# National Defense ISAC

## Code Signing

May 29, 2024

Authors:
Raul Barreras
Terence Ho
Allan Jacob
Waldemar Pabon
Andrew Zuehlke

Reviewers:
Troy Vuyovich
Will Jimenez
Renee Stegman

## About the Authors

**Dr. Waldemar Pabon, ND-ISAC Lead & Senior Contributor**

Dr. Waldemar Pabon is a Cyber Security Architect with over 28 years of experience in software engineering. Dr. Pabon leads the Application Security Working Group, Software Security Automation, and the COTS Software Assessments Subgroups at the ND-ISAC. Under his leadership, the Software Security Automation Working Group has published seven white papers. The papers provide ND-ISAC members and the industry with a roadmap on how to adopt application security best practices while leveraging automation as a catalyst to achieve efficiencies. Dr. Pabon has a Doctor of Science in Cybersecurity degree from Capitol Technology University.

**Raul Barreras, ND-ISAC Senior Contributor**

Raul is an Information Security Professional with more than 20 years of experience. His multiple roles throughout his career include information security officer, systems administrator, developer, teacher, and occasional pen tester. In recent years, Raul has had the opportunity to use the accumulated experience in a new role: application security. This role has allowed him to discover how much he enjoys being a developer advocate and how fun and useful it is to build a community of security champions while improving the quality of the organization's software. This is Raul's second opportunity to contribute to application security papers at the ND-ISAC.

### Terence Ho, ND-ISAC Contributor

Terence Ho is a Cloud Application Cybersecurity Specialist with 4 years of experience in the Information Security field with a primary focus on developing automation, integrating security tooling, testing, and standardizing innovative solutions to enhance the security of applications in the Enterprise. He joined ND-ISAC in December 2022 and became a member of the Application Security working group to learn more from the Information Security community and contribute his expertise to the group. Terence graduated from the University of Washington with a Bachelor’s Degree in Computer Science and Software Engineering with a focus on Information Assurance and Cybersecurity.

### Allan Jacob, ND-ISAC Contributor

Allan Jacob is a Cloud Engineer with 4 years of experience in network engineering, network automation, and network security. He focuses on developing automation within cloud environments, standardizing cloud deployment, and monitoring. Allan joined the Security Group in January 2022. A graduate of Rutgers University with a Bachelor's Degree in Information Technology, Allan is also pursuing a Master's Degree in Information Systems at the New Jersey Institute of Technology with a focus on Network Security.

**Andrew Zuehlke, ND-ISAC Senior Contributor**

Andrew Zuehlke is a Cyber Security Architect with six years of experience in the information security field. Andrew graduated from Appalachian State University in 2017 with a Bachelor of Science Degree in Computer Science and Computational Mathematics. In his former role, Andrew served as the lead administrator of SIEM and EDR solutions. In early 2020, Andrew moved to his current role as Cyber Security Architect, primarily supporting Research & Development and Information Technology. Since joining the ND-ISAC in December 2020, Andrew has become a member of the Application Security working group as well as the Cloud Security and Architecture Working Group. Andrew has co-authored multiple white papers with ND-ISAC's Application Security Working Group.

## Executive Summary

The protection of software requires a strong security posture in the Software Development Lifecycle (SDLC). All we must do is look at the way the industry is trying to enforce not only the protection of the source code but also the concern of producing well-secured software (NIST, 2022). Under this paradigm, security controls are embedded at various stages of the process to enable faster risk detection. From Static Application Security Testing (SAST) to Software Composition Analysis (SCA), each security control has the responsibility of identifying risk proactively at three distinct stages:

- during development
- use and inclusion of dependencies
- integration of components

Unfortunately, this is not enough. While these security controls are key security components, even with well-secured software, attackers can embed malicious code in software without anti-tampering protection. Having anti-tampering coverage requires software producers to perform an attestation that no one has modified in any way the software produced through the SDLC. A more perfect attestation involves the use of code signing. Code signing involves the use of digital certificates to provide proof to the operating system (OS) that no malicious actor has tampered with the implementation.

Any attempt by an attacker to embed malicious code in software binaries stamped with code signing will break the chain of trust created by the digital certificates. This attempt will result in the digital signature being invalidated which will warn the OS not to trust the software. This type of protection ensures that software created using security controls remains secure through the delivery process to the end user. This anti-tampering protection completes the security lifecycle of software producers.

Recent industry changes have impacted how organizations use public certificate authorities (CAs) to perform code signing. The new Federal information Processing Standard (FIPS) requirement to store and protect the private key is moving the industry to adopt products such as the Hardware Management Service (HSM) to ensure the key used during the code signing will never leave the secure environment. In this paper, cloud services and HSM are presented as secure mechanisms to meet this requirement. Furthermore, the integration of code signing in the DevOps pipeline presents a good opportunity to integrate with any of these different services to create efficiencies while maintaining a strong security stance in the protection of the binaries.

As noted, embedding security controls in the SDLC alone is not enough to secure software end to end. Understanding the diverse types of services organizations can use to integrate and implement code signing as an anti-tampering protection mechanism will elevate the security posture of software and minimize the ability of attackers to weaponize code.

Transitioning the organization to more agile SDLC methodologies helps take advantage of efficiencies while enforcing strong security requirements for the protection keys used during the code-signing operation.

Table of Content

# Introduction

## Objective

One of the most common processes in software engineering is the compilation process. The compilation process is used to produce one or more binaries needed to support the implementation of software in the infrastructure. This process itself does not necessarily represent a security threat. Unfortunately, attackers realize that binaries can be modified. This creates a great opportunity for them to embed malicious code in what normally would be a process without major concerns.

There are many examples of how attackers are able to modify software and compromise infrastructure. A Mandiant report (Proska, Hidelbrandt, Zafra, Brubaker, 2021) shows how the injection of malware in executable binaries has increased over 1000% from 2015 through 2021. How can industry protect against attackers using binary tampering attempts to exploit infrastructure? The answer is simple, code signing.

Code signing provides an attestation to the consumer of the software that no one has tampered with the original implementation of the binaries. The code signing process adds digital signatures to binaries which identify the organization that is providing the attestation of the anti-tampering protection. The absence of the digital certificate in binaries is a clear indication of the absence of anti-tampering protections (either because the organization never signed the code or because an attacker broke the digital signatures by tampering with the binaries).

In this paper, we address multiple important aspects of code signing:

- Strategies to sign code
- How to integrate into the DevOps pipeline
- Considerations for cloud environments
- Best practices
- Compliance requirements

The discussion of these topics will provide a comprehensive understanding of not only the options available to support anti-tampering protections but also how to create efficiencies while embedding this practice as part of the Software Development Lifecycle (SDLC).

## Audience

The audience of this white paper includes security engineers, software engineers, cloud architects, and product managers responsible for securing software used by end users.

## Structure of the paper

This paper introduces the importance of code signing as a mechanism to ensure software has not been tampered with. The paper first introduces the concept of code signing. Manual and automated processes to achieve code signing are discussed. Cloud considerations, best practices and compliance concerns are finally discussed to cover additional important aspects of code signing.

## Code Signing

In the ever-evolving software development landscape, the security, integrity, and authenticity of digital assets have never been more critical. One of the more common and well-established security practices used to protect software is known as code signing. At its foundation, code signing establishes trust in software applications, mitigates risks, and helps ensure compliance with industry standards and regulations. The process of signing code involves the use of a stamp containing digital signatures, which will serve as an attestation to the end user that no one has tampered with the software. Figure 1 provides an explanation of the code signing process.

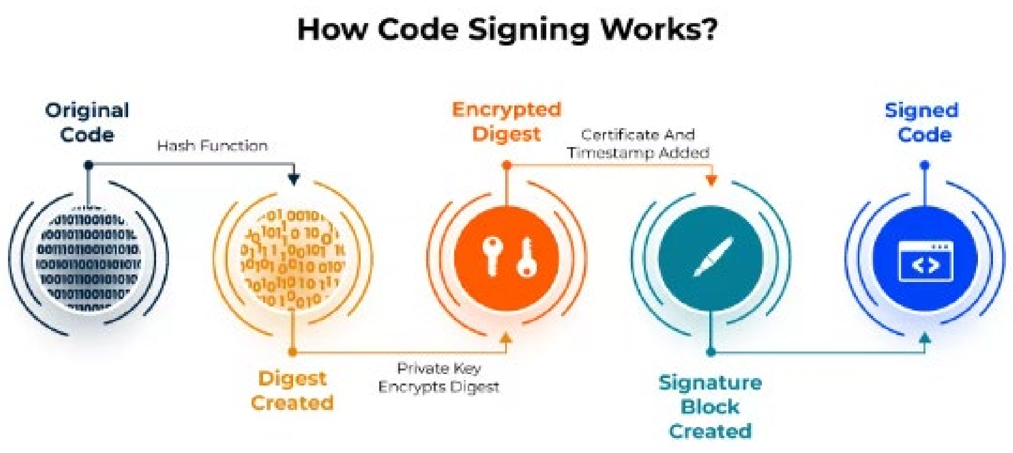


Figure 1 Code Signing process explained. From *How Does the Code Signing Process Work? – Understand the Code Signing Architecture* [Image], by J. Mehta, n.d., (https://signmycode.com/resources/code-signing-architecture-how-does-it-work)

The following sections will further investigate the multifaceted world of code signing, exploring various code signing strategies, its integration into DevOps workflows, considerations for cloud applications, best practices, and its pivotal role in meeting compliance requirements.

## Strategies Available to Sign Code

As with most software development practices, code signing is not a one-size-fits-all practice, instead it encompasses a spectrum of strategies tailored to various application types, platforms, and deployment scenarios. This paper will introduce common code signing methodologies, from a manual approach to cloud considerations, to the integration of code signing in agile methodologies such as DevSecOps.

## DevOps Integration

As highlighted in previous white papers published by ND-ISAC's Application Development Working Group, integration into the Software Development Lifecycle (SDLC) and the Continuous Integration / Continuous Delivery (CI/CD) pipeline is essential to an efficient and secure software development program:

- Software Security Automation: A Roadmap Towards Efficiency and Security
- How to protect Cloud Native Applications

The integration of code signing into CI/CD pipelines is no exception. This paper will explore how organizations can bridge the gap between development (Dev) and operations (Ops) through code signing practices that ensure each iteration of software is signed, secure, and ready for deployment.

## Cloud Applications Considerations

Just as cloud computing has revolutionized how applications are developed, deployed, and scaled, it also caused code signing to take  an important role in addressing additional risks. This paper identifies multiple considerations and best practices for signing cloud applications hosted on public, private, or hybrid cloud infrastructures.

## Code Signing Best Practices

As an established practice, code signing has well defined standards and best practices to ensure organizations secure their software supply chain. This paper provides a comprehensive guide to code signing best practices, from the protection of the keys to the need to establish a strong revocation process.

## Meeting Compliance Requirements

In a world marked by increasingly stringent regulatory requirements and industry standards, compliance is essential to software solutions and deployments. This paper highlights how code signing can play a pivotal role in meeting compliance requirements to support either commercial or government requirements. The paper discusses some of the most common compliance standards impacting code signing.

## Strategies available to sign code

To provide anti-tampering protections to software binaries, organizations can use two standard methods available to industry:

- manual
- automated

Because of the surplus of use cases associated with the delivery and creation of software, it is difficult to just adopt a single approach. Understanding the main differences between the automated process as well as automation options will enrich any discussion associated with selecting an anti-tampering approach. The next sections include discussions on the manual and automated processes available, as well as automation paths to facilitate decision making to support the development team’s needs.

### Manual Process

The adoption by organizations of agile methodologies associated with the SDLC is increasing year over year. Nevertheless, the reality is that not all organizations are necessarily ready for it.  The organizations specific use case could potentially dictate a different approach when applying digital signatures. If an agile methodology is not implemented, then the organization will require teams to use a manual code signing process. Fortunately, there is a manual path teams can pursue.

Imagine a scenario where a developer may want to sign his/her binaries as soon as the compilation of the code is completed in the Integrated Development Environment (IDE). Under such a scenario, there is no feasible path to implement a DevOps approach since the code signing process is taking place right at the development environment and not in the Continuous Integration (CI) space. This use case requires development teams to manually add the digital signatures right from the IDE.

Developers can use multiple tools to ensure their compiled software receives digital certificates, which are available from multiple software vendors (Microsoft, DigiCert, etc.), and provide tools that support code signing. The IDE itself provides a mechanism for developers to execute macros as soon as the compilation event takes place, including a menu option. This scenario enables developers to execute the code signing directly from the IDE without the need of spawning additional infrastructure, other than their development environment and the environment where the private keys for the digital certificates are stored.

For example, the IDE could be configured to communicate with a REST API in a cloud environment to send the binary to the cloud service and get the digital certificate applied. There are specific security concerns with this approach that must be addressed (i.e., the protection of the access key to the REST API) but putting that discussion aside, it enables a feasible scenario for developers that allows them to sign any binaries and eventually send them to their destination (secure file exchange, web site upload, etc.).

In other scenarios, tools from vendors can be used by a development team to achieve the same anti-tampering objective. Vendors such as Microsoft, DigiCert, etc. provide tools to perform the embedding of the digital certificates in the code. Traditionally, these tools provide the following common capabilities:

- specify which digital certificate to use.
- the time stamp to apply.
- which file digest algorithm to apply.
- specify which store name to use.
- provide the password protecting the certificate.
- which binary to apply the digital certificates to.

Based on new industry requirements to protect the private key used during the code signing process (DigiCert, 2023), the use of these tools is applicable to private digital certificates and not public certificates. The use of public certificates requires additional security controls to enable a successful code signing operation. For example, if an organization attempts to sign a binary utilizing any of these tools and a public certificate, the use of a supported hardware token to act as the private key will be required. This hardware token will enforce strong authentication to ensure the private key never leaves the hardware and forces the user to provide a password every time the certificates are applied to a software binary.

Even though this approach will guarantee the binaries provide an attestation no one has tampered with them, the reality is that the manual process is not an effective mechanism to create efficiencies and support agile implementations. In fact, this manual approach will create friction and will hinder the ability of organizations to use automation. This creates a predicament for organizations looking to create efficiencies that require the use of other services to ensure digital certificates are applied as part of an automated effort. Thankfully, there are some automation methods for Code Signing which ensure efficiency is at the forefront of the business value discussion.

## Automation

The preferred method should be the use of automation. Through automation, organizations would not only have a standardized process for anti-tampering but also gain efficiency. As more organizations adopt a DevOps approach in the Software Development Lifecycle (SDLC), it should be easy to embed code signing as one of the stages in the workflow.

Including a code signing stage in the DevOps pipeline enables the process to apply the digital certificates needed for the attestation (which states that no one has tampered with the software) are included before the binaries are stored in their destination. Deciding where to perform the code signing stage will require the evaluation of multiple elements.

For example, in a highly efficient environment where the time it takes for the pipeline to complete is critical, performing the code signing before the test cycle has been completed would not be ideal. Performing the code signing before gate checks pass would waste time since the binaries would not be the version deployed. The cycles spent signing the code will be wasted as failing integration testing (even with code signed) will dictate to discard that version of the code due to the gated failures in the Test cycle.

Imagine a scenario where binaries are signed, and the next step performs a security verification that identifies critical vulnerabilities. In such a scenario, any DevOps pipeline must stop. The time spent signing the binaries was wasted as the pipeline will not continue in the subsequent stages. It would be better to wait until security vulnerabilities and integrity checks have been vetted, and then apply the digital signatures. As the binaries get deployed during the test stage to the test environments, the process would evaluate the same conditions we would find in a production environment. This approach will guarantee the binaries are protected against any tapering attempt  by the time we are sending them to their destination (ftp site, server, cloud resource, etc.). Figure 1 provides an overview of a common code signing flow.

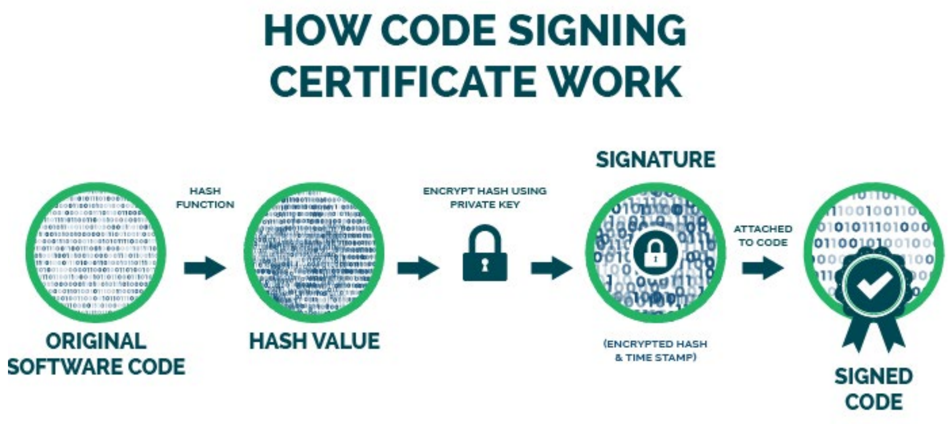

HOW CODE SIGNING CERTIFICATE WORK
ORIGINAL SOFTWARE CODE
HASH FUNCTION
HASH VALUE
ENCRYPT HASH USING PRIVATE KEY
SIGNATURE
(ENCRYPTED HASH & TIME STAMP)
ATTACHED TO CODE
SIGNED CODE

*Figure 2 Code signing process.* From *What is Code Signing Certificate? How* [Image], by ClickSSL, 2023, (https://www.clickssl.net/blog/what-is-code-signing-certificate)

The important question to answer as part of this discussion is: what is available to achieve anti-tampering protection? To answer this question, we would need to look at:

- services offered by public Certificate Authorities (Cas).
- services provided by cloud environments.
- HSMs

**Services offered by public CAs**

Recognizing the changes introduced in industry to protect the private key used during the code signing exercise, some public CAs have introduced REST API platforms anyone can use to receive a binary from a customer and perform the code signing. The result of this process often results in a JSON response that can be used to reconstruct the binaries with the corresponding digital certificates. This type of service provides a good platform for organizations that are trying to avoid maintaining internal implementations and leveraging code signing as a service.

These services are offered and billed just like utilities. Organizations can subscribe to these services and consume them. Some of these services are billed per request; the higher the number of requests to perform code signing, the higher the cost. In use cases where organizations are creating software very often (every day, every couple of hours), cost should be evaluated to verify if it is a cost-effective approach versus deciding to establish an internal implementation to control the process.

**Services offered by Cloud environments**

Cloud providers integrated in their service offerings the ability to perform code signing. These services are required to host the pair of certificates (public and private certificates; lower and higher-level certificates in the chain of trust)  as well as the private key that is used during the signing process. Just like the public CA services, these capabilities are offered and billed based on consumption. The signing service has a combination of REST API services as well as CLI capabilities which can be used in very different use cases. The availability of REST APIs enables organizations to embed code signing in their DevOps pipelines.

One important aspect of this type of approach in cloud services is the need to secure the API calls to the REST APIs used to perform code signing. Any security misconfiguration in these services could lead to an attacker gaining access to the cloud environment. Therefore, when defining access to these services, access control must ensure the principle of least privilege to minimize the blast radius of a potential compromise. Also, the token used to access these services must be protected and secured to prevent clear text access to it which could enable an attacker to use the organization's digital signature in any malware injection against the software binaries. For more details on API Services Security, please access the Software Security Controls: Application Programming Interface (API) Services white paper.

**HSMs**

Hardware security modules (HSM) are a device or a virtual service that provides tamper resistant services and secure storage for secrets, as well as encryption and decryption capabilities. One of the key characteristics of HSMs is the fact that these devices are validated and certified against industry standards such as FIPS 140-2 (Entrust, n.d.). HSMs can be found embedded as services from software producers as well as part of cloud services. Most of the major cloud providers include the ability to have a virtual HSM hosted in a cloud environment and enforcing industry best security practices.

As a matter of fact, the use of an HSM supports new industry requirements regarding the protection of the keys. Since the keys are well protected inside the HSM, they can be used to support safe code signing operations. HSMs can provide, in addition to code signing, many other capabilities such as:

- Application-level encryption
- Database encryption

- TLS/SSL
- Credential management
- Secret management
- Private cloud encryption
- Encryption key management
- Payment credential issuing/provisioning
- Container encryption

Because of their strong security nature to enforce security standards and their strength against any tapering attempt, HSMs are ideal at helping meet compliance requirements such as FIPS 140-2, PCI, etc. For more information regarding compliance requirements, please refer to section "Meeting Compliance Requirements."

It is understandable not all organizations are going to have the same requirements or compliance obligations. Nevertheless, the availability of all these different options to perform code signing provides strong flexibility for any organization which will help them meet any type of use case and scenario. Regardless of the approach used, focusing on the concept of strong protection of the key used for code signing should be at the forefront of any consideration since the objective is to prevent attackers from gaining access to the code signing keys. Focusing on protecting code signing keys hinders the ability of an attacker to inject malware in software and then pass it on as if no one has tampered with the software implementation and the potential nefarious consequences of such actions.

## DevOps Integration

One of the most common agile methodologies adopted by organizations is DevOps. As an automation vehicle, DevOps enhances multiple efficiency opportunities for software development teams. Through the integration of different tools, such as the Source Code Manager (SCM) and different security controls, a DevOps pipeline provides the flexibility to configure different stages to conduct separate important actions (build the software, run a security scan, deploy the code, etc.). It is within the confines of this process that SW Engineers are able to integrate code signing. The following section provides an overview of the most important concerns surrounding code signing and DevOps.

## Sign Early, Sign Often Principle

Software build pipelines are being targeted more frequently than ever before. It is not enough to secure your software's code but to ensure and validate each step in the software development pipeline. An effective strategy and best practice in the development process is to "sign early, sign often." This strategy reassures developers to sign their code in the early stages of development and sign it at multiple steps throughout the development lifecycle.

Implementing code signing during the early stages of the development lifecycle is recommended. It is advisable to sign the code as soon as it is ready to be uploaded to a central codebase or repository. By signing the code early, it is possible to identify potential vulnerabilities and address them at an earlier stage. Changes and adjustments to code frequently occur in the DevOps pipeline. Each modification, whether it involves code or added artifacts, requires a re-signing of the item to uphold trust within the environment and ensure that these changes are properly validated.

## Signing Commits and Tags

If there is a breach in repository access, it opens the door for potential malicious actors to insert counterfeit source code into the application without detection by the developers. While solutions such as GitHub offers certain safeguards like branch protection and GitLab offers a similar solution like Protection Branch, which restricts sensitive operations on specific repository branches, these controls while valuable, can easily be disabled by an intruder who has compromised the environment. To prevent unintended access to repositories, another layer of security is necessary to further strengthen security, especially in security sensitive environments.

Source Code Managers (SCMs) offer numerous features for tracking changes made to a repository over time. Among these features, one stands out in ensuring the legitimacy of source code: commit and tag signing through PGP (Pretty Good Privacy). The essence of signing in Git involves applying cryptographic signatures to individual patches or tags using keys that developers keep confidential.

Developers will need to generate a PGP key pair. Tools like GnuPG can help generate one. The following steps below show how to generate a PGP key generated on a local machine represented by its fingerprint (GitHub, n.d.). Then the steps indicate how to configure Git to sign commits and tags, and how to demonstrate the use of the PGP key identified using its fingerprint.

```
$ gpg --gen-key
$ gpg --fingerprint Allan Jacob
pub   rsa3072 2023-09-18 [SC] [expires: 2025-09-17]
      193C A493 D1CB B626 D414  13F2 F65E A552 A852 9053
uid           [ultimate] Allan Jacob <example@example.com>
sub   rsa3072 2023-09-18 [E] [expires: 2025-09-17]
```

```
$ git config --global user.signingKey 193CA493D1CBB626D414 13F2F65EA552A8529053

$ git config --global commit.gpgsign true

$ git config --global tag.gpgsign true
```

Unauthorized or malicious code changes can have serious security implications. Verifying commit authors helps to quickly identify any suspicious or unauthorized changes. It is also useful for tracking access and permissions to the codebase to prevent security breaches. It is recommended to verify commit authors using auditing scripts periodically and proactively in the DevOps pipeline. It is noted that periodic auditing of git signatures has some downsides to consider. Vehent (2018) suggests that periodic auditing mechanism must run outside the CI/CD pipeline to prevent a compromise of the pipeline from any malicious change of the auditing script. It is recommended the auditing mechanism performed is in complete isolation.

### Certificate Authority

A standard method among organizations includes signing certificates issued by a CA. Obtaining a code signing certificate from a trusted CA is essential to guarantee reliability and safety. Afterwards, the certificate is installed onto the designated development machine or server responsible for uploading the code to a repository. For your version control system or code repository to verify changes committed, it is necessary to integrate the certificate into the system or repository. When code changes are made, the certificate signature is confirmed by the version control system, ensuring trust and validity to the commit.

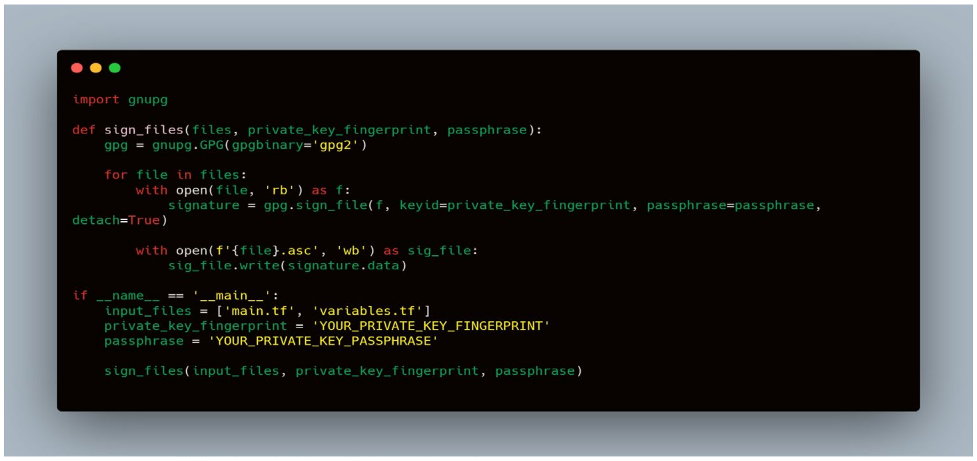
## PGP Method

PGP (Pretty Good Privacy) plays a pivotal role in securing software development through its multifaceted capabilities. Its primary functions encompass encryption, decryption, and the creation of digital signatures, all crucial for ensuring the integrity, authenticity, and confidentiality of digital assets. In the realm of code development, PGP's significance lies in code signing—a process where developers sign their code with their private key, generating a unique digital signature. This signature acts as a tamper-proof seal, verifying that the code remains unchanged and originates from the expected source. By signing code with PGP, developers establish a clear chain of trust, offering users a means to authenticate the origin of the software and ensure its legitimacy. This process prevents unauthorized alterations and reduces the risk of malware injection during distribution, fostering a secure development environment. Consequently, PGP-signed code not only validates the integrity of the codebase but also enhances confidence among users and collaborators, contributing significantly to the overall security posture of software development practices.

## Infrastructure as Code

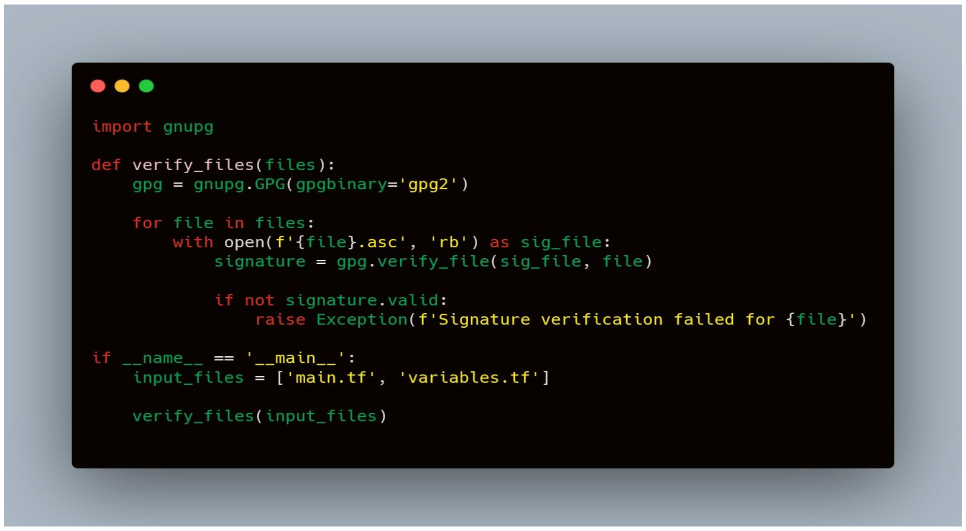
Infrastructure as Code (IaC) refers to the practice of defining and managing infrastructure (including servers, networks, and storage) using code and software development techniques. This means that infrastructure provisioning, configuration, and management are treated as software development tasks, which allows for automation, repeatability, and version control in infrastructure setups.

In the world of Infrastructure as Code (IaC), code signing is a strategic method that involves adding digital signatures to the IaC templates and configurations. This helps verify authenticity, integrity, and source of the automated infrastructure’s code. Adding code signing to the IaC workflow must include a specific signing step in the IaC pipeline. This step should come after the IaC templates are created before they are applied to the infrastructure. Custom script or suitable signing tools are options available to add a digital signature to the IaC templates by leveraging the digital certificate’s private key during the signing phase. After that, a verification step should be introduced to check the signature's validity before executing the IaC template. This verification process ensures the integrity of the template and confirms that it has not been altered since signing. Figure 3 below showcases an example of PHP code signing files while Figure 4 provides an example of a function performing integrity checks.



```
import gnupg

def sign_files(files, private_key_fingerprint, passphrase):
    gpg = gnupg.GPG(gpgbinary='gpg2')

    for file in files:
        with open(file, 'rb') as f:
            signature = gpg.sign_file(f, keyid=private_key_fingerprint, passphrase=passphrase,
detach=True)

        with open(f'{file}.asc', 'wb') as sig_file:
            sig_file.write(signature.data)

if __name__ == '__main__':
    input_files = ['main.tf', 'variables.tf']
    private_key_fingerprint = 'YOUR_PRIVATE_KEY_FINGERPRINT'
    passphrase = 'YOUR_PRIVATE_KEY_PASSPHRASE'

    sign_files(input_files, private_key_fingerprint, passphrase)
```

*Figure 3 Sign code example in PHP code.*



```python
import gnupg

def verify_files(files):
    gpg = gnupg.GPG(gpgbinary='gpg2')

    for file in files:
        with open(f'{file}.asc', 'rb') as sig_file:
            signature = gpg.verify_file(sig_file, file)

            if not signature.valid:
                raise Exception(f'Signature verification failed for {file}')

if __name__ == '__main__':
    input_files = ['main.tf', 'variables.tf']

    verify_files(input_files)
```

*Figure 4 Enforcing integrity checks in code*

To achieve work efficiency, it is a good idea to automate the verification step. This ensures that only correctly signed templates are deployed to the infrastructure. To keep track of all signed templates, signatures and deployment instances, it's important to have a comprehensive logging and auditing mechanism in place. This helps establish accountability and traceability throughout the process.

## Signing Containers

Containerization is a technology that packages applications and their dependencies, including libraries and runtime environments, into a single, lightweight unit called a container. These containers are isolated from the host system and can run consistently across different environments, making it easier to deploy and manage software applications. Code signing containers is crucial because it enhances the security and trustworthiness of containerized applications. By digitally signing container images with a private key or certificate, developers ensure that the image's content remains unaltered and authentic (Miller & Ramani, 2022).

## Docker Content Trust to secure container registries

In a DevOps pipeline, where the secure and reliable transfer of data is paramount, Docker Content Trust (DCT) emerges as a highly valuable consideration. When data moves across networked systems, especially over untrusted mediums like the internet, the integrity and authenticity of that data are crucial. DCT addresses these concerns by enabling the use of digital signatures for data exchanged with remote Docker registries (Docker, n.d.). These signatures facilitate client-side or runtime verification of the data's integrity and its source, ensuring that what is received from a registry is both unaltered and published by a trusted source.

DCT empowers image publishers to sign their images, creating a verifiable chain of trust, while image consumers can confidently verify the authenticity of the images they fetch. This assurance is essential, particularly in DevOps pipelines, where you need to guarantee the consistency and reliability of the components you are deploying. By implementing DCT, you establish a robust security layer, safeguarding against potential tampering and unauthorized alterations to container images. It gives control over which image tags are signed, allowing for a controlled release process. In essence, DCT is akin to a security "filter" for a registry, ensuring that only trusted, signed images are visible and accessible.

## Code Signing Policy

Incorporating code signing policies in a DevOps pipeline is a vital step to improve the security and dependability of software delivery. These policies guarantee that software components undergo a standardized signing and verification procedure before deployment, instilling confidence in the code's integrity. To govern the code signing process, various effective policies can be implemented. We recommend requiring code signing for all software releases, no matter how they are deployed. This step can easily be included in release automation scripts or tools, so it becomes a natural part of the deployment process. Another policy, previously discussed above, is to automate code signing within the Continuous Integration/Continuous Deployment (CI/CD) pipeline. This helps ensure consistency and reduces errors by providing that every piece of code is verified before moving on to the next stage of the release process. These policies reinforce the reliability of the software release process.

To ensure accountability and authenticity of crucial updates, code signing policies can be enforced through Git tags or commits. Requiring signed Git tags or commits for important milestones or critical changes is a good practice. For changes that have a significant impact, it is recommended to establish a policy of multiple signatures. This approach demands code signatures from multiple authorized team members, introducing an extra layer of review and approval. It is particularly beneficial for high-priority modifications.

Keeping code signing keys secure requires a strong policy that includes secure storage and strict access controls. The keys should be stored in a trusted key management system or Hardware Security Module (HSM) to prevent unauthorized access. It is also important to include detailed provenance information, such as commit IDs and build information in signed artifacts or metadata. This helps establish a clear chain of custody and confirms the authenticity of the signed artifacts. Additionally, the key stored in the key management system should only be configured for the CI/CD system to access the key and to also support key rotation.

A clearly defined code signing policy requires signature verification during deployment. This means that any item that fails the signature verification process will not be deployed, creating a strong defense against possible tampering. Along with these policies, it is important to have thorough auditing and logging practices that document code signing activities and enable precise tracking for accountability and problem-solving.

## Considerations

Code signing provides a strong layer of security, but like any security measure, it is not entirely immune to all potential risks. It significantly enhances software trustworthiness and mitigates many threats, but it is essential to understand its limitations and potential vulnerabilities.

## Binary Provenance

Ensuring the accuracy of automated tasks is crucial. What is the correct setup of the pipeline? How is it possible to impose specific guarantees for certain deployment settings that do not apply to all?

To tackle these issues, utilizing explicit deployment policies that outline the expected attributes for each unique deployment setting is required. From there, these deployment environments can compare policies with the source and origin of the deployed artifacts to guarantee compliance. By embracing this approach, the software supply chain is streamlined and minimizes the need for unspoken assumptions, thereby simplifying the process of scrutiny and verification. This method also brings a heightened level of transparency to every stage within the software supply chain, mitigating the risk of inadvertent misconfigurations.

This allows for a single signing key for each build step rather than for each deployment setting. This efficiency stems from the newfound ability to rely on the binary origin as the basis for making informed deployment choices. Imagine overseeing a network of software modules, where each module must be made exclusively from code sourced from its designated repository. Relying solely on conventional code signing would mean managing a distinct signing key for every source repository. This could lead to configuration complexities and potential missteps.

Alternatively, a provenance-centric deployment policy is an approach where the CI/CD system generates binary provenance records. The records serve as clear markers of the source repository for each piece of code. All these records are signed consistently using a single, central key. Each module's deployment policy then explicitly lists the approved source repositories. This simplifies the verification process and minimizes the risk of errors. All crucial information about each module can be found in a single, easily accessible repository, making management tasks simpler.

By consolidating all critical information about each module in a single, readily accessible repository, this approach streamlines management and enhances transparency and accountability within the software ecosystem. It provides a holistic view of the software supply chain, making it easier to track and verify the origins and integrity of each component. This method offers a robust foundation for auditing and compliance, as it simplifies the process of ensuring that each module adheres to its designated source repository, contributing to a more secure and reliable software development and deployment process.

Here are rules to help mitigate the threats suggested by Heather Adkins et. al. (2020):

- Source code was submitted to version control and peer reviewed.
- Source code came from a particular location, such as a specific build target and repository.
- Build was through the official CI/CD pipeline.
- Tests have passed.

- Binary was explicitly allowed for this deployment environment. For example, do not allow “test” binaries in production.
- Version of code or build is sufficiently recent.
- Code is free of known vulnerabilities, as reported by a sufficiently recent security scan.

As covered in the Code Signing Policy section, these policies help ensure that only authorized and validated changes are released into production environments.

## Considerations for Signing Cloud Applications

In today's technical landscape, cloud computing has become the standard and pillar of modern businesses, enabling scalable, flexible, and easily accessible software solutions. Cloud applications, ranging from web services to microservices, play a pivotal role in this paradigm shift. However, despite the increase of cloud-based services, the need to ensure and maintain the integrity and authenticity of these applications remains critical. This section investigates multiple considerations and best practices for code signing and cloud applications.

### Cloud Applications: A Transformative Landscape

Cloud applications are a diverse and dynamic class, residing on public, private, or hybrid cloud infrastructures and catering to a global audience. They empower organizations to deliver software to users and clients with unprecedented agility. Despite this dynamic landscape, however, cloud applications also introduce a set of distinctive security challenges, which have resulted in specialized code signing practices being established specifically for cloud environments.

## Key Considerations for Code Signing in the Cloud

### *1. Trust in Cloud Providers*

Cloud applications often rely on the infrastructure and services provided by the underlying cloud service providers (CSPs). Consequently, it is important to evaluate and establish trust in the CSP’s security measures—this includes verifying that a selected CSP adheres to any industry- applicable security protocols and compliance requirements. As many organizations adopt multi-cloud strategies, it is equally important to implement code signing processes that are adaptable across multiple CSPs.

### *2. Secure Key Management*

The scale and dynamic nature of code signing in the cloud requires robust key management practices. Therefore, it is of utmost importance to implement state-of-the-art key storage solutions, such as Hardware Security Modules (HSMs) or cloud-based key vaults, to secure code signing keys against unauthorized access. Access to these key storage solutions should follow industry best practices for least privileged access and separation of duties.

### *3. CI/CD Integration*

As illustrated in previous ND-ISAC whitepapers, software development—especially as it relates to cloud-native technologies—frequently relies on Continuous Integration/Continuous Deployment (CI/CD) pipelines to enable fast updates and deployments of solutions. By incorporating code signing into the CI/CD pipeline, an organization can ensure every release of a cloud application is properly signed before deployment. The automation of code signing in the CI/CD pipeline also ensures consistency and minimizes human errors in code signing workflows.

Code signing can be further extrapolated and applied to verify the authenticity of third-party libraries and dependencies used in an application, resulting in a well-defined and secure software supply chain.

### *4. Scalability and Load Balancing*

Just as cloud applications are designed for scalability, code signing should be built to scale with an application to accommodate high traffic loads while simultaneously maintaining secure signing processes.

### *5. Incident Response*

While no organization hopes to execute an incident response plan in response to a security incident, having a robust incident response plan in place is nevertheless critical; for software developers it is essential that this incident response plan include procedures for re-signing and re-deploying cloud applications if needed.

### *6. Compliance and Auditing*

While not applicable to all solutions, software sometimes is required to comply with industry standards and regulations. Such regulations can often include a requirement to maintain a well-documented audit trail of code signing activities, thereby maintaining accountability and traceability.

While not an exhaustive list, the incorporation of these considerations into cloud application development and deployment will further increase the security of, mitigate risks with, and maintain trust of applications in the dynamic world of cloud computing.

## Code Signing Best Practices

Code Signing is essential in ensuring a piece of code has not been altered; however, code signing alone is not enough to ensure authenticity. Best practices for code signing should be followed to ensure that a binary that has been code signed is still valid and unaltered.

### Identifying and Authenticate

Users who have access to sign code should be identified and provided access based on an Identity and Authentication method. This identifies users authorized to sign code or submit code to be signed.

### Authorize Trusted Users

After determining the identity of users, authorization should be given to trusted users to sign code. This list should be maintained and regularly reviewed. For example, employees that are no longer with a company should not have access to perform code signing using the company's certificate.

### Separate Roles within the Code Signing Service

Critical roles within a Code Signing Service should be separated. The role of administrator should be separate from the role of the authorized code signer. This way, not all users are administrators, and the authorized code signer role can only perform code signing with a given restriction and is unable to add or remove users, grant privileges or more.

### Policies and Procedures for Vetting Approved Code

Signed code is code that an organization has given their name and stamp of approval to. Before code signing, a review process should be done to ensure that the code is legitimate and trustworthy. An organization should implement this into their secure software development process such as CI/CD and peer reviews before code signing.

## Strong Cryptography

When performing code signing, it is important to utilize a certificate that is at least the industry key size minimum as stated by the Certificate Authority Browser (CAB) forum. For example, using code signing certificates with 3072 bit or larger RSA keys.

## Protect the Signing Keys

It is important to follow industry security requirements as stated by the Certificate Authority Browser (CAB) forum for secure storage of signing keys (certificates). The CAB forum has mandated that code signing certificates must be stored in a hardware token or HSM as of June 2023 (CAB, 2023, p. 47). This ensures that certificates are securely stored. Keys are tougher to crack due to the private key being stored on the hardware itself and being unique to each device.

## Use a Separate Code Signing Certificate for Development

Separation of production signing and development code signing should be done to separate what can be used for production vs development. The development certificate should not be chained back to the same root keys as the production keys (NIST 8) to differentiate the signature and trust authority.

## Isolate and protect the Code Signing Service

The Code Signing Service should be protected as a critical asset, as compromise of this service could result in bad actors being able to release code that looks legitimate and validated to users. As Code Signing has matured over the years, more and more commercial code signing services have made it easier to sign code in a modern development lifecycle and comply with additional security requirements without fully isolating the ability to perform code signing.

### Audit and Logging

To have traceability in the event an action takes place without permission, users who have access to sign code should be identified and associated with signings in logs. For example, identification through logging is required to authenticate the user's identity and permission to sign code using a specified certificate.

### Utilize a Reputable Certificate Authority

If a trusted Certificate Authority is used, the trust is established by the binding of the signer and the public key. This creates a certificate chain that can be validated and does not solely rely on the private signing key alone if a trust Certificate Authority was not used.

### Utilize a Reputable Time Stamp Server

A time stamp is used to ensure the integrity of the time of code signing. This makes the signature display a time when the signature was valid and proves that code was trusted at that time when it was signed. If a malicious time stamp server was used, it could have a spoofed time, tricking users into thinking that the signed code is valid when it is not supposed to be.

### Manage Trust Anchors

Trust anchors should only be changed when properly authorized. This needs to be stored in a way that it cannot be changed outside of an auditable update process.

## Manage Code Versions

If protection against the use of previous versions is desired, then code should be signed with the version information as part of the metadata in the package and verified by the verifier during a software update process where the package is installed. The update should only be processed with the supplied code which is newer than what is already on the system where the package is present.

## Validate Certificates

The verifying mechanism must be able to determine that the certificate containing the public signature verification key is valid as shown by the Certificate Authority. It should also check that the signature on the certificate is valid, from the authorized Certificate Authority, and critical fields such as validity period and key usage fields are used in accordance with the code's signing needs.

## Validate Time Stamps

Code with time stamps must be validated by the verifying mechanism to ensure that the signature on the code was valid at the time specified on the time stamp or it is not valid at the time of verification.

## Validate Signatures

The verification mechanism must be able to validate the signature and determine if the public key associated with the signature can be chained back to an authorized signer, typically to a trusted Certificate Authority.

### Check for Revoked Certificates

There are some instances where certificates should be revoked before the end of their validity period due to the compromise of certificates. The verifying mechanism must be able to check if a certificate was revoked or have a secure mechanism for updating the trusted anchors (NIST 9). Revoking a certificate that has signed code will cause the signed file to fail certificate validation based on the signature, thus the code will need to be signed again to function if certificate validation is required on the system.

## Meeting Compliance Requirements

As section "Code Signing" describes, a subscriber will use a certificate issued by a Certification Authority to sign a piece of code. In this section, we will cover the fundamental areas of code signing covered by several regulations:

- identity proofing
- key generation
- use of FIPS
- key size requirements
- storage
- private key protection

Several organizations are responsible for creating regulations for using code signing certificates. Those regulations could be specific to some industries and accepted internationally or only applicable to government institutions and their business partners. In the private field, for instance, we have The Internet Engineering Task Force (IETF), the Certification Authority Browser Forum (CA/Browser Forum or CA|B), and the Payment Card Industry Data Security Standard (PCI-DSS). The National Institute of Standards and Technology (NIST) and Department of Defense (DoD) are the more prominent institutions regulating this area in the public sector.

We will review the following documents and how they fit in the Code Signing process.

- IETF's "ACME End User Client and Code Signing Certificates"
- CA/Browser Forum's "Baseline Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates"
- Payment Card Industry Security Standards Council's PCI (Payment Card Industry) DSS
- Secure Software Development Framework (SSDF), NIST Special Publication 800-218
- NIST Special Publication 800-53
- DoD's Cybersecurity Maturity Model Certification

## ACME End User Client and Code Signing Certificates

Automatic Certificate Management Environment (ACME) describes a framework for automating the issuance of digital certificates. IETF, the creator of ACME and responsible for the technical standards that power the Internet, in their Internet Draft "ACME End User Client and Code Signing Certificates" (draft-ietf-acme-client-07) document (Moriarty, 2023), extends the ACME protocol to support code signing certificates.  The document defines the process to retrieve a code signing certificate from a Certification Authority, when a subscriber requests it. Validating that a subject is who they claim to be is called Identity Proofing. NIST Special Publication 800-63-

3 Digital Identity Guidelines supplies the procedures and requirements for Identity Proofing of users requesting Code Signing Certificates. For environments with stricter security requirements out-of-band initial authentication and identity proofing are recommended.

## CA/Browser Forum’s "Baseline Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates"

CA/Browser Forum is a voluntary group of certification authorities (CA's), vendors of Internet browser software, and suppliers of other applications that use X.509 v.3 digital certificates for code signing and other applications. They also set up the rules and requirements to become a Certification Authority. In CA/B's "Baseline Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates," we can find technical security controls for the CAs (Certificate Authorities) to manage the certificates throughout their lifecycle.

### Identity Proofing for Extended Validation Code Signing Certificates

When validating code signing certificates, pay attention to the following concerns:

- Verify applicant's existence, identity, legal, physical, and operational aspects.
- Confirm applicant's authorization for EV Code Signing Certificate, including roles, contracts, and approvals.

**Key Generation**

Section 6 of Technical Security Controls defines how private keys should be generated, their recommended key length, and how they should be stored.

- Private Key Generation: The subscriber's private key is generated, stored, and used in suitable FIPS (Federal Information Processing Standard) 140-2 Level 2 or Common Criteria EAL 4+ compliant hardware.
- Key Sizes:
    - If the Key is RSA, the modulus MUST be at least 3072 bits long.
    - If the Key is ECDSA, the curve MUST be one of NIST P-256, P-384, or P-521.
    - If the Key is DSA, one of the following key parameter options MUST be used:
    - Key length (L) of 2048 bits and modulus length (N) of 224 bits
    - Key length (L) of 2048 bits and modulus length (N) of 256 bits

## Payment Card Industry Security Standards Council's PCI DSS

The Payment Card Industry Security Standards Council's PCI DSS is a standard oriented to members of the payments industry designed to establish robust security practices and guidelines for safeguarding sensitive payment card data and ensuring secure transaction processing. Controls B.2.8 and C.1.1 of the PCI Software Security Framework (PCI-SSF) recommend that all software files should be cryptographically signed to enable cryptographic authentication of the software files by the payment terminal firmware, and all software components and services should be documented in a software bill of materials (SBOM). Although PCI-SSF does not explicitly say that SBOMs must be signed, they refer to some SBOM standards like CycloneDX, SPDX, and SWID. CycloneDX is an OWASP (Open Web Application Security Project) flagship project, and OWASP, in their Software Component Verification Standard, sets up the signing of SBOM as a verification requirement. PCI-DSS indicates what to do, not how to do it.

## NIST Secure Software Development Framework (SSDF)

The Secure Software Development Framework (SSDF), NIST SP (Special Publication) 800-218, was developed by NIST in response to the U.S. Federal Government's Executive Order 14028, "Improving the Nation's Cybersecurity," which mandates that NIST will supply guidance on "practices that improve software supply chain security". Code signing is cited in NIST-SP 800-218 as an example of one of the methods used to achieve the stated goals of the Protect Software practices: what organizations should do to protect software components from tampering and unauthorized access. They refer us to NIST SP 800-53. In their Special Publication 800-53, NIST also supplies a comprehensive set of security controls for federal information systems in the United States. Code signing is mentioned as part of the families of controls "System and Communications Protection" and "System and Information Integrity," specifically in controls SI-7 "Software, Firmware, And Information Integrity”, SC-8 "Transmission Confidentiality and Integrity," SC-12, “Cryptographic Key Establishment and Management,” and SC-18 "Mobile Code."

All NIST documents refer to SP 800-56A, SP 800-56B, and SP 800-56C for guidance on cryptographic key establishment schemes and key derivation methods. For cryptographic key management, publications SP 800-57-1, SP 800-57-2, and SP 800-57-3. SP 800-175B should be consulted. The algorithms that can be used to generate a digital signature are defined on the standard Federal Information Processing Standards (FIPS) 186-5 Digital Signature Standard (DSS).

**Key Generation**

All keys shall be generated within a FIPS 140-validated cryptographic module or obtained from another source approved by the U.S. Government for the protection of national security information. Long-term keys shall be inventoried. [NIST SP 800-57 "Recommendation for Key Management", Section 8.1.5]. NIST SP 800-131A provides the approval status of the NIST-approved cryptographic algorithms and their key lengths.

### Digital Signature Algorithms

This is the current list of approved Digital Signature Algorithms.

- RSA
    - Minimum key size 2048
    - Digital signature key pair shall not be used for other purposes (e.g., key establishment)

- ECDSA
    - Minimum key size 224
    - ECDSA keys shall not be used for any other purpose (e.g., key establishment)

- EdDSA
    - Minimum key size 224
    - Cryptographic function H

- For Ed25519, SHA-512 shall be used
- For Ed448, SHAKE256 (as specified in FIPS 202) shall be used
- Ed25519 is intended to provide approximately 128-bits of security, and Ed448 is intended to provide approximately 224-bits of security.

DSA is no longer approved for digital signature generation. However, DSA may be used to verify signatures generated prior to 02/03/23.

## Department of Defense's Cybersecurity Maturity Model Certification (CMMC)

The Department of Defense Cybersecurity Maturity Model Certification (CMMC) 2.0, designed to enforce the protection of sensitive unclassified information that is shared by the Department with its contractors and subcontractors. CMMC 2.0 "Advanced" level (Level 2) will be equivalent to the NIST SP 800-171, "Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations." The "Expert Level" (level 3) is currently under development and will be based on a subset of NIST SP 800-172, "Enhanced Security Requirements for Protecting Controlled Unclassified Information." In this case, code signing is used to enforce the integrity of critical or essential software, as explained in CMMC 2.0, control CM.5.074.

## Conclusion

Mitigating the risk of software weaponization requires a continuous improvement process that includes careful consideration of not only security controls but also anti-tampering protections. While security controls will provide a proactive way to detect risk in software, without an actual attestation that software has not been tampered with, an attacker could potentially take advantage of this gap and compromise any organization. Implementing code signing requires careful consideration of the current state of the SDLC. In industry, there are several key approaches that could be used to secure the integrity of the software.

These approaches include manual as well as automated strategies. Regardless of the current state of SDLC in the organization, these different code signing options laid out in this paper have the objective of providing enough flexibility to accommodate for any type of implementation. As stated in the paper, organizations should aspire to implement efficient environments where code signing is performed as an integral part of their automation initiatives to minimize friction and create efficiencies. But if the organization is not there from a maturity standpoint, this gap should not be a factor when enforcing an integrity attestation.

As organizations transition to protect the software end-to-end, special considerations must be pondered when dealing with containers, cloud, images and hashing mechanisms. Recent changes in the industry are forcing more compliance requirements upon any code signing holder. These considerations are captured in different compliance standards presented in this paper such as SSDF, PCI, FIPS, among others. Achieving consistent implementations of code signing practices must leverage standardization. Adopting best practices will ensure that not only the software is secure, but also the process implemented achieves this objective.

## References

Adkins , H., Beyer, B., Blankinship, P., Lewandoski, P., Oprea, A., Stubblefield, A. (2020). *Building Secure and Reliable Systems*. O'Reilly.

CAB. (2023, December 7). *Baseline Requirements for the Issuance and Management of Publicly-Trusted Code Signing Certificates*. https://cabforum.org/wp-content/uploads/Baseline-Requirements-for-the-Issuance-and-Management-of-Code-Signing.v3.5.pdf

ClickSSL. (2023, November 27). *What is Code Signing Certificate? How Does Code Signing Certificate Work?* ClickSSL. https://www.clickssl.net/blog/what-is-code-signing-certificate

Digicert. (2023, May 1). *Code Signing Changes In 2023*. https://knowledge.digicert.com/alerts/code-signing-changes-in-2023

Docker. (n.d.). *Content trust in Docker*. Docker. https://docs.docker.com/engine/security/trust/

Entrust. (n.d.). *What is a Hardware Security Module (HSM)*? Entrust. https://www.entrust.com/resources/hsm/faq/what-are-hardware-security-modules

GitHub. (n.d.). *Signing commits*. GitHub. https://docs.github.com/en/authentication/managing-commit-signature-verification/signing-commits

Mehta, J. (n.d.). *How Does the Code Signing Process Work? – Understand the Code Signing Architecture*. Signmycode. https://signmycode.com/resources/code-signing-architecture-how-does-it-work

Miller, Z., Ramani, R. (2022, June 23). *Cryptographic Signing for Containers*. AWS. https://aws.amazon.com/blogs/containers/cryptographic-signing-for-containers

Moriarty, K. (2023, August 3). *ACME End User Client and Code Signing Certificates*. https://datatracker.ietf.org/doc/html/draft-ietf-acme-client-07

National Institute of Standards and Technology. (2022). *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for Mitigating the Risk of Software Vulnerabilities*. https://doi.org/10.6028/NIST.SP.800-218

Proska, K., Hidelbrandt, C., Zafra, D. K., & Brubaker, N. (2021, October 27). *Portable Executable File Infecting Malware Is Increasingly Found in OT Networks*. Mandiant. https://www.mandiant.com/resources/blog/pe-file-infecting-malware-ot

Vehent, J. (2018). *Securing DevOps: Security in the Cloud*. Manning Publication Co.