Security Considerations for Code Signing

David Cooper
Andrew Regenscheid
Murugiah Souppaya
Computer Security Division
Information Technology Laboratory

Christopher Bean
Mike Boyle
Dorothy Cooley
Michael Jenkins
National Security Agency
Ft. George G. Meade, Maryland

January 26, 2018

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

Abstract

A wide range of software products (also known as code)—including firmware, operating systems, mobile applications, and application container images—must be distributed and updated in a secure and automatic way to prevent forgery and tampering. Digitally signing code provides both data integrity to prove that the code was not modified, and source authentication to identify who signed the code. This paper describes features and architectural relationships of typical code signing solutions that are widely deployed today. It defines code signing use cases and identifies some security problems that can arise when applying code signing solutions to those use cases. Finally, it provides recommendations for avoiding those problems and resources for more information.

Keywords

application container; code signing; digital signature; firmware; mobile application; operating system; software update

Acknowledgements

The authors wish to thank their colleagues who reviewed drafts of this document and contributed to its technical content during its development. In particular, we wish to thank Docker, HP Inc., Intel Corporation and Microsoft for providing feedback, which was incorporated in this whitepaper.

Disclaimer

Any mention of commercial products or reference to commercial organizations is for information only; it does not imply recommendation or endorsement by NIST, nor does it imply that the products mentioned are necessarily the best available for the purpose.

Additional Information

For additional information on NIST's Cybersecurity programs, projects and publications, visit the Computer Security Resource Center, csrc.nist.gov. Information on other efforts at NIST and in the Information Technology Laboratory (ITL) is available at www.nist.gov and www.nist.gov/itl.

Feedback

Feedback on this publication is welcome, and can be sent to: code-signing@nist.gov.

1 Introduction

Recent security-related incidents indicate the need for a secure software supply chain to protect software products (also referred to as code) during the development, build, distribution, and maintenance phases. Of particular concern is provisioning and updating software that plays a critical role in platform security. A wide range of software products, including firmware, operating systems, mobile applications, and application container images must be distributed and updated in a secure and automatic way to prevent forgery and tampering.

An effective and common method of protecting software is to apply a digital signature to the code. Digitally signing code provides both data integrity to prove that the code was not modified, and source authentication to identify who was in control of the code at the time it was signed. When the recipient verifies the signature, he is assured that the code came from the source that signed it, and that it has not been modified in transit.

This white paper targets software developers and product vendors who are implementing a code signing system or reviewing the security of an existing system, with the goal of achieving improved security and customer confidence in code authenticity and integrity. System integrators and administrators who are concerned about the trustworthiness of the applications that are installed and run on their systems will learn the properties they should expect from a code signing solution to protect their software supply chain.

This white paper describes features and architectural relationships of typical code signing solutions that are widely deployed today. It defines code signing use cases and identifies some security problems that can arise when applying code signing solutions to those use cases. Finally, it provides recommendations for avoiding those problems, and resources for more information. Properly applied, these recommendations will help to ensure that the software supply chain is resistant to attack.

NIST plans to develop further guidance to help organizations evaluating, deploying or managing code signing systems. The high-level recommendations described in this document are expected to form the basis for more detailed recommended practices for code signing.

2 The Basics of Code Signing

This section provides high-level technical details about how this process works. There are multiple roles in the process: developer, signer, and verifier.

The *developer* is the entity responsible for writing, building, and/or submitting the code that will be signed. This entity maintains a secure development environment, including the source code repository, and will submit code to the signer after it has completed the organization's software development and testing processes.

The *signer* is the entity responsible for managing the keys used to sign software. This role may be performed by the same organization that developed or built the software, or by an independent party in a position to vouch for the source of the code. The signer generates the code signing private/public key pair on a device that is sufficiently protected, as the security of this process relies upon the protection of the private key. In many cases, the signer then provides the public key to a certification authority (CA) through a certificate signing request. The CA will confirm the signer's identity and provides a signed certificate that ties the signer to the provided public key. Anyone can use the public key associated with this certificate to validate the authenticity and integrity of code signed with this key pair. If no CA is used, the public key must instead be distributed using a trusted, out-of-band mechanism.

The signer ensures through technical and procedural controls that only authorized code is signed. When code is submitted by developers for signing, the signer verifies their identities and their authority to request a signature. The signer may also take additional steps to verify the code is trustworthy. Ultimately, two or more trusted agents of the code signing system may be needed to approve the request and generate a digital signature. In some cases, the signed code may also be provided to a time stamp authority to indicate when the code was signed.

The *verifier* is responsible for validating signatures on signed code. The verifier may be a software component provided by the same developer as the signed code (e.g., for a signed firmware update), or it may be a shared component provided by the platform (e.g., the operating system).

3 Architectural Components

The code signing architecture is composed of a set of logical components that are responsible for different aspects of the code signing process. The code signing/verifying architecture represented in Figure 1 potentially has four distinct components: the code signing system (CSS), the certification authority (CA), the time stamp authority (TSA), and the verifier(s).

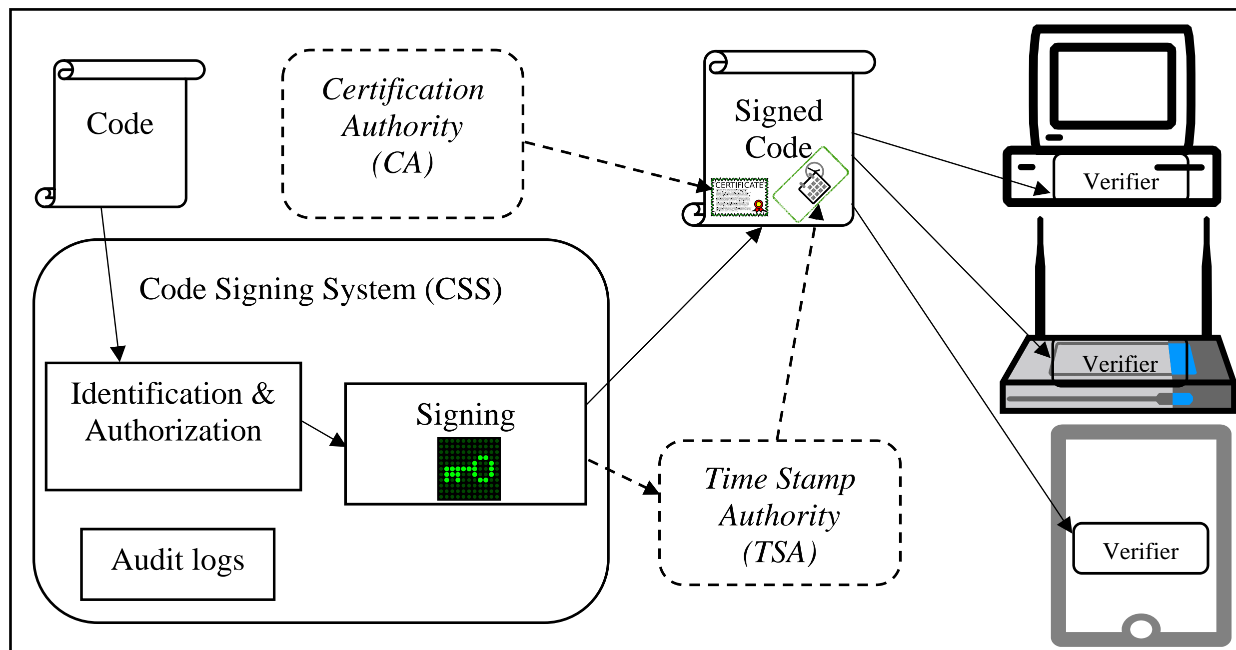

Figure 1: Code Signing Architecture

3.1 Code Signing System (CSS)

The first component, the CSS, receives code submitted for signing, authenticates and authorizes the submitter, and generates the signature. To generate these signatures the CSS has one or more private signing keys, which need to be carefully protected from extraction or unauthorized use.

3.2 Certification Authority (CA)

Typically, a CSS utilizes a CA to enable authenticating the identities of signers. CAs issue certificates to signers in accordance with certificate policies, which specify the security controls and practices the CA follows when issuing certificates, and impose requirements on the subjects of the certificates. NIST Interagency Report 7924 [1] is a reference certificate policy that specifies most of the requirements for a CA that issues code signing certificates. There are also industry groups, such as the CA/Browser Forum [3] and the CA Security Council [4], that have published requirements documents for the issuance of code signing certificates.

3.3 Time Stamp Authority (TSA)

Some code signing architectures use a TSA to demonstrate when a particular piece of code was signed. When a TSA is used, signatures are sent to the TSA, which applies its own signature and signing time to the package. The TSA operates independently from the CSS and keeps its clock synchronized to an authoritative time source. This allows an entity verifying code to accept the signature on the code as valid if the signing key was valid at the time the code was signed, even if the key has already expired at the time of verification or the key was compromised sometime after the code was signed.

3.4 Verifiers

Verifiers are responsible for validating the signature and any certificates and time stamps used by the signers. The verifiers also manage any trust anchors that are used to validate certificates. Either the signer or an independent party may be responsible for the verification component.

3.5 Trust Anchors

A critical aspect of managing the verification component is the management of trust anchors that are used to validate signatures, usually by verifying code signing certificates. Trust anchors are data objects, generally public keys, that are installed and securely stored on the verifying platform. The trustworthiness of a trust anchor is based on the method of its installation, storage and management.

Trust anchors are generally not used to verify code directly, but are typically public keys of root CAs that sign over certificates of code-signing-CAs. This provides a means to recover from incidents resulting in the theft of loss of a private key owned by individual code-signing-CAs or code signers. While trust anchors must be stored in a protected manner, it is not advisable that trust anchors be permanent - if the corresponding private key were compromised, this would render the platform vulnerable without a means to remediate that vulnerability.

4 Code Signing Use Cases

The organization establishes a policy that defines the developer(s) authorized to submit code to be signed by the CSS at particular stages of development. The implementation of the system is dependent on the use case for the CSS. This section describes four example use cases: firmware signing, driver signing, trusted application stores, and application software signing. Please note that these examples are for illustrative purposes; they neither reflect all instantiations of each use case nor are they comprehensive of all code signing use cases.

4.1 Firmware Signing

Secure distribution of firmware is an example of a software supply chain where all aspects of the code signing activity—software development, authorization of the code to be signed, administration of the CSS, and development of the verifier—are typically under the control of a single organization. A typical work flow includes a team of developers who track contributions to the project in a version control system before the code is passed along to a build system. During development, the CSS might sign non-production versions with development or test keys. These signatures would be used in testing but would not be used for verifying code in fielded products.

When the code is ready to be shipped, the developers submit it for signing. An authorized approver (or multiple approvers) examine the code per organization policy and submit the code, along with an approval. The CSS inspects the package (code, approvals, and metadata) to determine whether all requirements are met. The package is then signed with a production key. This key can be unique per project. All production keys are centrally held, likely in a hardware security module (HSM). Each corresponding public key is securely installed in the verifier component on the platform for which the firmware is intended.

4.2 Driver Signing

Device drivers are privileged processes that run close to a computer's system kernel, controlling and communicating with hardware components. A typical personal computer will have several hardware devices with associated drivers. While drivers are typically developed by the associated hardware vendor, all other aspects of the code signing activity are still typically controlled by a central authority. While details vary from platform to platform, device drivers often are signed by a central, preconfigured trusted authority. For operating system (OS) drivers, the approval and signing process is often performed by the OS vendor. Firmware-level drivers (e.g., Unified Extensible Firmware Interface (UEFI) drivers) often rely on a central, third-party signing service, with the trust anchor managed by the original equipment manufacturer (OEM).

In this example, the driver developers do not obtain their own code signing keys and certificates. Instead, they submit their code to the central trusted authority (e.g., the OS vendor or OEM), who verifies the identity of the developer and provides a signature for the submitted driver. If a driver is later discovered to be untrustworthy or malicious, the trusted authority has mechanisms to revoke the particular signature or signatures issued to that developer.

4.3 Trusted Application Stores

Modern computing devices use application stores to distribute signed applications (apps) to users, including software developed by third parties. Different platforms perform code signing in different ways, but in most cases, application stores are operated by the OS or platform vendor to distribute only those applications that have undergone some level of vetting based on a defined policy (see NIST Special Publication (SP) 800-163 [2]). While individual developers may have their own signing keys, either for development purposes or to authenticate to the store, the private keys used to sign applications are under the control of the store, which serves at the CSS. Developers submit their applications to the store, and their identities are verified before the apps undergo vetting. If everything is in order, the store signs the app and publishes it, making it available for download. Devices can be configured to only install apps that have been signed by a key controlled by the application store.

4.4 Application Software Signing

Operating systems generally include support for applications developed and distributed by independent vendors. Depending on the configuration of the system, a valid code signature may be a prerequisite to installation, or the OS may grant access to protected system resources only to signed applications. For these signatures to be accepted by the OS, application developers must obtain a code signing certificate from a CA that is trusted by the OS vendor. The OS vendor may operate a CA for this purpose, or it may partner with commercial CAs to provide this service. The developer generates a code signing key pair, and requests a public-key certificate from the CA. The CA verifies the identity of the developer before issuing the certificate.

The developer uses its private key to sign any applications it develops for that OS. These signatures establish the identity of the developer, and allow the OS to verify that the application has not been tampered with. The developer may also obtain time stamps from a TSA so that the OS will be able to verify the signatures even after the corresponding certificate has expired. The platform OS can determine the provenance of the application. The system administrator or user may be asked to decide whether to trust code from the developer, since the OS vendor does not vet the application. The trusted CA(s) may, however, aid the user by refusing to issue certificates to developers who are known to have signed malicious code in the past.

5 Threats to the CSS

This section enumerates some potential threats to the CSS.

Theft of private signing key: Private signing keys that are not properly protected are at risk of theft, allowing a successful attacker to sign arbitrary code. Limited revocation mechanisms in some systems that rely on code signing exacerbate this threat.

Issuance of unauthorized code signing certificates: Weak protections on CA private keys used to issue code signing certificates, or weak vetting procedures used to issue those certificates, could allow an attacker to obtain one or more unauthorized code signing certificates.

Misplaced trust in certificates or keys: Verifiers could trust certificates or keys for code signing that were never intended to be used for code signing or that otherwise should not be trusted to sign code from a particular source. In some cases, verifiers may allow users to extend trust to untrustworthy certificates or signers.

Signing of unauthorized or malicious code: Code signing procedures could allow malicious or unauthorized code to be inadvertently signed, either as the result of a legitimate mistake, poor governance controls, an insider attack or a successful intrusion into software/firmware development or code signing systems. Similarly, intrusions into development systems or the code signing infrastructure itself, could result in malicious code being signed.

Use of insecure cryptography: Use of weak or insecure cryptographic algorithms or key generation methods could allow cryptanalytic or brute-force attacks to recover private keys or obtain fraudulent certificates. Future developments, such as new cryptanalysis, implementation vulnerabilities, or the development of a cryptographically-significant quantum computer, could render a previously-deployed system insecure.

6 Recommendations

This section presents recommendations for improving the security of the software supply chain by protecting the code during the development, build, distribution, and maintenance phases.

Identify and authenticate (I&A) trusted users: Implement an I&A scheme that adequately identifies the users who have the ability to sign code or submit code to be signed.

Authorize trusted users: Determine who has authorization to submit code for signature. Maintain and regularly review/audit a list that specifies who can submit code and what code they can submit (e.g., for what projects they can submit). Consider requiring two people to approve code for signing and/or the use of multi-factor authentication.

Separate roles within the CSS: Separate critical roles for running the CSS and assign them to different people. In particular, separate the administration role of the CSS from the role of authorized submitter of code.

Establish policies and procedures for reviewing, vetting and approving code: Signed code reflects an organization's acceptance of that firmware or software. Prior to signing code, organizations should review and vet code to ensure it is trustworthy. Organization's existing secure software development and build processes should support these procedures.

Use strong cryptography: Use a well-designed, tested, and maintained cryptographic library, and choose NIST-approved algorithms for hashing and signing.

Protect the signing keys: The secrecy of the signing keys must be maintained. They should be isolated on a machine that has minimal applications and connectivity. If possible, the keys should be stored in an HSM with minimal functionality (e.g., signing).

Use a separate CSS for development: If code needs to be signed regularly as part of development, create a physically or logically distinct CSS that will be used during development and not used for signing production code. As part of this separation, development keys should not chain back to the same root keys that the production keys do.

Isolate and protect the CSS: The CSS is a foundational security system and should be protected as a critical asset. To the greatest extent possible, isolate the CSS from the production network (as well as any external networks).

Utilize auditing and periodically review logs: Audit the use of signing keys. Review the logs regularly for anomalous usage of keys. Protect the logs so that a breach of the CSS does not allow past entries of the log to be overwritten.

Utilize a reputable[1] CA: If a CA is being utilized to establish trust in the binding of the signer and the public key, then either operate a secure CA or purchase certificates from a reputable CA. If a CA is not being utilized, then trust is established by possession of the private signing key alone. In this case, protection of the signing key is paramount as it cannot be easily updated later.

Utilize a reputable time stamp server: If a time stamp server is being utilized to ensure the integrity of the time of code signing, then either operate a secure time stamp server or use the services of a reputable time stamp server.

Manage trust anchors: In many cases, the trust anchor can be updated. If this is the case, care should be taken that the trust anchor only changes when properly authorized. It needs to be stored in such a way that it cannot be changed outside of a well-defined update process.

Manage code versions: If protection against rollbacks is desired, the version of the code should be signed as part of the metadata in the package, and verified by the verifier during a software update process. The update should proceed only if the supplied code is newer than what is already present on the system.

Validate certificates: If a CA is utilized, the verifier must be able to determine that the certificate containing the public signature verification key is valid. It should check that the signature on the certificate is valid and from an authorized CA. Critical fields including the validity period and key usage fields should also be checked.

Validate time stamps: If the signed code includes a time stamp, the verifier must be able to determine that the time stamp was signed by a TSA that it trusts, particularly if the signature on the code was valid at the time specified in the time stamp but is no longer valid at the time of verification.

Validate signatures: The verifier must be able to validate the signature and determine that the public key associated with the signature can be chained back to an authorized signer. Note that in some cases, there will be no CA, and the public key determines authorization.

Check certificate revocation: Sometimes certificates need to be revoked before the end of their validity period. The verifier should either be able to check revocation or have a secure mechanism for updating the trust anchors.

---

[1] Reputable service providers operate under documented policies and practices that have been designed to ensure secure operations and undergo periodic audits by independent auditors to demonstrate that they are operating in compliance with their policies. For instance, audit schemes such as WebTrust are designed to ensure that a CA's documented policies and practices are appropriate and that its audits are performed by a qualified auditor.

7 Conclusion

This white paper describes the security considerations for a code signing solution to protect the software supply chain. Code signing provides assurance that the software is authentic and has not been tampered with during the development, build, distribution, and maintenance phases, and the verifier can validate these properties at runtime. The use cases dictate the deployment model but the core components of the solution include the CA, the TSA, the CSS, and the verifier.

Appendix A—References

[1] H. Booth and A. Regenscheid, NIST Internal Report (IR) 7924, *Reference Certificate Policy, Second Draft*, National Institute of Standards and Technology, Gaithersburg, Maryland, May 2014, 94pp.
https://csrc.nist.gov/publications/detail/nistir/7924/draft

[2] S. Quirolgico, J. Voas, T. Karygiannis, C. Michael, and K. Scarfone, NIST Special Publication (SP) 800-163, *Vetting the Security of Mobile Applications,* National Institute of Standards and Technology, Gaithersburg, Maryland, January 2015, 44pp.
https://doi.org/10.6028/NIST.SP.800-163

[3] *Guidelines for the Issuance and Management of Extended Validation Code Signing Certificates, Version 1.4*, CA/Browser Forum, July 2016, 15pp.
https://cabforum.org/wp-content/uploads/EV-Code-Signing-v.1.4.pdf

[4] *Code Signing Whitepaper*, CA Security Council, December 2016, 16pp.
https://casecurity.org/wp-content/uploads/2016/12/CASC-Code-Signing.pdf