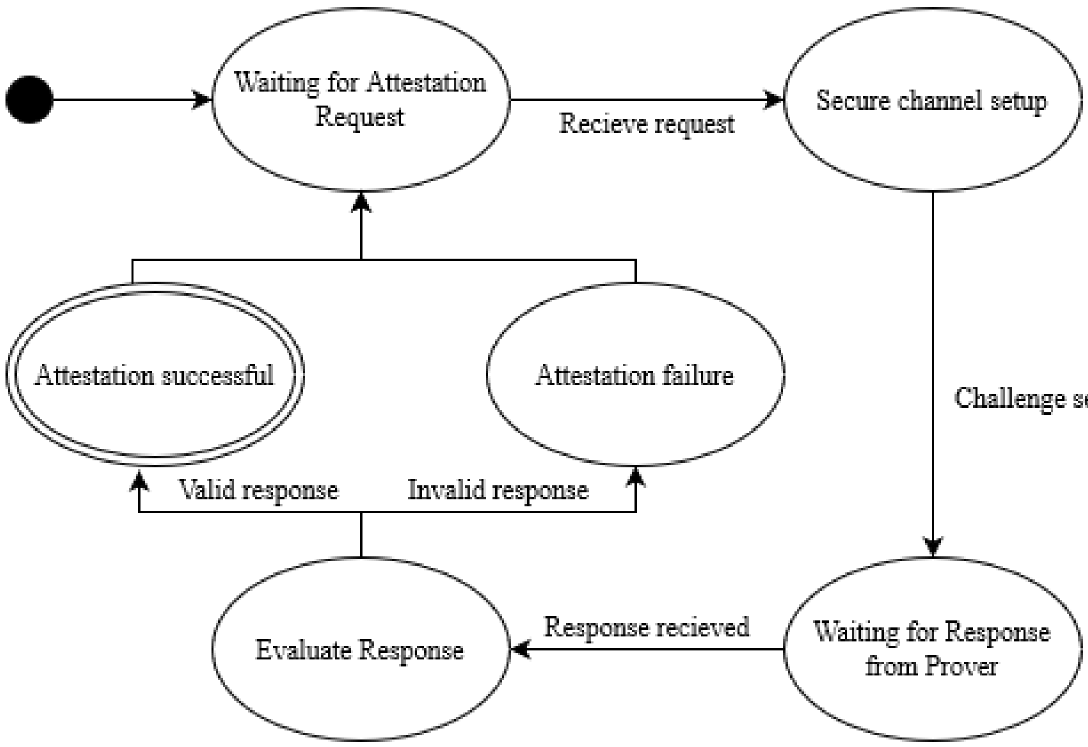


Figure 5.3: State machine of the Verifier

Every attestation is either started from the Prover TA inside the TEE, or by sending an initiating message to the REE, which then initializes the Prover TA, starting the attestation this way. In both cases, the Prover TA sends the first message in the protocol. Up until the Verifier receives the initial message, it is waiting for a request from a Prover. In the first message, the Prover TA initiates the secure channel setup, hence advancing the state of the Verifier. The setup of this secure channel is described in detail in Section 5.2. Once the secure channel is ready, the Verifier sends an attestation challenge to the Prover TA. The exact format of this message; how the Prover TA handles the message; how integrity checks are relayed to the Rootkit Detector TA; and the response format is detailed in Section 5.1. After receiving the challenge response, the Verifier evaluates it. The evaluation is done by performing all the calculations that the Prover TA was requested to do. This is the reason that the data stored on the Verifier server is essential. In Section 5.3.1 we discuss the data held on the server in more detail. Finally, the Verifier checks if the response sent by the Prover matches the local calculations. If at any given point, the response of the Prover TA differs from the expected, the attestation is deemed unsuccessful, otherwise the Prover passes. Following a successful attestation or after an error at any given place during the protocol, the server resets the state associated with the Prover.

5.3.1 Verifier data

Server side challenge verification requires data stored on the server that can help verify the state of the Prover. This could be done in various ways. In order to support all verification checks, the output of each check needs to be taken into consideration. The list of possible checks is provided in Section 5.1.

- The Verifier can request the list of running processes on the device. The result of this check is a list of the names of the running processes. On the Verifier server, we store a list of process names that are whitelisted, hence allowed on the devices. When verifying an attestation, process list names can be compared.
- There are multiple file system related checks, hashing directories and single files are both possibilities. The Verifier stores a mirror of the file system of each device. If any file system hashing is requested, the Verifier hashes the same path relative to where the mirror file system is stored, then the two outputs can be compared. In case an update of the IoT device under attestation is required, this way, the mirror file system should be replaced, and no further modification is required. We note that it makes sense to check only the non-volatile parts of the file system, such as the `/bin` folder, where the executables are located.
- Checking network hooks does not require any data on the Verifier, since checks are made on the device, and only the result is relayed to the Veirifer.
- Last, there are kernel related checks, such as hashing the kernels text segment. During the installation of the device, these hashes can be computed and stored on the Verifier. If changes are made to the device, that affect parts of the kernel under measurement, these hashes should be recomputed and updated on the Verifier.

5.4 Prover TA

The Prover TA is situated in the TEE of the IoT device under attestation. Its main purpose is:

1. to establish a secure connection with the Verifier,
2. relay the requested tasks to the Rootkit Detector TA
3. and send the results back to the Verifier.

The establishment of the secure channel is detailed in Section 5.2. It consists of running an authenticated version of ECDH key exchange with the Verifier, where the messages of the key exchange are signed with pre-shared ECC keypairs. Once the channel is ready, the Prover TA is ready to receive the attestation challenge from the Verifier. A single attestation challenge can contain one or more tasks. These tasks are parsed by the Prover TA, and executed one after the other. The result of each individual tasks is accumulated in a buffer. When the Prover TA finished all required tasks, it encrypts and signs the buffer, with all required formatting detailed in Section 5.2, and sends the response back to the Verifier.

Chapter 6

Integrity checks

The integrity checks implemented by the Rootkit Detector TA perform rootkit detection on the IoT device. Its successful execution guarantees the malware-free-state of the device. The Rootkit Detector TA is capable of accessing both REE and TEE, thus having access to the whole memory and the file system. More details about the implementation can be found in the paper by Roland Nagy et al. [15]. Using the previously mentioned capabilities, we implement several checks, each aiming to detect different rootkit techniques or ensuring the components of the system running inside the REE.

6.1 Process listing

One of the checks implemented by the Rootkit Detector TA, is to get the list of processes running on the device. The Linux kernel organizes the processes into so called tasks. Each task is approximately equivalent to a thread. Single-threaded applications consists of one task, while multi-threaded applications have one task for each thread. These tasks are organized into multiple dynamic data structures. According to the paper previously mentioned [15], the Rootkit Detector TA is currently able to list process IDs and names of the processes. This data is accumulated from the previously mentioned data structures. These structures include:

- Doubly-linked list, where each process has a `next` and `prev` pointer. You can traverse this list from the `init_task`, which is the first kernel thread started at boot.
- Process tree where the root node is the aforementioned `init_task`. When a process starts another process, it becomes its child, while the new process

refers to its spawner as parent. You can also get the list of processes by traversing this tree.

- Finally, you can extract this information from runqueues. Runqueues are used by the process scheduler. These structures do not hold every process, only runnable ones.

6.2 Memory integrity checks

The Rootkit Detector TA can also check the integrity of the device's memory. It checks for two memory areas of the Linux kernel frequently targeted by rootkits: the system call table and the kernel's text segment [15]. The system call table is an array of function pointers, pointing to system calls such as interaction with files, network sockets, etc.. Rootkits often replace pointers in this array in order to re-implement certain system calls. Another common attack employed by attackers is inline hooking [9]. In this case an attacker modifies the code of an existing function by rewriting the first few instructions and hijacking the control flow of the application. To detect such attacks, SHA-256 hash of both the system call table and the entire text segment of the kernel is computed [15]. If we detect alteration in the hashes, it signals that an adversary modified the device in a malicous way.

6.3 File system integrity checks

Another functionality provided by the Rootkit Detector TA is the ability to check the integrity of the file system. To do so, the implementation provides two functions, `hash_file` and `hash_dir` [15]. The first function expects an absolute path as its only parameter. If the given file exists, its SHA-256 hash is returned. The latter function has more arguments: besides the path parameter, one can optionally call the function recursively on all subdirectories. A blacklist can also be provided, so files or directories with volatile content can be avoided. For sake of consistency, all folder content is sorted alphabetically.

6.4 Network integrity checks

Lastly, checks targeting the network stack of the kernel, are also implemented by the Rootkit Detector TA. Rootkits usually implement two kind of attacks: hiding open connections, and implementing "magic packet" functionality. Roland Nagy et al. identified a way to hide open sockets as well as three mechanisms that can be abused by attackers to implement magic packets [15]. The most common way of implementing magic packets is using the Netfilter subsystem, which serves as the backbone of Linux firewall solutions. Firewall rules are stored in so called chains, where each chain contains an array of Netfilter hooks, storing function pointers. When a packet is checked against a chain, all hooks in the chain are involved, and the packet is only accepted, if all hooks accept it. An attacker can implement a rule, that executes his payload. The Rootkit Detector TA traverses all hooks of every chain. If any function pointers store a value pointing outside of the text segment of the kernel, it is considered to be a sign of rootkit presence [15]. Checks targeting hidden network connections are also implemented. Files in the `/proc/net` directory contain information about open network connections. In these files, objects store specific function pointers. Rootkits often target these to hide open connections, therefore the check of these function pointers is also implemented in the same manner as previously mentioned.

Chapter 7

Implementation

In this section, we present a prototype of the previously described system. In the upcoming sections, we assume the following setup:

- the Verifier is running on Ubuntu 18.04.6 LTS.
- The target architecture of our IoT device is ARM,
- the IoT device is emulated in QEMU[1].
- Open Portable Trusted Execution Environment[2] (OP-TEE) is used as the TEE implementation,
- the implementation of the REE comes with OP-TEE, which is a forked version of the Linux kernel containing OP-TEEs drivers.
- The mbedTLS[3] library is used as the cryptographic library of choice, due to its availability both on the server and inside OP-TEEs build environment.

We discuss the implementation of the Verifier and the Prover TA components in Section 7.1 and 7.2, respectively.

7.1 Verifier

In Section 5.3, we discussed the main purpose of the Verifier. To summarize, its tasks include:

[1]https://www.qemu.org/, Last visited: Oct 18, 2021.
[2]https://www.op-tee.org/, Last visited: Oct 18, 2021.
[3]https://github.com/ARMmbed/mbedtls, Last visited: Oct 18, 2021.

1. initiating or accepting an attestation with a remote device,
2. setting up a secure channel with the Prover TA,
3. generating a remote attestation challenge and sending it to the Prover TA,
4. recieving the attestation response and processing it.

In the current prototype initiating the remote attestation from the server is not yet possible. It would require the Attestation Service inside the REE to listen on a port for connections, as well as an option on the server to send an initiation message. Currently the attestation process is initiated from the command line of the IoT device through the Attestation Service, running in the REE, which is accessible within QEMU. The Attestation Services function is to start up the Prover TA by initiating a context switch. The Prover TA connects to the Verifier then, which is already listening on a predefined port.

At this point, the Verifier and the Prover TA run the key exchange described in Section 5.2. Both the Verifier and the Prover TA use ECC keypairs in order to sign the ECDH messages, which are generated ahead of time using OpenSSL[4]. The second and third messages of the key exchange are signed, however all three of them undergo header verification, such as checking for correct id, length and sequence numbering. After the public parameters of the ECDH[5] are exchanged the Verifier generates the shared secret. This shared secret is then used with PBKDF2[6] in order to generate two 32-byte keys. One is used for encryption with AES in CBC mode, the other is used for message authentication with HMAC using SHA-256. Since AES in CBC encrypts the data in 16-byte blocks, PKCS#7 is used to pad the data to be encrypted. This concludes the setup of the secure channel. For future messages, the attestation challenge and response, symmetric key encryption is used with message authentication.

The next step in the remote attestation protocol we designed, is to generate an attestation request, which is sent to the Prover TA. Randomizing the attestation request is a challenge in itself. In the prototype, we decided to store several precompiled attestation requests in a file. Each attestation request consists of multiple tasks, which are described in 5.1. We use the standard C library function `rand()`, which is seeded using time with `srand()` ahead of its usage, to select one of the attestation requests at random. The selected attestation is then parsed from the file, encrypted

[4]https://www.openssl.org/, Last visited: Oct 19, 2021.
[5]https://github.com/ARMmbed/mbedtls/blob/development/programs/pkey/ecdh_curve25519.c, Last visited: Oct 19, 2021.
[6]https://tls.mbed.org/api/pkcs5_8h.html, Last visited: Oct 19, 2021.

using AES in CBC mode, signed using HMAC with SHA-256 and packaged with the headers described in Section 5.2. I note, the header is not encrypted, however, it is included during the calculation of the signature.

After receiving the attestation response from the Prover TA, the last step is to verify the correctness of the response. The verification is based on the same attestation challenge that was sent to the device. The Verifier parses the same challenge, that was previously sent to the Prover TA. Based on the task and its parameters, there are four possibilities:

- If the check requested the hash of the syscall table or the hash of the kernels text segment, the hash is compared to a hardcoded hash.
- In case of Netfilter hooks checks, the Prover TA should return “0” in case of success. This is checked by comparing the result to a single “0” byte. If it matches, the devices passes the test.
- If the process list was requested by the Verifier, the result –list of proccess names– is compared to a whitelist. In case any process found on the device is missing from the whitelist, the device fails the checks.
- Last, the file system related checks are handled by calculating the hashes requested by the Verifier. This requires the Verifier to implement the same checks as detailed in the article describing the various integrity checks [15].

In contrast to the previously mentioned paper [15], which implements the file system checks inside OP-TEE, checks implemented on the Verifier are not limited by the TEEs file system access. The file system of the IoT device is stored on the Verifier, and file system related check parameters are prepended with a fix path, so the parameters point to the appropriate paths on the Verifier. The file hashing is implemented in the same way as in the Rootkit Detector TA with the exception of the RPC calls being required in order to access the file’s contents. Directory hashing is modified as well, files can be access similarly as in the case of file hashing. The given directory is opened and traversed using `opendir()` and `readdir()`, respectively, both being standard C library functions. After collecting each entry from the directory stream, the array of file paths are sorted using an adapted version of `strcmp()`. We note, that special directories, such as `./.` and `./..`, are excluded from the list in order to avoid hashing the same file twice and to avoid infinite loops. Additionally, sorting the array of file paths is crucial, since any difference between the Verifiers implementation and the implementation of the Rootkit Detector TA would lead to different hashes. Once the array of paths are accumulated, we use `stat()` to determine whether the given path points to a file or a directory. In case the `stat()` function finds a directory, and the recursive option is provided, the `hash_dir()` function is called recursively with an updated path. Other parameters, such as the recursive option and the blacklist, are passed from the previous call.

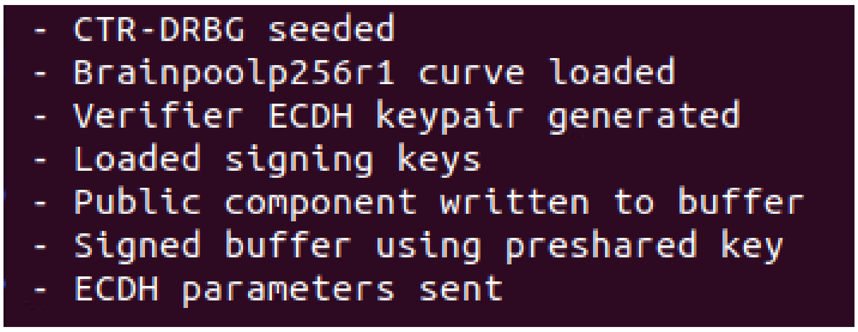
7.2 Prover TA

Prover TA is the component running inside the TEE of the IoT device. It can be initiated from the REE using the Attestation Service. Once the control is passed to the TEE, the Prover TA begins with setting up the secure channel with the Verifier. This phase consists of running an authenticated version of ECDH key exchange, where the second and third messages are authenticated using ECDSA with preshared ECC keypairs. After the common secret is established, the Prover TA generates two 32-byte keys using the common secret with PBKDF2. The two keys are used for encryption and message authentication with AES in CBC mode and HMAC with SHA-256, respectively. PKCS#7 is used to pad the data to be encrypted with AES-CBC, since CBC mode requires the data to be in 16-byte chunks. After the secure channel is established, the Prover TA receives the attestation challenge sent by the Verifier. The challenge consists of multiple tasks, each separated with a newline character. The exact format of the challenge is detailed in Section 5.1. To parse the challenge we implemented `strtok()`, which is part of the standard C library, however only part of the library is available inside the TEE. Some library functions, mainly the ones that require dynamic memory allocation, are not implemented inside the TEE, since the C memory allocation functions, such as `malloc()` and `free()` are not available. TEE implements these functions in a more secure manner, and the maintainers have not yet implemented some of the required library functions, such as `strtok()` and `atoi()`, which we use to parse the challenge sent by the Verifier. We implemented all necessary standard library functions with TEE specific memory allocation, which is documented in the TEE Internal Core API Specification [11]. The attestation request is first parsed into lines using `strtok()`, where each line contains a separate task. Parameters are extracted from each line, if they have any, then the appropriate method of the Rootkit Detector TA is called. The Rootkit Detector TA exposes its methods through an API. They are accessible using `TEE_OpenTASession()` and `TEE_InvokeTACommand()`, which are documented in the Core API Specification [11]. We implemented wrappers for these functions. The result of each function call is accumulated in a buffer. Once all challenges are calculated, the buffer containing all responses is encrypted, prepended with the header, which is described in Section 5.2, and signed. We note, that the signature is calculated on both the header and the encrypted data. Last, the response is sent back to the Verifier. After sending the response, the Prover TA terminates, handing back the execution to the Attestation Service.

Chapter 8

Evaluation and discussion

In this Section, we evaluate the implementation, which is described in the previous Section. We discuss how the secure channel is set up, and used to transfer data between the Verifier and the Prover TA. Finally, we evaluate the result of file system related integrity checks.

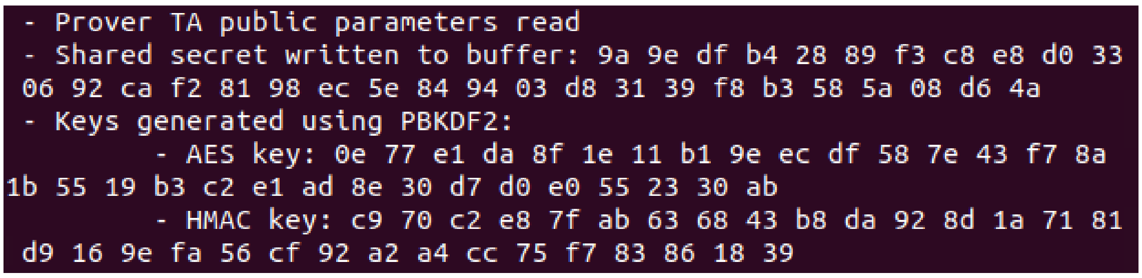


```
 - CTR-DRBG seeded
 - Brainpoolp256r1 curve loaded
 - Verifier ECDH keypair generated
 - Loaded signing keys
 - Public component written to buffer
 - Signed buffer using preshared key
 - ECDH parameters sent
```

Figure 8.1: Authenticated ECDH key exchange implementation of the Verifier

In order to communicate between the Verifier and the Prover TA, a secure channel is set up. This channel uses symmetric cryptography, AES in CBC mode with HMAC message authentication, in order to ensure only the attestation participants can read the messages correctly. This requires the two parties to generate a common secret. For this ECDH is used with message authentication, where the latter is achieved with ECDSA signatures. In Fig. 8.1, the basics of the ECDH implementation is shown. Both parties seed a pseudo-random number generator, load the predefined curve, we chose this curve to be BrainpoolP256r1[1], generate both public and private domain parameters, then exchange the public parameters with each other in an authenticated manner.

[1]https://datatracker.ietf.org/doc/html/rfc5639#section-3.4, Last visited: Oct 20, 2021.



```
 - Prover TA public parameters read
 - Shared secret written to buffer: 9a 9e df b4 28 89 f3 c8 e8 d0 33
 06 92 ca f2 81 98 ec 5e 84 94 03 d8 31 39 f8 b3 58 5a 08 d6 4a
 - Keys generated using PBKDF2:
        - AES key: 0e 77 e1 da 8f 1e 11 b1 9e ec df 58 7e 43 f7 8a
1b 55 19 b3 c2 e1 ad 8e 30 d7 d0 e0 55 23 30 ab
        - HMAC key: c9 70 c2 e8 7f ab 63 68 43 b8 da 92 8d 1a 71 81
 d9 16 9e fa 56 cf 92 a2 a4 cc 75 f7 83 86 18 39
```

Figure 8.2: Generation of symmetric cryptography keys on the Verifier

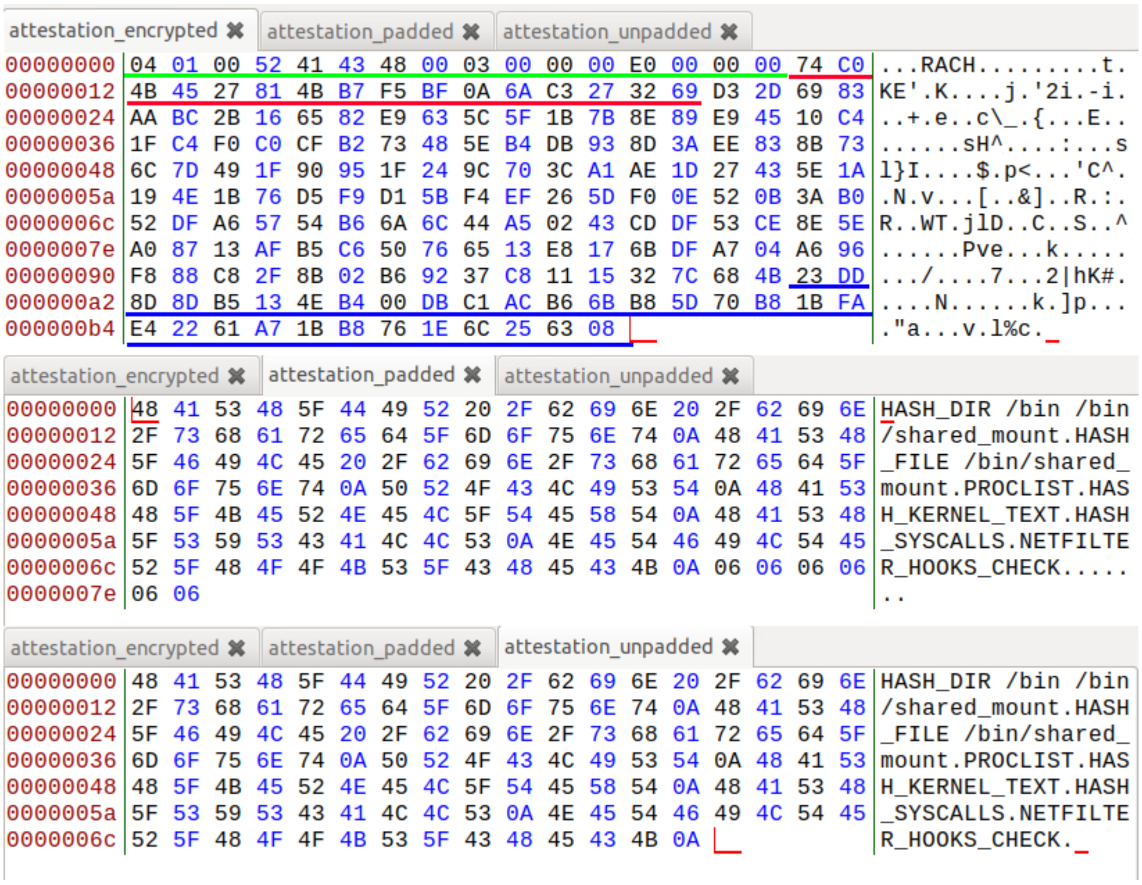
Each party then loads the public parameters they received, and finish the key exchange, thus calculate the common secret. This common secret is then used to generate the symmetric encryption keys used to encrypt and authenticate messages. This process is shown in Fig. 8.2. We note, that both screenshots are taken from the Verifier's console, however the process is the same on the Prover TA aswell.

This secure channel is used to transfer the attestation request and response messages. Fig 8.3 shows the phases receiving and decrypting the attestation request. The message transferred on the secure channel contains several fields, which get checked before decryption. The header field of the request is underlined with green, the Initialization Vector (IV), required to decrypt the message, is underlined with red, and the HMAC signature of the message is underlined with blue. After successful validation of both the header values and the HMAC, the message can be decrypted, revealing the padded attestation message. The attestation is now in a human-readable form[2], however before parsing, the PKCS#7 padding needs to be removed.

The file system related checks are verified by computing the same tasks on the Verifier, that was requested from the Prover TA. This required the reimplementation of file system related checks, such as `hash_dir()` and `hash_file()`. During implementation, Roland Nagy, the author of the Rootkit Detector TA [15] and we made sure, that the same result is achieved both on the Verifier and on the IoT device. We traverse the directories in the same fashion and use the same string comparison method to sort the list of files in a directory, which is necessary in order to obtain the same hash. The only difference being that the absolute path to each file to be hashed is prefixed with the location of where the mirror file system is actually stored.

In Fig. 8.4, we show a side-by-side comparison of the calculations made on the Verifier and on the IoT device. The Verifiers and the Prover TAs output is shown on the left and right side console, respectively. Both sides parse the task in the

[2] 0x0a is the hexadecimal value for Line Feed (LF), more commonly known as `\n` on Linux



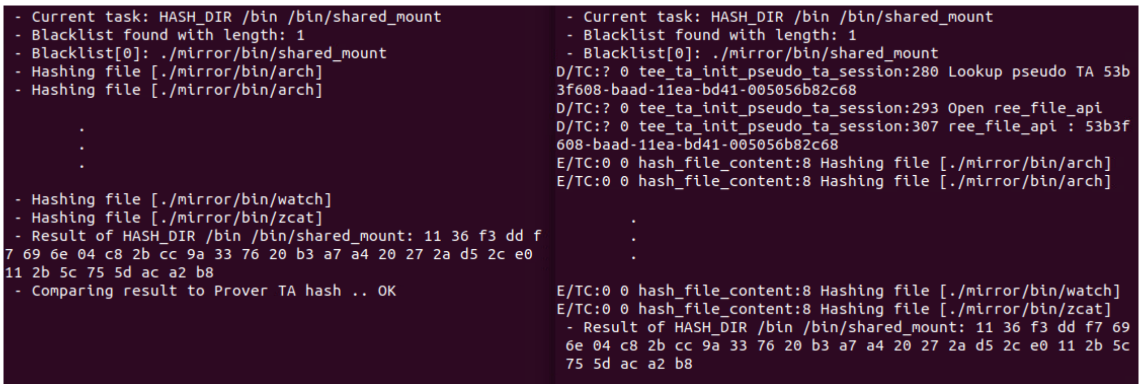
```
00000000 04 01 00 52 41 43 48 00 03 00 00 00 E0 00 00 00 74 C0 ...RACH.........t.
00000012 4B 45 27 81 4B B7 F5 BF 0A 6A C3 27 32 69 D3 2D 69 83 KE'.K....j.'2i.-i.
00000024 AA BC 2B 16 65 82 E9 63 5C 5F 1B 7B 8E 89 E9 45 10 C4 ..+.e..c\_.{...E..
00000036 1F C4 F0 C0 CF B2 73 48 5E B4 DB 93 8D 3A EE 83 8B 73 ......sH^....:...s
00000048 6C 7D 49 1F 90 95 1F 24 9C 70 3C A1 AE 1D 27 43 5E 1A l}I....$.p<...'C^.
0000005a 19 4E 1B 76 D5 F9 D1 5B F4 EF 26 5D F0 0E 52 0B 3A B0 .N.v...[..&]..R.:.
0000006c 52 DF A6 57 54 B6 6A 6C 44 A5 02 43 CD DF 53 CE 8E 5E R..WT.jlD..C..S..^
0000007e A0 87 13 AF B5 C6 50 76 65 13 E8 17 6B DF A7 04 A6 96 ......Pve...k.....
00000090 F8 88 C8 2F 8B 02 B6 92 37 C8 11 15 32 7C 68 4B 23 DD .../....7...2|hK#.
000000a2 8D 8D B5 13 4E B4 00 DB C1 AC B6 6B B8 5D 70 B8 1B FA ....N......k.]p...
000000b4 E4 22 61 A7 1B B8 76 1E 6C 25 63 08                   ."a...v.l%c.
```

```
00000000 48 41 53 48 5F 44 49 52 20 2F 62 69 6E 20 2F 62 69 6E HASH_DIR /bin /bin
00000012 2F 73 68 61 72 65 64 5F 6D 6F 75 6E 74 0A 48 41 53 48 /shared_mount.HASH
00000024 5F 46 49 4C 45 20 2F 62 69 6E 2F 73 68 61 72 65 64 5F _FILE /bin/shared_
00000036 6D 6F 75 6E 74 0A 50 52 4F 43 4C 49 53 54 0A 48 41 53 mount.PROCLIST.HAS
00000048 48 5F 4B 45 52 4E 45 4C 5F 54 45 58 54 0A 48 41 53 48 H_KERNEL_TEXT.HASH
0000005a 5F 53 59 53 43 41 4C 4C 53 0A 4E 45 54 46 49 4C 54 45 _SYSCALLS.NETFILTE
0000006c 52 5F 48 4F 4F 4B 53 5F 43 48 45 43 4B 0A 06 06 06 06 R_HOOKS_CHECK.....
0000007e 06 06                                                 ..
```

```
00000000 48 41 53 48 5F 44 49 52 20 2F 62 69 6E 20 2F 62 69 6E HASH_DIR /bin /bin
00000012 2F 73 68 61 72 65 64 5F 6D 6F 75 6E 74 0A 48 41 53 48 /shared_mount.HASH
00000024 5F 46 49 4C 45 20 2F 62 69 6E 2F 73 68 61 72 65 64 5F _FILE /bin/shared_
00000036 6D 6F 75 6E 74 0A 50 52 4F 43 4C 49 53 54 0A 48 41 53 mount.PROCLIST.HAS
00000048 48 5F 4B 45 52 4E 45 4C 5F 54 45 58 54 0A 48 41 53 48 H_KERNEL_TEXT.HASH
0000005a 5F 53 59 53 43 41 4C 4C 53 0A 4E 45 54 46 49 4C 54 45 _SYSCALLS.NETFILTE
0000006c 52 5F 48 4F 4F 4B 53 5F 43 48 45 43 4B 0A             R_HOOKS_CHECK.
```

Figure 8.3: Attestation request received by the Prover TA

same manner. The Prover TA then delegates the task to the Rootkit Detector TA, which is exposed to us via the `ree_file_api`. File hashing is done in the same manner, except the Verifier prefixes the path to the previously mirrored file system of the device. After the Verifier receives the calculations of the device, the produced hashes are compared. In our case, the hashes match, and the challenge is completed successfully.



```
 - Current task: HASH_DIR /bin /bin/shared_mount
 - Blacklist found with length: 1
 - Blacklist[0]: ./mirror/bin/shared_mount
 - Hashing file [./mirror/bin/arch]
 - Hashing file [./mirror/bin/arch]

        .
        .
        .

 - Hashing file [./mirror/bin/watch]
 - Hashing file [./mirror/bin/zcat]
 - Result of HASH_DIR /bin /bin/shared_mount: 11 36 f3 dd f
7 69 6e 04 c8 2b cc 9a 33 76 20 b3 a7 a4 20 27 2a d5 2c e0
11 2b 5c 75 5d ac a2 b8
 - Comparing result to Prover TA hash .. OK
```

```
 - Current task: HASH_DIR /bin /bin/shared_mount
 - Blacklist found with length: 1
 - Blacklist[0]: ./mirror/bin/shared_mount
D/TC:? 0 tee_ta_init_pseudo_ta_session:280 Lookup pseudo TA 53b
3f608-baad-11ea-bd41-005056b82c68
D/TC:? 0 tee_ta_init_pseudo_ta_session:293 Open ree_file_api
D/TC:? 0 tee_ta_init_pseudo_ta_session:307 ree_file_api : 53b3f
608-baad-11ea-bd41-005056b82c68
E/TC:0 0 hash_file_content:8 Hashing file [./mirror/bin/arch]
E/TC:0 0 hash_file_content:8 Hashing file [./mirror/bin/arch]

        .
        .
        .

E/TC:0 0 hash_file_content:8 Hashing file [./mirror/bin/watch]
E/TC:0 0 hash_file_content:8 Hashing file [./mirror/bin/zcat]
 - Result of HASH_DIR /bin /bin/shared_mount: 11 36 f3 dd f7 69
 6e 04 c8 2b cc 9a 33 76 20 b3 a7 a4 20 27 2a d5 2c e0 11 2b 5c
 75 5d ac a2 b8
```

Figure 8.4: HASH_DIR task calculation and verification

Finally, to evaluate the security guarantees provided by our TEE-based remote attestation, we evaluated how the solution stacks up the the minimum requirements of a remote attestation described in "A minimalist approach to remote attestation" [7]:

1. Exclusive access to the attestation key is provided, since it is generated with authenticated ECDH key exchange, and the key used to sign the attestation remains only available for the Prover TA inside the TEE, being inaccessible from the potentially compromised REE.

2. Strict memory separation is enforced by the TEE, and memory allocation and erasure is safely implemented inside the TEE according to the TEE Internal Core API Specification [11].

3. Trusted Applications are integrity protected by the TEE, ensuring similar properties as a ROM would. This means, that neither the Prover TA nor the Rootkit Detector TA can be modified from the REE, hence protecting the calculation of attestation.

4. Interrupts inside the TEE can be disabled and re-enabled. Disabling them when the Rootkit Detector TA is invoked guarantees uninterrupted execution of our integrity checks. This is however only available inside the TEE, which poses some threats during our file system checks, which we discuss later.

5. Controlled invocation is also provided by the TEE, ensuring the correct entry point and execution of our remote attestation.

Unfortunately, the prototype we presented has two known weaknesses, that undermine the 4th requirement. The first weakness occurs during the file system checks.

Since the file system is by default not reachable from the TEE in OP-TEEs implementation, we are forced to delegate the checks to the REE via RPC calls, where they can actually be interrupted. On top of that, our file operations take a long time, which means that they will most likely be interrupted by the task scheduler. We note, that this is the case with OP-TEE, other TEE implementations may provide a secure way of accessing the file system from within the TEE.

The other possible weakness stems from the multi-core processors themselves. On multi-core processors, such as most ARM processors, other, potentially malicious, processes may run parallel to the execution of our integrity checks. For example, a malware running parallel to the Rootkit Detector TA could remove hooks from the system call table before it is hashed, then put the hooks back once hashing is completed.

Chapter 9

Conclusion

In this document, we proposed a TEE-based remote attestation scheme for IoT devices. Prior to designing the system, we assessed the possible threats to our scheme, constructed an attacker model, and identified the security requirements. The proposed TEE-based remote attestation is a hybrid attestation scheme. It is mostly based on software and relies only on the hardware requirements of the TEE itself. Considering that TEEs are already widely supported, our scheme can be considered as affordable and viable for many IoT solutions.

We showed that our TEE-based remote attestation scheme has similar security properties to those of HYDRA, a hybrid remote attestation scheme proposed in 2017. Our scheme however, is capable of performing more complex integrity checks, aiming to detect rootkits both in memory and in persistent storage. While our prototype implementation is capable of detecting rootkits both in memory and in persistent storage, unfortunately, it has some weaknesses due to the TEE implementation we use, OP-TEE, as well as the parallelism provided by multi-core processors. We note that with proper file system availability inside the TEE, and some limitations on the processors themselves during attestation, these problems could be mitigated.

Currently our implementation is tested on, and supports only a single device. Future work includes the handling of multiple devices simultaneously, and to provide a console interface on the Verifier, in order to interact with the system.

Bibliography

[1] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai botnet. In *USENIX Security Symposium*, pages 1093–1110. USENIX Association, August 2017.

[2] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *ACM Conference on Computer and Communications Security (CCS)*, pages 400–409, 2009. DOI: `10.1145/1653662.1653711`.

[3] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *Network and Distributed Systems Symposium (NDSS)*, 2012.

[4] Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. HYDRA: Hybrid design for remote attestation (using a formally verified microkernel). In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec)*, pages 99–110, 2017. DOI: `10.1145/3098243.3098261`.

[5] ENISA. Guidelines for securing the Internet of Things. ENISA study, November 2020.

[6] ETSI. CYBER; Cyber security for consumer Internet of Things: Baseline requirements. ETSI TS 103 645 v2.1.2, June 2020.

[7] Aurélien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. A minimalist approach to remote attestation. In *Conference on Design, Automation & Test in Europe (DATE)*, pages 1–6, 2014.

[8] Global Platform. Security evaluation standard for IoT platforms v1.1 (SESIP). Global Platform Standard, June 2021.

[9] J. Gu, M. Xian, T. Chen, and R. Du. A Linux rootkit improvement based on inline hook. In *Proceedings of the 2nd International Conference on Advances in Mechanical Engineering and Industrial Informatics*, pages 793–798. Atlantis Press, 2016. DOI: `10.2991/ameii-16.2016.155`.

[10] Vikas Hassija, Vinay Chamola, Vikas Saxena, Divyansh Jain, Pranav Goyal, and Biplab Sikdar. A survey on IoT security: Application areas, security threats, and solution architectures. *IEEE Access*, 7:82721–82743, 2019. DOI: `10.1109/ACCESS.2019.2924045`.

[11] GlobalPlatform Inc. Globalplatform technology tee internal core api specification version 1.1.2.50 (target v1.2). `https://globalplatform.org/wp-content/uploads/2018/06/GPD_TEE_Internal_Core_API_Specification_v1.1.2.50_PublicReview.pdf`. Last visited: Oct. 19, 2021.

[12] European Telecommunications Standards Institute. Etsi ts 102 165-1 v5.2.3 (2017-10) cyber; methods and protocols; part 1: Method and pro forma for threat, vulnerability, risk analysis (tvra). `https://www.etsi.org/deliver/etsi_ts/102100_102199/10216501/05.02.03_60/ts_10216501v050203p.pdf`. Last visited: Oct. 12, 2021.

[13] Yanlin Li, Yueqiang Cheng, Virgil Gligor, and Adrian Perrig. Establishing software-only root of trust on embedded systems: Facts and fiction. In *International Workshop on Security Protocols*, pages 50–68, 2015. DOI: `10.1007/978-3-319-26096-9\_7`.

[14] Microsoft. Stride, threat modelling framework by microsoft. `https://docs.microsoft.com/en-us/azure/security/develop/threat-modeling-tool-threats#stride-model`. Last visited: Oct. 12, 2021.

[15] Roland Nagy, Krisztián Németh, Dorottya Papp, and Levente Buttyán. Rootkit detection on embedded IoT devices. *Acta Cybernetica*, August 2021. DOI: `10.14232/actacyb.288834`.

[16] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–282, 2004. DOI: `10.1109/SECPRI.2004.1301329`.

[17] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered

code execution on legacy systems. *ACM SIGOPS Operating Systems Review*, 35(5):1–16, 2005. DOI: `10.1145/1095809.1095812`.

[18] Sean Turner. Transport layer security. *IEEE Internet Computing*, 18(6):60–63, 2014. DOI: `10.1109/MIC.2014.126`.