Acta Cybernetica **25** (2021) 369–400.

Rootkit Detection on Embedded IoT Devices*

Roland Nagy[ab], Krisztián Németh[ac], Dorottya Papp[ad], and Levente Buttyán[ae]

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 propose 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.

Keywords: embedded systems, Internet of Things, security, malware

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

*The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004), 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.

[a]CrySyS Lab, Department of Networked Systems and Services, Budapest University of Technology and Economics, Hungary
[b]ORCID: 0000-0003-2305-3271
[c]ORCID: 0000-0001-6677-6862
[d]ORCID: 0000-0002-9976-614X
[e]ORCID: 0000-0003-4233-2559

DOI: 10.14232/actacyb.288834

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 [1, 16] and gaining momentum [25]. 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* [5]. 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 TustZone 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

[1] `https://developer.arm.com/ip-products/security-ip/trustzone`, Last visited: Sep 20, 2020

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 paper is organized as follows: In Section 2, we introduce the main features of rootkits and explain why they are difficult to detect. In Section 3, we give an overview of our rootkit detection approach, which is based on checking the system memory and the persistent storage from the TEE. These two components of our approach are described in more details in Sections 4 and 5, respectively. We implemented our approach using OP-TEE[2], an open source TEE implementation, and Section 6 contains the details on this implementation effort. In Section 7, we report on the evaluation of our rootkit detection mechanisms, which we performed with custom rootkits that we developed for this specific purpose. Finally, we discuss some limitations of the proposed rootkit detection approach and possible future extensions in Section 8, and conclude the paper in Section 9.

2 Rootkits

The term *rootkit* [21] 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

[2] `https://www.op-tee.org`, Last visited: Sep 20, 2020

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 [7, 21, 22]. *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[3], Jynx2 [8], BEURK[4], vlany[5] and bedevil[6]. Other rootkits, such as HORSEPILL [19], 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`[7], mitigating this attack. Other techniques in this category are kernel-space hooking [28] 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 administrator. There exist a number of rootkits employing these techniques, including Adore-NG[8], LilyOfTheValley[9], the work presented in [14], OSOM [11], SucKIT [9] and Suterusu[10].

Many proof-of-concept rootkits compromise the virtualization layer, the BIOS and/or the firmware of hardware components [10, 15, 17, 18]. 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 [24]. *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[11] and Rootkit Hunter[12]. The main lim-

[3] `https://github.com/chokepoint/azazel`, Last visited: Sep 16, 2020
[4] `https://github.com/unix-thrust/beurk`, Last visited: Sep 16, 2020
[5] `https://github.com/mempodippy/vlany`, Last visited: Sep 16, 2020
[6] `https://github.com/wiperpaul/bdvl`, Last visited: Sep 16, 2020
[7] `https://lore.kernel.org/lkml/18778.1508769258@warthog.procyon.org.uk/`, Last visited: Sep 16, 2020
[8] `https://github.com/yaoyumeng/adore-ng`, Last visited: Sep 16, 2020
[9] `https://github.com/En14c/LilyOfTheValley`, Last visited: Sep 16, 2020
[10] `https://poppopret.org/2013/01/07/suterusu-rootkit-inline-kernel-function-hooking-on-x86-and-arm/, Last visited: Sep 16, 2020`
[11] `http://www.chkrootkit.org/`, Last visited: Sep 17, 2020
[12] `http://rkhunter.sourceforge.net/`, Last visited: Sep 17, 2020

itation 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 [12], 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[13] 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[14]. Kernel data structures can also be monitored for malicious changes, e.g. system call re-maps can be detected using StMichael[15], running processes can be listed with KSTAT[16], and Gibraltar [3] uses 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 [2] is an example of the former approach. It defines protected zones for memory and files, and performs integrity checks from the hypervisor. Copilot [20], 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.

Our proposed method incorporates ideas from cross-view-based and integrity-based detection methods, including integrity checks on files similarly to other file integrity checkers [27], and anomaly detection in kernel data structures similarly to Strider GhostBuster [26]. We provide a more in-depth discussion of our detection methods in Section 3. Unlike other approaches, however, our proposed method

[13] `https://securiteam.com/tools/5jp0m1f40e/`, Last visited: Sep 21, 2020
[14] `https://www.la-samhna.de/samhain/`, Last visited: Sep 17, 2020
[15] `https://sourceforge.net/projects/stjude/`, Last visited: Sep 21, 2020
[16] `http://www.s0ftpj.org/docs/lkm.htm`, Last visited: Sep 17, 2020

does not need additional hardware: we leverage a TEE to prevent the rootkit from interfering with our rootkit detection process. We also use the TEE to safely store baseline values (e.g., file hashes and hash values of memory images).

3 Overview of our approach

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. Moreover, some of the detection functions we present are agnostic to the OS used.

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 `lookup` function in the operation structure of the inode 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 4.1.

- **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 itself, a rootkit may remove its task structure from the process list or process tree, while it must keep itself 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 4.2.

- **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 4.3.

- **Looking for persistent rootkit components:** Finally, 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 may be 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 reinstalled 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 disables all file access operations (including execution of programs) before invoking our TA and re-enables them only after the TA completes its job. More details on this are provided in Section 5.

Figure 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

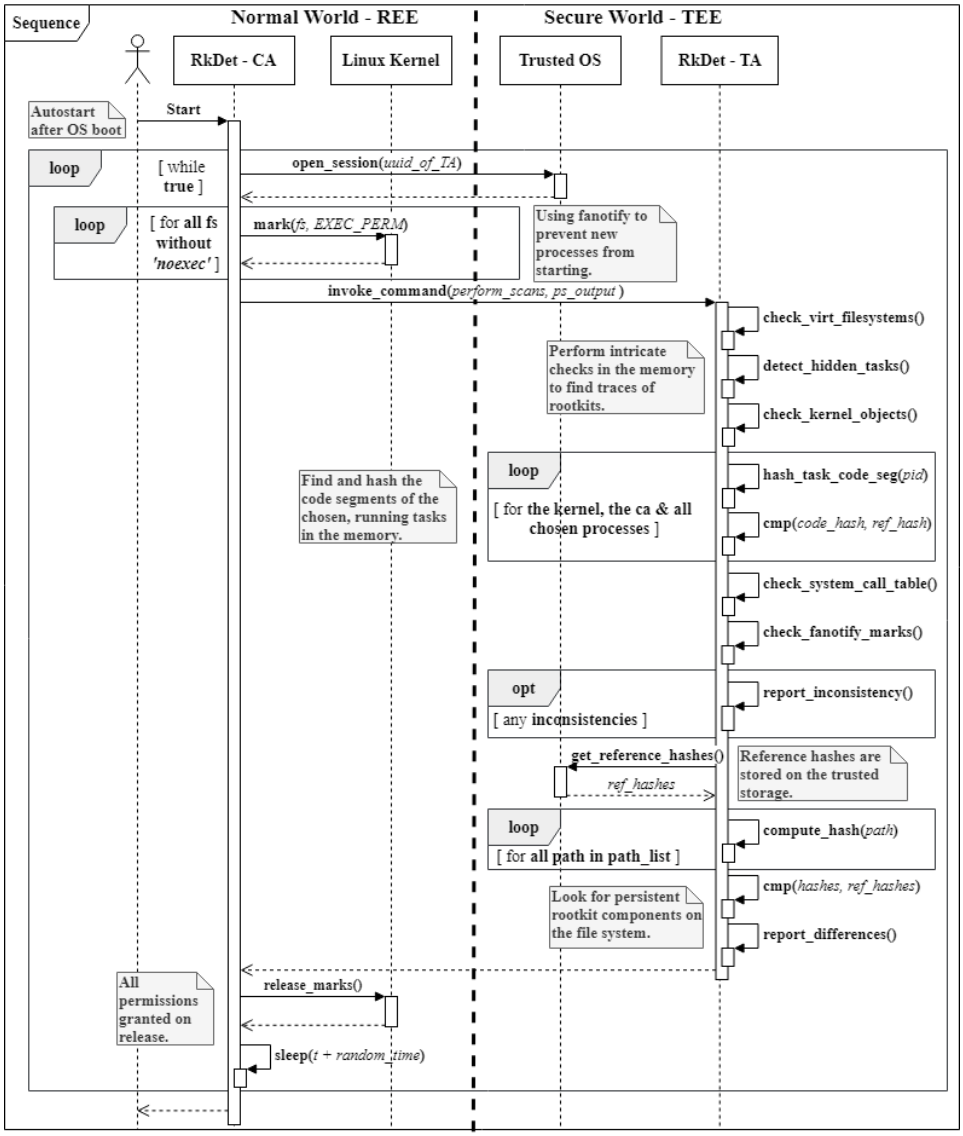

Figure 1: High level overview of our rootkit detection components, their interactions, and the operations they perform.

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

4 Detection of rootkit components in the kernel memory

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 rootkit. 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 a rootkits. In Subsection 4.1, we present the Linux kernel's Virtual File System (VFS) and our technique to detect rootkit attacks against its structures. In Subsection 4.2, we discuss direct kernel object manipulation (DKOM) in details and our techniques to counter this attack. Finally, in Subsection 4.3, we describe our proposed checks to examine the integrity of the analyzed system.

4.1 Detecting hooks in the Virtual File System

The Virtual File System (VFS) [4] 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 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 inode.

Open files are represented by so-called *file* structures in the context of a process. Tampering with these is impractical from the rootkit authors point of view. For example, to modify the way a process reads a file, the hook must be scheduled between the open and the first read call, and this must be performed every time a process opens the file. There are easier and more stable solutions to implement

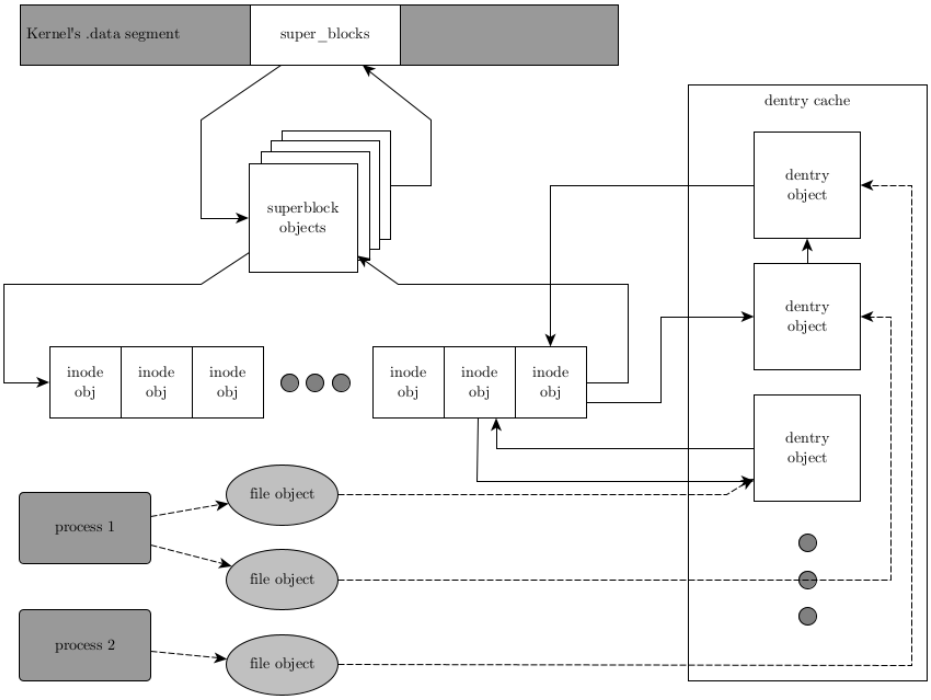

Figure 2: Relationship of VFS objects. White objects are checked, gray ones are ignored by our checks.

this behavior, so we consider files structures out of the scope of our integrity and consistency checks.

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` 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 inode corresponding to the `/proc` directory, and hook the `lookup` member of its `inode_operations`. 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 7 different operation structures. They cover all the operation structures used by superblocks, inodes, and dentries. 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; and `export_operations`, which are operations for the nfs daemon to communicate with file systems. 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.

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 the `/proc` file system as hooking its inode operations is a common technique to hide rootkit processes.

4.2 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[17], containing pointers to pid structures[18] (`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[19]. In recent kernel versions, this mechanism populates the `/proc` directory.

[17] `https://lwn.net/Articles/175432/`, Last visited: Mar 19, 2021
[18] `https://lwn.net/Articles/195627/`, Last visited: Mar 19, 2021
[19] `https://lore.kernel.org/patchwork/patch/834401/`, Last visited: Mar 19, 2021

Direct kernel object manipulation (DKOM) is a rootkit technique which modifies the data structures traversed by the kernel when asked to retrieve information about tasks. 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 Reptile rootkit[20] 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`. Another example is the deadlands rootkit[21]: 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.

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 `/proc`, 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 is found, the TA performs a depth-first search on the task tree and compares the pid list created in 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. 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.

[20] `https://github.com/f0rb1dd3n/Reptile`, Last visited: Sep 18, 2020
[21] `https://github.com/majdi/deadlands`, Last visited: Sep 18, 2020

4.3 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 layers of translation tables[22].

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 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 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. [6]

We were able to remove the write protection using the `update_mapping_prot` function[23], changing the last parameter from `PAGE_KERNEL_RO` to `PAGE_KERNEL`. 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

[22] `https://www.kernel.org/doc/html/latest/arm64/memory.html`, Last visited: Mar 19, 2021

[23] `https://elixir.bootlin.com/linux/v5.1/source/arch/arm64/mm/mmu.c\#L525`, Last visited Sep 18, 2020

calling `update_mapping_prot` on the `text` segment[24]. 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 of any files until the CA approves it. While our checks are performed, the CA denies 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 to the TA. 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. Using the list of marks, the TA checks if all file system partitions accessible from user-space are indeed marked; if this is not the case, then an attack is detected. In order to ensure that the TA is aware of all available partitions, the CA discovers the partitions and saves their in-memory locations to the TA during the init process, right after secure boot, when the system is unlikely to be already compromised. In addition, we extend the VFS checks described in Subsection 4.1 such that the TA also checks if there exist partitions not saved previously. This is needed, because without checking this, the rootkit could create a new partition, which would not be among the saved ones, and hence, the TA would not check if there is a mark on it. By checking that all existing partitions have been saved before, we detect this potential attack too.

5 Detection of persistent rootkit components

Rootkits may try to evade our detection mechanisms discussed in Section 4 by ceasing all malicious activities, restoring kernel objects to their clean states and hiding on the file system using persistence mechanisms. Interested readers may find an in-depth discussion of the topic in [23], we only discuss a few techniques here. The most basic way of achieving persistence is using functions provided by the host OS for task automation and scheduling. On Linux distributions, this includes initialization scripts that execute commands automatically upon system boot or launching the shell, and task schedulers, e.g. `cron`, `anacron` and `at`.

[24] `https://elixir.bootlin.com/linux/v5.1/source/arch/arm64/mm/mmu.c\#L445`, Last visited: Mar 19, 2021

Existing software binaries can be easily modified or replaced on the disk by malware, resulting in the malware being executed when the infected binaries are executed. Libraries can be placed and loaded through the dynamic linker, resulting in the backdoored processes executing malicious code without modifying the program binary itself. Kernel modules that load during the boot process are also possible targets for achieving persistence. These persistence mechanisms leave traces which can be revealed by scanning the file system. We note that there are other mechanisms as well, some of which are not detectable from the file system (e.g., modified boot drivers and firmware). Their detection requires different protective measures, like a secure boot implementation.

We detect rootkits trying to evade our memory checks via persistence mechanisms by scanning the file system for integrity violations. We describe the known uninfected state in Subsection 5.1 and present the scanning process in Subsection 5.3. Our detection method assumes a specific structuring of the file system, detailed in Subsection 5.2. The baseline state to compare hashes against is stored in the TEE's trusted storage, the details are discussed in Subsection 5.4.

5.1 Design decisions and hashed file system entries

Our approach for detecting the persistent parts of rootkits on the file system is to recursively compute hash values for selected directories. To compile such a list, we originally experimented with parsing the crontab files of the system. Our goal was to extract all file references from the scheduled shell commands and use these results together with scanning a selected set of important directories recursively to reduce running time. We used a static analysis approach based on the source code of the bash command parser and `cron`'s crontab entry parser. It is possible to extract the scheduled commands from the crontab files, however, one must pay attention to the specified environment variables and keep track of which user's permissions and `HOME` setting will be applied during execution. We built an abstract syntax tree for each command in the crontab file, extracted the possible file paths from the command's parameters (including the invoked command's own file), and expanded all valid paths into absolute paths using the `PATH` and `HOME` variables.

However, this approach has serious issues which may result in the target list being incomplete. For example, if the crontab file includes a program which dynamically loads and executes another binary, our static analysis would miss that binary. What is more, our static analysis would need to be able to handle paths which are constructed dynamically (e.g., using loops). In order to overcome this challenge, either a "cron policy" is needed which limits how the users can define scheduled commands or the analysis should be extended with dynamic tools which run the commands in a controlled environment and extract the directories and files accessed during execution.

Another automated approach to constructing the list of file system entries to hash would be to identify all executable files on the file system. However, we cannot identify such files using the `x` permission flag, because file permissions are easy to change. Recognizing ELF binaries by reading the first few bytes for the magic

value is doable but the device may contain other types of executable files (e.g., script files). Differentiating between script files for different interpreters, such as Python and bash, and simple text files is challenging without executing them.

As our method is designed for embedded IoT devices, we assume a small local file system. Thanks to the small file system, we can expect to perform a thorough scan in a reasonably short amount of time. Therefore, we compiled a general list of file entries to hash which consists of many of the well-known top-level Linux directories, like `/bin`, `/lib` and `/etc`. There are directories with highly volatile contents, such as `/tmp, /var` and `/dev`, which we do not include in our scanning process. We do not rely on the common and default locations and naming schemes of the persistent components of known rootkits. We also do not rely on any malware signature databases, only on our own reference hashes. This way, we could potentially detect previously undiscovered rootkits, and are prepared for ones that randomize the names and locations of their components.

5.2 Recommended file system structuring

We now discuss a few basic structuring practices regarding the file systems mounted on the device which serve to greatly reduce the false positive rate of the file system scans. Our detection of persistent rootkit components on the disk was designed with the assumption that the device adheres to these practices.

As we use hashing to detect inconsistencies, it is necessary to separate the mounts containing static, unchanging files like program binaries and linked libraries from the ones containing dynamic, often-changing ones, such as log files. It is recommended to add the `noexec` flag to every mount apart from the root file system, which should be the one containing all programs and scripts executed during normal operation.

For files that change during runtime and cannot be moved from the root file system, we provide the option to be excluded from the hashing process. We recommend to use this option only for non-executable files for the following reason. Even if rootkits infect these files to achieve persistence, they must also stop their execution to evade our checks. Therefore, they need another component to execute the files in which they hide to re-infect the system. If the file in which rootkits hide is never executed, the rootkit cannot re-infect the system using this evasion method.

5.3 The hashing process

The hashing process is performed entirely in the Trusted Execution Environment (TEE). We take a predefined list of directories and/or file paths to be hashed and create a hashing context for each entry. In the case of directories, we recursively read the contents of all files from all subdirectories, and load them into the hashing context. We produce a single SHA-256 digest for every individual entry in the target list, as shown in Figure 3. The list of paths to be hashed is shuffled before each new scan to provide a degree of unpredictability to the hashing order; this serves

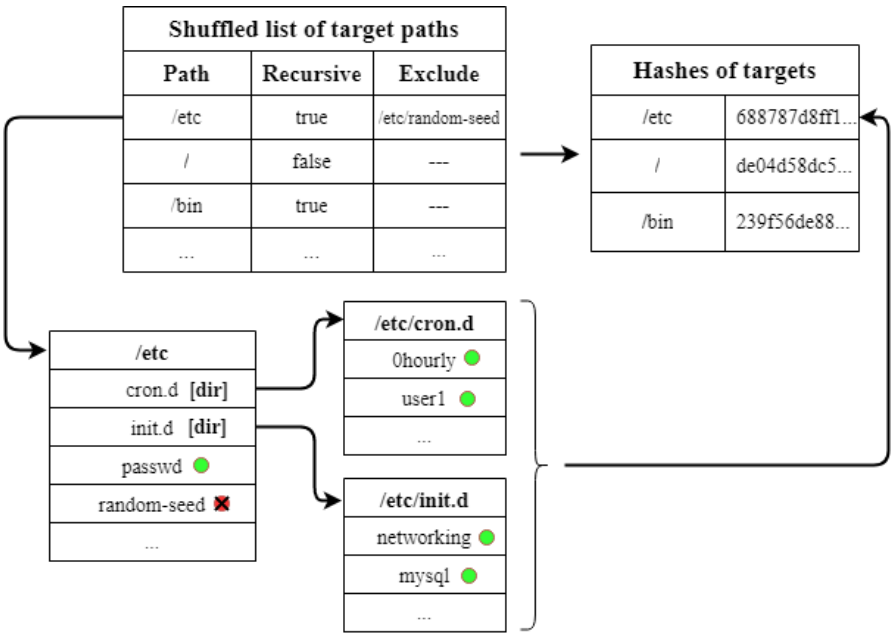

Figure 3: Each entry in the list of targets consists of an absolute path, a flag to indicate whether to traverse the subdirectories recursively or not, and a list of files to exclude from the hashing. For each target entry, a single SHA-256 digest is generated. If the recursive flag is set to false, only the files in the top level directory are calculated into the hash. If it is set to true, all subdirectories are recursively parsed until the maximum file path length of Linux, and all files not explicitly excluded are calculated into the hash.

as a secondary hardening technique against timing window attacks, supplementing the use of the fanotify API (see end of Subsection 4.3). This does not affect the digests themselves, since the order in which we read files inside directories remains unchanged.

The reference hashes to which we compare the computed ones are initialized on first launch, based on the state of the file system at that moment, and are placed into the TEE's trusted storage to protect them from tampering. Ideally, this first launch should occur before putting the target system into active use, but after the environment has been configured and all files necessary for operation have been written to the disk. The stored reference hashes become outdated after a system update or reconfiguration, and have to be replaced. In this paper, we do not discuss how this update can be solved, but an automated solution for this problem is part of our future plans.

The reference hashes are loaded from the trusted storage and compared to the computed hashes. Targets with mismatching hashes are noted and should be

examined further by the operators. It is recommended to keep a full reference hash set based on the initial state of the storage in order to find the exact cause of the mismatch. Pin-pointing mismatching locations would require keeping a 32-byte hash on the trusted storage for every entity present on the file system. This approach, however, requires much storage space, which is a drawback in the case of IoT devices, as they are often equipped with only a small flash memory for storage. As a result, a trade-off must be made between the amount of storage available and the precision with which mismatching locations can be identified.

5.4 Using the TEE's trusted storage

In this subsection, we briefly discuss a few important features of the TEE's trusted storage API that are necessary for protecting the authenticity and integrity of our reference hashes. It is important to mention that according to the specification of the trusted storage API [13], a single, designated Trusted Storage Space is provided for each trusted application. As a result, our stored reference hashes are accessible only by authorized TAs running on the same device in the same TEE as when the data was created.

The inner workings of the trusted storage are highly dependent on the TEE implementation, and it may not be entirely separated from the REE file system. Consequently, file modifications from a root privileged user account could be a valid concern. The trusted storage is expected to provide confidentiality and authenticity through the use of authenticated encryption. This protects our reference hashes against targeted modification and replay attacks. For additional protection and separation from the REE file system, the reference files can be stored on a Replay Protected Memory Block (RPMB) partition[25].

The attacker could attempt to delete the reference hashes from the trusted storage, because if this file cannot be found, we assume that our detection method is in the initialization phase and file hashes are computed. According to the specification, the trusted storage is protected from such attacks because a stored object should not be accessible from outside the TA that created it. How this functionality is achieved in practice is, again, highly dependent on the TEE implementation. In our chosen implementation (see next section), the TEE recognizes the data corruption and generates an alert.

6 Implementation details

We use the Open Portable Trusted Execution Environment[26] (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

[25] `https://lwn.net/Articles/682276/`, Last visited: Sep 18, 2020
[26] `https://www.op-tee.org`, Last visited: Sep 20, 2020

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`. OP-TEE-related library calls are handled by this daemon which 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 Subsections 6.1 and 6.2, respectively, and describe the checks we implemented in order to protect the REE components of OP-TEE in Subsection 6.3.

6.1 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`, which can be unfolded into three other macros, which in turn are also unfolded into other macros. All in all, we implemented 30 macros based on `__virt_to_phys_nodebug` to be able to translate Normal World virtual addresses to physical addresses. The Normal World virtual addresses are obtained from the `System.map` file of the compiled kernel, except for runqueues. Since they are not intended to be used by any kernel module, we had to implement a patch to dump these addresses into the boot log, and then we hard coded them into our TA.

In order to access the physical memory in which the Linux kernel resides, we implemented 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[27] by default.

6.2 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 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[28], 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 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

[27] `https://optee.readthedocs.io/en/latest/architecture/crypto.html`, Last visited: Sep 18, 2020

[28] `https://github.com/OP-TEE/optee\_os/pull/3962`, Last visited: Sep 18, 2020

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.

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

6.3 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 4.3, however, our implementation currently verifies only the `tee-supplicant` daemon and the CA.

7 Evaluation

We evaluated our implemented method on different types of rootkits. Our results in detecting kernel space rootkits are presented in Subsection 7.1. In Subsection 7.2, we present the results of detecting a custom rootkit's persistent components. The performance of our implementation is presented in Subsection 7.3.

7.1 Detecting different rootkits in kernel memory

In order to test our rootkit detection method, we wrote multiple kernel space rootkits. We also wanted to test it against real kernel space rootkits found in the wild,

but we were unable to find any that would work in our environment (the ones we found on github were written for older kernels or different architectures). However, we did manage to make user space rootkits work in our environment. In this subsection, we will describe these rootkits and the way our method was capable of detecting them.

Syscall hook: This test rootkit was implemented as a Linux kernel module. When it is loaded, it disables the write protection of the kernel's `.rodata` segment and replaces one of the function pointers in the system call table. Finally, it reenables the write protection. On unload, it restores the modified pointer. We are capable of detecting the corruption of the system call table by comparing the freshly computed hash to one created from the intact table.

Inline hook: This rootkit is similar to the previous one: in this case, we remove the write protection of the text segment of the kernel and tamper with the first few bytes of a chosen function, then restore it on unload. Again, this modification will change the hash of the text segment, and comparing it with the original one will reveal the presence of the rootkit.

DKOM I: This rootkit creates a backdoor process and attempts to hide it via removing it from the initial pid namespace's IDR. For a regular `ps`, the backdoor process is invisible, however it is possible to connect to it. Our solution detects the backdoor because it was not removed from the list of all the tasks, but it was missing from the output of `ps`.

DKOM II: In this test, we extended the functionality of the previous rootkit: we remove the backdoor process from the task list and the task tree as well, thus, our solution's only chance is to find it in any of the run queues. At this point, detection success depends on the rootkit's payload. In our case, it was a bind shell, which spends little time in run queues; it mostly waits, therefore, it is in wait queues. As currently our implemnentation does not check wait queues (the issue of wait queues will be explained in Section 8), we did not detect this rootkit. However, if the payload had been a computationally intensive task, such as a cryptocurrency miner, it would probably have been detected.

A HORSEPILL variant: HORSEPILL [19] 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[29], and starting the init process happens as follows: First, the Linux

[29] `https://buildroot.org/`, Last visited: Sep 18, 2020

kernel calls `/init`, a minimal shell script, and sets the 3 default file descriptors to `/dev/console`. Then, it invokes `/sbin/init`, which is a symlink to busybox[30]. 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.

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.

7.2 Detecting persistent components

In order to illustrate how our solution can detect the persistent components of rootkits, we created a proof of concept demonstration based on a potential vulnerability in an earlier version of our rootkit detection approach. Before we decided to use fanotify to prevent executables from starting during our checking process executed by our TA, we experimented with a technique that a sophisticated rootkit could use to avoid detection and stay persistent even after both the kernel memory and the file system checks have finished. The concept is the following:

1. Let's assume that the system has been compromised and the rootkit process is present in the memory. Let's also assume that the rootkit is able to predict when the next scan will happen and has an accurate idea of how long each scanning phase takes to finish.

2. Right before the scan is about to launch, the rootkit places a binary on the file system (which it has been storing on the heap), which, when executed,

[30] `https://busybox.net/`, Last visited: Sep 18, 2020

would load the rootkit back into memory and reinfect the system. It also runs `chmod` to make the file executable.

3. The rootkit then creates a crontab entry that is scheduled to execute the binary in the time window between the memory and the file system scan.

4. The rootkit process terminates at this point in order to avoid detection. Since it has no process loaded into memory and made no modifications to the kernel, the first phase of the scan finishes without finding any inconsistencies.

5. The scheduled task executes, the rootkit is loaded into memory. It immediately deletes the previously placed binary from the file system, and restores the modified crontab file.

6. Since every modification on the file system was reverted, the second phase of the scan finds no mismatching hashes.

This vulnerability was deemed very hard to exploit in practice due to the limited time window available for the attack. It is important to mention that the versions of cron based on Vixie-cron [31] were not designed to execute tasks in such an accurate manner either, it only "wakes up" at each minute. The task's execution could be delayed by an arbitrary number of seconds by using the sleep instruction in the scheduled task. Accurate timing would be hard to achieve reliably, considering that the interval between the scans is randomized, but we still considered it an architectural weakness, for which we needed a proper solution, not just hardening and mitigations.

With the current architecture, this vulnerability does not exist because we place the `FAN_OPEN_EXEC_PERM` fanotify mark on every file system. Fanotify would prevent the malicious crontab entry from executing the rootkit binary (more accurately, it would prevent the cron daemon from creating the bash session to run the shell command executing the binary), and the file system scan would detect the mismatches in the directory containing the rootkit binary.

As a demonstration, we prepared a small, statically linked C program which stores its own binary's bytes and is able to place and delete the binary on the file system at will. To simulate the perfectly timed scenario, instead of using cron, we used a small bash script to execute the binary on time, and disabled the memory checks.

The results were the following: The script tried to relaunch the program after it placed its binary in `/bin` and terminated, but fanotify prevented the relaunch. The script was waiting for permission to execute the binary, but the request was put on hold while the file system scan was running. The file system scan then detected the mismatch in `/bin`. Without the fanotify marks in place, the program would have been able to launch and delete the binary, and avoid detection.

[31] `https://directory.fsf.org/wiki/Vixie-cron`, Last visited: Sep 18, 2020

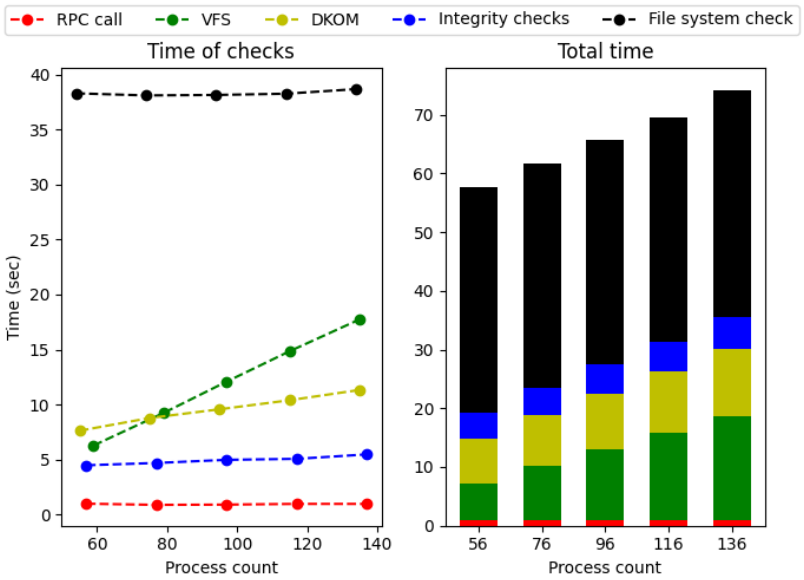

Figure 4: Execution time in seconds depending on the number of processes

7.3 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. Our implementation runs in a virtualized environment with 1057 MBs of RAM and 2 Cortex-A57 cores.

Since most 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 4 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 the TA performs normalization of its input. As expected, the number of running processes is irrelevant in this case.

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.

Finally, 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.

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 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.

8 Discussion and future work

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

8.1 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.

A source of another limitation is our decision of not checking the entire VFS layer, only the objects accessible from the superblock of `/proc`. As a result, the kernel-level integrity of other file systems are not checked, but we rely only on file hashing. A possible attack against the persistence check might be to hook the `read` function in the default file operations of an inode to show the original content of the file for every entity except for the one executing it.

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.

8.2 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 [7], therefore, rootkits often use them and try to hide their modules. Consequently, we disabled module support in the Linux kernel configuration, making rootkit installation more challenging. However, without module support, all necessary drivers must be compiled into the kernel, which is a functional restriction. In addition, there are other kernel resources worthy of being checked as well, such as components of the network stack.

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.

9 Conclusions

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.

References

[1] Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., and Zhou, Y. Understanding the Mirai botnet. In *USENIX Security Symposium*, August 2017. `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`, Last visited: Mar 19, 2021.

[2] Baliga, A, Chen, X, and Iftode, L. Paladin: Automated detection and containment of rootkit attacks. Technical report, Rutgers University Department of Computer Science, 2006. `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.74.9742&rep=rep1&type=pdf`, Last visited: Mar 19, 2021.

[3] Baliga, Arati, Ganapathy, Vinod, and Iftode, Liviu. Detecting kernel-level rootkits using data structure invariants. *IEEE Trans. Dependable Secur. Comput.*, 8(5):670–684, September 2011. DOI: `10.1109/TDSC.2010.38`.

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

[5] Blunden, William. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones and Bartlett Learning, 2012. ISBN: 9781449626372.

[6] Bravo, P. and García, D. Rootkits survey: A concealment story. Technical report, University of Oviedo, 2011. `https://pablo-bravo.com/files/survey.pdf`, Last visited: Sep 21, 2020.

[7] Bunten, A. Unix and Linux based rootkits techniques and countermeasures. In *16th Annual First Conference on Computer Security Incident Handling*. Budapest, Hungary, 2004. `https://www.first.org/resources/papers/conference2004/c17.pdf`, Last visited: Mar 18, 2021.

[8] Carbone, R. Malware memory analysis of the Jynx2 Linux rootkit. Technical report, Defence Research and Development Canada, 2014. `https://apps.dtic.mil/dtic/tr/fulltext/u2/1004190.pdf`, Last visited: Mar 19, 2021.

[9] Devik, S. Linux on-the-fly kernel patching without LKM. *Phrack Magazine*, 2001. `http://phrack.org/issues/58/7.html`, Last visited: Sep 21, 2020.

[10] Embleton, Sh., Sparks, Sh., and Zou, C. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013. DOI: `10.1002/sec.166`.

[11] Espensen, M. and Hoej, N. Rootkits: Out of sight, out of mind. Technical report, University of Copenhagen, 2014. https://github.com/NinnOgTonic/Out-of-Sight-Out-of-Mind-Rootkit, Last visited: Sep 16, 2020.

[12] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. 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. https://www.usenix.org/legacy/events/hotos07/tech/full_papers/garfinkel/garfinkel.pdf, Last visited: Mar 19, 2021.

[13] GlobalPlatform Technology. TEE system architecture – version 1.2. Technical report, 2018. http://globalplatform.org/specs-library/?filter-committee=tee, Last visited: Sep 19, 2020.

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

[15] Heasman, J. Implementing and detecting an ACPI BIOS rootkit. *Black Hat Europe*, 2006. https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-Heasman.pdf, Last visited: Mar 18, 2021.

[16] Herwig, S., Harvey, K., Hughey, G., Roberts, R., and Levin, D. Measurement and analysis of Hajime, a peer-to-peer IoT botnet. In *Network and Distributed Systems Security (NDSS) Symposium*, 2019. DOI: 10.14722/ndss.2019.23488.

[17] Hudson, T., Kovah, X., and Kallenberg, C. Thunderstrike 2: Sith Strike. *Black Hat USA Briefings*, 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Hudson-Thunderstrike-2-Sith-Strike.pdf, Last visited: Mar 19, 2021.

[18] Hudson, T. and Rudolph, L. Thunderstrike: EFI firmware bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, pages 1–10, 2015. DOI: 10.1145/2757667.2757673.

[19] Leibowitz, M. Horse Pill: A new kind of Linux rootkit. In *Black Hat USA*, 2016. https://www.blackhat.com/docs/us-16/materials/us-16-Leibowitz-Horse-Pill-A-New-Type-Of-Linux-Rootkit.pdf, Last visited: Mar 19, 2021.

[20] Petroni, Nick L., Fraser, Timothy, Molina, Jesus, and Arbaugh, William A. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, page 13, USA, 2004. USENIX Association. https://www.usenix.org/legacy/publications/library/

proceedings/sec04/tech/full_papers/petroni/petroni.pdf, Last visited: Mar 19, 2021.

[21] Rudd, E. M., Rozsa, A., Günther, M., and Boult, T. E. A survey of stealth malware attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys Tutorials*, 19(2):1145–1172, 2017. DOI: 10.1109/COMST.2016.2636078.

[22] Shah, A. and Giffin, J. Analysis of rootkits: Attack approaches and detection mechanisms. Technical report, Georgia Institute of Technology, 2008. http://beefchunk.com/documentation/security/RootkitsReport.pdf, Last visited: Mar 19, 2021.

[23] The MITRE Corporation. MITRE ATT&CK Tactics: TA0003 - Persistence, 2019. https://attack.mitre.org/tactics/TA0003/, Last visited: Mar 19, 2021.

[24] Todd, A., Benson, J., Peterson, G., Franz, T., Stevens, M., and Raines, R. Analysis of tools for detecting rootkits and hidden processes. In Craiger, Philip and Shenoi, Sujeet, editors, *Advances in Digital Forensics III*, pages 89–105, New York, NY, 2007. Springer New York. DOI: 10.1007/978-0-387-73742-3_6.

[25] Vervier, P.-A. and Shen, Y. Before toasters rise up: A view into the emerging IoT threat landscape. In *IoT Security Foundation Conference*, 2018. DOI: 10.1007/978-3-030-00470-5_26.

[26] Wang, Y., Beck, D., Vo, B., Roussev, R., and Verbowski, C. Detecting stealth software with strider ghostbuster. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 368–377, 2005. DOI: 10.1109/DSN.2005.39.

[27] Wichmann, Rainer. File integrity checkers: A comparison of several host/file integrity monitoring programs, 2009. https://www.la-samhna.de/library/scanners.html, Last visited: Mar 3, 2021.

[28] 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. DOI: 10.1109/CSSS.2011.5974667.