Versatile Endpoint Storage Security with Trusted Integrity Modules

Javier González
Philippe Bonnet

IT University Technical Report Series **TR-2014-TR 2014-173**

ISSN 1600–6100 **February 2014**

ISSN 1600–6100

ISBN 978-87-7949311-7

Copies may be obtained by contacting:

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S
Denmark

Telephone: +45 72 18 50 00
Telefax: +45 72 18 50 01
Web `www.itu.dk`

Versatile Endpoint Storage Security with Trusted Integrity Modules

Abstract

The immersion of personal devices throughout our daily life is changing the nature of computer security: It is getting personal. Many people are now concerned about the loss of privacy, the financial consequences or even the personal risks that could result from device theft or hacker attacks. To counter such threats, and to remain trustworthy, personal devices should enforce storage security. State-of-the-art storage security solutions rely on hardware protected encryption. They cannot be deployed, as such, on personal devices either because they require additional hardware (e.g., NetApp's SafeNet), or because they are constrained to specific hardware/software combinations (e.g., McAfee's DeepSafe). In this paper, we propose a solution for personal devices equipped with a Trusted Execution Environment and a Secure Element. We propose Trusted Integrity Modules, separated by hardware from the operating system and applications, that guarantee the durability, confidentiality and integrity of a configurable subset of the filesystem data and meta-data. While, we detail our design with the Linux virtual file system, we expect that our results can be applied to a range of different file systems. As Trusted Execution Environments also become available in Cloud environments, we envisage that Trusted Integrity Modules could constitute the solution of choice for endpoint storage security on both clients and servers.

1 Introduction

The advent of secure hardware embedded in all forms of personal devices, at the edges of the Internet, constitutes a sea change for computer security [1]. They provide an avenue to tackle the well documented problems related to leakiness and creepiness [14], loss of privacy [3] or simply surveillance. In particular, they should enable storage security, which has so far been reserved to server environments.

In [12], Ganger et al. proposed host-based intrusion detection systems (HIDS). The goal there was to detect intrusions in a system by observing sequences of system calls [7], patterns [11] or storage activity [12]. The main issue for host intrusion detection is that a smart attacker could feed the HIDS component with false information about the system and therefore bridge its security. Approaches such as [12] where the storage activity is monitored by looking at system primitives allow to decouple the OS processes from the processes running on the storage device. Illegitimate accesses can be detected even when the OS is compromised. This approach assumes however that the storage system is separated from the OS and cannot be compromised. In [16] Wurster et al. take this approach further, monitoring the file system while storage and OS run in the same physical device. While they present an interesting modification of the Linux's VFS to mark monitored files, the integrity mechanisms run in a demon in user space. Thus, they assume that the the user of the system is trusted not to be malicious. All these assumptions are a leap of faith; attackers can use widely spread rootkits to obtain root control of a system. If devices do not offer a hardware supported security perimeter, these attacks can be leveraged remotely without the user's knowledge, or published on the Internet as device class attacks.

Existing storage security solutions thus rely on hardware protected encryption. However solutions such as NetApp's SafeNet[1] require dedicated hardware, while McAfee's DeepSafe[2] is restricted, which relies on Trusted Platform Module (TPM) is restricted to a specific hardware (Intel-based processor equipped with TPM) and software (Windows) combination. The latter form of storage security is effective, but rigid and it restricts the user's ability to configure her computer [2].

[1] `http://www.netapp.com/us/products/storage-security-systems/storagesecure-encryption/`
[2] `http://www.mcafee.com/us/solutions/mcafee-deepsafe.aspx`

In contrast, we argue that Trusted Execution Environment (TEE) equipped with a Secure Element (SE, e.g., tamper resistant smart card) are ideally suited to support versatile storage security on personal devices. First, TEE-Based personal devices can readily implement trusted storage. While the TEE provides the runtime security perimeter to process and encrypt sensitive data (secure area), the SE provides persistency. Data and metadata are stored and replicated in untrusted storage, and encryption keys reside in the tamper-resistant unit. We detailed in [5] the design and implementation of such a Trusted Storage Module. Second, and this is the focus of this paper, TEE and SE, can be leveraged to guarantee the durability, confidentiality and integrity of a configurable subset of the filesystem without *a priori* restrictions on the nature of this file system. But, what are the core mechanisms that should be embedded in the TEE to provide storage security? Can they be extended to provide advanced intrusion detection services? How to gracefully integrate these mechanisms with the file systems readily available on personal devices equipped with Android or iOS? What is the impact on performance?

We note that many smartphones, tablets or set-top box computers are already equipped with ARM TrustZone that provides a Trusted Execution Environment. Hardware platforms and Software Development Kits are readily available from companies such as Trustonic, Xilinx or SierraWare. We described the platform we use for our experimentations in [4].

In this paper, we describe the design of a Trusted Integrity Module (TIM) that is responsible for enforcing integrity for a portion of the file system in the secure area provided by a Trusted Execution Environment. We detail its integration with the Linux Virtual File System, which is a good representative of file systems available on the current generation of personal devices. Finally, we show how simple usage control mechanisms emerge from the TIM design.

2 TIM Architecture

In the Linux kernel the principal file system data structures are the directory entry (dentry) and the inode [15] [9]. The inode structure contains most of the data and metadata concerning a file, as well as pointers to the actual blocks on disk. The dentry contains information concerning the specific instance of a file in a given path. The dentry points to one or several inodes. These structures describe each file perfectly, from its size or permissions to the date of last modification. Each time a file is accessed using a file object, these attributes are modified to describe the file's current status.

Our proposal is to mark some files as **secure**. When files are marked secure, a cryptographic hash of the file and of its metadata (i.e., inode and dentry) are kept in the secure area. Operations on this file are executed through the secure area and logged there. Such a Log contains the path to the file, the id of the process modifying it and time. Logs and hashes are stored in trusted storage using a trusted storage module (TSM) [5], i.e., data is encrypted and stored as secure files and the encryption keys are stored in a tamper-resistant unit, which is only accessible from the secure area. Secure files can then be checked for integrity [8]: if the hash of a file does not correspond with the hash stored in the secure area it means that either the file has been corrupted or that the integrity functions in the secure area have been bypassed, which can be considered as an attack. Simple usage control policies emerge from this mechanism, since users can make decisions based on regular integrity checks. They can come back to an old non-corrupted version of the file, they can log the anomaly and continue the operation for later audit, or they can lock the system down for later inspection. Mandatory Access Control (MAC) policies are also simple to implement. They can be executed as tasklets in the secure area that execute prior the file operation is carried out. Also, when a file is modified using the secure area, logs cannot be modified unless the tamper-resistant unit is bypassed.

Integrity checks are not reserved for secure files. We also mark some files as **monitored**. System files that are subject to be modified by rootkit attacks should for example be monitored. Such files are stored in clear. They are used by the rich area, but they are monitored from the secure are to detect modifications that can lead to security breaches. Examples of files targeted by attackers are: system commands that can help administrators to detect an attacker (e.g., ps, ls, wc), initialization scripts (e.g., init, rc), and system libraries (e.g., libc, libssh, libssl). To recap: secure files are always encrypted by the TSM, monitored files are stored in clear, for both files the metadata are handled from the secure area. This distinction is transparent to the rich area. In the rest of the paper, we refer to secure files as those files whose metadata is handled in the secure area.

The distinction between rich and secure files is implemented directly in the file system. The inode structure is modified to add an environment owner, *eid_t*, that determines to which environment actually manages the file's metadata; just as *uid_t* and *gid_t* determine user and group owners. When the System-Call Interface (SCI) catches a system call

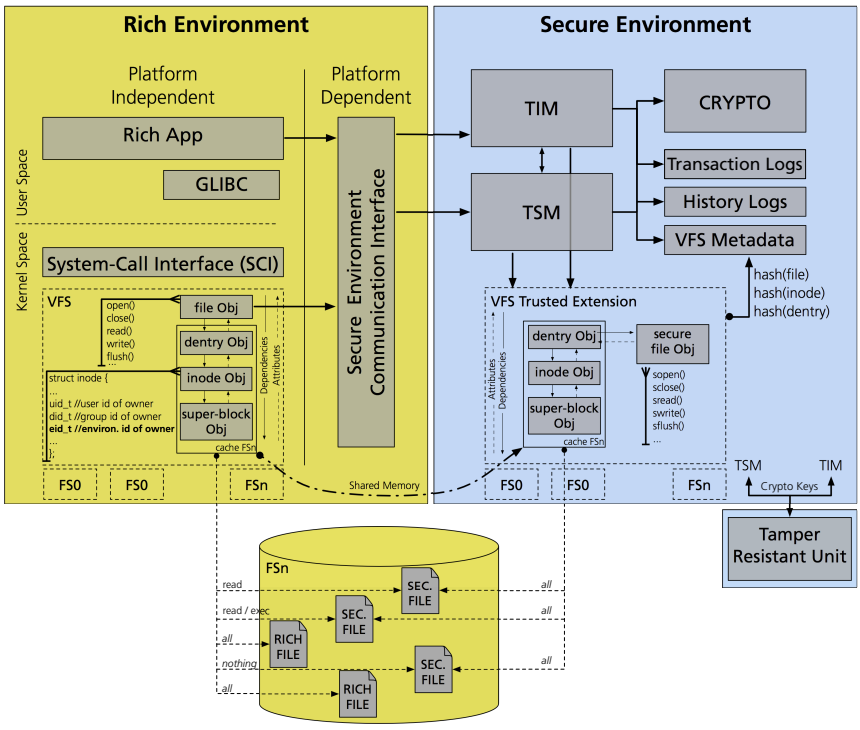

Figure 1: TIM Architecture. Yellow: Rich environment and untrusted components. Blue: Secure environment and trusted components. The inode structure is extended with the environment owner id (*eid_t*).

that requests a file operation, the system call is mapped to the Virtual File System (VFS) and a file object is created [9]. Before carrying out any operation, the file object checks the properties of the inode and dentry objects. If the object belongs to the rich area, the VFS continues with its normal operation (i.e., system operation in VFS file system's method $\rightarrow$ physical media). However, when the file belongs to the secure area, the system call is mapped to a VFS trusted extension implemented in the secure area where a secure version of the system operation is implemented. By moving the logic associated to secure file operations to the secure area, we guarantee that the operations carried out on secure files are logged in trusted storage, preventing attackers from compromising the hash creation, storage and checking. We refer to the module carrying out the integrity operations in the secure area as Trusted Integrity Module (TIM). Figure 1 depicts the architecture.

2.1 VFS Trusted Extension

Current file systems support the implementation of complex access control policies, specially if extended with Extended Attributes (AEs). Complex MACs have been implemented by extending traditional Unix permissions using AEs [6]. However, if an attacker obtains root access to a device, all security implemented in software is compromised and can easily be removed without physical access to the device itself. As a consequence, attackers can bypass the security, modify the system to allow their software to survive a reboot, and modify logs and system files to hide their processes from system administrators. Attackers use for this legitimate VFS primitives. By complementing the VFS with a trusted extension in the secure area, it is possible to store metadata on critical parts of a FS in trusted storage, therefore providing a first nod of trust to evaluated the integrity of the FS.

The VFS Trusted Extension implements a secure file object in the secure area. Since TEEs support sharing memory between rich applications and secure tasks, the VFS Trusted Extension can make use of all the VFS structures implemented in rich area, thus simplifying its design and minimizing code replication. In this way, operations over a secure file are carried out by a secure file object. This allows to add enforcement (*a priori*) and audit (*a posterior*)

policies, as well as checks to secure file operations. How restrictive these policies are, is decided by the administrator of the system. Usage control policies emerging from this design will be discussed in Section 3. In the following subsections we will discuss which structures are used by the VFS Trusted Extension to ensure the integrity of the secure files in the FS.

2.2 VFS Metadata

In order to guarantee the integrity of secure files, TIM stores the hashes of the contents and the metadata describing a secure file, i.e., the hash of the secure file and the hashes of the inode and dentry pointing to the secure file. Whenever a secure file operation is executed, the hashes are recalculated. This allows to evaluate both accesses entailing modifications to the file itself (i.e., write), and accesses without modifications (e.g., read, change ownership). All hashes are stored in trusted storage using the TSM, thus protected by cryptographic keys and tamper-resistant storage.

At any moment in time, the hashes of a secure file and its metadata must match the hashes handled by the secure area. This can be tricky due to the VFS and the FS not providing transaction properties, such as database system do; upon a failure, the contents of the *dcache* and the *icache* could be lost, therefore resulting in a mismatch between the data and metadata producing the hashes in the secure area, and the data and metadata present in disk. We will discuss this further in Subsection 2.4.

2.3 History Logs

One of the main problems that system administrators face when dealing with a system that has been attacked is figuring out how the attacker got inside, when, and what she modified. Since most attacks are perpetrated using root privileges, system logs containing this kind of information can be modified. Keeping history logs in the secure area is a way to prevent attackers to clean their tracks. When a secure file object is created, all the actions carried out in that file are logged in the secure area. The logs record: file, file inode, file dentry and pid of the process creating the file object and its owner. If an attacker supplants an identity that is allowed to make changes in a file, she will succeed making those changes. However, history logs in the secure area are neither visible nor accessible from the rich area, not even for root. Indeed, history logs are stored in trusted storage by the TSM. In this way, attacks would always be logged and available for later audit, helping system administrators to learn from successful attacks to improve their systems. Also, since TEEs make it possible to run complex authorized software on the secure area, a machine learning algorithm could be used to learn from successful attacks. Evaluating threats and learning from them to prevent future attacks from the secure environment is work in progress.

2.4 Transaction Logs

Since file images and metadata are both stored in disk and in RAM (*dcache* and *icache*), unexpected failures can cause inconsistencies. In order to avoid this problem, there are two things we need to enforce for secure files: (i) the hashes of metadata on disk must match the file image on disk, and (ii) the hashes of metadata in RAM must match the file image in RAM. RAM images and disk images may differ, as long as the image on disk remains consistent in event of a cache failure.

Whenever a *fsync* occurs, both metadata and disk images must be updated with the cache contents atomically. For this we can rely on the same mechanisms that InnoDB uses in the *doublewrite buffer* to deal with partial page writes [13]: a staging buffer on disk and on a log. Atomicity of writes to the log ensure global atomicity. In our case, all data, metadata and hashes are written to a staging buffer on disk and the operation is logged. A log sequence number (LSN) is held in order to bind log record and writes in the staged buffer. In case of a failure midway through the write on the staging buffer, it is like nothing was written (i.e., no log record is written) and the contents of the staged buffer can be discarded for the given LSN. In case of failure after the log record is written, the write operation can resume from the staging buffer until it completes in place. A new log record is then entered that marks the completion of the write. The contents of the staged buffer can then be discarded for the given LSN. Finally, If the system fails after this second log record is written and before the contents of the staged buffer is cleared, then the write can be re-executed from the staged buffer.

Transaction logs are treated like history logs, indeed history logs are in part created from successful transaction operations. Transaction logs are also stored using trusted storage by means of the TSM.

3 Embedded Usage Control

The design of TIM and its components (VFS Trusted Extension, VFS Metadata and the Logs) provides, in its simplest form, a way to keep track of the operations affecting secure files. However, simple usage control features emerge naturally. By combining the VFS metadata and the history logs stored in the secure area, TIM provides system administrators with three powerful storage-based tools: (i) enforcement: evaluation of policies prior execution file operation; (ii) embedded behavior: possibility to embed actions in file primitives; and (iii) audit: post-use evaluation of file system activity.

Enforcement. The fact that the VFS Trusted Extension implements a secure file object in the secure area provides a simple method to attach rigid access control policies to secure file operations. While some interesting work has been proposed to provide complex and customized MAC using extended attributes (EAs) and access control lists (ACLs) [6], critical system files need a simple access control policy: while in normal operation, critical system files should just not be modified, not by root, not by anybody. Secure file system primitives running in a TEE are the perfect place to implement these special access control policies. This is not an alternative to ACLs, but a way to complement them.

Embedded behavior VFS metadata allows to detect inconsistencies in secure files. When the hash of a file does not correspond with the hash handled by the secure area, TIM is certain that the file is corrupted, either because of a file system malfunction or as a result of a successful attack. System administrators can then attach actions to this inconsistencies in form of secure tasks in TIM. These actions are dependent on the requirements of the system: a system prioritizing accessibility could be configured to take conservative actions such as notifying the incidence, saving the corrupted file and its metadata for later audit, and loading an uncorrupted version of the file from a backup; a system prioritizing security could be configured to take more drastic measures such as notifying the incidence, stopping all communications, backing up the current state, and shutting the system down. Since the usage control actions are embedded on file system primitives, the decision making process does not depend on complex system evaluations. This results on the overhead introduced by the evaluation of storage-based usage control policies being minimal, reducing response time and consequently the impact of a successful attack.

Audit. History logs allow to evaluate file system activity. While the main purpose of logs is that humans or automated processes analyze them to gain knowledge of the usage of the system, actions can also be attached to the operations appending records to the log. This allows to embed storage-based usage control policies right into file system primitives. While evaluating the policies requires a policy engine running on the side of the TIM, attaching file operations to logs as if they were atomic operations provides a simple and effective way to feed the policy engine. Defining a usage control model and implementing it is work in progress. However, given the tight relationship between usage of a system and its secondary storage, we consider TIM as a fundamental part of a trusted usage control model embedded in TEE-enable devices.

All operations on secure files are executed inside a TEE, and all metadata and logs concerning secure files are stored using trusted storage. As a consequence, successful attacks require being more intricate than just gaining root access to a system, and would require physical access to the device itself. This constitutes and extra prevention layer for remote attacks over the Internet, and class attacks targeting specific devices.

4 Security Analysis

The changes introduced by the TIM in the rich area are minimal: an environmental flag (*eid_t*) in the inode structure, and a system call mapping to the VFS Trusted Extension. These changes are not visible from user space, which means that even if an attacker gains root privileges, she would not be able to determine which files are secure and which are not without making changes in the kernel. What is more, since the VFS extensions, the logs and the VFS metadata are stored in trusted storage, an attacker would have to either bypass the TEE gatekeeper, force the TSM communication interface, or physically attack the tamper-resistant unit in order to obtain them [5]. Under this circumstances, an attacker could design three attacks to bypass the TIM and modify a secure file: (i) software attacks against the rich VFS modifications

in the kernel; (ii) software attacks against the TIM rich - secure communication interface; and (iii) hardware attacks against the tamper resistant unit storing the TSM keys, and TrustZone itself.

An attacker with root privileges could make modifications in the kernel in order to prevent triggering TIM. To achieve this, she would have to either modify the VFS structures in the kernel so that a secure file is not marked as secure, or unmap the VFS Trusted Extension system call. If the attacks succeeds, the attacker would be able to access files readable from the rich area (i.e., not encrypted), while avoiding the integrity checks and logs in the secure area. However, when an integrity check occurs and TIM is executed, all modifications to secure files would be identified, the files marked as corrupted, and the actions specified by the system administrator carried out. Even if the attacker succeeds in disabling all communications with the secure area without users or system administrators realizing it, all sensitive information would still be protected by trusted storage in the TSM. This is, files containing sensitive being encrypted, and the keys being stored in a tamper-resistant unit.

Attacks to the TIM rich - secure interface could be launched in order to gain control of TIM and make modifications of the history logs and hashes. Even assuming that the attacker is able to take control of the interface, the integrity checks, usage control policies and the logging processes are embedded in secure file operations. This means that an attacker would have to replace TIM primitives interacting with VFS Trusted Extension. These kind of changes would need to recompile the kernel in the secure area, which is unrealistic for a software attack.

Finally, TrustZone is not tamper resistant, which means that a hardware attack is feasible. Bypassing the hardware would mean the collapse of the TIM security. Regarding the encryption keys, the upper boundary to which they can be safe is again, the level of tamper resistance of the hardware in which they are stored. Since the keys are stored in a secure element, hardware attacks need to be very advanced to succeed. The assumption should be that with enough time and money any security system can be bypassed. Nonetheless, we consider that client devices such as smart phones, tablets or laptops offer an intrinsic level of tamper evidence, giving users the chance to minimize the effects of a successful hardware attack.

In case of a failure in the secure environment, the effect on rich applications is the same as a system call failure. If the TSM crashes, the call issued inside the secure environment will not be catched and an error will be returned; if the whole TEE crashes, the system call issued by the gatekeeper in order to switch contexts will not be catched either, and a similar error will be returned.

5 Conclusion

We presented the design of TIM, Trusted Integrity Module, and its components VFS Trusted Extension, VFS Metadata and the Logs, that together with the underlying Trusted Storage Module, enable flexible storage security on Posix-like file systems running on platforms equipped with Trusted Execution Environments and Secure Elements. TIM is thus well suited for a wide range of personal devices. As trusted execution environment are appearing in cloud environments[3], TIM might become relevant for endpoint security on both clients and servers. Much work remains to be done before TIM is ready for prime-time though. We must evaluate its impact on performance and thoroughly explore the design space for log management (Ori's grafts might serve as an inspiration there [10]), file system integration and usage control. These are topic for future research.

References

[1] N. Anciaux, P. Bonnet, L. Bouganim, B. Nguyen, I. Sandu Popa, and P. Pucheral. Trusted cells: A sea change for personal data services. *CIDR*, 2013.

[2] C. Doctorow. Lockdown, the coming war on general-purpose computing.

[3] A. P. et al. Personal data: The emergence of a new asset class. World Economic Forum. January 2011.

[4] J. González and P. Bonnet. Towards an open framework leveraging a trusted execution environment. In *Cyberspace Safety and Security*. Springer, 2013.

[3] ARMv8 64 bits server support TrustZone.

[5] J. González and P. Bonnet. Tee-based trusted storage. Technical report, IT University of Copenhagen, 2014.

[6] A. Grünbacher. Posix access control lists on linux. *USENIX*, 2003.

[7] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 1998.

[8] G. H. Kim and E. H. Spafford. In *Proceedings of the 2Nd ACM Conference on Computer and Communications Security*, New York, NY, USA, 1994.

[9] R. Love. *Linux Kernel Development (3rd Edition)*. Developer's Library, 2010.

[10] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières. Replication, history, and grafting in the ori file system. In *SOSP*, New York, NY, USA, 2013.

[11] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. *Neuronal Networks, IJCNN'02*, 2002.

[12] A. G. Pennington, J. L. Griffin, J. S. Bucy, J. D. Strunk, and G. R. Ganger. Storage-based intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 13(4):30:1–30:27, Dec. 2010.

[13] B. Schwartz, P. Zaitsev, V. Tkachenko, J. D. Zawodny, A. Lentz, and D. J. Balling. *High Performance MySQL(2nd ed.)*. O'Reilly, 2008.

[14] I. Shklovski, S. Mainwaring, H. Skuladottir, and H. Borgthorsson. Of leakiness & creepiness in app space: User perceptions of privacy and mobile app use. In *CHI 2014*.

[15] W. Stallings. *Operating Systems: Internals and Design Principles (7th Edition)*. Prentice Hall, 2011.

[16] G. Wurster and P. C. van Oorschot. A control point for reducing root abuse of file-system privileges. In *CCS*, New York, NY, USA, 2010. ACM.