Patterns for Secure Boot and Secure Storage in Computer Systems

Hans Löhr, Ahmad-Reza Sadeghi, Marcel Winandy
Horst Görtz Institute for IT Security, Ruhr-University Bochum, Germany
{*hans.loehr,ahmad.sadeghi,marcel.winandy*}*@trust.rub.de*

***Abstract*—Trusted Computing aims at enhancing the security of IT systems by using a combination of trusted hardware and software components to provide security guarantees. This includes system state integrity and the secure link between the software and hardware of a computing platform. Although security patterns exist for operating system security, access control, and authentication, there is still none of Trusted Computing aspects. In this paper, we introduce security patterns for secure boot and for secure storage, which are important basic Trusted Computing concepts. Secure boot is at the heart of most security solutions and secure storage is fundamental for application-level security: it ensures that the integrity of software is verified before accessing stored data. Our paper aims at complementing existing system security patterns by presenting the common patterns underlying the different realizations of secure boot and secure storage.**

***Keywords*-security patterns; trusted computing; secure boot; secure storage;**

I. Introduction

The literature on security patterns includes numerous patterns related to operating systems, e.g., concerning authentication and access control. However, until now, Trusted Computing (TC) has received very little attention from the pattern community. TC aims to employ a combination of hardware security mechanisms and software to address security problems that cannot be solved by software alone. Particularly relevant among these are security threats related to malicious software, such as Trojan horses and viruses. By now, there is not only a tremendous amount of research in this field, but TC concepts can also be found in a wide variety of products, ranging from embedded devices, such as mobile phones, to servers equipped with expensive tamper-resistant secure co-processors. The best-known approach to TC is based on the specifications released by the Trusted Computing Group (TCG) [1].

In this paper, we present the patterns underlying two fundamental TC concepts: *secure boot* and *secure storage*.

Secure boot guarantees that violations of integrity properties of the software stack that is booted on a platform can be prevented, i.e., software that violates the integrity properties cannot be loaded. A variant of this pattern, termed *authenticated boot*, does not prevent software from being loaded, but allows reliable verification of the load-time integrity of the software that has been booted later on. Secure boot is a building block at the heart of many TC-based solutions (including implementations of secure storage).

Secure storage is a crucial application-level requirement in many scenarios. Simple encryption is often not enough to protect sensitive data: it must also be ensured that an attacker cannot obtain the decryption key. Secure storage solves this issue by using hardware (and software) to enforce access restrictions on the stored data. Before access is granted to an application, the integrity of the software is verified.

Secure storage and secure boot are essential concepts for TC systems. For instance, a Common Criteria protection profile for security kernels with TC support has been evaluated and certified recently [2], which also includes secure boot and secure storage. The security patterns described here could be helpful to implement these features for security kernels that aim to comply with this protection profile.

This paper describes the common pattern underlying various existing realizations of secure boot [3], [4], [1], [5], and of secure storage [4], [1], [5].

II. Secure Boot Pattern

Intent: This pattern addresses how to ensure that violations of integrity properties of the software stack that is booted on a platform can be either prevented (secure boot) or detected (authenticated boot).

A. Example

Consider a user who wants to use a computing device that was left unattended or that was used by another person before. How can the user be sure that the system software is in the intended operational state, i.e., that no critical component of the operating system or other software applications has been modified in a malicious or unauthorized way? Typically, a file integrity checker program can check the integrity of system and application files. However, any file integrity checker program must rely on trusted reference values and that those values have not been tampered with. Moreover, the user wants also to be sure that the file integrity checker itself is not tampered with or deactivated at all.

B. Context

Users of security-sensitive applications want to be sure about the operational integrity of their applications and execution environment. Unauthorized changes to the application code or the operating system may lead to unintentional program behavior or violation of security goals. Users trust the hardware, but they need a way to verify that the software loaded on this hardware has not been tampered with.

C. Problem

On conventional platforms, software can be manipulated or exchanged. Local users cannot verify if they are interacting with the "correct" software, and remote platforms cannot verify the software of their communication partner.

Before applications can be used on a computer system, the system has to be bootstrapped. Typically, the hardware starts by loading a piece of code (firmware, or BIOS on PCs), which in turn loads the bootloader from a pre-defined place from main storage (the disk drives). The bootloader loads the operating system kernel, and the operating system kernel loads system services, device drivers, and other applications.

At any stage of the bootstrap process, software components could have been exchanged or modified by another user or by malicious software that has been executed before.

The following forces have to be resolved:

- You want to ensure the integrity of the loaded software on the system, otherwise malicious software could run without being noticed.
- You want the computer system to always boot in a well-defined secure state. Otherwise, attackers could violate security goals by putting it into an insecure state.
- You want to allow modifications of the operating system or application binaries. Otherwise, software installation and updates would be problematic.

D. Solution

Based on the assumption that the hardware of the computer system is correct, the integrity of lower layer boot modules is checked and control is transferred to the next stage if and only if the integrity of that stage is valid. Hence, every stage is responsible for checking the integrity of the next stage. Integrity checking can be performed in different ways: two common methods are comparing hash values or verifying digital signatures. In the first case, hash values of program binaries are computed using a cryptographic hash function (e.g., SHA-1) and compared to reference values. If the computed value does not match the reference value, the binary has been modified. In the second case, each program binary is cryptographically signed by its vendor with a signature key that is only known to the vendor. The signature of the binary can be verified to check that it has not been modified since its signature generation.

If one stage detects an integrity violation, execution is stopped and the system halts. The sequence of these integrity checks builds the *chain of trust*. Thus, the user knows implicitly that the system has booted valid program code if it is running at all.

If the first module of such a sequence of integrity checks has already been modified in an unauthorized or malicious way, then the user cannot trust on subsequent integrity checks. The modified module could cheat or even skip any integrity checking. Therefore, the very first boot module is the *root of trust* for the whole chain of integrity measurements and needs to be protected against unauthorized modifications. The integrity verification data (hash reference values, or signature verification keys) must be protected, too.

To protect the initial boot module (and its verification data) and to reliably build the chain of trust, the root of trust is realized in hardware. Hardware is assumed to be more secure than software because it cannot be changed or read out as easily as software. Moreover, hardware security modules can be protected against various physical attacks – at least to some extend.

1) Structure: Figure 1 shows the elements of the Secure Boot pattern. The Root of Trust for Measurement is the first module in the bootstrap chain and realized and protected by hardware. A Bootstrap Module has a link to the next Bootstrap Module, which is "measured" for its integrity (typically, by computing a hash value) before control is transferred. Each Bootstrap Module has to maintain (and protect) its corresponding integrity verification data. The verification data of the Root of Trust module is also protected by the hardware, e.g., stored in protected memory registers.

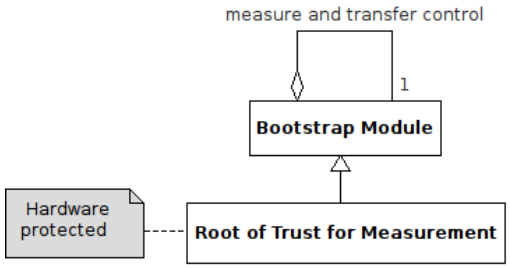

Figure 1. Elements of the Secure Boot pattern.

Usually, the last Bootstrap Module is the operating system, which can load several different applications and not only one. In addition, applications can also start other applications or load libraries. In this case, applications also have to measure the integrity of the corresponding components.

2) Dynamics: When the computer system is started, the Root of Trust for Measurement is executed. It loads and measures the program code of the subsequent Bootstrap Module, and verifies the integrity of this code. If this fails, the Root of Trust stops the execution and the system is halted. Otherwise, the Root of Trust transfers control to the subsequent Bootstrap Module.

The Bootstrap Module loads and measures the code of the next Bootstrap Module, and verifies the integrity of this code based on the verification data. If this fails, the system is halted, otherwise control is transferred to the next Bootstrap Module. This continues until the last module in the chain of trust has received control (typically, applications).

E. Example Resolved

Based on secure boot, the user can be sure that the correct system has been booted on the computer without

changes, because any modifications would have caused the boot process to abort. Moreover, it is not possible to boot a completely different system, because the boot chain starts with a root of trust protected by hardware.

F. Variants

Authenticated boot refers to an architecture that ensures that local or remote parties can verify properties of the software that has been booted. While secure boot interrupts the boot process if a modification of the loaded modules is detected, authenticated boot continues the boot process even in that case. However, any modification is detected and recorded securely for later inspection or verification.

Authenticated boot can be implemented using a hardware security module in the following way: The first module in the bootstrap chain is trusted by assumption (e.g., implemented in hardware), and measures the integrity of the next module. This integrity measurement is then stored in protected hardware registers. The measurements taken at load-time of the software can later be reported by the security module. For this, the module signs the stored values.

G. Known Uses

There are various implementations of secure boot:

- AEGIS [3] is a secure boot architecture for PCs, where a chain of trust is constructed at boot time by hashing components, and comparing the result with digitally signed reference values. An expansion card implements the necessary hardware root of trust for measurement.
- The Cell Broadband Engine processor [4] implements a secure boot mechanism to load application code into an isolated execution mode of one of its multiple processor cores. The Cell processor provides a hardware root of trust that verifies the integrity of the loaded code cryptographically. Only if the verification succeeds, the code is executed. Moreover, every time an application wants to use this isolated execution environment, it has to pass the secure boot process. Within the isolated mode, the application code is protected from software running on other cores and even from the operating system running on the supervisor core. The Cell Broadband Engine is used in IBM Cell blade servers and also in the Sony Playstation3 game console.
- For mobile devices, requirements for secure boot have been collected in Open Mobile Terminal Platform (OMTP) recommendations [5] that take into account different industrial standards and specifications, including specifications from the Open Mobile Alliance (OMA), the TCG, and the 3rd Generation Partnership Project (3GPP). The document describes the mechanisms for secure boot and authenticated boot in an abstract way, and hardware manufacturers implement them differently.

There is a well-known example for authenticated boot:

- The TCG specified authenticated boot based on the TPM [1]. The TPM contains special-purpose registers called Platform Configuration Registers (PCRs) that can be used to store hash values. These PCRs cannot be overwritten but only "extended" by computing hash values. Starting from the root of trust for measurement, all software modules in the boot chain of a TCG-enabled PC (BIOS, boot loader, operating system kernel, etc.) are first hashed, a PCR is extended accordingly, and then control is transferred. By this, the entire bootstrap chain is measured and recorded in PCRs. The TPM also offers a function `TPM_Quote`, where the recorded hash values are digitally signed and reported to the caller.

H. Consequences

The benefits from the secure boot pattern include:

- The software integrity state is verified at boot time, and only software that passed this verification is booted.
- With the variant authenticated boot, it is possible to boot any software, however, the integrity state of the software at boot time can be checked later.

The liabilities from the secure boot pattern include:

- Integrity verification data must be installed and updated (e.g., due to revocation) in a secure fashion (preserving integrity, authenticity, and freshness of the data).
- Software updates could be an issue, because after an update, the integrity state is changed (the software is modified). Hence, specific mechanisms for updates are needed to allow the system to boot properly afterwards.
- The integrity of modules during runtime needs to be ensured by additional mechanisms.
- The integrity verification of large modules (e.g., an entire OS) may be time consuming. Hence, the OS may require adaption to include integrity verification of its modules and applications.
- This pattern adds complexity and overhead. It needs to be coordinated with the other protection mechanisms.

I. Related Patterns

Boot Loader [6] describes the boot process as a sequence of single bootstrap stages. Each stage loads the image of the next one and transfers control after validation of the image according to certain requirements, typically verifying error correction checksums. In contrast, Secure Boot requires a secure verification of the integrity of the image in each stage, e.g., based on cryptographic hash functions. In addition, Secure Boot defines a root of trust for measurement and requires its protection by trusted hardware to establish a chain of trust throughout all stages.

Authenticator [7] verifies the identity of a subject and creates a proof of identity for later use, e.g., in access control decisions. However, it is typically intended to authenticate users and it does not include a protection for the Authenticator itself. Hence, it cannot provide a chain of trust. But

Authenticator can be combined with Secure Boot to extend the proof of identity to the underlying system components.

In [8], the authors introduce patterns for TPM usage. Their patterns concern the communication of software with the TPM, whereas the Secure Boot pattern describes more abstract concepts that can be realized based on TPMs, or based on alternatives, such as secure co-processors.

III. Secure Storage Pattern

Intent: Secure storage provides confidentiality and integrity for stored data, and additionally enforces access restrictions on entities that want to access data. In particular, the secure storage can verify the integrity of software components, and it grants access to the data only to authorized, unmodified components that meet these restrictions.

A. Example

Consider the problem of storing passwords (e.g., for web services) securely on a computer. Users want to protect their passwords, such that attackers who might install malicious software on the computer (e.g., phishers using malware) cannot access them. In this case, simply encrypting the passwords with a passphrase does not help: The software installed by the attacker might look exactly like the legitimate software (i.e., the password manager they normally use) to users, so they enter their passwords. Thus, the attacker obtains the passphrase, and can decrypt and steal all passwords. We need to store passwords in such a way that only the legitimate password manager can access them.

B. Context

You need to provide storage that protects the confidentiality and integrity of stored data, and which verifies the integrity of the software before granting access to the data. You trust the hardware, but you need to be able to verify that only authorized, unmodified software can access data stored in the secure storage.

C. Problem

Cryptographic techniques exist to protect the confidentiality and integrity of data, e.g., encryption and digital signatures. However, the secret keys need to be protected against unauthorized usage. Operating system access controls are not sufficient because the access control component could be manipulated or replaced by an attacker.

The following forces have to be resolved:

- You need to protect the confidentiality and integrity of data, even when the system is not running. Otherwise, an attacker could read or modify protected data.
- You need to protect secret cryptographic keys from unauthorized access and usage. Otherwise, an attacker could use the keys, e.g., to decrypt confidential data.
- You want to allow modifications of the operating system or application binaries. Otherwise, software installation and updates would be problematic.

D. Solution

A Root Key is used to encrypt and decrypt data and other keys (which in turn can protect data or other keys). The usage of the Root Key is controlled by a Root Key Control component. Root Key and Root Key Control are both protected by trusted hardware, i.e., the secret part of the Root Key never leaves the hardware, and Root Key Control cannot be manipulated or replaced by users or other software programs. Root Key Control verifies the integrity of Applications and their Execution Environment before it performs cryptographic operations on behalf of an Application.

1) Structure: Figure 2 shows the elements of the Secure Storage pattern. Root Key Control has solely access to the Root Key, and both are protected by hardware. The Root Key can protect an arbitrary amount of Data. A special case of Data are Application Keys, which Applications can use to protect their application-specific data. The Root Key Control maintains a mapping of what Data can be accessed by which Application. It uses integrity verification data (hash reference values, or digital signatures) to verify the integrity of Applications and their Execution Environment.

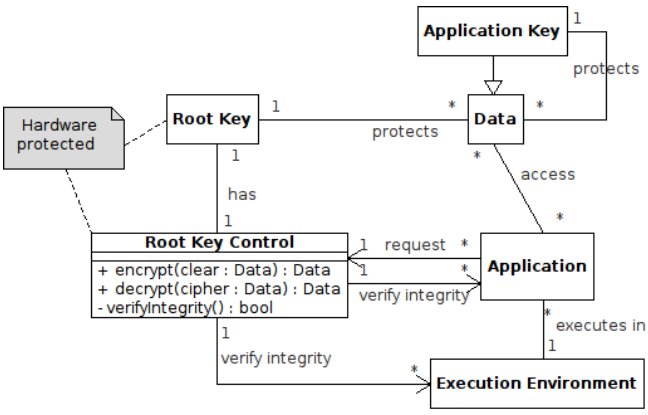

Figure 2. Elements of the Secure Storage pattern.

2) Dynamics: When an Application requests to access Data, the Root Key Control verifies first the integrity of the Application and of the software components of the Execution Environment the Application is executed in (typically, the operating system components). Only if the integrity verification succeeds, Root Key Control uses the Root Key to decrypt (or encrypt) the Data. Note that the Root Key is never passed to an Application. Instead, the Root Key Control performs the corresponding encryption and decryption operations and only returns the result.

E. Example Resolved

Passwords are stored in secure storage, ensuring that the integrity of the application and its trusted computing base (TCB) are verified before the application can access

the passwords. Access is granted only to the authorized password manager, and only if it is running in a secure execution environment on a secure operating system. It is important to include the TCB (here: operating system) into the verification process, because on an insecure system, malware might be able to read the memory of the password manager and hence gain access to the passwords.

F. Known Uses

There are various implementations of Secure Storage:

- The Cell processor [4] features storage that can only be accessed when the processor is in a "secure state". In this state, software that is running on the processor has been measured by a secure boot mechanism. This way, the Cell processor can provide secure storage.
- TPM sealing [1] is a mechanism specified by the TCG and implemented in the TPM. With this functionality, a key which can be used only inside the TPM can be restricted in its use by defining specific values for the PCRs. Since the PCRs securely store the recorded measurements from the authenticated boot process, usage of the key can be restricted to software that passes the integrity verification.
- The OMTP TR1 recommendation [5] includes detailed requirements for secure storage on mobile devices.

G. Consequences

The benefits from the secure storage pattern include:

- Only software where the integrity verification succeeded can access the protected data. In combination with a secure operating system, this can prevent malware from accessing sensitive data.
- Data can be stored on a system, such that it can be accessed only when the authorized operating system and software has been started.

The liabilities from the secure storage pattern include:

- Backup strategies become more involved, because the encryption keys are protected in hardware. Data must be encrypted with a different key for the backup system, or a mechanism is needed to backup hardware-protected keys to other secure hardware.
- After an update, software cannot access protected data anymore, if no additional mechanisms are in place. Thus, software updates become more difficult.
- This pattern adds complexity and overhead. It needs to be coordinated with the other protection mechanisms.

H. Related Patterns

Secure Storage requires **Secure Boot** to protect the integrity verification data. If Applications and their Execution Environment are also part of the Secure Boot, then Root Key Control can rely on the integrity verification during Secure Boot and does not need to perform it explicitly.

Secure Storage also requires **Controlled Virtual Address Space** [7] for providing isolated memory for each process to protect private data against unauthorized access. Otherwise malicious processes could wait until the authorized process has access to the decrypted data, and simply copy the data from the memory of this process.

Information Obscurity [7] also addresses how to protect data stored on a system from unauthorized access. Application components obscure data by applying encryption, and encryption keys are hidden in a protected location. Secure Storage describes how to realize such a protected location.

In **Controlled Execution Environment** [7], a process also requests to access a protected resource, and the access is granted or denied based on the access rights of the process. However, the protection of such resources is based on access control rules and requires the underlying **Reference Monitor** [7] of the operating system to be trusted. In contrast, Secure Storage protects the data (using a trusted component such as a security module) even if the operating system has been manipulated or completely replaced. Moreover, granting access in Secure Storage is bound to the integrity of the requesting program and its execution environment.

IV. Conclusion

Secure boot and secure storage are fundamental concepts of Trusted Computing that are vital to a variety of security solutions. In this paper, we introduced the common patterns that are at the core of different implementations of these concepts. We expect our patterns to become a valuable addition to existing operating system security patterns.

References

[1] Trusted Computing Group, "TCG TPM Specification, Version 1.2, Revision 103," https://www.trustedcomputinggroup.org/specs/TPM, 2007.

[2] H. Löhr, A.-R. Sadeghi, C. Stüble, M. Weber, and M. Winandy, "Modeling trusted computing support in a protection profile for high assurance security kernels," in *2nd International Conference on Trusted Computing (TRUST 2009)*. LNCS 5471, Springer, 2009, pp. 45–62.

[3] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *IEEE Symposium on Security and Privacy*. IEEE, 1997, pp. 65–71.

[4] K. Shimizu, "The Cell Broadband Engine processor security architecture," http://www.ibm.com/developerworks/power/library/pa-cellsecurity/, Apr. 2006.

[5] Open Mobile Terminal Platform Consortium, "OMTP advanced trusted environment: OMTP TR1 (v1.1)," 2009, recommendation document, available at http://www.omtp.org/Publications.aspx.

[6] D. Schütz, "Boot loader," in *Proceedings of the 11th European Conference on Pattern Languages of Programs (EuroPLoP 2006)*, 2006.

[7] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. J. Wiley & Sons, 2006.

[8] S. Gürgens, C. Rudolph, A. Maña, and A. Muñoz, "Facilitating the use of TPM technologies through S&D patterns," in *18th Intl. Workshop on Database and Expert Systems Applications (DEXA'07)*. IEEE, 2007, pp. 765–769.