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

Rootkit detection with memory snapshot inspection on embedded IoT devices

Master's Thesis

Author
Roland Nagy

Advisor
dr. Levente Buttyán

May 11, 2021

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Nagy Roland*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2021. május 11.

Nagy Roland
hallgató

Kivonat

IoT rendszereket is érnek kibertámadások, mint például rootkit fertőzés. A rootkit-ek olyan káros szoftverek, amik tipikusan magas jogosultsággal futnak, ezért nehéz őket detektálni. Ebben a dolgozatban ezzel a problémával foglalkozunk: egy olyan rootkit detekciós eljárást ismertetünk, ami egy trusted execution environmnet (TEE) segítségével kínál megoldást erre a problémára IoT eszközök számára. A TEE egy izolált környezetet nyújt a detekciós algoritmusunk számára, és meggátolja, hogy rootkit-ek beavatkozzanak annak futásába, még ha a rootkit root jogokkal is fut. A detekciós eljárásunk olyan módosításokat észlel, amiket egy rootkit a kernel kódjában, felhasználói programokban, vagy olyan adatokban visz véghez, amik befolyásolják a kernel kódjának végrehajtását (pl. rendszerhívások hook-olása). Emellett képes detektálni, ha egy rootkit inkonzisztens állapotban hagy bizonyos adatstruktúrákat (pl. azokat, amelyekben a folyamatokkal kapcsolatos információk tárolódnak). Emellett rootkit-ek perzisztens komponenseit is detektálja az általunk bemutatott megoldás. Amellett, hogy bemutatjuk, hogyan terveztük meg ezt a detekciós alkalmazást, bemutatunk egy prototípust is, amely implementálja ezeket az ellenőrzéseket, valamint azt, hogy hogyan teszteltük az alkalmazásunkat több olyan rootkit segítségével, amiket direkt erre a célra fejlesztettünk.

Abstract

IoT systems are subject to cyber attacks, including infecting embedded IoT devices with rootkits. Rootkits are malicious software that typically run with elevated privileges, which makes their detection challenging. In this paper, we address this challenge: we propose a rootkit detection approach for embedded IoT devices that takes advantage of a trusted execution environment (TEE), which is often supported on popular IoT platforms, such as ARM based embedded boards. The TEE provides an isolated environment for our rootkit detection algorithms, and prevents the rootkit from interfering with their execution even if the rootkit has root privileges on the untrusted part of the IoT device. Our rootkit detection algorithms identify modifications made by the rootkit to the code of the operating system kernel, to system programs, and to data influencing the control flow (e.g., hooking system calls), as well as inconsistencies created by the rootkit in certain kernel data structures (e.g., those responsible to handle process related information). We also use algorithms to detect rootkit components in the persistent storage of the device. Besides describing our approach and algorithms in details, we also report on a prototype implementation and on the evaluation of our design and implementation, which is based on testing our prototype with rootkits that we developed for this purpose.

Chapter 1

Introduction

The Internet has grown beyond a network of laptops, PCs, and large servers: it also connects millions of small embedded devices. This new trend is called the Internet of Things, or IoT in short, and it enables many new and exciting applications such as smart homes, intelligent transportation systems, smart factories, and personalized healthcare. At the same time, IoT also comes with a number of risks related to information security. The lack of security, however, cannot be tolerated in certain applications of IoT, including those in the domains of healthcare, transportation, and industrial automation. In such applications, security failures may lead to substantial monetary loss, physical damage of expensive equipment, or even loss of human life. Therefore, one of the biggest challenges today, which hinders the application of IoT technologies in many cases, is the lack of security guarantees.

Unfortunately, IoT systems are notoriously insecure. One of the reasons is that they are built from cheap embedded devices that are easy to compromise by exploiting weaknesses in the way they are operated and vulnerabilities of the software components running on them. A consequence of this is that malware for IoT devices has appeared [4, 20] and gaining momentum [29]. Malware designed for IoT devices is similar to malware designed for other types of computers: it compromises the integrity of the device by installing unwanted, and potentially harmful, software components on it. These software components can then be used to cause other types of compromise such as allowing the attacker to access the device remotely by installing a backdoor, tampering with messages sent by the device to other devices, or making data stored on the device unavailable by deleting or encrypting them.

Sophisticated malware tries to maintain its presence on infected devices while remaining invisible for the operators of those devices. This sort of malware is called *rootkit* [8]. Typically, rootkits run with elevated (root) privileges and they modify system commands and/or code and various data structures in the operating system (OS) kernel such that their files and running processes do not appear in the output of various system tools used to monitor the operation of the devices. Detecting such a rootkit is challenging, mainly because any detection program running at the same or lower privilege levels than the rootkit may also be compromised or may be misled by the tricks used by the rootkit to hide itself.

In this work, we aim at rootkit detection on embedded IoT devices, and we address the above challenge by running our rootkit detection mechanisms in a Trusted Execution Environment (TEE), which is isolated from the main OS of the device, and hence the rootkit – even running with root privileges on the main OS – cannot interfere with its operation. Such a TEE is supported on many embedded platforms, including the popular ARM platform that supports the establishment of TEEs by its TrustZone[1] technology. A TrustZone enabled ARM processor can execute in two modes: in untrusted mode (called Normal World), it runs a common OS (e.g., Linux) and applications on top of it (often referred to as the Rich Execution Environment, or REE for short), whereas in trusted mode (called Secure World), it runs the trusted OS and the trusted applications of the TEE. Isolation between these two modes are ensured by hardware based protection mechanisms. As a result, software components running in the Normal World cannot access some of the resources (including a part of the system memory) of the device, whereas trusted software components of the TEE running in the Secure World have unlimited access to all resources. Thus, the TEE provides two advantages for rootkit detection: it can protect the integrity of some trusted rootkit detection code by keeping it inaccessible for potentially malicious software running in the REE, and it can provide a safe execution environment for that rootkit detection code where it can access and inspect all system resources.

However, as the same processor is shared between the trusted and the untrusted mode, the execution of untrusted software is suspended when the processor switches to trusted mode and starts executing trusted software. This means that our trusted rootkit detection code cannot observe the behavior of untrusted software components during their execution, but it can only inspect their memory images reflecting their state at the time of their suspension. In other words, our rootkit detection approach is based on analyzing memory snapshots of the untrusted system (OS and applications), and it consists of identifying the anomalies caused by the rootkit in the state of the kernel data structures representing processes, as well as computing the hash values of the memory images of running processes and comparing them to known good hash values stored safely in the TEE. In addition, as the rootkit may delete all its components from the memory before our rootkit detection code is invoked and save itself to persistent storage for later execution, we also scan and hash files stored on the flash disk of the device from within the TEE, and compare the computed hash values to known good hashes stored safely in the TEE.

The rest of the thesis is organized the following way: In Chapter 2, we introduce the main features of rootkits and describe the challenges what make their detection complicated. Chapter 3 will describe multiple techniques rootkits can apply to hide certain resources, and our solutions targeting these methods. The prototype we developed attempts to detect persistent components of rootkits as well, however this part will not be described

[1]https://developer.arm.com/ip-products/security-ip/trustzone, Last visited: 20.09.2020

in details. More information about the persistence check can be found in the thesis of Krisztián Németh [24]. Chapter 4 contains details about OP-TEE[2], the Trusted Execution Environment we used to implement our solution, the test environments we used for development and evaluation and detailed description of components and patches we had to use in order to be able to implement our detection solution securely and properly. In Chapter 5, we present rootkits we used to test our solution; some of them are open source, or ports of existing solutions, but most of them were developed by us to demonstrate the capabilities and limits of our solution against common rootkit techniques. In Chapter 6 we discuss future work and possible improvements, while in Chapter 7 we conclude the thesis.

[2]https://www.op-tee.org, Last visited: 09.12.2020

Chapter 2

Rootkits

The term *rootkit* [26] refers to malware or modules of pieces of malware whose primary goal is to maintain stealth on infected devices while allowing continued access to its resources. Remaining hidden is in the best interests of the attackers: if the operators detect the infection, they will try to eliminate the responsible pieces of malware by reinstalling the system, and they will also try to patch the exploited vulnerability rendering the system unavailable for the attackers.

Rootkits employ a wide variety of techniques to remain hidden on infected devices [10, 26, 27]. *User-mode* techniques target the user space of devices and include techniques such as the manipulation of log files, modification of disk-resident system files (e.g., `ls`, `top`) or hooking libraries used by executables. One well-known user space hooking technique is the abuse of the `LD_PRELOAD` environment variable to load malicious libraries before benign ones, therefore overriding their provided functionalities. This technique is used by rootkits such as Azazel[1], Jynx2 [11], BEURK[2], vlany[3] and bedevil[4]. Other rootkits, such as HORSEPILL [23], abuse valid kernel features offered to user space programs to maintain stealth presence on the device.

Kernel-mode techniques, on the other hand, target the internal data structures and functionalities in the operating system's kernel. In the past, modifying the kernel memory image was a widely used technique to remain hidden on Linux systems. However, recent Linux kernel versions restrict access to `/dev/mem` and `/dev/kmem`[5], mitigating this attack. Other techniques in this category are kernel-space hooking [30] and direct kernel object manipulation. The former includes the use of malicious device drivers and the modification of the system call and interrupt descriptor table. Direct kernel object manipulation tampers with the integrity of the kernel by targeting dynamic kernel data structures. This technique can be used, for example, to hide processes from the system administra-

[1]https://github.com/chokepoint/azazel, Last visited: 16.09.2020
[2]https://github.com/unix-thrust/beurk, Last visited: 16.09.2020
[3]https://github.com/mempodippy/vlany, Last visited: 16.09.2020
[4]https://github.com/wiperpaul/bdvl, Last visited: 16.09.2020
[5]https://lore.kernel.org/lkml/18778.1508769258@warthog.procyon.org.uk/, Last visited: 16.09.2020

tor. There exist a number of rootkits employing these techniques, including Adore-NG[6], LilyOfTheValley[7], the work presented in [18], OSOM [16], SucKIT [12] and Suterusu[8].

Many proof-of-concept rootkits compromise the virtualization layer, the BIOS and/or the firmware of hardware components [15, 19, 22, 21]. Such rootkits are called *OS-independent* rootkits and their advantages are manifold. Rootkits in the lowest-level components can survive reboots and re-installations, and they leave no traces on the disk. Their detection is particularly challenging because they do not make visible changes to the operating system.

In response to the rising threat of rootkits, a number of detection methods have been proposed [28]. *Signature-based* methods scan the files on the disk for byte sequences and use a signature database to detect known rootkits. Tools that implement this method include chkrootkit[9] and Rootkit Hunter[10]. The main limitation of signature-based methods is similar to those of signature-based intrusion detection systems and virus scanners, namely, that they cannot detect very recent or sufficiently modified old rootkits.

Behavior-based detection methods detect deviations from "normal" patterns of system behavior [17], e.g. timing discrepancies and irregularities in resources such as the Translation Lookaside Buffer (TLB). Such methods, however, require *a priori* measurements of the analyzed system in a controlled environment. Differences between the real and the controlled environment can decrease the accuracy of such approaches. The baseline measurements must also be stored securely on the device, otherwise, malware can influence the detection method.

Cross-view-based methods assume that there is no perfect rootkit which can perfectly emulate all aspects of the system. In order to detect compromises, they enumerate system parameters in at least two different ways and compare the results. However, the kernel consists of many dynamically changing structures: a change in an enumerated structure during cross-checking can lead to false alarms. To overcome this challenge, Carbonite[11] preempts scheduling, thereby prohibiting new processes to spawn. However, such an approach has an impact on performance.

Integrity-based detection methods compare a snapshot with a trusted baseline. In the case of files, the trusted baseline can be the hash value of a file computed in a controlled environment, which can be checked with tools such as Samhain[12]. Kernel data structures can also be monitored for malicious changes, e.g. system call re-maps can be detected using StMichael[13], running processes can be listed with KSTAT[14], and Gibraltar [6] uses

[6]https://github.com/yaoyumeng/adore-ng, Last visited: 16.09.2020

[7]https://github.com/En14c/LilyOfTheValley, Last visited: 16.09.2020

[8]https://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/, Last visited: 16.09.2020

[9]http://www.chkrootkit.org/, Last visited: 17.09.2020

[10]http://rkhunter.sourceforge.net/, Last visited: 17.09.2020

[11]https://securiteam.com/tools/5jp0m1f40e/, Last visited: 21.09.2020

[12]https://www.la-samhna.de/samhain/, Last visited: 17.09.2020

[13]https://sourceforge.net/projects/stjude/, Last visited: 21.09.2020

[14]http://www.s0ftpj.org/docs/lkm.htm, Last visited: 17.09.2020

a set of automatically derived data structure invariants for monitoring purposes. The main challenge of integrity-based detection is the secure storage of the baseline: if the kernel is assumed to be compromised, then no file or memory on the system is adequate for storage. Rootkits in the kernel can modify the return value of system calls necessary for reading files and may compromise the Virtual Memory Management unit of the operating system to access and/or tamper with data stored in memory.

Reliable detection of rootkits requires that the detector runs with higher privileges than the rootkit itself; as a result, detectors are placed in ever lower levels in the devices' architecture or even into a separate hardware. Paladin [5] is an example of the former approach. It defines protected zones for memory and files, and performs integrity checks from the hypervisor. Copilot [25], on the other hand, is an example of the latter: it is a coprocessor-based kernel integrity monitor implemented as a PCI card which connects the monitored system to the remote detector. By contrast, our proposed method does not need additional hardware: we leverage a Trusted Execution Environment to protect the monitoring process from interference. Our detection method also incorporates ideas from cross-view-based, integrity-based and signature-based detection methods. We also use the TEE to safely store baseline values (e.g., file hashes and hash values of memory images).

Chapter 3

Design approach

3.1 High level overview

As mentioned previously, the basic idea of our rootkit detection approach is to leverage the TEE (Trusted Execution Environment) for running functions aiming at detecting integrity violations and inconsistencies in the state of the untrusted system components of the REE (Rich Execution Environment) that may have been caused by a rootkit. The primary targets for checking integrity and consistency are the kernel code and the kernel data structures (in particular, data structures representing processes, as well as data structures holding function pointers) of the untrusted OS, as rootkits typically modify those to achieve their goals. The kernel code and data structures can be accessed from the TEE by reading the memory snapshot of the REE that is left behind when execution is transferred to our rootkit detection function in the TEE. In addition, besides checking code and data structures in the memory, we must be prepared for malware that tries to remove its traces from memory before the invocation of our rootkit detection function. Because it cannot remain in memory, it may try to hide itself in persistent storage in such a way that it can execute again later when the memory has been checked. Hence, our rootkit detection approach also includes checking the persistent storage for signs of malware.

To achieve our goals, we deploy two software components: a Trusted Application (TA) running in the TEE and a client application (CA) running as a user space process on top of the untrusted OS in the REE. Our CA should be started when the system is booted and then it should run continuously. The main role of the CA is to invoke the TA periodically and to pass certain data to the TA collected from the REE (e.g., the list of running processes as seen by the ps program when executed on the untrusted OS). The TA performs rootkit detection by executing different integrity and consistency verification functions as described below. In order to ensure that the TA is indeed invoked periodically, a watchdog timer can be started during the boot process that can only be reset by the TA; therefore, if the TA is not invoked, the timer expires and the device reboots itself. When the TA finishes its execution, control is returned to the CA, which runs concurrently with other applications and services in the REE.

In the sequel, we assume that the OS running in the REE is Linux. Some of our rootkit detection functions described below are specific to Linux, because rootkits often operate at low level in the system architecture and exploit specific features or mechanisms of the OS kernel. Yet, the principles even behind these Linux specific functions are sufficiently general to be applied for other operating systems as well. More over, some of the detection functions we present are agnostic to the OS used.

First some initialization needs to be done: our TA implements a function which can only be called once. This creates reference hashes from the files and directories checked later and discovers the available file systems, so we can detect tampering with these.

When invoked, our rootkit detection TA performs the following verification steps aiming at detecting inconsistencies in the data held by certain kernel data structures or modifications of kernel code:

- **Looking for hooks in the Virtual File System (VFS):** Rootkits often target the so called *operation structures* of the VFS and replace (hook) function pointers there such that file operations are handled by attacker code instead of legitimate kernel code. For instance, the rootkit may hook the `iterate_shared` function in the file operation structure of the `/proc` directory, and ensure that certain process IDs are removed from its output, and hence, become invisible to certain system utilities. Thus, in each operation structure of the VFS, we check if function pointers point inside the address space where the kernel code segment is located. Any function pointer pointing outside that address space is considered to be hooked. More details on detecting hooks of the VFS file operations are provided in Subsection 3.2.2.

- **Detecting hidden tasks:** Another way of hiding certain processes is to manipulate kernel data structures (a.k.a. Direct Kernel Object Manipulation or DKOM for short) representing them. At the kernel level, processes (threads) are represented by tasks, and there are different data structures, such as the task list, the task tree, and so called PID namespaces that hold information about existing tasks. In addition, tasks also appear in queues used by the kernel for scheduling them. Rootkits rarely modify these data structures in a consistent manner. For instance, in order to hide it, a rootkit may remove a task structure from the process list or process tree, while it must be kept in the run queue to be scheduled and have the chance to be executed on the CPU. Therefore, we check all those data structures that hold information about existing tasks and we compare the list of tasks obtained from them to each other and to the process list received from the CA running in the REE. Any inconsistency among these lists is interpreted as an integrity violation of the system. More details on detecting hidden tasks are provided in Subsection 3.2.3.

- **Integrity checks:** Besides manipulating task related kernel data structures, rootkits can also modify other important data structures in the kernel, as well as the code of running processes. For instance, a common rootkit technique, called *system call table hooking*, is to replace function pointers in the system call table such that when certain system calls are invoked, attacker code is executed before control is given to the legitimate function that handles those system calls. Another technique, called *inline hooking*, has similar effects, but in this case, the system call handling functions themselves are modified by inserting a jump instruction at the beginning of the function that points to some attacker code. Similarly, the code of any processes in the memory may be modified by the rootkit including the kernel code segment, system programs, and user space applications. For this reason, we perform integrity verification of the system call table, the kernel code segment (which includes the functions that handle system calls), system programs currently executing, and the code segment of our CA running in the untrusted execution environment. This integrity verification is based on accessing these pieces of data structures and code in the REE memory from our TA, computing the hash values of their memory image, and comparing the computed values to known reference values stored securely within the TEE. These reference values are computed and saved in secure storage provided by the TEE after system installation when the system runs for the first time. More details on the integrity checks we perform are provided in Subsection 3.2.4.

- **Looking for persistent rootkit components:** As rootkits may remove their components from memory before our TA is invoked and our consistency and integrity verification is executed, and hide themselves on persistent storage, we must also look for these persistent components. For this, our TA accesses the persistent storage of the device, recursively hashes all files in a pre-selected set of folders, and then compares the computed hash values to known reference values stored securely in the TEE. These reference values are computed and saved in secure storage provided by the TEE after system installation when the system runs for the first time. The folders are pre-selected in such a way that they contain all the binaries and scripts that could legitimately execute on the device. This requires a certain organization of the files in the file system, notably to separate files (including programs) that should not change from those that has variable content (e.g., files used mainly for data storage). However, this is not a serious limitation of our approach, because this kind of separation is also useful for many other reasons related to the maintenance and troubleshooting of the device.

  An issue that we have to consider is that while our TA is waiting for I/O operations (i.e., reading file contents) to complete, control is given back to the REE. When this happens, pre-scheduled jobs may be executed by the job scheduler (e.g., `cron`). Hence, in theory, a rootkit can hide its persistent component in a program file A and schedule the execution of A before removing itself from memory. Then, program A (and hence the rootkit) could be executed by the job scheduler during the file hashing operation performed by our TA, and when executing, the rootkit can move itself from program file A into program file B. If this move operation happens after file B has been hashed already and before file A being hashed, then the computed hash values would be good, and we would not detect the rootkit, which can then be re-installed when file B is executed. As I/O operations are usually slow, our TA is mostly waiting during the file hashing, which means that control is mainly at the untrusted execution environment, and hence, chances of the above described scenario happening are not negligible. To cope with this issue, our CA suspends file access operations for execution before invoking our TA and re-enables them only after the TA completes its job.

  The component responsible for file checks is an integral part of pur solution, however it won't be described in details, like other checks. The TA code responsible for creating and verifying file signature hashes was the work of Krisztián Németh, for more details, see his thesis[24].

- **Network checks:** Finally, we perform some checks regarding the network stack; two common functionality implemented by rootkits in this component of the kernel are hiding open ports and implement "magic packets". This means executing certain payloads triggered by specially crafted network packets. For example, the rootkit opens a shell when the victim machine receives a TCP packet, on a specific port, from a specific port, with a specific payload. The network stack uses several function pointers, so this functionality can be implemented many ways. We do not attempt to check the integrity of all of these pointers, we only check the Netfilter subsystem, the most common target of rootkits, and some other structures containing function pointers, what we were able to hook from our test rootkits. Port information is provided to the userspace via the `/proc` file system, where many files are using so called `seq_ops` structures. These encapsulate function pointers, what are used to generate the content of the file. We check the function pointers the same way as we did in the VFS layer.

Figure 3.1 gives a high level overview of our rootkit detection components (i.e., the CA and the TA), their interaction, and the operations they perform. As it can be seen, the CA is started at boot time, it continuously runs, and it invokes the TA at random time intervals. Before invoking the TA, the CA disables execution type access to files, such that new programs cannot be started during the checks of the TA. Then the TA performs the above described consistency and integrity checks on the kernel data structures and the code segments of the kernel, running system programs, and our CA. If integrity violations or inconsistencies are found, then they are reported to the operator of the device via some remote attestation protocol, but this is out of the scope of this paper. If the verification of the memory is successful, then the TA proceeds with hashing the files in the pre-selected folders in the persistent storage, and comparing the computed hash values to the stored reference values. Again, if an integrity violation is found, then it is reported. Otherwise,

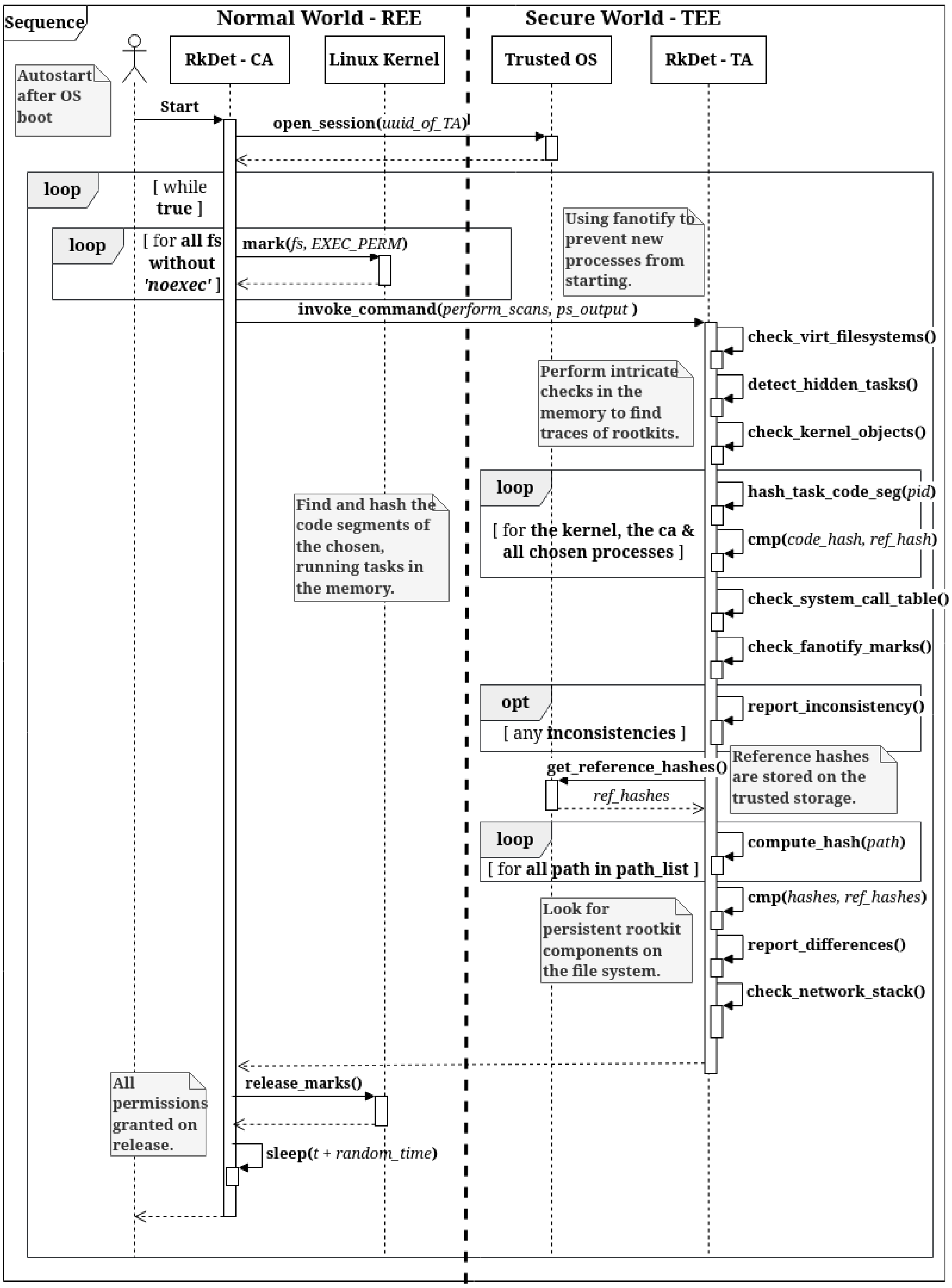

Figure 3.1: High level overview of our rootkit detection components, their interactions, and the operations they perform.

control is returned to the CA, which re-enables file access, and sleeps until the next round of all these operations.

3.2 Design details

In this section, we discuss the checks we implemented in order to detect the traces of rootkit infections in the Linux kernel memory. These checks focus on the integrity of and consistency between kernel objects and target common techniques applied by rootkits. In the following subsections, we briefly introduce certain kernel objects, describe attacks against them, and present our methods to determine whether the system is infected by rootkits. In Subsection 3.2.2, we present the Linux kernel's Virtual File System (VFS) and our technique to detect rootkit attacks against its structures. In Subsection 3.2.3, we discuss direct kernel object manipulation (DKOM) in details and our techniques to counter this attack. In Subsection 3.2.4, we describe our proposed checks to examine the integrity of the analyzed system, and finally in Subsection 3.2.5, we describe certain components of the network stack, attacks against them and our methods to detect tampering in this subsystem.

3.2.1 Initialization

Before we can execute our checks, we need to initialize our solution. This routine can only be invoked once, and it's called right after the setup of the Normal World components of the TEE. Here we compute and store reference hashes of the file system, used by the persistence checks, and explore the file system objects in the kernel. We collect all the partitions the device has, so we can detect later if new ones are created, and we also collect the ones, where users can execute binaries from, so we can ensure that no new process is created while we perform our checks.

3.2.2 Detecting hooks in the Virtual File System

The Virtual File System (VFS) [7] is an API in the Linux kernel which hides the differences of the various file system drivers. It uses 4 main data structures to abstract away the details of different file system implementations, shown in Figure 3.2. The *superblock* structure represents a mounted partition and stores metadata about the partition itself, which is usually present in the first block of the underlying physical device. Superblocks are chained together into a doubly linked circular list, which is accessible from the data segment of the kernel binary. Each superblock maintains a circular, doubly linked list of the *inodes* stored on the disk. An inode is the physical representation of a file or a directory stored on the device. An inode can be used by one or more directory entries, or *dentries* for short. For example, if we create a new file and a hard link pointing to it, then we will have one inode and two dentires referencing the same inode. Open files are represented by so-called *file* structures in the context of a process and can be accessed via file descriptors.

At code level, VFS uses so called *operation structures* which contain function pointers. Each function pointer implements a functionality used by the layers above VFS (e.g. `lookup` to retrieve the contained dentries in the inode of a directory or `get_inode_usage`

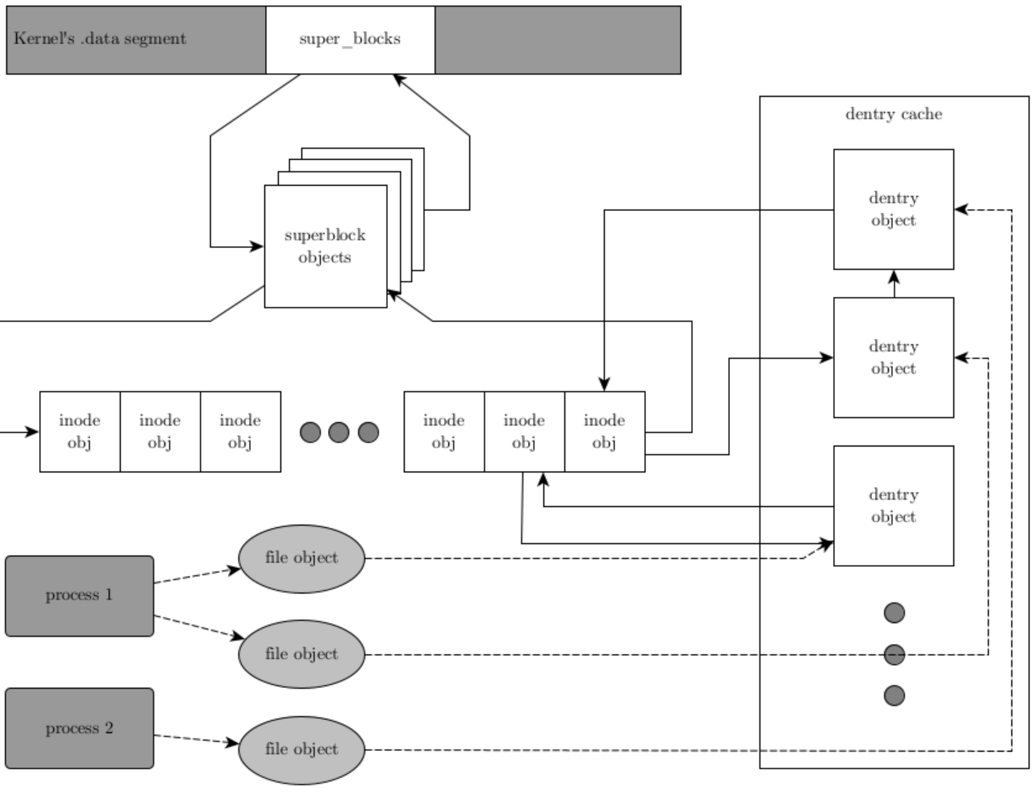

Figure 3.2: Relationship of VFS objects. White objects are checked, gray ones are ignored by our checks.

to find out how many inodes are used on a partition), where the implementation is provided by the underlying file system drivers. These operation structures appear in all 4 of the previously mentioned structures.

Rootkits often target the operation structures: by hooking certain function pointers, they can alter the results returned by these functions. For example, an attacker can create a Linux kernel module, find the `file_operations` of the `/proc` directory, and hook its `iterate_shared` function, which implements the functionality of `readdir`. With a properly designed replacement, attackers can hide their presence by excluding certain process IDs whenever process information is retrieved from `/proc`.

We perform integrity checks on 9 different types of operation structures. They cover all the operation structures used by superblocks, inodes, dentries and files. For superblocks, we analyze `super_operations`, which implement general file system operations (e.g., allocating inodes); `dquot_operations`, which handle quota objects on disk; `quotactl_ops`, which manage quotas on the file system; `export_operations`, which are operations for the NFS daemon to communicate with file systems and the default `dentry_opartions`, which is assigned to dentries of the superblock. For inodes, we analyze `inode_operations` and `file_operations`. The former provides operations to manage the inode, including `rename`, `unlink`, etc. The latter one contains the file operations assigned to a file structure when a process opens a dentry pointing to this inode. This structure contains function pointers like `read`, `write`, `flush` and many more. For dentries, we check `dentry_operations`, which hold directory entry related operations. For example, `delete` removes the dentry, but the inode remains intact, just in case other dentries use the inode as well. We also check several file operation structures from the `.rodata` section of the kernel by their addresses.

For each operation structure, we examine the function pointers, and consider one to be hooked if it points outside of the address space where the code segment of the kernel is located. We perform the check recursively, i.e., we inspect every inode of a superblock and every dentry of an inode. Unfortunately, inspecting all superblocks is time consuming, therefore, we only focus on `/proc` and the root file system as hooking their inode operations is a common technique to hide rootkit files and processes.

We also check if no new superblocks are added to the list: this is a valid restriction for IoT devices, the appearance of new file systems is considered to be tampering by an attacker.

3.2.3 Detecting hidden tasks

Processes and threads are represented by tasks in the Linux kernel. A task is approximately equivalent to a thread: single-threaded processes consist of a single task, while multi-threaded ones are made up of several tasks sharing the same address space. Each task is represented with the so-called `task_struct` structure. In Linux 5.1, there are three data structures which contain all of the existing tasks.

Task list: All task objects are linked together into a doubly linked circular list. In earlier versions of the kernel, this list was used to populate the `/proc` file system.

Task tree: Tasks are also organized into a tree via the relation of creation. When a task creates other tasks, they become its children and they refer to their creator as their parent. The root of this tree is the so called `init_task`, the kernel task which starts the init process (the first process in the user space).

Pid namespace, IDR and the `struct pid`: Pid stands for process identifier and IDR is the rewritten version of the old ID allocation API. Linux provides pid namespaces as an isolation feature. By default, there is only one such namespace, the initial pid namespace. Each namespace maintains a radix tree[1], containing pointers to pid structures[2] (`struct pid *`). Pid structures contain lists of pointers for the tasks with an equal ID, thread group ID, process group leader or session ID. The Pid namespace is responsible for keeping track of taken pids and for fast access to tasks via their pids[3]. In recent kernel versions, this mechanism populates the `/proc` directory.

[1]`https://lwn.net/Articles/175432/`
[2]`https://lwn.net/Articles/195627/`
[3]`https://lore.kernel.org/patchwork/patch/834401/`

Direct kernel object manipulation (DKOM) is a rootkit technique which modifies the data structures traversed by the kernel when asked to retrieve information about resources. For example, rootkits can use DKOM to hide a process by removing it from the data structure queried for process information. Many rootkits target the task list to hide a backdoor on older systems; the same can be achieved on modern Linux by removing a pid structure from the IDR of the initial pid namespace. DKOM is rarely applied thoroughly, rootkits usually remove processes only from the necessary data structures, thus leaving the kernel memory in an inconsistent state.

For example, the deadlands rootkit[4]: it is capable of removing a process from the list of tasks and a hash table (used by older kernel versions for fast access of tasks by their pids), but the hidden task can still be found in the task tree.

Another example is how the Reptile rootkit[5] hides the kernel module implementing its functionality by removing the module from the list used by the kernel to populate `/proc/modules`. However, it is still visible under `/sys/module`.

Our analysis attempts to detect these inconsistencies. We assume that a trivial goal of an attacker is to hide a process from the output of the `ps` utility. First, in the REE, we execute `ps` and pass its output to our TA running in the TEE in order to compare it to pid lists we can extract from the REE kernel memory. The pid of `ps` itself must be excluded from this list because it is not running when we perform our check. Our TA, after checking the VFS layer, iterates through the task list, saves the pid of every task, and then looks for pids present in the list, but missing from the output of `ps`. If no hidden process found, the TA performs a depth-first search on the task tree and compares the pid list created this way with the sample from `ps`. If no inconsistency is found, the TA performs the same check for the IDR. If it does not find any hidden processes, it tries to determine if there is any process present in any of the mentioned data structures, but missing from any of the others. This is done by taking the union of the three lists and comparing each list to the union.

The most important feature of the kernel is the capability to schedule processes. A task is considered to be *runnable*, if it is not waiting for anything (usually I/O) and is not stopped. Runnable tasks ready to be executed are collected in separate data structures; these are called run queues. Run queues are per-CPU data structures, i.e., each CPU has its own run queue, and every run queue wraps sub-runqueues implementing the data structures used by the scheduling algorithms. These scheduling policies are implemented via a structure called `sched_class`, which works similar to the operation structures at VFS: it contains multiple function pointers, and each scheduling policy must implement these. Linux 5.1 supports the following three schedulers:

Completely Fair Scheduler (CFS)[1] This is the default scheduler. It stores a tree of scheduling entities (task sets), each maintaining a red-black tree of tasks. CFS

[4]https://github.com/majdi/deadlands, Last visited: 18.09.2020
[5]https://github.com/f0rb1dd3n/Reptile, Last visited: 18.09.2020

orders tasks by their *virtual run time*, meaning how long each should run, and always picks the one associated with the smallest time.

Real-Time Scheduler[3] It stores 140 lists (one for each priority level), and a bitmap to speed up lookup of non-empty lists. The next task is searched by testing the bits of the bitmap, and on the first set bit, its position is used as an index to get a list. The first task in the list is chosen.

Deadline Scheduler[2] On multiprocessor systems, the scheduler maintains two red-black trees of tasks. In the first one, the tasks are ordered by their deadline, and the leftmost element of the tree is picked. The other one contains tasks what can be pushed to other cores. In case of one single core, only the first tree is present.

We assume that removing a task from a run queue would make it unschedulable permanently, which is why we include run queues in our consistency check. Our TA collects the pids of all of the tasks found in the data structures of the schedulers and then compares them to the union of pids created earlier. If it finds a task in a run queue which is not present in the union, then the kernel is considered to be compromised.

3.2.4 Integrity checks

So far, we focused on revealing hidden processes by verifying the integrity of VFS components and by performing consistency checks on task-related data structures. In this subsection, we leave the concept of hidden processes and instead target common rootkit techniques; some of the checks presented here ensure the integrity of the system itself, while others defend our solution from a possible attacker.

First, we must ensure that our CA can be trusted. To this end, the CA is compiled as a static executable (i.e., its binary contains the code for all the used library functions as well). This allows us to protect it against LD_PRELOAD hooks, a common rootkit technique applied in the user space. The CA passes its own pid as a parameter to our TA, such that the TA can look for the corresponding task in the kernel memory. Each task has a pointer to a memory map structure, which stores information about the task's memory mappings. From this structure, we can determine the start and the end of the task's code segment, and we can use it to translate the virtual addresses of the process to physical addresses. Via physical addresses, we can access the contents of the memory pages storing the code of the task, and we can compute its hash to check whether the code of the CA has changed. The address translation depends on the system's configuration; in our case, the kernel uses 4 Kb pages and 4 (ARM64[6]) or 2 (ARM32[7]) layers of translation tables.

Next, we check the system call table. System calls are the interface between user space and kernel space. Whenever a user space process needs to perform an operation that is the

[6]`https://www.kernel.org/doc/html/latest/arm64/memory.html`, Last visited: 12.04.2021
[7]`https://people.kernel.org/linusw/arm32-page-tables`, Last visited: 12.04.2021

kernel's responsibility, it invokes the appropriate system call. The system call table is an array of function pointers indexed by the system call number. On the ARM and ARM64 architecture, the system call table resides in the `.rodata` section of the kernel binary, which is marked read-only at boot time. If an attacker can remove the write protection, it is possible to overwrite pointers in the array and alter the kernel's control flow to execute a different implementation of the system call. This is the most popular target of kernel space rootkits. [9]

We were able to remove the write protection for both architectures. On ARM, we could use the `set_kernel_text_rw`[8] function to disable write protection, and we could re-enable it with `set_kernel_text_ro`[9]. Despite their names, these functions cover the write protection of the `.rodata` section as well. On ARM64 we used the `update_mapping_prot` function[10], borrowed from the boot sequence. We changed the last parameter from `PAGE_KERNEL_RO` to `PAGE_KERNEL` to disable write protection. Re-enabling the write protection can be done with the same function.

If kernel space randomization is disabled, we are able to retrieve the location of the system call table after the kernel is compiled. With this information and the number of entries in the table, we can easily determine the memory area to check. We do this by creating a hash and comparing it against the one computed on the intact system call table.

Another common technique is *inline hooking*. In this case, the attacker chooses a function to hook and replaces its first few bytes with an unconditional jump instruction and a pointer to the implementation he wishes to execute instead of the original one. The previously mentioned write protection is applied to the kernel's text segment as well, but similarly to the system call table, it can be removed by calling `set_kernel_text_rw` or `update_mapping_prot` on the `text` segment[11]. We detect inline hooking, by hashing the kernel's entire text segment and comparing the hash value to a reference value. The location and size of the kernel's text segment is determined after the kernel is compiled. Note that this solution does not support self modifying kernel code and kernel address space layout randomization. The very same approach is used to verify the integrity of the text segment of the system processes and our CA running in the REE.

The Linux kernel provides APIs for monitoring file system operations (e.g., fanotify and inotify, which both use the same underlying kernel mechanism, fsnotify). We use fanotify in our CA to ensure that no executable is started while our TA performs our checks. We do this by placing marks at every mount point to make sure the kernel does not allow execution until the CA approves it. While our checks are performed, the CA suspends

[8]`https://elixir.bootlin.com/linux/v5.1/source/arch/arm/mm/init.c#L666`, Last visited: 22.04.2021

[9]`https://elixir.bootlin.com/linux/v5.1/source/arch/arm/mm/init.c#L675`, Last visited: 22.04.2021

[10]`https://elixir.bootlin.com/linux/v5.1/source/arch/arm64/mm/mmu.c#L525`, Last visited: 18.09.2020

[11]`https://elixir.bootlin.com/linux/v5.1/source/arch/arm64/mm/mmu.c#L445`, Last visited: 12.04.2021

all execution requests. When the checks are completed, the marks are removed and every execution request is approved.

Since we use another kernel functionality and we assume that the kernel is compromised, our TA needs to check if the fanotify marks placed by our CA are intact. Therefore, the CA passes the file descriptor returned by fanotify and the pid of the CA. The TA then locates the fsnotify group among the open files using the received pid and the file descriptor. This group stores a list to the marks placed. The TA checks if all the necessary superblocks are marked.

3.2.5 Detecting hooks in the network stack

The checks we present in this subsection focus on the integrity of the subsystem of the kernel responsible for networking. This component is called the network stack and rootkits often target it. The two features rootkits implement here are hiding open network connections and a so-called "magic packet" functionality. "Magic packet" is when an attacker implements some mechanism which can be triggered by a specially crafted network packet. The payload can be arbitrary, but usually a shell is opened when the infected system receives the packet. The common way to implement this functionality is to use the Netfilter subsystem, the backend of Linux firewall solutions, but since the network stack uses many function pointers, other solutions are possible as well. We present 3 different checks to cover magic packets; this cannot be considered a full solution, probably there are other pointers what can be hooked by attackers. However, our solution covers all common methods, and some uncommon ones as well. We also present a check to discover port hiding.

First we check the data structures of Netfilter. Netfilter uses chains to store firewall rules. Every supported protocol (like IPv4, IPv6 or ARP) has 5 chains for different sections of packet processing. These chains are called PRE_ROUTING, INPUT, FORWARD, OUTPUT and POST_ROUTING, and their relationship is presented at figure 3.3.

Each of these chains act like an arraylist for so-called Netfilter hooks. These hooks have a function pointer, and when a packet is processed by a chain, every entry's function is invoked on the packet. If all of them return with `NF_ACCEPT`, the packet is accepted and it's forwarded to the next entity.

The trick rootkit authors use here is that these hook functions can have side-effects. Netfilter is a common choice, since it's relatively easy to use, only a filter function needs to be implemented, which executes the payload if all conditions are met.

We traverse all chains of all supported protocols and a hook in any of the chains is considered to be malicious, if the function pointer points outside of the kernels text segment.

The next structure we check is called `icmp_control`[12]. The kernel has an array of these structures (called `icmp_pointers`), and when it receives ICMP messages, it uses the type

[12]https://elixir.bootlin.com/linux/v5.1/source/net/ipv4/icmp.c#L193, Last visited: 12.04.2021

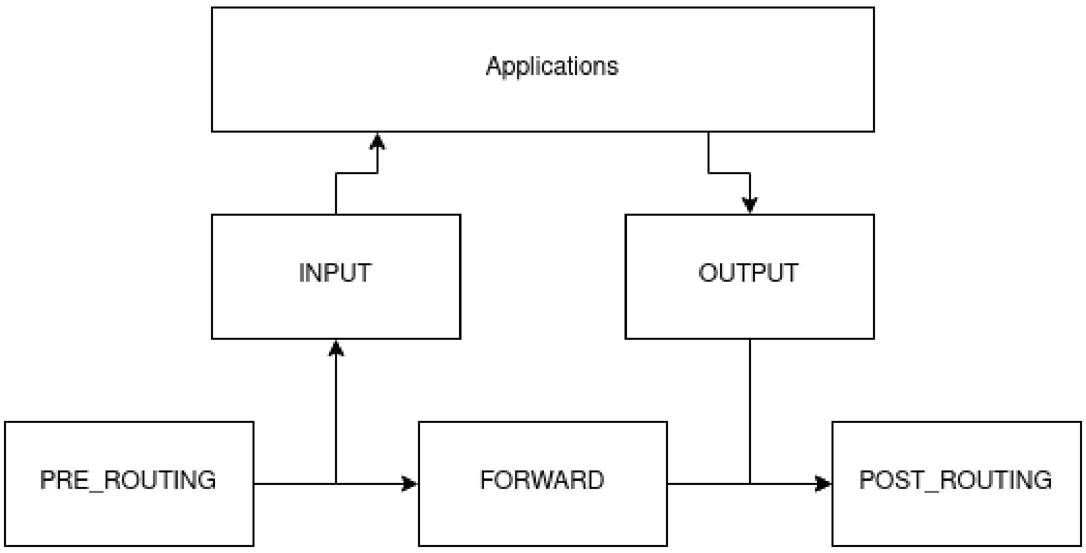

Figure 3.3: Relationship of the 5 Netfilter chains. Arrows show the directions of packet processing.

field of the message header as an index to determine which handler function must be executed. We check the integrity of these structures the same way as we did previously: if the function pointer points outside of the kernel's text segment, it's considered to be hooked. We didn't find any rootkit implementing magic packet functionality this way, but we were able to create a proof of concept rootkit which hooks one of these pointers.

`net_protocol`[13] structures can also be hooked. These are used by the kernel when the network stack is initialized to register handler functions for different protocols, like TCP, UDP or IGMP. These structures contain handler, error handler and demultiplexer function pointers. Their integrity check is identical to the previous function pointer checks. Again, we didn't find any application of this technique in the wild, but we managed to create a proof of concept rootkit.

Finally, we check for hidden sockets. Many files in the `/proc` directory use `seq_ops` structures, used to generate the content of the file. These structures contain function pointers called `start`, `stop`, `next` and `show`. The first two are called at the beginning and end respectively, while `next` returns an entity which will be displayed by `show`. This is how the Linux kernel supplies information about open ports via files like `/proc/net/tcp` or `/proc/net/udp`. We collected as many `seq_ops` structures as we could and perform an integrity check on all of them by verifying their functions pointers. Rootkits usually hook `show` because it's easier to implement a replacement for it, then implementing one for `next`.

[13]`https://elixir.bootlin.com/linux/v5.1/source/include/net/protocol.h#L41`, Last visited: 12.04.2021

Chapter 4

Implementation

We use the Open Portable Trusted Execution Environment[1] (OP-TEE) as the trusted execution environment. It was initially developed by ST-Ericsson and currently owned and maintained by Linaro. Our implementation uses version 3.6.

Besides the TEE itself, which is essentially a minimal OS running in the Secure World, OP-TEE consists of Normal World components as well: Linaro has its own fork of the Linux kernel, which includes an OP-TEE driver. This driver is responsible for shared memory allocation between the two worlds and provides the RPC functionality through which the CA and TA can communicate. The driver exposes its functionality to the user space via a block device. However, in order to increase usability, the driver also has a counterpart in the user space, a daemon called `tee-supplicant`.

This daemon is capable of performing REE user-space actions and it communicates with the driver using ioctl. This is a bidirectional channel, meaning that the driver is also capable of requesting operations from the daemon.

There is also a method for extending the functionality of the OP-TEE kernel: pseudo-trusted applications (PTAs). These applications must be compiled into the OP-TEE OS and they are capable of exposing core functionality to TAs or CAs. We used this feature to implement functionality necessary for accessing non-secure memory, which is otherwise forbidden for TAs. We discuss how we read REE memory and files via PTAs from the TEE in Sections 4.1 and 4.2, respectively, and describe our watchdog driver in Section 4.3. We also implemented checks in order to protect the REE components of OP-TEE, these are discussed in Subsection 4.4. Finally we describe how our solution is built and what environments we used for development and testing. These are covered by Sections 4.5 and 4.6, respectively.

[1]`https://www.op-tee.org`, Last visited: 20.09.2020

4.1 Normal World memory API

To be able to read the memory of the REE kernel, we had to instruct the memory management unit of the OP-TEE OS to map the physical memory range where the Linux kernel resides. We determined this range from the boot log and modified `core_mmu.c` to register it as non-secure RAM. This makes it accessible in the context of the TEE, but TAs cannot read it due to address translation issues. The Normal and Secure Worlds are using two distinct address spaces, which means that if we take the address of a Linux kernel object, OP-TEE will not be able to process it. We solved this issue by translating Normal World addresses to physical address, and back to Secure World virtual addresses. Fortunately, OP-TEE provides us with tools to accomplish the latter task, so we only needed to implement the former one. The Linux kernel uses a macro called `__virt_to_phys_nodebug` for kernel addresses, and so-called page global directories for user-space address translation. Since the implementation of address translation is highly platform dependent, we discuss it later in Section 4.6.

In order to access the physical memory in which the Linux kernel resides, we had to use a PTA. The PTA implements the following interface:

read_mem This function expects two parameters, a memory region to copy data into and a physical address to copy data from. It translates the address to a Secure World virtual address, and after performing the necessary checks, it populates the buffer with the requested amount of data.

hash_mem This function is used to create a hash from non-contiguous REE memory ranges. It expects an array of `phys_mem_range` structures (containing a pointer to a memory region and a size), and a buffer where to store the created hash. It initializes a hash context, iterates through the array of physical memory ranges, translates the addresses back to virtual addresses and feeds the specified regions to the hash function. When it is done, it copies the produced hash into the output buffer. We chose the SHA-256 hash function and OP-TEE OS is configured to use the libtomcrypt library[2] by default.

This API was originally written by Szilárd Dömötör and István Telek[13, 14], we only had to apply minor modifications to make it work in our environment.

4.2 Normal World file API

OP-TEE does not provide access to the REE file system by default. However, we noticed that OP-TEE's trusted storage stores encrypted data on that file system, hence we suspected that there must be a way to access the REE file system from OP-TEE. After digging

[2]`https://optee.readthedocs.io/en/latest/architecture/crypto.html`, Last visited: 18.09.2020

the source code, we discovered that the `tee-supplicant` daemon can be instructed via RPC calls to perform REE file operations. However, the set of RPC functions available to perform file operations were not written for general purposes, but they were designed specifically for the trusted storage. As a result, to be able to pass arbitrary filenames as parameters, we had to modify the `open` and `opendir` functions. Function `readdir` also had to be modified due to a bug we discovered. Our pull request with the fix is merged already[3], but it is only included in the 3.10 release. We modified the previously mentioned functions by making copies of them in our PTA and applying the necessary changes to the copies. In addition, as the root of the trusted storage is `/data/tee/` and filenames are prefixed with this string in `tee-supplicant`, the PTA expects absolute filenames and prefixes them with `../..` before passing them to `open` and `opendir`. The PTA implements the following interface:

hash_file This function expects a filename as a string and a buffer to store the computed hash in. It opens the requested file (if exists), reads its content by 4096 bytes and passes these blocks to a hash function. When the end of the file is reached, it finalizes the hash and copies it into the output buffer.

hash_dir This function takes four parameters: a directory name, an output buffer, an integer to indicate if we want to hash recursively (0 for false, 1 for true) and a pointer to a blacklist, a null-terminated array of strings. It creates a hash context, opens the specified directory (if exists) and reads its content. For every entry, we check the blacklist first (to avoid hashing files with changing content, e.g. `/etc/random-seed`). If the entry is not on the blacklist, we try to determine if it is a regular file or a directory. Since we have no `stat`-like primitive, we do this by invoking `opendir`. If `opendir` fails, we have a file, otherwise, a directory. For files, we do the same as above: read the file by blocks and feed every block to the hash function. If the entry is a directory and we hash recursively, then the function calls itself recursively on the entry. Otherwise, the entry is skipped. Finally, the hash is copied to the output buffer. We use the SHA-256 hash function from the libtomcrypt library. This function supports two ways of recursive hashing: it is capable of feeding all file contents into one hash, or create one hash per file/directory, and the parent's hash is computed from the hashes of its children. This method was implemented to comply with later remote attestation solutions, where reference hashes might be computed on other devices, by inspecting firmware images.

In order to ensure the proper functioning of the above described PTA, we had to apply two patches to the `tee-supplicant` daemon. First, we had to give it root privileges, otherwise it cannot read certain files. Second, `tee-supplicant`'s `readdir` handles certain directories improperly: if a directory only stores hidden files, it is considered to be empty. From the aspect of our persistence checks, this behavior can be fatal, so we had to patch

[3]`https://github.com/OP-TEE/optee_os/pull/3962`, Last visited: 18.09.2020

the appropriate function to only skip `.`, `..` and `.nfs*` (these files are created by an NFS server, when an open file is deleted).

4.3 Secure World watchdog API

To properly handle discovered rootkit infections, our solution needs to be able to utilize the functionality of a watchdog timer. Unfortunately, the watchdog driver implementation in OP-TEE is rather incomplete, so we had to provide our own implementation in the form of another PTA. This application is used only when we test our solution on an i.MX board, which will be described in detail later in this chapter.

The i.MX 6 SoC (System-on-Chip) we used contains two watchdog timers, namely `WDOG1` and `WDOG2`. The former one can be used both from the Normal and Secure World, while the later one can only be used by software running in the TrustZone. When these timers are set and enabled, they start to count down from the supplied timeout value, and if zero is reached, it emits a signal which will reset the entire system. Resets can be avoided, if predefined values are written into the appropriate register of the watchdog before zero is reached. In this case, the countdown starts again. This mechanism is typically used to detect hardware failures that might be solved by the reset.

NXP (manufacturer of the i.MX SoCs) use the same watchdog hardware in every SoC of the i.MX series, named `imx2-wdt`. It has 4 registers, each 16 bits wide:

Watchdog Control Register (WCR) : to configure properties and timeout of the watchdog. The lower 8 bits are used to enable/disable certain features of the timer, while the top 8 bits hold the timeout value (`0x0 = 0.5 sec, 0x1 = 1 sec, ..., 0xff = 128 sec`).

Watchdog Service Register (WSR) : this is used to "service" the timer, so it will restart the countdown instead of resetting the board. First the value of `0x5555` must be written into this register, and then it must be overwritten with `0xAAAA` to complete the service routine.

Watchdog Reset Status Register (WRSR) : stores the source of the last reset for debugging purpose (not used by our implementation).

Watchdog Miscellaneous Control Register (WMCR) : for additional configurations (not used by our implementation).

These registers must be mapped into the virtual address space; in our case, this is performed by OP-TEE OS. We can get the virtual base address by translating the timer's physical address to virtual. From the base address of the timer, we can access the registers simply by adding their offset to the base address.

The watchdogs are connected to other hardware components, namely the System Reset Controller (`SRC`) and the Secure Non-Volatile Storage (`SNVS`). The `SRC` performs the reset operations, while the `SNVS` can wipe sensitive information from its registers before invoking the `SRC`. This is done only if `WDOG2` reaches zero. They are both connected to the Generic Interrupt Controller (`GIC`), since watchdogs can generate interrupts before the timeout.

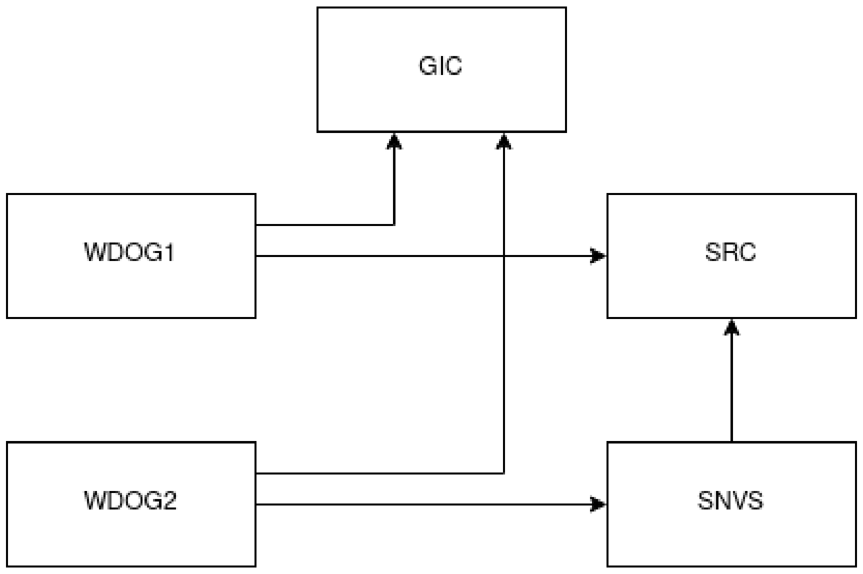

Figure 4.1: System-level connectivity of watchdogs.

It's important to note that our PTA is not a full-featured driver, it only implements the functionality absolutely necessary for our solution to work. This can be accessed via the following interface:

setup this function performs the initialization of the watchdog. It sets the appropriate configuration, the timeout received as parameter and starts the countdown. Its only parameter is an 8 bit long unsigned integer.

set_timeout this function can be used to set a new timeout value. When a new value is set, the timer won't start to use it immediately, it will only be loaded when the watchdog is "serviced". Setup uses this function to set new timeout. Its only parameter is an 8 bit long unsigned integer.

keep_alive writes the appropriate values into the `WSR`, so the timer won't reset the board. Setup uses this to start the timer. This function does not have any parameters.

reset_now this function is used to reset the board immediately. To do this, it sets the timeout to zero (0.5 sec) and loads this new value by invoking `keep_alive`. This function does not have any parameters either.

Currently our implementation is able to use `WDOG1`, but not `WDOG2`. We suspect, this might be because `SNVS` is not configured to react to the security violation signals emitted by the TZ watchdog.

4.4 Verification of OP-TEE specific components

As we do not trust software components in the REE, but our implementation relies on using OP-TEE's Normal World components, we needed to ensure the integrity of them. For OP-TEE's Linux driver, we did not need to implement any additional checks, because it is compiled as a part of the kernel, so its integrity is verified when our TA checks the integrity of the kernel code. The daemon `tee-supplicant`, however, had to be slightly modified before we applied the same technique as we used for checking the integrity of our CA. First, we built it as a static binary. Next, we had to ensure that the pages containing the code of the daemon are present in the memory, and not swapped out. We implemented a Linux driver to create an entry under `/proc`. If pids are written to it, it looks for the corresponding task, and if it is a thread of `tee-supplicant`, it generates a page fault for every page of its code segment. With these preparations, we can apply the same check on `tee-supplicant` as we did on the CA, except that we compute the hash of every task, if it has the proper name (`tee-supplicant`). This check is executed right after we check the code of our CA. We could extend this kind of integrity check to other tasks as well, as described in Subsection 3.2.4, however, our implementation currently verifies only the `tee-supplicant` daemon and the CA.

4.5 Building

OP-TEE provides makefiles for the example applications which we can reuse for building our solution and a framework for building TAs. Our solution however requires some extra steps to be built successfully, so we summarize these steps in this section.

At this point, we must note that we actually compile 2 CAs and 1 TA. The two client apps are responsible for triggering the initialization routine of the TA and invoking the checks repeatedly. These could have been implemented in one binary as well, but we found this approach more comfortable.

Building a CA is simple: we use an example makefile to cross-compile our application. All `.c` files are compiled by `gcc` to object files and the linker creates an elf file from these. The only customization we implemented is to compile statically linked executables, meaning that all necessary library functions are part of the executable and not linked to it in runtime. We do this to be able to verify the integrity of the code section. This way we only need to implement one hash check, we don't need to look up all the used libraries in the address space of the running application. To achieve this, we added `-static` to the linker flags.

TA building is more complicated, many variants can be created. We use a legacy TA stored on the REE filesystem. These are digitally signed but not encrypted. OP-TEE provides us build scripts to compile such an application. The TA file is formatted as:

```
hash = H(<struct shdr> || <stripped ELF>)
```

```
signature = RSA-Sign(hash)
legacy_binary = <struct shdr> || <hash> || <signature> || <stripped ELF>
```

shdr is a structure containing metadata like the used cryptographic primitives, size of the hash and signature and others.

The build scripts provided by OP-TEE produce this file, we only need to enumerate the source files we want to be compiled and optionally locations for header files, compiler and linker flags.

To be able to interact with the memory image of the kernel, we had to implement some preparation steps. First, for interpreting the data found in the memory image, we needed to know the layout of several data structures used by the kernel, like tasks or inodes. A possible solution would have been compiling these manually, but the structures we use usually depend on many more structures, what also have dependencies, so we abandoned this approach. Instead, we use a script called `dwarfparse`[4], which is capable for generating header files based on the debug section of an executable. In our case, this executable was a dummy kernel module which uses the data structures we need in our application. These headers were generated and manually fine-tuned for each platform we support.

Another issue is the location of data structures in the kernel memory. Many are dynamically allocated, so we cannot determine their base address and access them directly, however they might be accessible from some global variable, like tasks through the task list from the initial task structure, or the list of superblocks, accessible from a list defined in the data section of the kernel. To obtain addresses of these data structures, we use the `System.map` file of the kernel. This is essentially a symbol table used by the kernel. Using a simple script, we can generate a header file defining addresses. The input of the script is the symbol table we use to look up addresses from a list of pre-defined symbols.

Many of our checks has to be performed only if the certain functionality is compiled into the kernel. This can be achieved by writing configuration manually for each kernel, but if the kernel is reconfigured, our config needs to be changed as well. We solved this issue by using another file generated by the kernel, called `autoconf.h`. This header is generated from the kernels configuration and contains macros the developers can use to check if certain configuration options are enabled or not. Before compiling our application, we copy this file to our include directory. It is also used to check if certain features we rely on are enabled, and if not supported ones are disabled.

Besides using `autoconf.h`, other aspects of our application can be configured as well. For example, we were not able to clearly determine all parameters of userspace address translation from `autoconf.h`, and directories needed to be checked by the persistency component can also vary between platforms. We solved this issue the same way as Linux developers did: we created a `Kconfig` file to describe the possible configurations of our application and borrowed a minimal set of python script from the Kconfiglib[5] project.

[4]`https://github.com/realmoriss/dwarfparse`, Last visited: 08.04.2021
[5]`https://github.com/ulfalizer/Kconfiglib`, Last visited: 08.04.2021

We created default configurations for all of the supported platforms and our build system uses these and the scripts to generate a header file, containing macros for checking the configuration options.

4.6 Development environments

4.6.1 Qemu

As one of our development environments, we used OP-TEE's official Qemu-based distribution. OP-TEE has a github repository[6] containing manifest files. By using these, users can clone several repositories and build the whole solution with a few commands only. One of these manifest files describe an environment based on Qemu for Aarch64. It includes the OS, client, test, benchmark and example repositories of OP-TEE, the ARM Trusted Firmware, a bootloader, Buildroot and a custom Linux kernel with the OP-TEE driver included. Also there is a build package, which can be used to compile all the mentioned components and boot a virtual machine suitable for development and experimentation with OP-TEE. The process of setting up an environment like this is briefly described in the documentation of OP-TEE[7]

This was an ideal and comfortable development environment, but as we proceeded with the development, we discovered special needs of our rootkit detection solution, thus we had to customize certain components.

First, to fit our needs better, we had to modify the configuration of busybox. Busybox is software suit aiming to be a minimal replacement of classic Unix utilities. It implements the functionality of more than 300 commands in one single executable, making it an excellent choice for embedded systems.

In our case, the functionality of `ps` is also implemented by busybox. It is also capable of listing threads, not processes, which suits our needs better, since we work with tasks. This feature can be enabled by `-T` flag. To compile it into busybox, we had to modify its config via `make busybox-menuconfig` and enable `Process Utilities/Support thread display in ps/pstree/top`. After this modification, we had to recompile Buildroot, which recompiled busybox with the requested modification and created a new rootfs file.

Some modifications also had to be performed on OP-TEE. These patches were discussed in details in Section 4.1 for OP-TEE OS and Section 4.4 for OP-TEE client.

In order to make it compliant with our solution, the configuration of the Linux kernel had to be modified as well. Since we check the hash of the entire code segment of the kernel, components using self-modifying code break this check. We changed the file used

[6]`https://github.com/OP-TEE/manifest`, Last visited: 06.04.2021

[7]`https://optee.readthedocs.io/en/latest/building/devices/qemu.html#qemu-v8`, Last visited: 06.04.2021

by OP-TEE's build system to generate kernel config, to not allow components of KVM to be compiled into the kernel. Since the intended use-case of our application is to protect IoT devices, the lack of virtualization support in the kernel is acceptable.

A new driver had to be introduced: as we check code segments of user-space applications, we need all their code containing memory pages to be in the physical memory, not in swap files or swap partitions. The easiest way to achieve this was to create a driver which can generate page fault events on-demand. These event occur when accessing a memory page fails because it is swapped out. The kernel reacts to this by reading the page into the physical memory. For our CA, we could do this easily within the application, but to be able to hash the text segment of arbitrary processes without modifying them, we had implement a Linux kernel module which triggers the page fault events. It creates an entity under the `/proc` directory of the system, and if a pid is written to this file, the driver checks if the corresponding process is present in its whitelist and triggers a page fault for all of the pages containing the code segment of the process. The heavy lifting is done by the `get_user_pages_remote`[8] function of the memory management subsystem, we only had to implement the process lookup and invocation of the function.

Although this module could be compiled separately from the kernel and inserted by the init system, in the design phase we decided to disable module support. This is an acceptable restriction for IoT devices, it makes installing rootkits significantly harder and solves the issue of differentiating malicious modules from benign ones. Therefore we decided to compile this module as part of the kernel. However, to make testing easier, module support wasn't disabled in our development environments; all the kernel-level test rootkits presented in section 5 are implemented as Linux kernel modules. In a production-ready environment, module support must be disabled and all necessary drivers must be compiled into the kernel!

In the previous section, we discussed how we use the symbol table to locate kernel objects in the virtual address space. However, not all the objects we use are meant to be accessible this way, like runquques. They are per-cpu data structures, meaning there is one for every CPU, and a special section of the memory is dedicated to contain such variables. To get the location of the runqueues, we had to patch the kernel to print the addresses somewhere in the boot process. We chose the point before the init system is invoked, namely the `run_init_process` function[9]. We placed one print call into the function for every runqueue, and obtained their addresses via the `cpu_rq`[10] macro. The addresses printed by the kernel are hard-coded into our application.

Another important and highly platform-dependent component of our solution is the one responsible for address translation. We need it because the Normal and the Secure World are using two distinct virtual address spaces. When the physical memory holding the

[8]`https://elixir.bootlin.com/linux/v5.1/source/mm/gup.c#L1114`, Last visited: 08.04.2021

[9]`https://elixir.bootlin.com/linux/v5.1/source/init/main.c#L1007`, Last visited: 11.04.2021

[10]`https://elixir.bootlin.com/linux/v5.1/source/kernel/sched/sched.h#L998`, Last visited: 11.04.2021

Linux kernel is mapped into the address space of OP-TEE, it can be accessed via TEE virtual addresses. The kernel data structures however use REE virtual pointers, and directly using these would result in a translation fault, since these are incomprehensible from OP-TEE's point of view. We solve this issue by translating REE virtual addresses to physical ones, what we can translate to TEE virtual ones. OP-TEE solves the later one for us, but REE address translation must be implemented in our TA.

Linux kernel address translation consists of two parts: kernel- and user-space translation. Both of them had to be implemented, since we need to access the memory of certain processes as well.

To translate kernel addresses, we copied the ARM64 version of a macro called `__virt_to_phys_nodebug` and all of its dependencies transitively. This resulted in approximately 30 macros and it's capable of translating kernel address to physical addresses by performing only simple arithmetic- and bit-operations.

User-space address translation is less straightforward: it's implemented using a so called page table hierarchy, where memory pages contain pointers to other memory pages, and parts of the virtual address can be used as indexes within the current page to look up the page on the next level. This way pages form a tree, and a lookup operation can be considered as a path in the tree, from the root to a leaf. The leaf contains the physical address of the memory page, corresponding to the virtual one.

In our Qemu based environment, the kernel uses 4 levels of page tables, each page having the size of 4kB. Translation is originally performed via macros capable of determining the offset of the next pointer in the current page, however, in our case, we need to access these pages via the normal world memory api. Figure 4.2 illustrates how these page tables are used to look up pages from virtual addresses.

4.6.2 i.MX 6

Using Qemu as a development environment was an ideal choice, since it's easy to use, works out of the box and very little customization was necessary. However it's not suitable for a full-featured proof of concept solution, since our application relies on mechanisms provided by certain hardware elements. Thus we implemented the rootkit detection solution for a real hardware as well, where we could use a watchdog timer to handle discovered infections properly.

To achieve this, we used an i.MX 6Quad SoC (System-on-Chip) with an Apalis Evaluation Board, a carrier board which provides a wide range of peripherals. In this environment, we could implement Secure Boot and we could utilize the functionality of a watchdog timer. This way we can test our solution in an environment, where we can defend our application against rootkits trying to disable the detection solution. Also Secure Boot makes it possible to handle infections by resetting the system, since we know it will be in a clean state when it finishes the boot process.

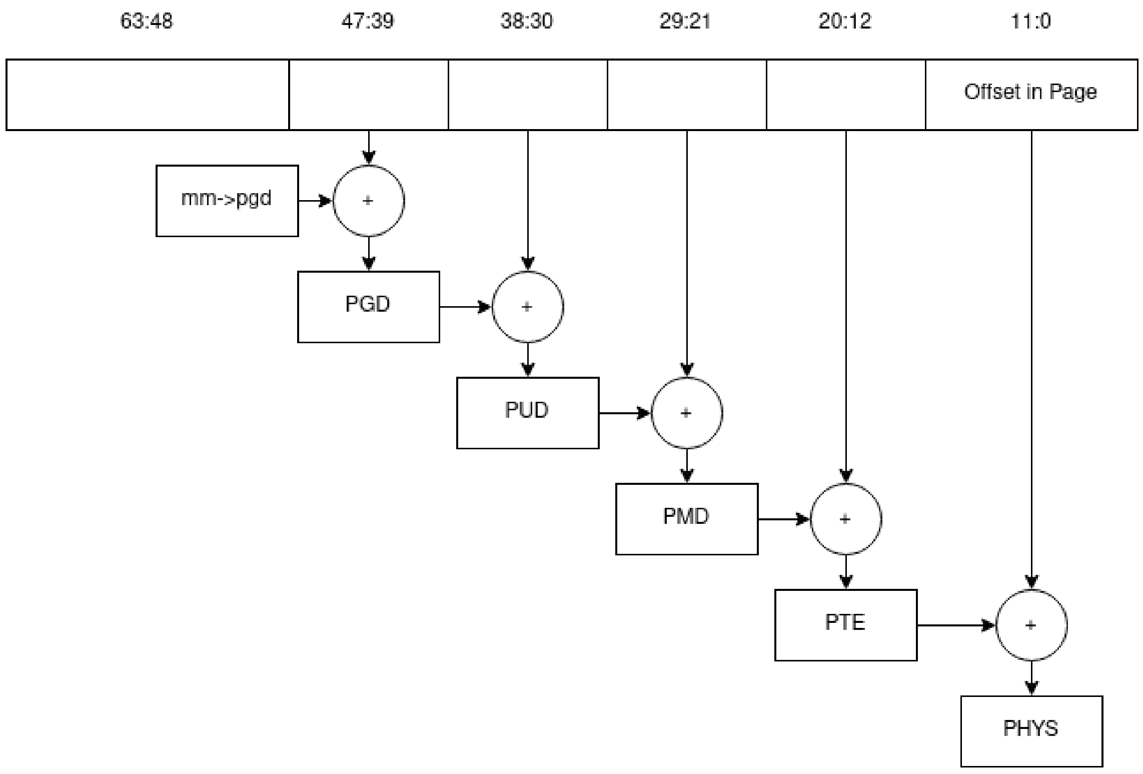

Figure 4.2: This figure illustrates how sections of a memory address are used to look up physical pages by traversing the page table hierarchy on ARM64. `mm->pgd` is a pointer in the memory management object of the task.

Unfortunately, OP-TEE does not support i.MX6 the same way as it supports Qemu or Rasperry Pi. OP-TEE OS is ported to i.MX6, but there are no manifest files, so we had to create our own build environment. We used Buildroot, just like OP-TEE's manifest files, but in this case, we had to setup a custom Buildroot project to collect, compile and assemble the necessary components.

We used version 2019.08-rc1 of Buildroot, which had to be patched first. It would use the version 3.5.0 of OP-TEE, however our solution was developed to 3.6.0. So we had to delete these packages from Buildroot and we supply the appropriate version of these components as external packages.

Our customized Buildroot project consists mostly of configurations, patches and custom packages. In this section, we first describe the used configurations. Then we discuss the applied patches and the packages added to Buildroot. Finally we describe how we had to modify some files on the created rootfs, how we implemented secure boot and what platform specific features we had to implement in our solution.

We added our own busybox configuration, where we enabled thread listing in process utilities, like described in the previous section. We also have our own Linux config. This file is based on the default iMX6 configuration of the kernel and contains the same customizations as the Qemu Linux config, like the pagefaulter driver and disabling KVM, and some more. We had to enable fsnotify, since it was disabled by-default. Also we had to disable `CONFIG_ARM_PATCH_PHYS_VIRT`[11] and `CONFIG_HIGHPTE`[12]. These are memory management related configurations, and their absence made it easier to implement address translation for this environment. Regarding OP-TEE, we don't have a config file to collection our configuration options, we had to place place them in the Makefile used to compile the package itself. Here we had to specify the platform we use, the architecture and the address where the Linux kernel will be loaded into the memory. We also want the device tree files to be embedded in the image and we enabled the Power State Coordination Interface (PSCI).

Our patches are pretty much the same as in the Qemu environment: we also had to patch the Linux kernel to get runqueue addresses and we also had to patch OP-TEE OS and the client. There is one additional patch: since OP-TEE OS doesn't have device tree files for the evaluation board we use, we had to copy some device tree files from the Linux kernel to the appropriate location in order to make sure that every hardware component can be used by the TEE.

The patches are collected into a directory, where every package has its own subdirectory. Patch files are placed in these folders and Buildroot is capable of applying them before it compiles the packages.

We also extended Buildroot with several external packages. We compile the test, client and OS packages of OP-TEE this way, so we can use the version we used when we developed on Qemu, and we can set our configuration for OP-TEE OS more easily. Although it's not an external package, but we must note that we use the same Linux kernel as we did for Qemu. Our PTAs are also added as external packages, but they are not compiled directly. Their source is copied to the appropriate directory of OP-TEE OS before it's built, and they are also copied to the build directory of Buildroot, so our TA can find the necessary header files. Our rootkit detection solution is also an external package. The build scripts of OP-TEE are used to compile the TA, while the CA is cross compiled with the toolchain used by every other package. We also have a test rootkit which will be discussed in details at the end of Section 5.1; it's built by Buildroot's makefiles for kernel modules. Finally, we created a vulnerable web application for demonstrational purpose. It doesn't require building, the PHP files are just installed to the appropriate directory of the target folder, which will be used to create the image of the rootfs.

We also have an overlay directory whose content is applied to the rootfs created by Buildroot before it packages the created file system into a single file. This overlay contains configuration files for the web server we use to host our demo application (described later at the end of Section 5.1), a configuration file for sudo (also used by our demo) and a network config file to assign a static IP address to the network interface of the board.

[11]`https://elixir.bootlin.com/linux/v5.1/source/arch/arm/Kconfig#L229`, Last visited: 19.04.2021

[12]`https://elixir.bootlin.com/linux/v5.1/source/arch/arm/Kconfig#L1622`, Last visited: 19.04.2021

We also managed to partially implement Secure Boot on the device. An ideal implementation of the boot process would be the following: The firmware checks its own integrity using a key stored in a write-once hardware component. Next, it checks if OP-TEE is intact and starts it. When OP-TEE finished initialization, it gives the control back to the firmware, who checks the integrity of the boot loader (U-Boot in our case). If the check succeeds, the firmware starts the boot loader. It then checks the Linux kernel and the rootfs, and then starts the REE OS. The integrity check of the rootfs can also be performed by the Linux kernel itself. Our implementation executes a slightly different flow and some of the low-level components are not checked. In our case, the firmware starts the boot loader, which executes OP-TEE. OP-TEE then starts the Linux kernel. Integrity checks are only performed on OP-TEE, the Linux kernel and the rootfs. Figure 4.3 illustrates the difference between the described approaches. The main reasons of this setup was the lack of Arm Trusted Firmware for i.MX6 and the lack of time for further refinements.

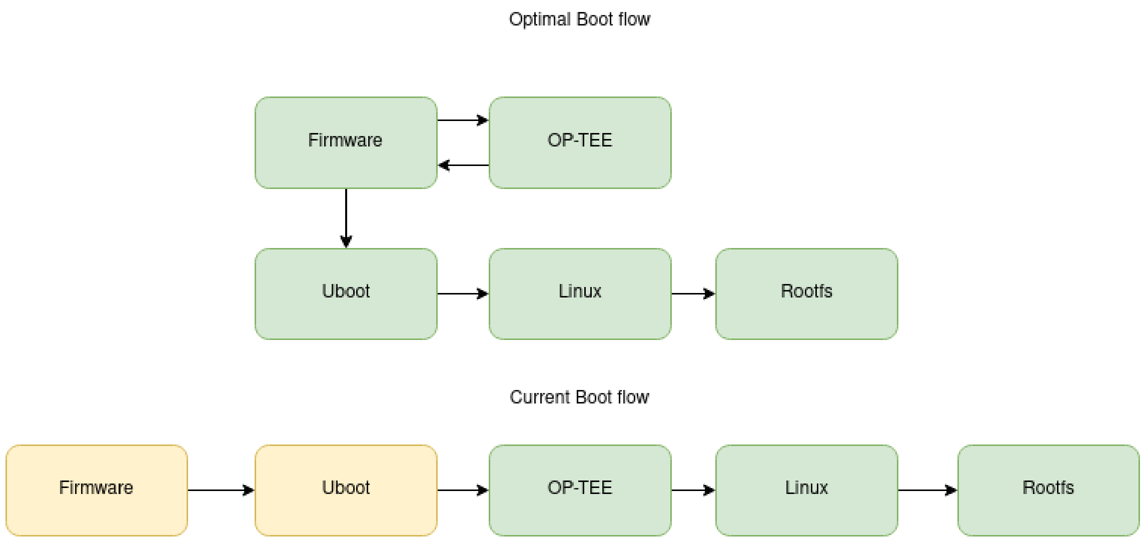

Figure 4.3: The optimal and currently implemented boot flow. Checked components are green, unchecked ones are yellow.

To achieve this, we created a script which is executed by Buildroot after it created binaries from the Linux kernel and OP-TEE. This script creates a so-called FIT image which can be loaded by U-Boot. It contains the kernel, OP-TEE, the rootfs and device tree blobs, all digitally signed. We modified the boot command in U-Boot, so it downloads this image via TFTP, and if all signatures are valid, it boots OP-TEE, which will pass the execution to the Linux kernel, when it finished initialization.

Finally we need to discuss the platform specific components of our solution. One of these is an error handler function which act as a wrapper around the TEE watchdog API, described in Section 4.3. This is required to maintain compatibility with the Qemu environment, where the watchdog API is not present. This function invokes the `reset_now` routine of the PTA when any of the checks fail. Beside this, we had to implement address translation for this environment as well. The translation of kernel addresses is even simpler than it was for ARM64: with the proper configuration (`CONFIG_ARM_PATCH_PHYS_VIRT` disabled) we only need one macro to compute addresses, which depends on two others.

User-space address translation is, however, just as complicated as it was for ARM64. Here we only have 3 layers of page tables (actually only 2, the kernel is tricked to handle them as 3 layers). Translation happens similarly as described in Subsection 4.6.1.

Chapter 5

Evaluation

In this chapter, we describe the rootkits we used for testing our solution. Unfortunately, only a few 3rd party rootkits could be tested against our solution, so many checks were tested by rootkits written by us. We describe what are the goals of each rootkit, how they implement it and why our solution is capable (or not capable) of detecting them. We also summarize which rootkit is detected by which check in table 5.1. Later, in Section 5.2 we describe some performance measurements regarding the checks we implemented.

5.1 Test rootkits

VFS proc fops hooking: The goal of this rootkit is to hide a process, and it does this by hooking a function pointer in the VFS layer. Two functions are responsible for populating the `/proc` directory: most of the files are returned by the `lookup` function of the inode operations structure, but creation of the pid directories is the responsibility of the `iterate_shared` member of the `proc_root_operations`, a file operation structure in the `.rodata` section of the kernel. By designing an appropriate replacement, we could hide a specific process in this layer; namely, we used init for testing, since it's always present in the system. Our replacement function is a copy of the original, except for one thing. It calls our version of `proc_pid_readdir`, which we also copied from the kernel and slightly modified to ignore tasks whose pid is equal to the one we intend to skip. Our solution can detect this rootkit, since it hooks a function of a well known file operation structure, which is checked at the beginning of the VFS layer checks. When we reach `proc_root_operations`, we detect that the value of `iterate_shared` is not in the allowed interval, so an error is raised.

VFS inode number 0: This rootkit achieves the same goal in the same layer with a different method. The inode structure has a member called `i_ino`, which holds the inode number. Some file system implementations, like the procfs, for example, handle 0 as an invalid value for the inode number. Thus such filesystems might not list inodes with invalid inode numbers. In case of the procfs, we were able to hide a process by setting the corresponding inode's number to 0. In our VFS checks, every time we check an inode, we inspect the inode number and raise an error, if it's 0.

Removeing from IDR: As we described earlier, Linux 5.1 uses a radix tree to store pid structures, and these structures have lists of tasks with equal ID, thread group ID, etc. As we stated earlier, this tree is used to fill `/proc` with the currently active processes. This rootkit removes a pid structure from this tree to hide the corresponding process. We can easily detect this kind of tampering: before executing our process targeting consistency checks, we query `ps`, and we find the hidden process in the list of tasks. However it's missing from the output of `ps`, thus we raise an error, since a hidden process was found.

Removeing from the task list: This rootkit is very similar to the previous one, except that it removes a task from the list of tasks. It was originally written to an earlier version of the kernel, since at an early stage of development we were experimenting with Linux 4.12, where `/proc` was populated from the task list. In our current environment, this rootkit is not capable of hiding processes, yet it can demonstrate a powerful property of our solution. We can detect this rootkit because it makes kernel objects inconsistent. When the list, the tree and the IDR are traversed, a union of the found pids is created, and every one of the collected lists is checked against the union. At this check, our solution raises an error, since there is a task which is present in the union, but missing from the task list.

Removeing from list, tree & IDR: An obvious improvement of the previous two rootkits is to remove a process from all three data structures containing information about processes. Removal from the task list and the IDR are trivial. The process tree however is a bit more complicated because of the way Linux implements the tree. Each process has a pointer to its first and last child, and the children are linked to each other via the siblings list. To successfully remove a process from this tree, we have to remove it from the list of its siblings and from the parent as well. In this case, we have four cases: if we remove the only child, the first, the last or if the parent does not reference the process directly. If all of these removals were successful, the detection depends on the payload of the rootkit, the removed process. If the payload is a computationally intensive task, like a crypto miner for example, it is likely to be detected, since it spends most of the time in runqueues, where we can find it. If the rootkit hides a backdoor, or something that spends most of its time waiting, it is unlikely to be detected.

System call hooking: Our next rootkit hooks the system call table. It overwrites a pointer in the syscall table with one pointing to its own code segment, so if the hooked syscall would be called, execution would jump to the function implemented by our rootkit. We can detect this, since when we hash the syscall table, the hash won't be the same as the hash of the intact pointer array.

Inline hooking: This rootkit modifies the text segment of the kernel. It overwrites the first few bytes of one of the functions implemented in the kernel. Just like in the case of the syscall table, the computed hash will change after the hook and we can detect this.

Netfilter hooking (ICMP& TCP): We implemented two rootkits targeting the Netfilter subsystem, one for ICMP and one for TCP. Both of them create a new hook with a function implemented by them. They are capable of executing arbitrary payload if an incoming packet matches the criteria implemented by these functions. Since these functions reside in the memory area where kernel modules are loaded, and not in the text segment of the kernel, by inspecting the function pointers of the hooks, we are capable of detecting these rootkits.

ICMP controls hooking: This rootkit hooks one of the pointers in the `icmp_control` array used to determine what function should handle the incoming ICMP packets. We hooked the function responsible for handling ICMP echo requests, so our own implementation is executed when the system is pinged. Again, since the value of the function pointer in one of the `icmp_control` structures is not in the allowed interval, we raise an error.

Protocol hooking: This time, we hooked the `handler` member of the `icmp_protocol` structure. This is the function which handles all ICMP packets and in fact, it calls the appropriate handler described at the previous rootkit. The mechanism of detection is identical to the previous one: we check all function pointers of all of the `net_protocol` structures, and as we identify one pointing outside of the kernel's text segment, an error is raised.

seq_ops hooking: This rootkit hooks a structure called `tcp4_seq_ops`, the `seq_ops` structure for IPv4 TCP. It is used when the content of `/proc/net/tcp` is printed. We overwrite the value of the `show` pointer with our own implementation, which calls the original, and if it finds the port to be hidden in the buffer where the output is collected, it sets the position back to the beginning of the line, so it will be overwritten by the next line. As we check all available `seq_ops` structures, when we reach `tcp4_seq_ops`, we find the invalid pointer and raise an error.

So far, we described the test rootkits that we created for testing our solution. These were specifically written to test certain features of our rootkit detection solution. However we were able to make some open source rootkits work in our environment. Next, we describe these and the way how our solution could detect them.

BEURK: BEURK is a userspace rootkit, the name stands for Beurk Experimental Unix RootKit. It exploits the linker's capability to load a library into the address space of a process before everything else, thus, it can hook certain library calls. It creates a backdoor when the `accept` function is called and certain conditions are met (local port matches its configuration, remote port is in range specified in its configuration). The created process is hidden from every other process, except its children. This is achieved by hooking the `readdir` and `readdir64` functions, which are wrappers around the `getdents` and `getdents64` system calls, respectively. When they are called on `/proc`, if the next entry is the pid of the backdoor, then it is skipped. We were able to detect the presence of the BEURK backdoor because it is missing from the output of `ps`, but present in the task list.

A HORSEPILL variant: HORSEPILL [23] is a ramdisk-based rootkit, which exploits namespaces. It infects `klibc`, a minimal library used in the early user space. It hides a process by creating a new pid namespace and executes `systemd` in this namespace. Normally, it would not be possible to see kernel threads with this setup, but HORSEPILL has a workaround to fake these in the freshly created namespace. Unfortunately, HORSEPILL is not compatible with our test environment, as we use a different init system, we do not use klibc, and we do not assemble ramdisks on the system like personal computers usually do. However, we were able to port the idea behind HORSEPILL to work in our environment. In our case, the ramdisk is assembled by Buildroot[1], and starting the init process happens as follows: First, the Linux kernel calls `/init`, a minimal shell script, that sets the 3 default file descriptors to `/dev/console`. Then, it invokes `/sbin/init`, which is a symlink to busybox[2]. We managed to start our HORSEPILL variant by replacing the symbolic link to another binary. This binary clones two new threads and executes the original init and our backdoor in them. Busybox is executed in a new pid namespace to hide the backdoor from the rest of the system. Our implementation lacks HORSEPILL's feature of faking kernel threads, therefore, they appear as hidden processes to our detection solution, much like the backdoor process. Originally, HORSEPILL fakes kernel threads by collecting their names, creating new processes in the namespace of init, and renaming them to the names of the kernel threads. The output of our detection mechanism would be the same in this case: we would find the original ones and the backdoor.

Diamorphine: Diamorphine is a kernel-space rootkit compatible with a wide range of Linux kernel versions (2.6.x-5.x). Originally it was supporting x86-based architectures only, but we were capable of porting it to ARM64 and our pull request[3] is already accepted. Diamorphine uses system call hooking to hide files and processes. `getdents` and `getdents64` are hooked to hide files and processes while `kill` is hooked to provide easy communication with the rootkit. Files are hidden if they have a magic prefix, while processes can be turned invisible if they receive signal 31. Since the rootkit is tampering with the syscall table, our hash check can detect it easily.

[1]`https://buildroot.org/`, Last visited: Sep 18, 2020
[2]`https://busybox.net/`, Last visited: Sep 18, 2020
[3]`https://github.com/m0nad/Diamorphine/pull/21`, Last visited: 14.04.2021

Finally we created a complex rootkit, a combination of our previously described rootkits, and an environment where we can properly demonstrate the capabilities of our solution. This test rootkit was designed to grant access to a compromised system and hide the backdoor as much as possible. First, it creates a Netfilter hook which spawns a backdoor process if a packet is received on a predefined port which is coming from a predefined port and its content is a secret defined at compile time. When the rootkit creates the backdoor process, it immediately hides it by removing it from the list of tasks, the process tree and the IDR as well. The payload of the rootkit is a bind shell written in go, so it opens a TCP port immediately. To hide this, our rootkit hooks the appropriate `seq_ops` structure. In order to operate as stealthy as possible, our rootkit implements a special dropper mechanism: the executable of the backdoor is stored inside the data segment of the rootkit and it's spawned by creating a temporary file which exists only as long as the process is started. This way we do not need to write it to a disk and start it from there. For a proper demonstration, we implemented a vulnerable web application running on the i.MX6 board. This application contains a local file inclusion vulnerability[4] which can be exploited by log poisoning[5]. This can be used to install our rootkit. Our detection solution is invoked periodically, and once the rootkit is installed, we can detect its presence by the Netfilter hook. If the backdoor is triggered, the hidden process might be detected, but since we implemented DKOM thoroughly, it's unlikely. The Netfilter hook however will be detected, and if our solution wouldn't stop scanning on the first suspicious result, it would detect the `seq_ops` hook as well. When the malicious Netfilter hook is detected, the watchdog driver is used to reset the board and it boots again from an clean, uninfected image containing the kernel, the TEE and the root filesystem. Signature verifications ensure the integrity of these components.

5.2 Performance measurements

Performance impact is always a concern with anti-malware solutions. In this subsection, we present the performance characteristics of each part of our implementation and review their impact on the system. These measurements were performed in our Qemu-based environment, which uses 1057 MBs of RAM and 2 Cortex-A57 cores.

Since many of our checks are process-related, we measured the execution speed depending on the number of currently running processes: we spawned new background processes with the `less` command, which did not consume relevant amount of CPU time slices, but increased the size of the data structures we needed to check. Figure 5.1 shows the execution time of the checks separately and in total as well.

RPC call stands for the communication between the CA and TA: all necessary input is collected, the TA is invoked and it performs normalization of its input. As expected, the number of running processes is irrelevant in this case.

[4]`https://www.acunetix.com/blog/articles/local-file-inclusion-lfi`, Last visited: 15.04.2021
[5]`https://owasp.org/www-community/attacks/Log_Injection`, Last visited: 15.04.2021

Check	VFS		K. obj.		Integrity		Network			
Rootkit	Ops	Ino	Adm	Sched	K. text	Syscalls	Nf	Proto	Icmp	seq_ops
vfs proc fops hook	X									
inode number 0		X								
idr remove			X							
list remove			X							
wait hider				?						
syscall hook					X					
inline hook						X				
icmp nf hook							X			
tcp nf hook							X			
protocol hook								X		
icmp ptr hook									X	
tcp seq_ops hook										X
prototype test				?			X			X
beurk			X							
HORSEPILL			X							
Diamorphine						X				

Table 5.1: The test rootkits and the checks detecting them.
(X: successful detection, ?: possible detection)

Checking the integrity of the VFS does not scale well. This is due to the nature of the `/proc` file system. For every process, approximately 300 new files are created by the kernel. These are regular files and symbolic links, resulting in an increasing number of inodes. Our checks scale linearly with respect to the number of inodes.

The DKOM check achieves better performance with respect to the growth in the process count: for n processes, it traverses a list made of n elements and two trees with n nodes each. It also sorts arrays of n pids and performs binary searches to find differences between the collected lists of pids.

Integrity checks scale very well, the hash check of the kernel's `text` segment and the system call table can be performed with constant time complexity, and interaction with task-related data structures is necessary only when it is looking for the task of the CA (for the purpose of hashing its `text` segment and verifying the integrity of the `fanotify` marks) and the tasks of `tee-supplicant`. However, extending this integrity check to other tasks would probably have an impact.

After the integrity checks, our solution performs the file system checks. These checks have no relation to the process count, they depend on the number of files included in the check and the overall size of those files. This part is by far the most time consuming, since it requires RPC calls and world switches to read the bytes of the files.

Finally, we perform checks on the network stack. As expected, the execution time of these checks is independent from the process count. The performance of the Netfilter check is

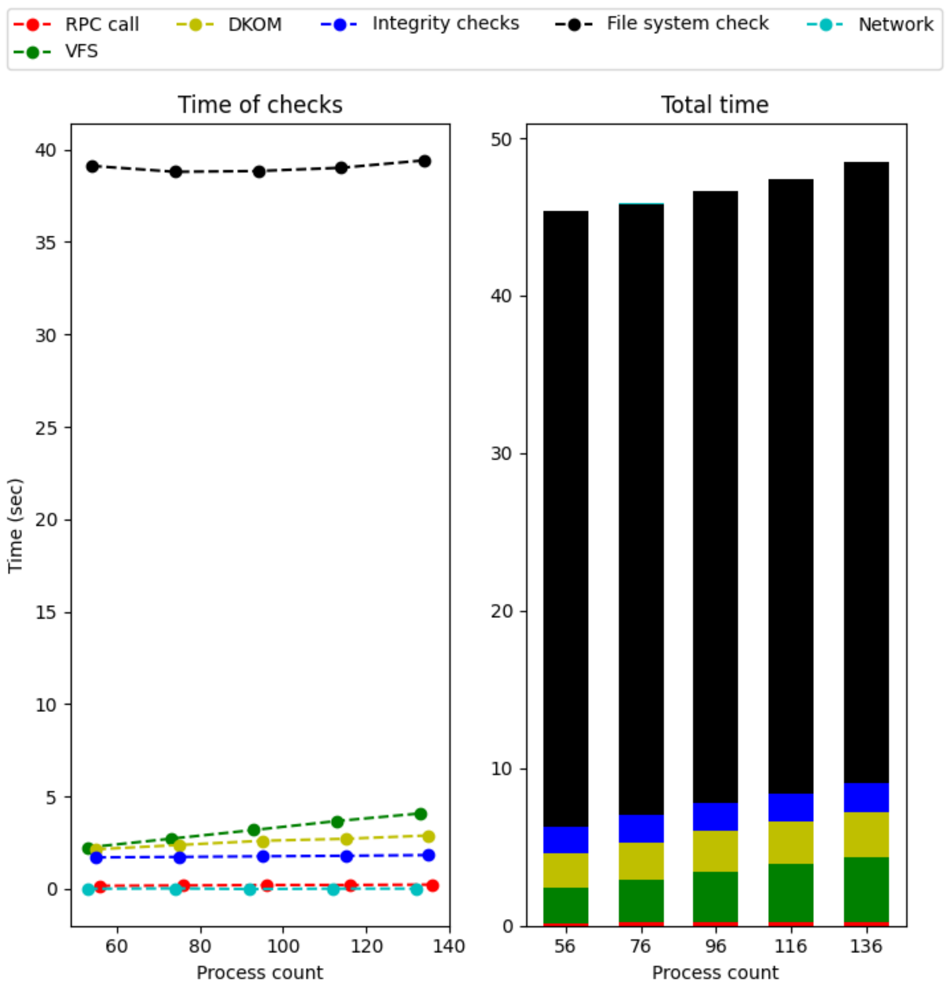

Figure 5.1: Execution time in seconds depending on the number of processes

linear to the number of active firewall rules, while all the other checks in this section has a constant execution time.

Our plot shows that these checks are sometimes executed faster, than the RPC call itself, which is part of all our measurements. This anomaly comes from the way we measure: we calculated averages from multiple execution, and due to their small performance impact, they are indistinguishable from the baseline. On our second plot, which shows total execution time, they practically disappear, since there we plot $network\_avg - rpc\_avg$. To circumvent this issue, we present the average execution times in Table 5.2

While our scanning process is being executed in the TEE, it uses only one of the cores. As a result, the REE can execute on the other core. Processes running in the REE are not halted while we perform our checks, but significantly less resources are available to them until the checks are completed. Until our TA reaches the file system checks, the core it uses can only execute REE code when an interrupt occurs that has to be handled in the REE. During the file system checks, however, our implementation needs to wait for a lot

Check \ Proc count	~55	~75	~95	~115	~135
RPC calls	0.168	0.189	0.214	0.215	0.236
VFS check	2.412	2.906	3.404	3.891	4.322
DKOM checks	2.311	2.568	2.82	2.928	3.129
Integrity checks	1.876	1.922	1.981	2.008	2.068
File system check	39.2871	38.988	39.055	39.232	39.654
Network checks	0.178	0.219	0.208	0.234	0.254

Table 5.2: Average execution times in seconds, the raw data presented in Figure 5.1

of I/O operations, and while waiting, control is given back to the REE, where the Linux kernel's scheduler can execute other tasks on the first core as well.

Chapter 6

Discussion

In this section, we present some known weaknesses of our approach and discuss ideas to overcome them.

6.1 Known limitations and possible solutions

The first limitations comes from OP-TEE and the way it handles interrupts. Interrupts are divided into two groups, foreign and native interrupts. The former one needs to be handled by the Normal World, and the latter one by the Secure World. If an interrupt rises and the CPU which should handle it is not in the appropriate world, the machine switches to the correct world, and the interrupt handler is executed. However, OP-TEE does not have its own scheduler, but it uses the Linux kernel's scheduler. When a CPU is executing code in the Secure World and a foreign interrupt occurs, the execution of the TA does not continue immediately after the handler exits. It only resumes execution when the scheduler gives the CPU to the thread associated with the TA.

In addition, on a system with multiple cores, it is possible for one core to execute in the Normal World and another in the Secure World. This behavior can make our inconsistency checks unreliable: it is possible for a new thread to start during our checks, making them fail despite the lack of any hidden processes. This issue might be resolved if we can disable other cores during our checks, and disable interrupts as well to ensure the uninterrupted execution of our checks. Disabling a core is possible from the REE, however, it has a negative impact on the performance of the system. Disabling interrupts is a bit more complicated, although, PTAs can do it while they are running. It might be possible to disable and re-enable interrupts for other Secure World threads as well, or the check itself can be implemented in a PTA in order to use this feature.

Another, less manageable issue is the way the Linux kernel handles waiting tasks. This is implemented via wait queues, which store the tasks waiting for the same event. When the event occurs, it is possible to wake up all the tasks or just one of them. The issue lies with how Linux creates wait queues: some exists as global variables, but most of them are created on stacks or as members of other structures. So far, we have not been able to find a way to enumerate all the currently existing wait queues in memory, thus, we are unable to check all of them, only those implemented as global variables.

Our current method is unable to handle self modifying features in the Linux kernel and Address Space Layout Randomization in the kernel. These features would break the integrity check of the kernel's `text` segment.

Finally, a way to bypass all of our checks would be for a malware to uninstall itself before the checks are performed. When the checks are completed, the malware could be reinstalled by exploiting the same vulnerability it originally exploited to infect the system. Note, however, that this can work against any rootkit detection approach, because there remains nothing malicious to detect in the system.

6.2 Future work

There are multiple features with which our method could be extended. First, our current implementation does not support updates. REE package updates would modify or add new files to the file system, so they would likely break the file system check for persistent rootkit components, and a kernel update would certainly break the integrity checks and probably the VFS and task-related ones too. In the future, we would like to address the former issue by recomputing reference hashes in a secure way. The latter issue may require re-implementing certain checks, making its automation challenging.

We discussed multiple checks for Linux kernel modules. They are the most convenient way to execute code in kernel space [10], therefore, rootkits often use them and try to hide their modules. Consequently, we can disable module support in the Linux kernel configuration, making rootkit installation more challenging. In our test environment, we only left it to be enabled in order to be able to test rootkits & the solution more easily. However, without module support, all necessary drivers must be compiled into the kernel, which is a functional restriction.

We would also like to make our solution compatible with other kernel security features. We reviewed a long list of possible configurations and our solution is incompatible with only two of them: structure randomization and Kernel Address Space Layout Randomization. In case of structure randomization, the members of selected kernel structures are randomized at compile time. We could make our method compatible with this feature by compiling a kernel module with access functions, for example, to get the next task in the task list, and use these functions from a library in our TA. Kernel Address Space Layout Randomization is a technique which places the kernel's text segment at a random location at boot time. This feature interferes with several of our checks and we do not have a solution so far that can support this feature.

Finally, we found an anomaly we cannot fully explain yet: when we execute our checks for the first time after the boot, everything works fine. However, from this point on, every time our TA is invoked, our process checks identify a task who is missing from the list of tasks, but present in the task tree, the IDR and the output of `ps`. This task bears the name of the `tee-supplicant` daemon, and by its pid, we suspect this might be the process which executes our TA. We suspect the cause of the issue might be something in the OP-TEE driver, where the `TA_FLAG_INSTANCE_KEEP_ALIVE` configuration is handled erroneously. Yet this is all hypothetical, this topic needs further investigation. Once we have more information about this anomaly, we will involve the developers of OP-TEE to resolve this issue.

Chapter 7

Conclusion

In this paper, we addressed the problem of detecting rootkits on embedded IoT devices. Rootkits are malicious software that typically run with elevated privileges, which makes their detection challenging. Our solution is based on identifying signs of a rootkit infection (i.e., modifications to the code of system programs and the operating system kernel, as well as inconsistencies in certain kernel data structures) using a trusted application that is running in an isolated trusted execution environment. Fortunately, such trusted execution environments are supported on many embedded platforms used in IoT applications, and their protection measures ensure that malicious code cannot interfere with our detection mechanisms even when running with root privileges. We described in detail how we check both the memory of the untrusted execution environment and the persistent storage from our trusted application, looking for integrity violations and inconsistencies. We also reported on a prototype implementation of our approach, including some specific implementation level issues that we had to solve to make our prototype working in practice. Finally, we evaluated our design and implementation by testing the prototype with rootkits that we developed for this purpose.

Our approach has some limitations that we discussed in the paper. In summary, we can detect modifications of the kernel code and system programs, as well as hooking attacks in the memory, and we can also detect the presence of rootkit components in the persistent storage of the IoT device. Detection of manipulations of process related kernel data structures is not complete, as we were not able to analyze certain data structures (e.g., wait queues). In addition, at the time of this writing, we do not support multi-core processors, address space layout randomization, and self-modifying code in the kernel. Some of these limitations can be addressed (e.g., the kernel can be statically compiled with all the drivers included), while others require more work in the future. Despite all these limitations, we believe that our work demonstrates that it is possible to protect even small embedded devices used in IoT applications from sophisticated and powerful software based attacks, and that IoT is not necessarily as insecure as it is commonly perceived.

Acknowledgements

The presented work was carried out within the SETIT Project, which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme (project no. 2018-1.2.1-NKP-2018-00004).

I would like to thank my colleague Krisztián Németh for his contribution with the persistency check, without that, our solution would be incomplete. Also I would like to say thank you to Szilárd Dömötör and István Telek whose prior work served as inspiration for this project. Finally, I am thankful for Dorottya Futóné Papp for her design advices and for Dr. Levente Buttyán for his guidance during this project.

Bibliography

[1] Cfs scheduler. `https://elixir.bootlin.com/linux/v5.1/source/Documentation/scheduler/sched-design-CFS.txt`. Kernel documentation at version 5.1.

[2] Deadline task scheduling. `https://elixir.bootlin.com/linux/v5.1/source/Documentation/scheduler/sched-deadline.txt`. Kernel documentation at version 5.1.

[3] Real-time group scheduling. `https://elixir.bootlin.com/linux/v5.1/source/Documentation/scheduler/sched-rt-group.txt`. Kernel documentation at version 5.1.

[4] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the Mirai botnet. In *USENIX Security Symposium*, August 2017. URL `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`.

[5] A Baliga, X Chen, and L Iftode. Paladin: Automated detection and containment of rootkit attacks. Technical report, Rutgers University Department of Computer Science, 2006.

[6] A. Baliga, V. Ganapathy, and L. Iftode. Detecting kernel-level rootkits using data structure invariants. *IEEE Transactions on Dependable and Secure Computing*, 8(5): 670–684, 2011.

[7] R. Bharadwaj. *Mastering Linux Kernel Development*. Packt Publishing, 2017. ISBN 9781785883057.

[8] William Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones and Bartlett Learning, 2012.

[9] P. Bravo and D. García. Rootkits survey: A concealment story. Technical report, University of Oviedo, 2011. https://pablo-bravo.com/files/survey.pdf, Last visited: 21.09.2020.

[10] A. Bunten. Unix and Linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling*. Budapest, Hungary, 2004.

[11] R. Carbone. Malware memory analysis of the Jynx2 Linux rootkit. Technical report, Defence Research and Development Canada, 2014.

[12] S. Devik. Linux on-the-fly kernel patching without LKM. *Phrack Magazine*, 2001. http://phrack.org/issues/58/7.html, Last visited: 21.09.2020.

[13] Sz. Dömötör and L. Buttyán. Run-time kernel integrity monitoring on embedded platforms. Technical report, Budapest University of Technology and Economics, 2018.

[14] Sz. Dömötör, M. Juhász, G. Székely, I. Telek, and L. Buttyán. Enhancing the security of the internet ofthings: Design and implementation ofsecurity mechanisms for embeddedplatforms. Technical report, Budapest University of Technology and Economics, 2018.

[15] Sh. Embleton, Sh. Sparks, and C. Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[16] M. Espensen and N. Hoej. Rootkits: Out of sigth, out of mind. Technical report, University of Copenhagen, 2014. https://github.com/NinnOgTonic/Out-of-Sight-Out-of-Mind-Rootkit, Last visited: 16.09.2020.

[17] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, USA, 2007. USENIX Association.

[18] 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. ISBN 978-94-6252-188-9. DOI: `https://doi.org/10.2991/ameii-16.2016.155`. URL `https://doi.org/10.2991/ameii-16.2016.155`.

[19] J. Heasman. Implementing and detecting an ACPI BIOS rootkit. *Black Hat Europe*, 2006.

[20] S. Herwig, K. Harvey, G. Hughey, R. Roberts, and D. Levin. Measurement and analysis of Hajime, a peer-to-peer IoT botnet. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019.

[21] T. Hudson and L. Rudolph. Thunderstrike: EFI firmware bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–10, 2015.

[22] T. Hudson, X. Kovah, and C. Kallenberg. Thunderstrike 2: Sith Strike. *Black Hat USA Briefings*, 2015.

[23] M. Leibowitz. Horse Pill: A new kind of Linux rootkit. In *Black Hat USA*, 2016.

[24] K. Németh, D. Papp, and L. Buttyán. Detection of persistent rootkitcomponents on embedded iotdevices. Technical report, Budapest University of Technology and Economics, 2020.

[25] N. L. Petroni, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium*, page 13, USA, 2004. USENIX Association.

[26] E. M. Rudd, A. Rozsa, M. Günther, and T. E. Boult. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys Tutorials*, 19(2):1145–1172,, 2017.

[27] A. Shah and J. Giffin. Analysis of rootkits: Attack approaches and detection mechanisms. Technical report, Georgia Institute of Technology, 2008.

[28] A. Todd, J. Benson, G. Peterson, T. Franz, M. Stevens, and R. Raines. Analysis of tools for detecting rootkits and hidden processes. In Philip Craiger and Sujeet Shenoi, editors, *Advances in Digital Forensics III*, pages 89–105, New York, NY, 2007. Springer New York. ISBN 978-0-387-73742-3.

[29] P.-A. Vervier and Y. Shen. Before toasters rise up: A view into the emerging IoT threat landscape. In *IoT Security Foundation Conference*, 2018.

[30] Zhihong Tian, Bailing Wang, Zixi Zhou, and Hongli Zhang. The research on rootkit for information system classified protection. In *2011 International Conference on Computer Science and Service System (CSSS)*, pages 890–893, 2011.