Google Cloud

Shifting left on security

Securing software supply chains

Mike Ensor, Drew Stevens

Solution Architects

February 25, 2021

Table of contents

Overview

Continuous delivery (CD) is a common practice in which software tools are designed to build and deploy software. Developers carefully consider requirements, coding best practices, and clever solutions. However, they often overlook security requirements if those requirements are not explicitly stated. According to Tripwire's State of Container Security Report (Q1 2020), 46% of responders indicated that they had known security vulnerabilities in production. The number of responders that did not know whether they had known security vulnerabilities in production was slightly higher at 47%.

This document is intended for software developers, DevOps engineers, and site reliability engineers interested in adding explicit trust attribution and verification to their build and deployment pipelines.

This article is also intended for readers interested in collecting fast feedback for exposed security vulnerabilities. While the document discusses VM images and containers designed for Kubernetes, the principles are applicable to all software that consists of build and deploy phases, including serverless applications and platform-as-a-service (PaaS) applications.

The document focuses on the responsibilities, practices, processes, tools, and techniques that both systematically increase confidence in the software development lifecycle (SDLC) and reduce security-risk concerns. It discusses securing continuous integration and continuous delivery (CI/CD) pipelines. It does not discuss traditional security topics like perimeter security, penetration testing, or active threat detection on live services. Activities implemented before the pipeline, like defensive programming, and activities applied after pipeline deployment, like threat monitoring, are also not discussed.

When developers are not given clear security requirements, or security requirements are not accessible, it is possible to introduce vulnerabilities into the codebase. These vulnerabilities create information-security issues. Security vulnerabilities are often added to applications inadvertently by using publicly available libraries and dependencies. Known security vulnerabilities are documented and categorized as Common Vulnerabilities and Exposures (CVEs). In StackRox's State of Container and Kubernetes Security report (Q1 2020), 44% of responders slowed or halted deploying applications to production because of security concerns. According to Google's DevOps Research & Assessment (DORA) group, deployment frequency is a key metric used to determine team and organization maturity. The ability to build and release code frequently provides organizations with rapid response to customers and rapid response to security exposures.

This whitepaper is broken into five sections. These sections derive from the distinct phases of the SDLC and modern CI/CD pipeline practice:

1. Discuss vulnerabilities and trust in source code.
2. Describe techniques that increase trust in the build process.
3. Explore how to increase trust in built artifacts before deploying to an environment.
4. Consider the recommended techniques and processes to further increase trust when deploying code into environments.
5. Review how trust is acquired and used to help ensure that the overall process reduces the security risks associated with software development.

Trust is established throughout the five phases. This document discusses how to capture and use this trust to increase confidence in the security posture of the software supply chain. Often, security has been the concern of groups without direct development responsibilities or visibility. That means that security tooling, practice, and implementation are often applied after artifacts are built and deployed into production environments.

Companion repository

As a complement to this whitepaper, an example repository shows how and where each of the principles outlined is implemented. The principles described in this document are independent of any particular tooling or vendor. Apply them to projects that match, as closely as possible, the principle you hope to achieve. Avoid combining multiple principles into one tool or step. Doing so allows for maximum granularity. That granularity provides output flexibility for multiple audiences.

Terminology

Continuous integration / Continuous delivery (CI/CD): The combined practices of continuous integration and continuous delivery.

CI/CD pipeline: A sequence of repeatable and reliable actions implemented with code that results in building, securing, publishing, and deploying software.

Artifact: A binary representation of a packaged or grouped software solution. Most common forms of artifacts are OCI-based images. For the purposes of this document, artifacts are built to be immutable and versioned.

Pull request (PR) / merge request (MR): A request to merge one source branch into another. This process is used to govern what code is accepted into primary branches intended for production use. A PR/MR facilitates code review, automated quality checks, and security tests.

Common Weakness and Errors (CWE): A classification of static code analysis designed to look for common coding mistakes, like using non-escaped user input for SQL statements.

Common Vulnerabilities & Exposures (CVE): A classification of known vulnerabilities for operating systems and libraries. CVEs are stored in a database and used to compare against artifacts for matches.

Accumulating trust

Fundamental to a CI/CD solution is the confidence gained through rigorous activities that increase the computational trust of people and systems in the new artifact. Continuous integration and continuous delivery have been mainstays in the software industry for over a decade, but the core principles are often lost during implementation.

Trust is defined as the firm confidence in the reliability, truth, ability, or strength of someone or something. Trust can be recorded as implicit or explicit. Both implicit trust and explicit trust carry different weights in terms of confidence when referenced outside of the pipeline context. For example, implicit trust is not verifiable and therefore an artifact inside of a repository might or might not have been pushed through the security-minded CI/CD pipeline. If explicit trust is recorded, a verifiable audit trail can be recreated for any artifact and therefore trust can be established outside of the pipeline context.

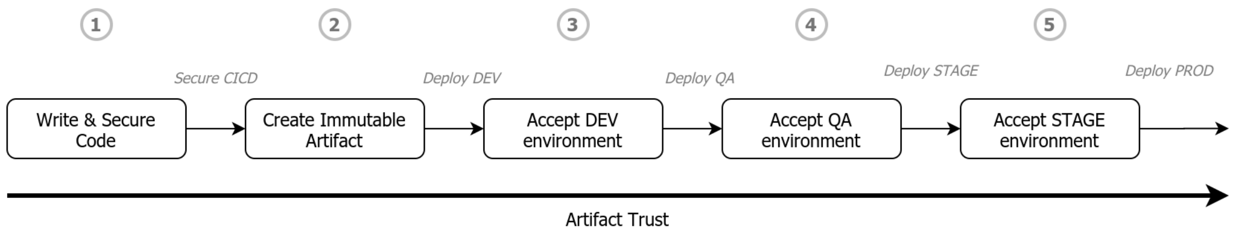
An example of implicitly recorded trust is when a quality assurance (QA) representative validates the quality of a software release version, informs others that they validated the build, but then does not record the validation. An example of explicitly recorded trust is when a QA representative validates the quality of a software release version, and then records the action. For example, they could write an email or digitally represent the date, time, and context that attests to the operation's success.

Recording trust is necessary to confidently reconstruct a series of stages. However, the need to record trust is often overlooked when not implemented explicitly. Recording trust is equivalent to accumulating trust.

Accumulated trust is built up through progressive stages in a defined series. Trust accrues as an artifact advances through the pipeline, from a lower-level environment through production. Each pipeline stage includes activities, like completing an automated test suite or a manual validation by exploratory testing. Upon successful completion of each activity, digital evidence that is cryptographically signed to ensure authenticity can be associated with the artifact.

As illustrated in the following diagram, trust is:

1. Implicit when developers finish feature development
2. Explicit when code security tools are complete
3. Implicit when Development members accept the DEV environment
4. Explicit when QA signs off on one or more releases
5. Explicit when stakeholders accept the staging environment.



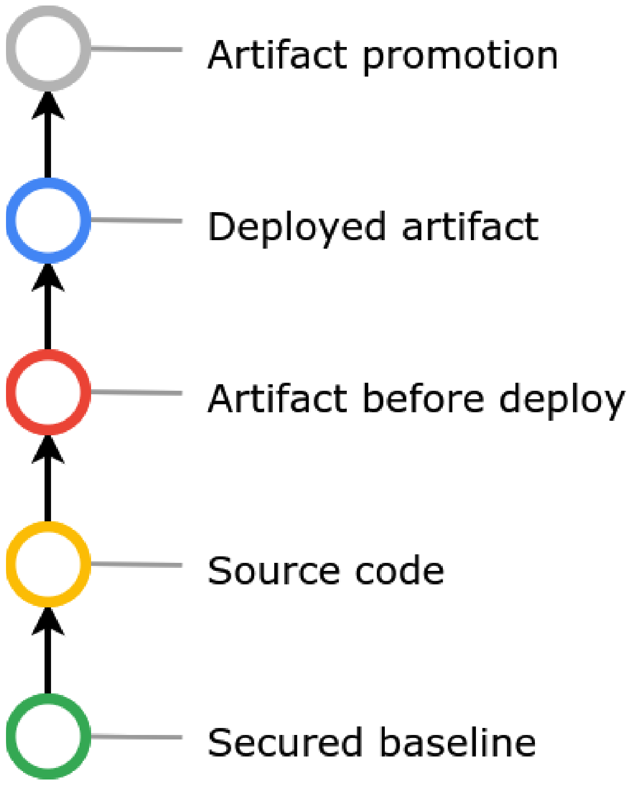
Historically, implicit trust was collected ad hoc from ticket systems, email, chat, or verbal communication. Explicit trust is the verifiably authentic digital evidence associated with an artifact. The aspiration is to replace implicit trust with explicit, digitally verifiable trust. Digital attestations are collected according to the lifecycle activities of a pipeline. Those trust artifacts are used to make deployment and runtime decisions for security, quality, and authenticity purposes.

To raise confidence, another key practice is to digitally represent all parties that are directly or indirectly associated with building software. For example, digitally representing a QA member who manually executes test suites and validates the results. When a test run successfully completes, an attestation is created for the artifact that associates the QA member with those tests. Digitally representing software supply chain actors is discussed in the Attestations, signers, and attestors section.

Applying a diverse array of digital tests that represent all aspects of the delivery lifecycle maximizes confidence in the software. It also exposes events that the pipeline can respond to. For example, the moment a failure occurs, the pipeline should be stopped and feedback should be given to everyone involved. This practice is called fast feedback. Fast feedback is a crucial component for driving success in any CI/CD pipeline process. During failure, the pipeline stops running; it no longer continues to accumulate digital trust. This lack of trust keeps the artifact from being deployed into target environments. Attaching signed digital evidence that verifies each successfully completed pipeline stage is important to accumulating digital trust. It is also important for increasing the overall confidence level of an artifact produced in the software supply chain.

Building secure delivery pipelines

There are five distinct software supply-chain phases that software goes through to be deployed to production. As shown in the following diagram, the five phases are physical infrastructure (secured baseline), source code, artifact before deployment, deployed artifacts, and artifact promotion.



Secured baseline

To build software, infrastructure changes are required to help support a secure baseline environment. From a networking perspective, this phase does not include the necessary techniques to physically secure or isolate resources.

Source code

The process of securing and verifying the software code.

Artifact before deployment

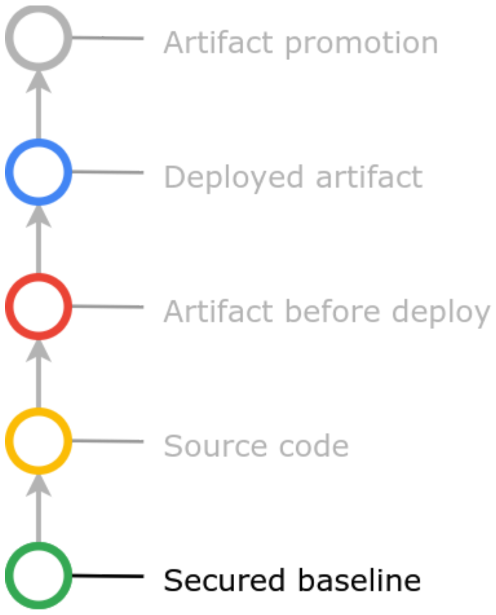
The activities and processes that are applied when building and releasing software. These activities occur before a versioned artifact is deployed to an environment.

Deployed artifact

Focuses on a single versioned and immutable candidate artifact.

Artifact promotion

The security-oriented activities and events associated with promoting an immutable versioned artifact through environments. For example, the development, quality assurance, staging, and production environments.



Securing a baseline environment

Starting with a secure infrastructure is an essential element of software security. Deploying to compromised infrastructure can introduce common vectors, like [person-in-the-middle attacks](). While this paper does not cover the full breadth of the methods used to help secure infrastructure, it does discuss the notable techniques related to securing CI/CD pipelines.

Declarative infrastructure

A key component of securing infrastructure is to use *declarative infrastructure*. Declarative infrastructure is commonly known as *infrastructure as code* ([IaC]()). It defines infrastructure components as code, manages the components in code repositories, and helps ensure that the infrastructure components undergo the same level of checks and balances as application feature code. Declarative infrastructure code should follow the same principles described in the following sections, including:

- Using a continuous integration server
- Implementing continuous delivery
- Running associated automated tests

- Using feature branches
- Code scanning and linting
- Enabling policy management

A secondary benefit of using declarative infrastructure is the ability to increase the frequency of re-deploying infrastructure resources, like Kubernetes clusters. A growing industry practice is to limit resource lifetimes to a fixed period that is measured in days. The benefit of periodic, forced infrastructure re-deployment is that it decreases the likelihood of configuration drift. Configuration drift occurs when resource configurations deviate from the needed state as found in the code repository.

DevOps best practices recommend treating infrastructure like paper plates as opposed to fine porcelain. The resource state should be deterministic in both creation and configuration. When the resource state is deterministic, it allows for a greater degree of automation across all resources. Ultimately, this practice leads to increases in operational efficiency.

A fully automated declarative infrastructure allows for the use of an immutable infrastructure implementation pattern. This implementation pattern, which reduces the overall security surface area, disallows changes to resource state through direct interaction. Instead, it requires changing the needed state, rebuilding new resources, and migrating stateful information to the new resources.

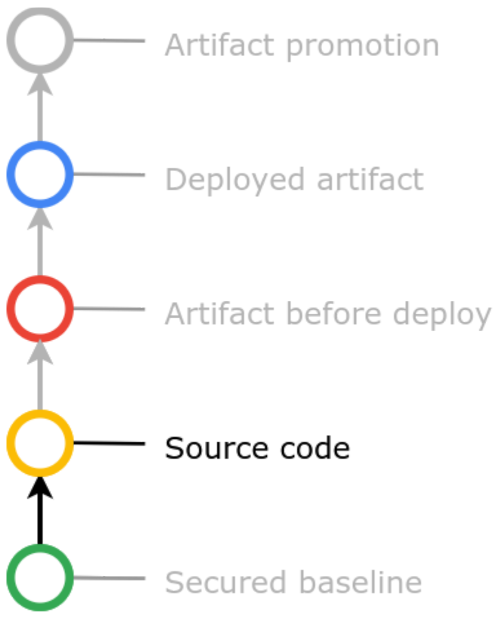
A critical aspect of a secure pipeline process is to ensure that the target deployment infrastructure is not compromised or unidentified. Using digital signatures to verify machine integrity and OS integrity prevents the introduction of rootkits or other malicious software hidden in the boot sequence. This capability is provided by Shielded VMs. Use Shielded VMs and actively monitor them for all Google Cloud VMs, including Google Kubernetes Engine nodes. As of GKE 1.18, Google implements Shielded VMs by default.

Docker containers have independent user space, but they use a shared kernel. Another method to help secure pipeline infrastructure is to run Docker containers in full isolation in a sandbox. Creating a container sandbox solution in a cluster can be achieved using the open source gVisor container runtime.

For GKE clusters, gVisor is exposed as a feature called GKE Sandbox. Using gVisor helps close the security gap present when using a shared kernel. It is the default for common container runtimes, like nvidia, crun, and runc. There are tradeoffs between using a fully sandboxed runtime like gVisor and an OCI-based runtime. Evaluate if the container can be run within an isolated runtime. For instance, because there is a performance cost incurred when gVisor intercepts system calls, consider whether the performance degradation of a sandbox is an acceptable tradeoff for the added security. Furthermore, kernel-specific modifications cannot be implemented in a gVisor kernel, as gVisor focuses on intercepting system calls instead of implementing a true kernel. The gVisor documentation provides more details on how system calls are intercepted and reviews the trade offs between a VM and a container approach.

Google offers a wide range of tools and processes to increase the security of your infrastructure. You can read more about Google's point of view on securing infrastructure in the following articles: Google Infrastructure Security Design Overview, Hardening your cluster's security, and Using Workload Identity.

Now that you know how systems prove their integrity and authenticity, you can focus on how to make the source code that underpins your CI/CD pipeline more secure.



Securing source code

This section focuses on the tools, processes, and techniques applied to source code. Many of the tools and techniques described here are implemented through CI/CD steps that identify security threats within the source code before the system creates an artifact.

Automated testing improves response time

All code committed to the software configuration repository should be run in a Continuous integration (CI) tool and use automated tests. These steps are fundamental to securing your source code. Having sufficient automated testing decreases the risks associated with deploying new code changes by reducing the chance of regressing the code or introducing new defects. The principles for securing code that are described in this section should be addressed early in the CI/CD pipeline process and executed as frequently as every commit.

The DORA research group's four key metrics highlight that deployment frequency and lead time for changes are both correlated with the efficacy of automated test suites. Test automation helps reduce the risk of regression defects and supports necessary refactoring. While automated tests do not directly increase security, having a broad suite of tests lowers risk by enabling a quicker response to both threats and discovered vulnerabilities.

In addition, automated test suites can save time by supplying proxy-scanner inputs for dynamic application security tests. Proxy scanners can be configured to run using an existing test suite or to run during automated user-level testing. The type of test suites should be user-level tests, like Selenium, or end-user tests such as build acceptance or full end-consumer regression tests. Proxy scanners are further discussed in the Dynamic application security tests section.

While the costs associated with creating and maintaining an automated test suite are large, they provide value to the SDLC by reducing late-stage defects. Further, using the same suite of tools for multiple purposes provides additional value. That means you can better justify the cost of ongoing test suite maintenance and upkeep.

Choose memory-safe software languages

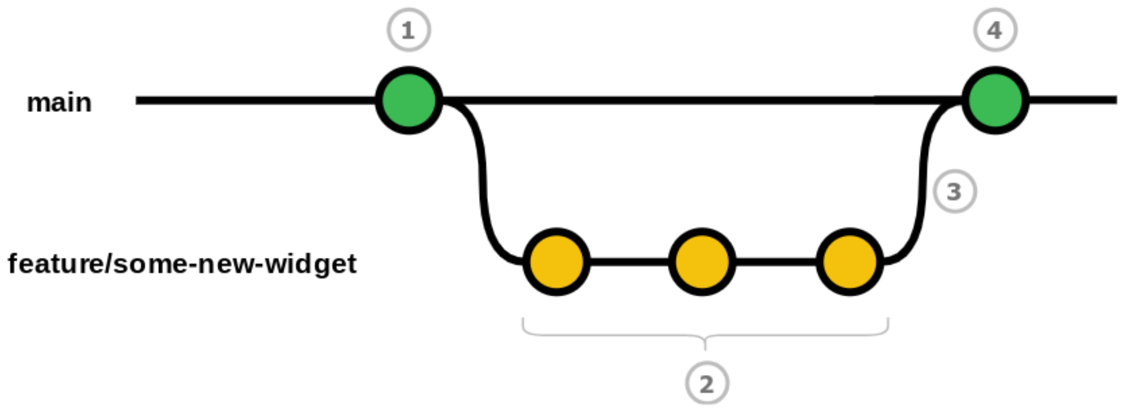
According to research at Microsoft, "~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues." Their research demonstrates a correlation between the use of memory-safe software languages and a reduced risk of memory-based vulnerabilities. Since memory-based vulnerabilities are difficult to detect, you can mitigate the risk by using languages that provide inherent support for memory safety. While not an exhaustive list, the Rust, Golang, Haskell, C#, F#, D, Java, Nim, and Ada languages are all considered to be memory safe.

Consider using statically typed, memory-safe languages to reduce the risk of memory-based vulnerability exploits. As an example, recent efforts by Google showed that using a memory-safe language would have prevented 53 of 95 potential memory exploits in the open source application curl. Where requirements and conditions are not conducive to memory-safe languages, consider using fuzz testing to stress your applications and identify vulnerabilities early. A recent blog from Google highlights these concerns and addresses fuzz-test tools like Fuzzing to assist in finding memory issues early before source code reaches any environment.

Flawless change management

A key finding in the DORA 2019 Accelerate State of DevOps report is that an optimized software change management lifecycle leads to lower costs, higher stability, and an increase in confidence. The report suggests that a best practice is to require a team member's approval of every code change as a part of code review. Perform the code review either before committing the code to version control, or before merging the code into the main branch during a pull request.

One approach to creating an organization-wide code-review practice is to implement a source code management (SCM) process that restricts primary branch access solely to the CI tool service account. Developers create a short-lived branch that not only isolates feature code, but also requires creating a pull request (also called a merge request). Creating the branch indicates that the feature code is ready to be reviewed, accepted, and merged into the main code branch. For more information, see page 49 of the DORA 2019 Accelerate State of DevOps report.



The preceding diagram shows the flow of an SCM. The developer:

1. Creates a feature branch
2. Commits and finalizes code for the feature
3. Issues a pull request
4. Merges commits back to the main branch after their peers and their security software approves the change.

Finding an appropriate peer group to perform code reviews can become a challenge. To aid in mitigating that challenge, the industry has created a pattern where project owners are added to the source code and automatically added to all code reviews.

A CODEOWNERS file (GitLab and GitHub) is a text-based file that indicates the owners and principles for either an entire repository or for sections of a repository. The file lets the platform automatically include named individuals in code reviews. GitHub introduced CODEOWNER files that were inspired by an internal Google code review process. In that process, a text file named OWNERS is used to list the responsible parties for different areas of the codebase. Adding CODEOWNERS functionality requires placing a file with valid syntax at the root of the code repository. Most code repository services automatically add owners to a pull request and sometimes automatically require acceptance during the review process. We recommend that CODEOWNERS files be created for all repositories, regardless of whether your current git repository service automatically includes responsible parties in code review.

Ensuring commit authenticity

A foundation of securing source code is the ability to authenticate contributor-added commits to the repository. Each of the major Git repository services has a security mechanism where contributors can digitally sign commits using asynchronous cryptographic key controls. These signatures are authenticated with publicly shared keys. By implementing a signed commits pattern and setting the security settings on your repository to disallow unsigned commits, you can verify the contributions to your code repositories.

Identify malicious code early

Nearly all modern programming languages and frameworks have tooling associated with code linting. Linting is the automated checking of your source code for programmatic and stylistic errors. Linting early in a CI/CD pipeline can identify easy-to-fix potential vulnerabilities derived from common syntax mistakes.

A linter, or static code analysis (SCAT) tool, only validates syntactic structure. It does not assess code functionality or identify logical errors. The syntax errors it identifies include potential vulnerabilities that result from unused variables, unreachable code, array index overruns, unused library references, and improper object reference usage that could lead to memory exploits. If you want to check for code functionality or logic errors, there are various open source and commercial automated testing tools that are triggered from a CI/CD job. These tools provide feedback that help determine whether to fail a build based on the feedback's severity.

Avoid exposing sensitive information

An often overlooked aspect of code security is the identification of code commits containing sensitive information. Sensitive information has a subjective scope and often includes, but is not limited to:

- Cryptographic keys
- Passwords
- Personally identifiable information (PII)
- Configuration secrets
- API keys or access tokens

Source code scanning tools designed to identify sensitive information are typically based on regular expressions. To avoid introducing sensitive information into code commit history, use pre-commit hooks. Pre-commit hooks are either local (client-side hooks) or remote (server-side hooks). These hooks can run software scripts as a result of commit events, or synchronizing events, such as fetch, pull, or push. If the scanning tool run by a pre-commit hook identifies sensitive information in the commit, the commit fails. The open source project Talisman is an example of using client-side hooks to help secure secrets. Hooks should be run server-side as well.

Local pre-commit hooks cannot be centrally enforced due to the distributed nature of Git. To be effective, a developer's environment needs to install, manage, and maintain the pre-commit hooks. Developers can circumvent pre-commit hooks, either inadvertently or maliciously, by failing to install client-side hook code, or by removing or replacing the client-side hook code. Regulating all of a developer's environments has historically been a challenge, especially when developing within a polyglot ecosystem. Most SaaS-based Git services do not provide server-side hooks. If you have a self-hosted Git instance, use server-side commit hooks to avoid circumventing commit-hook based enforcement policies. These policies prevent developers from introducing sensitive information to the repository.

Run the sensitive information scanner on commits made to all branches, including the primary and feature branches. If secrets are committed to history in a feature branch, mitigation should include rewriting history (rebase) and squashing merges after resolving the issue. Best practices for secrets management are covered in more detail in the Securing Secrets with Secrets Manager section.

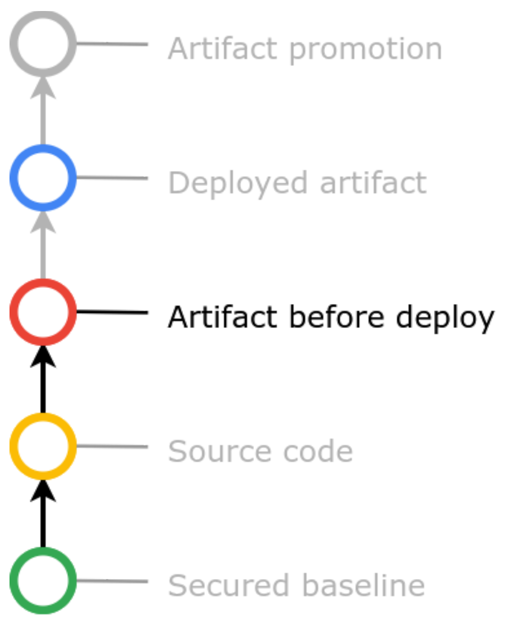
Logging and build output

CI/CD pipelines produce log files to help provide debugging and feedback information. Pay extra attention when deciding what logging output to generate from build jobs and script files. Consider which should be executed without any logging enabled. Secrets, PII, and sensitive information can be inadvertently leaked through logging output. To minimize risks, nearly all CI tools provide a masked process for hiding secrets embedded within a CI.

Alternatively, consider building scripts independently from the CI and executing them from within the build steps. Doing so means you won’t expose any secrets to your CI. The highest form of protection is to pipe information from one command to another without emitting sensitive sections to `stdout` or to log files.

License management

Software licenses indicate the acceptable usage and redistribution of code. Licenses are not strictly a security concern, but they do carry financial and compliance concerns. Ultimately these concerns become a security risk if a company is forced to expose software due to license restrictions. Copyleft, for example, prohibits proprietary software additions. At minimum, CI/CD pipelines should have a license collection tool to gather the open source software licenses of dependencies. If unwanted licenses are brought into the codebase, more sophisticated tools can use a shared policy to allow a pipeline to fail. Google offers an open source license scanner. Where possible, use license scanning tools that support policies in code defined by security stakeholders.



Securing artifacts before deployment

The previous sections discussed applying some techniques and tools to enhance the security posture of your source code. In this section, the focus shifts to the tools and techniques used to help secure an artifact before deployment into an environment. The goal is to identify vulnerabilities and threat risks to a composed application before introducing the component artifacts into any environment.

The 2019 Tripwire State of Security report notes that 60% of companies using containers say that they have experienced security-related incidents. Container vulnerabilities can result from open ports, elevated privileges, compromised base images, and application or library vulnerabilities, to name a few.

To help secure your artifacts before deployment, use various tools, processes, and techniques to transform source code into immutable and deployable artifacts consisting of application binaries and container images. The output from each subsequent step produces digital evidence to create a holistic, cryptographic-based attestation that represents artifact quality and security.

For example, once an artifact successfully passes all security tool scans, the system can create an attestation to represent that success. The Attestations, signers, and attestors section of this document is dedicated to the creation and usage of digital attestations.

Cloud-based application deployment has shifted from binary-based artifacts to artifacts based on a versioned Docker-based image format. The process of securing an immutable artifact starts by designing an artifact with only the minimal amount of access and required functionality.

Organizations should build artifacts with immutability in mind. Once built, the artifact's contents do not change when the artifact is deployed to a target environment. As an industry best practice, containerized applications should adhere to most, if not all, of the 12-factor app principles. After the initial release of the 12-factor app, additional factors, like immutability and a single codebase, were added in Beyond the 12-factor app. Immutability provides the foundation upon which artifact trust can be accrued, as attestations are accumulated through environmental promotions.

All Docker images are built atop a base image. Consequently, your image inherits any vulnerabilities associated with the operating system version of the base image. The two most commonly used Docker base images contain more than 500 known vulnerabilities. Enterprise policies often dictate that only known base images should be used. However, these policies are difficult to enforce. In fact, according to the Sysdig 2019 Container Usage report, 56% of private repository images are being deployed with known high CVEs.

A typical approach for building images involves two stages: building the binary, and configuring the binary to run when a container is started from the produced image. A mitigation for base image vulnerabilities is to implement a multi-stage Docker build which does not use the vulnerable base layer to create the image. The result is a runtime environment that is configured to only include the binary application and the contents of the final base layer.

Selecting a final base layer in a multi-stage Docker build can still introduce vulnerabilities, however. To mitigate this risk, the industry has produced base images with a bare minimum of operating system dependencies. For example, distroless base images.

Distroless base images

Distroless base images remove all non-essential operating system dependencies. This removal results in not only a thin final image, but also an artifact with a reduced number of threat vectors. Google maintains a suite of distroless base images. Organizations can apply this principled approach to the custom immutable image pattern using Google's managed images for Compute Engine VMs. An advantage of using distroless images is to avoid unnecessary binaries that an attacker might exploit. For example, removing `ssh` access and only exposing the application-specific ports reduces `ssh`-based attacks. If a container is compromised, eliminating shell access by removing `bash` significantly reduces an attacker's options.

Managed VM images

Enterprises are encouraged to develop their own immutable common base image or managed images. Images should be designed using automation tools, built with a CI/CD pipeline, and published to a private artifact repository. To provide consistent image builds that contain up-to-date OS patches and the latest libraries, use an image template designed with automated configuration tools. This pattern is often referred to as a *golden image*. You can find recommendations for creating base container images in the best practices for building containers guide and in the image management best practice documentation for VM images.

Artifact repositories

Artifact repositories originally provided a centralized hosting platform for shared dependency libraries and third-party libraries. Now, they often serve as a proxy for public repositories that caching public library systems with privately hosted repositories. Artifact repositories also provide a centralized source of managing libraries for your organization. This abstraction allows organizations to apply governing, privacy, and filtering policies for binaries according to their compliance needs.

Due to their central role in software deployment, consider artifact repositories a potential risk when performing a threat assessment. Because of the risk, they should be private-access only when possible. Give special attention to any publicly visible repositories or artifacts. Make sure to isolate and segregate public artifact repositories from your private repositories. Also, develop a deployment strategy for public artifact releases, if required by your business.

Many repositories provide consistency mechanisms, like hashes, to help ensure that artifacts have not been improperly altered. Even within repositories that digitally sign images, image references should use the image digest as a cryptographic signature of the artifact and a proof of integrity. If supported by the repository software, digitally signing artifacts using GNU Privacy Guard (GPG) or similar cryptographic software adds an extra layer of authenticity.

Access to publish repository artifacts should be configured with strict Identity Access Management (IAM) policies for read and write privileges. To avoid introducing compromised artifacts, only the CI/CD build service account should contribute to the repository. IAM policies that include both user and service accounts should control read access.

Artifacts should be versioned and considered immutable. Versioning artifacts provides a single reference because each version should have only one artifact.

Reusing an artifact version label is often called a snapshot build. The practice is typically adopted to reduce storage requirements. While convenient, snapshot artifacts are designed specifically for non-production use and should be avoided. There are more than three reasons to avoid them, but the following examples are representative:

- Avoid them due to the inherent complexities introduced by overwriting the existing artifact.
- Avoid them due to the inability to determine what functional content exists in the artifact.
- Avoid them due to the potential to inadvertently introduce vulnerabilities into an environment where more than one contribution pushes to the same snapshot artifact.

Google Cloud offers two products designed to manage the lifecycle of binary artifacts: Container Registry and Artifact Registry. Container Registry is purpose-built to provide both public and private storage exclusively for Docker-based images. Artifact Registry is purpose-built to store public and private Docker images, and application dependencies and libraries.

Both Artifact Registry and Container Registry have an embedded Container Analysis Scanner that can be configured to scan for vulnerabilities upon upload. Artifact Registry can manage encryption keys for stored artifacts. This feature lets you self-manage the encryption keys used to encrypt your artifacts. Discussed further in the policy management section, policies can be used with Open Policy Agent to restrict or allow container registries. For example, a policy can be configured to only allow the organization's private Container Registry and/or a specific public registry, like Docker Hub.

Container image analysis scanning

Container images published to an artifact repository can be scanned for CVEs. Scans that discover known CVEs produce metadata. That metadata is stored using the image digest. An image digest is a hash of the container contents. This image digest provides a cryptographic guarantee of container integrity and immutability. That guarantee removes any security concerns about altered contents.

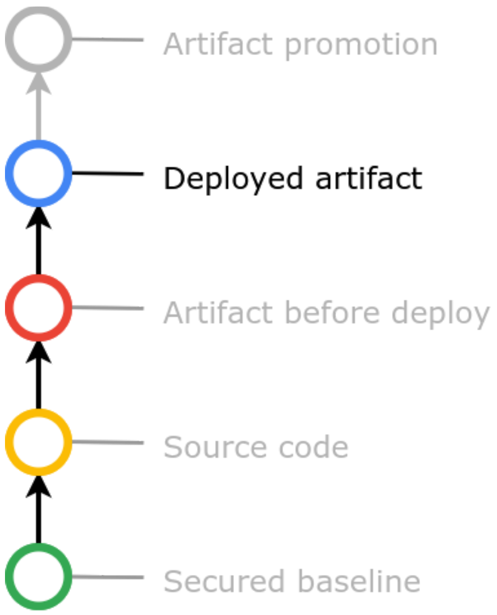
Grafeas is an open source project that provides a framework and a standard API for collecting, storing, and inspecting artifact metadata. Kritis is a companion tool created by the Grafeas project. It lets you query for metadata using an image digest and validate an image against a policy file. Policies can be configured, based on CVE severity or a list of allowed CVEs, to allow or disallow images. Integrating Kritis signer with Google Container Analysis is described in the Creating attestations with Kritis Signer guide.

Images pushed to Container Registry can be configured to be scanned automatically. The metadata of discovered CVEs can be stored with the Container Analysis API. The CI/CD pipeline should contain a stage to query the Container Analysis API and determine whether the scanned image meets the policy criteria. Following this pattern helps ensure that all images completing this CI/CD pipeline stage meet the organizational policy level for CVEs. Images that cannot be associated with the Container Analysis API, or do not pass the policy-based criteria, do not proceed. They cause the build pipeline to fail.

Application dependency analysis

Depending on the platform, language, and framework of your application, a container-level scanner might not discover all vulnerable dependencies within an application. To further assess the application dependencies, implement a library scanning build tool to help ensure that the build does not include vulnerable elements. For example, a Java application that compiles to Uber JAR, .NET, Maven, or Gradle can use the Open Web Application Security Project

(OWASP) dependency checker to validate dependencies. Tools also exist for Python (Safety) and for most modern frameworks that rely on external dependencies.



Securing deployed artifacts

At this stage in the pipeline, the immutable artifact is ready for deployment to an environment. Deploying immutable artifacts involves tools, processes, and techniques that add to, or deviate from, traditional deployment processes due to the static nature of the artifact. Challenges exist when injecting configuration, providing secrets, and addressing state. There are also access challenges when only service-specific ports are exposed. To reduce the likelihood of cross-feature functionality interference that can result in longer defect resolution times, test these configurations and state changes in isolation.

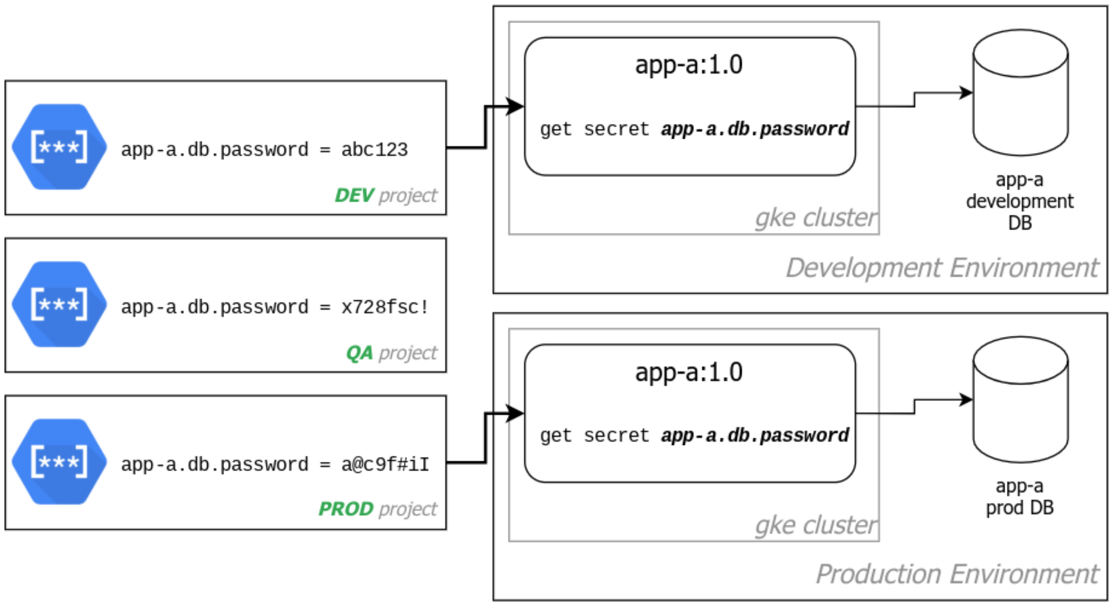
Securing secrets with Secret Manager

A secrets manager lets applications and pipelines access the values of named secrets based on permissions granted with IAM. Using a secrets manager, such as Google Secret Manager, not only provides an abstraction layer that safeguards secrets against unauthorized access, but also provides environment-specific values based on the same key.

When using a secrets manager, avoid environment prefixes for key labels. For example, the key labels `dev` and `qa` in `dev.db.password` and `qa.db.password`. While verbose, the labels are coupled to the application deployment environments, increasing complexity by requiring the application to be environment-aware. A pragmatic option is to create ephemeral and isolated testing environments.

A preferable secret storage technique to reduce naming complexity involves providing different values for the same secret based on the query environment. Applications can then make environment-independent references to the same secret without changing code or exposing environment variables.

In the following diagram, an application uses the same secret name, but the value is environment-dependent. This technique allows the application to remain immutable while still providing different, environment-specific values.



Entrusting secrets to a service that is external to your application prevents unintended secret leakage. Secret leakage often occurs through committing secrets to source code or by placing them in ad hoc tracking documents. Using Secrets Manager lets IAM grant or revoke secret access privileges. It also provides programmatic access to secrets so automated processes can access or mutate secret values. Furthermore, externalized secret tools help reduce security exposure. Security exposure can occur in several ways. For example, injecting values into environment variables that are inadvertently revealed is one common way to expose secrets. Another common way is when malicious software, included in a compromised Docker layer or system dependency, scans secrets from memory.

A high-quality secret manager provides centralized auditing. Centralized auditing has two main benefits. The first benefit is that focusing secrets in one location prevents those secrets from being distributed across your solution. Containing the secrets reduces the time to resolution when changing secret values during a compromise event or during a routine rotation. A secrets manager also provides an inventory of secrets, which simplifies automation. The second benefit is audit logs. When enabled, every interaction with Secrets Manager provides an audit trail. Use these audit trails to assist with forensics and compliance needs.

Artifact structure tests

As discussed earlier, building immutable images using distroless base images provides a solid security foundation for your image artifacts. Using artifact structure tests is a strong next step. Artifact structure tests focus on the structure of the finalized artifact image. These tools provide evidence that an image has the state and quality that policy guidelines require. These policy guidelines are usually established by operations, information security, and compliance teams.

Tests are run within an organization's CI/CD pipeline. The tests provide fast feedback that lets developers mitigate challenges before pushing images to a repository.

Docker container artifacts

Google has produced a container structure testing framework. Using a series of policies, it validates a container created from an image artifact. These policies are designed to inspect metadata, validate Docker layer output, and identify the presence or absence of files, executables, and folders within the container. A practical example is a set of checks that helps ensure that no libraries, other than the required artifact libraries and operating system libraries, are present. One approach would be to construct the artifact using a distroless image container as a base. Then create a container structure test to look for the presence of a few typical system binaries, like `ssh`, `bash`, or `sh`. This test validates that the intended image is configured correctly by checking that these system binaries are not present.

Virtual machine image artifacts

There are fewer tooling options for validating the structure of a VM image than for a container. To create artifact structure tests, organizations can use frameworks like KitchenCI, Gauntlet, or BDD Security. They can also be implemented with shell scripts using the Bash Automated Testing System (BATS) framework.

In addition to the physical aspects of security, security tests often extend to functional areas like network visibility, mTLS authentication, and checks for unneeded network port access. Tests might exercise authentication patterns with a RESTful API, scan ports to verify only necessary ports are open, and validate the hostname and configuration of network interfaces.

Dynamic application security tests

Dynamic application security testing (DAST) tests are designed to identify functional security vulnerabilities in deployed artifacts. Use DAST to identify possible exploits following known patterns of attack, like SQL injection, session field overloading, and cross-site scripting (XSS).

DAST tests are divided into two distinct methods: a scanner proxy or an application scanner. Scanner proxies are the most common DAST tool category. Placed between the client and the application endpoint, scanner proxies intercept the incoming and outgoing traffic as that traffic travels between the client and application. The intercepted traffic is used to evaluate potential vulnerabilities, like injection attacks, XSS, and insecure deserialization. A common pattern is to use the proxy to intercept traffic generated by an application's browser-level automated testing suite. This method removes the need to maintain two suites of browser-level test suites for generating traffic against the application. The OWASP ZAP proxy is an example of a scanner proxy method.

Application scanners crawl an application's exposed endpoints with known URL patterns, provided URL pathing configuration, or a dynamically generated tree structure that includes new endpoints as they are discovered. Arachni Scanner is an example of a dynamic crawling scanner. Crawler-based DAST tools implement fuzzy inputs that test known combinations of words, characters, and sequences that often trigger OWASP-listed vulnerabilities. One example of this technique is to query input parameters with SQL statement segments to identify code vulnerable to non-escaped input sequences.

Use a DAST tool immediately after deploying or starting an application, typically in an isolated testing environment. In a scenario where speed is required over completeness, use a crawling DAST tool. Where completeness is required over speed, use a proxy method paired with an automated user interface testing tool like Geb, Test Robot, or Selenium. There is no hard requirement that you select one DAST method or the other. It's common to apply the crawler method before deploying to an initial test environment, and to apply the proxy method before deploying to production, or to a canary production.

Deploying code after DAST testing can instill more confidence that the application is free of known vulnerabilities and common software patterns that expose it to unidentified vulnerabilities.

Isolated testing environments

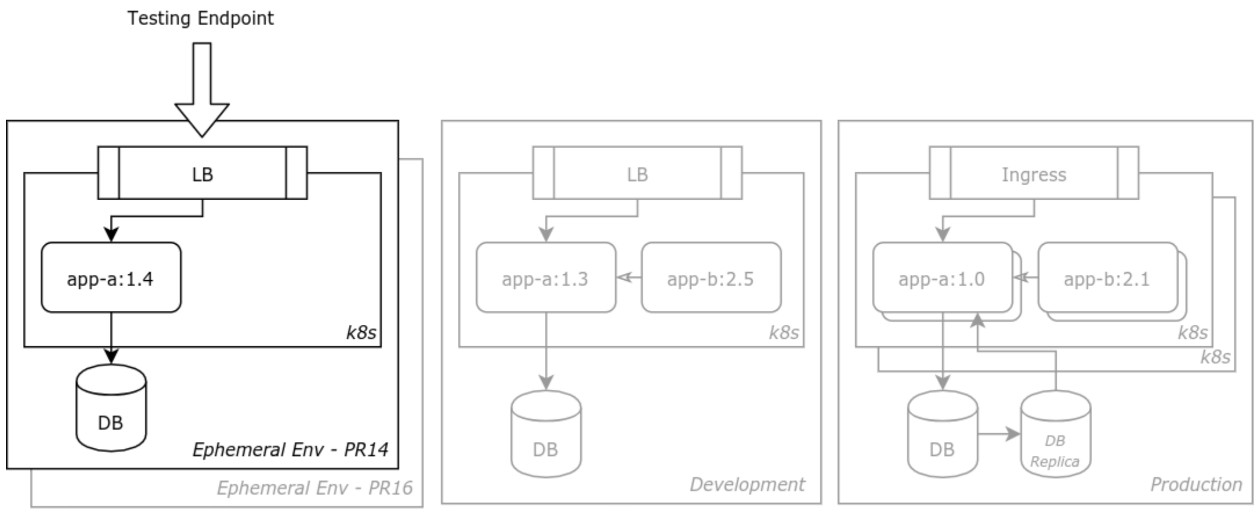
Progressive development, testing, staging, and production environments provide the platform for your organization's software value chain. Despite the tools and scripts intended to maintain parity between these largely static environments, environment inconsistency is not uncommon. In addition to the complexity of maintaining consistency between environments, development teams are often required to share environments for feature functionality releases. Frictions arise when new features have overlapping functionality or conflicting dependencies. This overlap can lead to logistical complexity. Organizations might be forced to progressively schedule their deployment of static environments by release version.

Static environments are challenging. One solution is to use the elastic nature of cloud platforms to create isolated testing environments. These isolated testing environments consist either of many environments or of ephemeral environments that are created on-demand. To create environments where you can test new functionality in isolation, use Declarative infrastructure. These environments are ephemeral. They only live as long as the feature test cycle is active. An environment can be created for a single feature request or a coordinated series of feature requests. After the ephemeral environment has served its purpose, the environment is removed and the build pipeline proceeds to the next stage. A common pattern is to create an ephemeral, isolated, environment specifically as a target for a single DAST execution. Isolation prevents coincident changes—like data mutation from concurrent tests—from influencing the outcome of the test run.

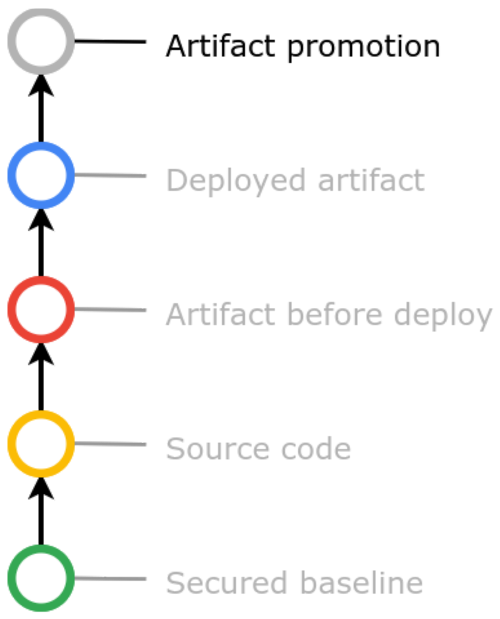
Isolated testing environments can be small or large. They can range from a locally running container inside the CI build pipeline, to a complete production environment clone using declarative infrastructure code.

In addition to testing new functionality, ephemeral environments are useful for debugging or security vulnerability testing. Combining IaC with immutable and versioned artifacts helps with recreating environments to test discovered vulnerabilities, their impacts, and fixes for the defect.

The following diagram shows an ephemeral environment created to isolate functionality built on pull request branches. The isolated environments are short lived, in contrast to the longer running Development and Production environments.



An often overlooked benefit of isolated testing environments is that they can provide a locale for isolated feature-functionality verification. Isolated environments provide stakeholders with a direct link to verify the functionality of a given feature request. While this aspect of an isolated environment is intended to verify business features and is not directly security-related, using isolated environments adds to the overall value of ephemeral environments. Designing a system to create ephemeral environments is an investment. Having multiple use cases reinforces the business justification for that investment.



Securing artifact promotions

Artifacts are usually promoted from one environment to another to minimally reduce regressions and to ensure basic functionality before exposing the artifact to the consumer. Throughout this process, trust is gained when verifying functionality in one environment to the next. Capturing this trust requires the recognition of implicit trust, the introduction of tools to capture the trust, and the use of additional processes to leverage the trust associated with the finalized artifact.

Upon reaching the final stage in the SDLC, the immutable trusted artifact is available for deployment to a client or production environment. Verifiable trust has been accumulated through the rigors and stress tests of the preceding pipeline stages. The trusted artifact has been verified in this progressive series of CI/CD environments; each environment has goals and qualifications to pass validation. Each validation produces digital records attesting to its successful completion. The ultimate goal is a comprehensive validation for production environment deployment. This final step of the artifact's value chain is known as artifact promotion.

Some of the technical tooling described in this section is specific to container-based artifacts. VM deployment pipelines can use the same principles to obtain similar outcomes. For example, adopting a resource labeling system as a substitute for a metadata tracking system like Grafea would work to emulate the digital attestation solution.

Attestations, signers, and attestors

Team members have historically held the primary responsibility to manually decide whether a particular build was functionally correct and regression-free. They were also responsible for endorsing their trust in a given piece of software. Today, automation replaces the manual, repetitive processes once performed by QA or security team members. Evidence of artifact trust is digitally gathered and cryptographically signed.

Attestations

By definition, an attestation is cryptographic evidence or proof of an observation. In the context of a CI/CD pipeline, the proof or evidence is related to a pipeline event result. For example, the successful completion of a test suite run, or the successful results of a satisfactory container scan.

With binary authorization, a structured text document that includes the container image digest manifests the attestation.

Signers

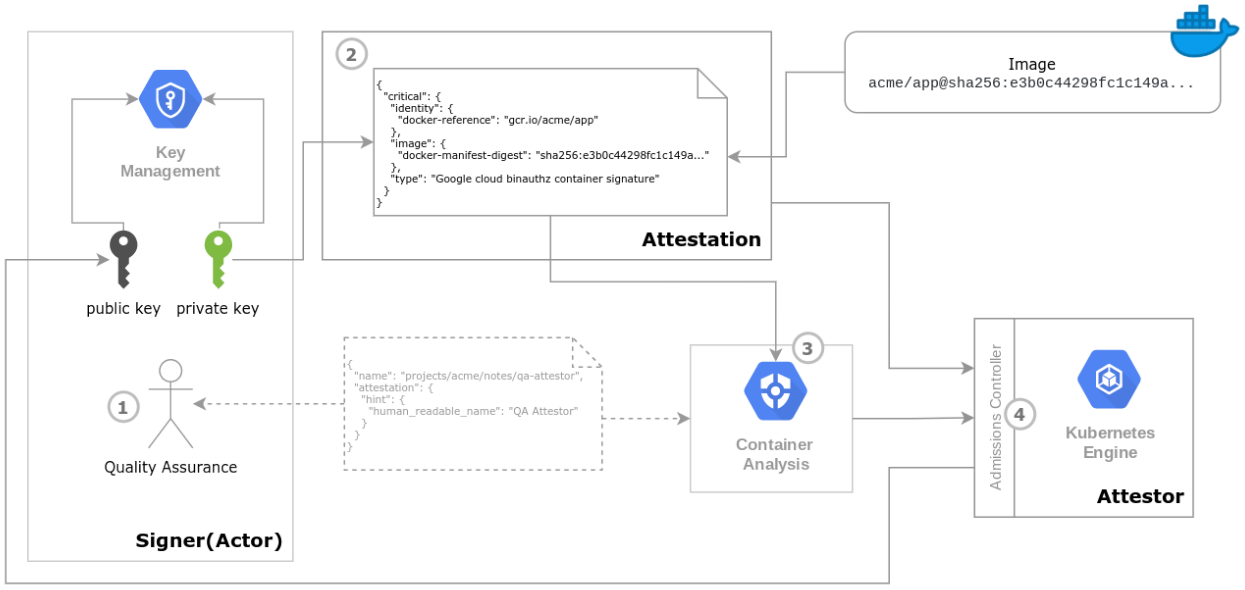
The entity that uses a private key to cryptographically sign artifact evidence is known as a Signer. A signer can digitally create and cryptographically sign evidence through manual or automated tooling. This signed digital evidence is the attestation.

One implementation pattern using signers is to digitally represent the manual trust chain represented in the build process. For example, a QA team member (signer) is responsible for attesting that the target build does not produce regression defects. The quality assurance actor can be digitized as a signer and the associated encryption keys can be used to digitally sign notes representing the success of a non-regressed build.

Attestors

An attestor is a process that uses the Signer's public key to verify an attestation. Typically, the verification is undertaken before some action is allowed to occur. With binary authorization, the attestor is used at deployment to control which container images are deployed. Only images with an accompanying and verifiable attestation, created before deployment, are permitted.

The following diagram illustrates how an attestation is created. A signer (1) provides the private key through the Cloud Key Management Service (KMS) to digitally sign a note memorializing a moment or event. This action creates an attestation (1+2). The attestation is stored in the metadata server contained within the Container Analysis Service (3). At deployment time, the attestor verifies the attestation (4) before deployment can be performed.



```
{
  "critical": {
    "identity": {
      "docker-reference": "gcr.io/acme/app"
    },
    "image": {
      "docker-manifest-digest": "sha256:e3b0c44298fc1c149a..."
    },
    "type": "Google cloud binauthz container signature"
  }
}
```

```
{
  "name": "projects/acme/notes/qa-attestor",
  "attestation": {
    "hint": {
      "human_readable_name": "QA Attestor"
    }
  }
}
```

Binary authorization

Evidence of successful events collected as attestations can be aggregated to represent the overall quality, security, and functional completeness of the artifact. Google's Binary Authorization service is a deploy-time security control. It helps ensure that only trusted container images are deployed. The binary authorization toolset consists of a Kubernetes cluster, an admission controller, a structured policy document protocol, and programmatic access to the digital attestations associated with a given container. Policies can be created to either allow or deny deployment based on the presence or absence of attestations. Policies are applied at a cluster level, so different policies can be established for different environments. This distinction allows for progressive attestation requirements as environments get closer to production.

While CI/CD pipelines can automate the process, codify trust, and install higher confidence, they are not without risk. Since artifacts are pulled from a repository external to the pipeline, there is a potential, without additional security checks, that a compromised artifact can be deployed. binary authorization policies are designed to mitigate this risk and should be used to restrict deployments to those only coming from a known list of repositories.

A key principle in securing pipelines is to focus on accumulating trust for the given artifact progressing through the pipeline. Attestations are the evidence used to validate the trust gained throughout the pipeline. These specifically formatted text documents that include the Docker image digest are created throughout the pipeline. Each of the documents indicate the success or failure of some event, such as an automated test suite completion. Acquiring these attestations results in a verifiable chain of trust that enables an informed decision about deploying an artifact to a particular cluster or environment.

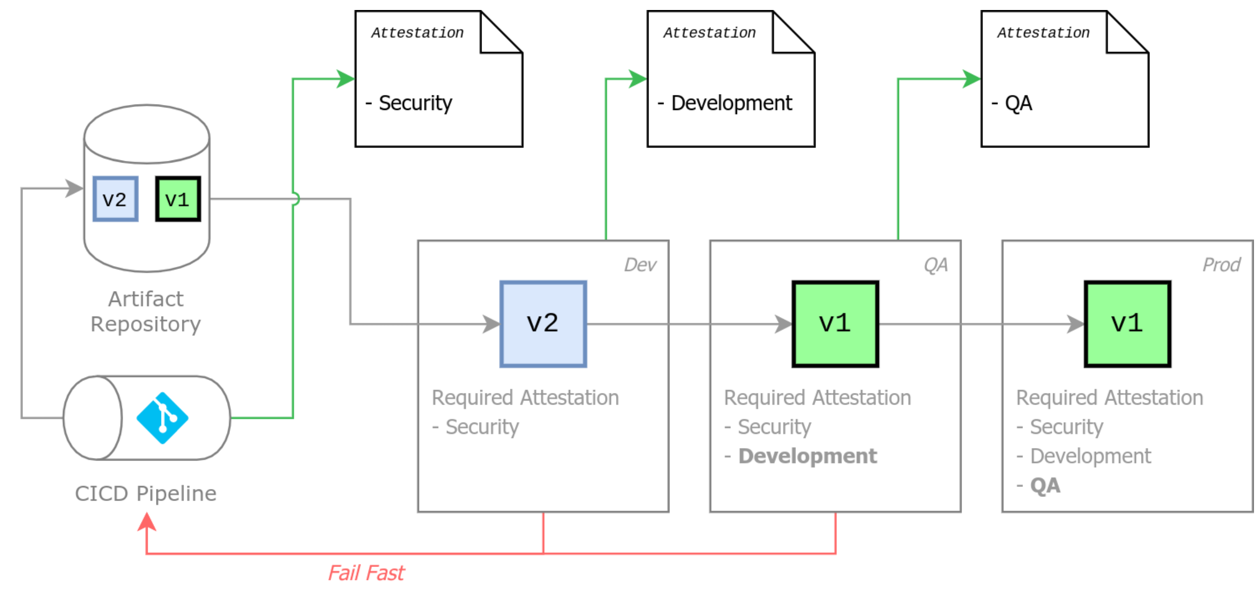
Artifact promotion

A common SDLC pattern starts with developing in a developer's environment, building an artifact, and pushing the artifact to a series of environments to perform various tests and validations. After the system determines that the goals established for an artifact in each environment have been met, it creates an attestation. The attestation memorializes the trust established when the goals in each environment are met. Informed from the evidence of trust accumulated through the previously completed stages, the artifact is promoted through progressive environments according to manual or automated decision processes. The ultimate goal is a production environment deployment.

When manually verified, trust is often conveyed through ad hoc communication attesting that the build is ready to proceed to the next environment. For example, an email might be sent to notify downstream users that the artifact passed validation. A loosely coupled communication pattern requires the verifier to provide verbose context, like relevant build information or the history of the artifact. Of course, manual processes are inherently error prone due to the potential for communication breakdowns, employee churn, vague feelings of doneness, or the need—for any arbitrary reason—to release a new version of the software.

Instead of relying on ad hoc human communication or configuration parameters to build and deploy, create attestations for each stage and/or signer responsible for the validation at each environment. For example, a QA team member can digitally sign off on a build after verifying functionality in a QA environment, while a product owner can digitally sign off in a staging environment. Applying policies that require this progressive series of signed attestations protects environments from accidental, or malicious, artifact deployments.

The following diagram illustrates the path of a versioned artifact, labeled v2, after it has completed the CI/CD pipeline process and has a security attestation automatically created for it.



In the preceding diagram, v2 has been deployed from the Artifact Repository to a development environment, replacing the previously deployed v1 artifact. In this environment, an automated tool or committee assesses the artifact quality. If these tests pass, the automated tool or committee triggers an automated sign-off attestation. A similar, though likely more rigorous, process would then be repeated for the QA environment. The final step relies upon the collected attestations and deploys the v2 artifact to the production environment. If all the previous environment completions produce the necessary artifact attestations, the production environment accepts the deployment, and replaces the v1 artifact with the v2 artifact.

Fully automated builds are designed as pipelines and can be thought of as a workflow or as a series of orchestrated events. Pipelines are often not designed with the idea of trust-building. Pipelines can be configured—based on events—to automatically push builds to any environment, including production. Thus, there is a need to capture and use the trust that is established through artifact promotion.

Policy management

A growing trend in declarative infrastructure is to codify company governance and compliance policies into a digital, code-like, format. This concept is equivalent to x-as-code philosophies, where software code, and structured data documents, are used to describe the declared state or configuration for a given system. For policy management, the common industry term is p*olicy as code (PaC)*.

Policy management code provides a transparent and portable digital method to enforce policies. It also gives an operator the ability to test, review, version, and automate changes to policies across the entire SDLC.

Policies are used to encode business or security decisions as configuration. These decisions are often used to allow or disallow an action, like deploying an artifact or failing a build. A relevant example would be a policy that requires an artifact to obtain all necessary attestations before it can be deployed.

Within the Kubernetes ecosystem, the use of Open Policy Agent (OPA) has been adopted to serve as both the policy protocol and the enforcement of policies. OPA is a generic policy ecosystem designed to provide the building blocks for implementing a flexible and comprehensive policy management framework that scales. For Kubernetes, policies are built using the Rego query language and exposed using Kubernetes resource model objects.

There are two primary methods to implement OPA-based policy enforcement: imperatively and at deployment. With the imperative method, a CI/CD pipeline stage uses Google's Kubernetes Packaging Toolkit (kpt) functions to validate the compliance of all Kubernetes Resource Model (KRM) resources within a change set. The benefit in using the imperative function is that it can fail fast rather than waiting until deployment. In the deployment method, the ability to deploy or modify cluster state is provided by a Kubernetes operator called Gatekeeper. Gatekeeper is an admissions controller that vets, for policy compliance, any changes to the cluster state. Google provides a library of common policies that can be configured for use in either method. Refer to Validating apps against company policies in a CI pipeline for additional context and an example implementation.

Establish consistent thresholds

Policy management tools often establish multi-tiered threshold levels and various runtime enforcement modes. Thresholds are a magnitude or intensity (bounded by lower and upper levels) of a measurable element. Thresholds can be categorized as a single bound, but are often a tiered, enumerated, series of thresholds. For example, “low, medium, high,” or “debug, info, warn, error.” Thresholds typically group a range of values on a continuum into more manageable categories that policies can use.

To help ensure consistency across applications and services, especially when working within a polyglot ecosystem, organizations should define categorization levels. What do these categories mean from a semantic and pragmatic perspective for the threshold levels of each tool? An example would be defining the impacts of `LOW`, `MEDIUM`, and `HIGH` levels for the Container Analysis API. In the Container Analysis API, CVEs are categorized based on their Common Vulnerability Scoring System (CVSS) score. An organization can then decide the risk impact of each level and codify that decision into a policy document used at deploy time.

The following code sample is an example of a policy document:

```
apiVersion: kritis.grafeas.io/v1beta1
kind: VulnzSigningPolicy
metadata:
 name: vulnerability-policy
spec:
 imageVulnerabilityRequirements:
 maximumFixableSeverity: LOW
 maximumUnfixableSeverity: LOW
 allowlistCVEs:
 - projects/goog-vulnz/notes/CVE-2020-10543
 - projects/goog-vulnz/notes/CVE-2020-1751
```

Runtime enforcement modes

Policy management tools can be run in different runtime modes that influence both the level of policy enforcement and the subsequent actions taken when issues are identified. Policy enforcement can range from drastic enforcement actions, like denial of deployment, or light enforcement actions, like logging the issue and continuing. In the latter scenario, violations are logged but no punitive actions are taken. The deployment is allowed to proceed.

Policy management tools that enable users to customize thresholds and runtime enforcement modes provide you with operational flexibility. For instance, when designing a break glass process, configuring different modes that temporarily lower the enforcement of policies during specific scenarios can be useful.

External policy repositories

Policies should be stored in code repositories that are independent of the application source code repository. They should follow the same CI/CD patterns outlined in this document. Policies have a broad reach over environments and software applications. As an example, a single policy file can be created and applied to restrict network access to applications, systems, and whole environments. A defect, vulnerability, or misconfiguration of that policy file can affect the operation of many resources.

Creating an independent policy source repository allows policies to be treated like all other software code. Policies should follow standard versioning and deployment techniques. The policy source code repository should also include an automated suite of tests. Automated testing helps ensure that policy changes produce the intended functionality and reduce the risks that policy changes might negatively impact production environments. Any defects discovered in policies should be resolved and, according to standard CI/CD principles, respective test cases should be added. Adding test cases helps ensure that functionality does not regress in future releases.

Policies that are outside of an application's code repository must be pulled into the CI/CD pipeline at the appropriate pipeline stage. As an example, a pipeline can use a kpt function with public function validators to fetch the public policy manager constraint library. The resource configuration toolkit can apply the policies against the application's Kubernetes resource files that are stored in the repository.

Fail open or fail closed

When crafting a testing suite for policies and a release cycle, consider how to handle policy and security control failures. If a policy or a security control fails, design the solution to either fail open or fail closed according to your specific requirements. The terms typically relate to how an application should behave when it encounters errors and exceptions. For example, if the Container Analysis APIs were not available, would the system allow the artifact to advance (fail open), or would it deny the artifact and keep it from advancing (fail closed). Carefully consider whether a particular part of a system defaults to allow or deny when a failure occurs.

Artifact deployment policies

Binary artifact deployment policies can be created to define the response to the presence or absence of attestations. An environment's clusters might reject artifacts that fail to conform with an existing deployment policy. The admissions controllers use the attestations associated with the image digest and compare with the cluster's policy to approve or reject the image from the cluster. These restrictions are called admission rules:

```
name: projects/example-project/policy
globalPolicyEvaluationMode: ENABLE
defaultAdmissionRule:
 evaluationMode: REQUIRE_ATTESTATION
 enforcementMode: ENFORCED_BLOCK_AND_AUDIT_LOG
 requireAttestationsBy:
 - projects/example-project/attestors/secure-build
```

Applying a policy for each cluster creates scale challenges as the number of clusters increase. Binary authorization policies have two types of admission rule sets: default rules and cluster rules. The default is required, and is used when a policy is not matched in a cluster-specific rule set. Defining specific cluster policies can reduce the need to maintain many policies. That's because one policy file can contain the rules for multiple policies. This approach is recommended for clusters within the same environment, but not for cross environments. Review the Google Cloud policy YAML reference documentation for policy semantics and structure.

Configure policy files for each environment. Only allow artifacts that correspond to the required attestations for that environment level. When you use this pattern, artifacts that have not been through an earlier pipeline stage environment cannot be accidentally deployed to a later stage environment. For example, policies for a QA environment would require an attestation stating that the QA checks have passed, whereas the policies for the development environment would not require those checks. To allow for experiments and for the development of new software, consider letting development or experimental clusters deploy containers from a series of less restrictive container repositories. On the contrary, require that QA, Stage, or Production environments only pull containers from vetted container repositories.

In cases when binary authorization is newly added to an existing environment, or the container supply chain is not verified, you can run clusters in a dry-run enforcement mode. In this mode, rejections are logged but not acted upon. To specify how policies would be enforced without actual enforcement, enable the dry-run mode in the policy file. Doing so reduces the risk of introducing blocking changes. Dry-run policy enforcement is a viable option for a development environment or for a low-risk environment where flexibility is needed.

Infrastructure policy enforcement

Policy management applies to all stages of a CI/CD pipeline. Using IaC implemented as a CI/CD pipeline is a best practice. Doing so lets you enforce PaC against pre-deployed infrastructure. See Validating apps against company policies in a CI pipeline to learn more.

Research from DORA shows how teams are shifting left to support security alongside quality, operations, and development. In CI/CD pipelines, best practices indicate that security should be implemented using policies that fail fast. For example, you can configure your policies to identify and stop a build for any discovered vulnerabilities, exposures, or violations before the artifact can be deployed.

Code-based policy best practices

Tools used to enforce software-based security often have an associated configuration file that lets you either establish multiple thresholds or ignore features. This configuration provides the flexibility needed to establish and maintain a security baseline. Where possible, use tools that can establish threshold levels of at least WARN and FATAL, where FATAL thresholds would result in a negative decision.

As important as policies and thresholds are, having out-of-date policies or thresholds are often more detrimental. Be sure to schedule time to reassess your policies and their thresholds over time as applications, threats, and technologies change.

Break glass

One drawback to having a fully automated and comprehensive delivery process is that security and quality are prioritized over speed and variability. What happens when a vulnerability is found, a fix is known, and a rapid response is needed? Site reliability engineering principles state that all solutions should have a planned break-glass process. In this process, tests are run with a high priority and with a smaller set of policies to rapidly push through a change.

Break glass does not mean removing all processes. It strictly allows for lower-priority services to be lifted temporarily: either for a short time, or for a single action. Immediately upon resolution, the circumvented path should return back to the original secure pattern. Apply updates to the pattern if the pipeline was the root cause that required breaking glass.

One break-glass solution involves a CI/CD pipeline triggered by a Git tag. Modern CI/CD platform tools can be configured to protect tag creation. Using this mechanism, a break-glass solution can be protected and only run by a known list of individuals. As a benefit, all break-glass builds have an audit trail in the Git repository history.

Conclusion

The key takeaway from this document is to design a code pipeline that accumulates trust throughout the progression of the build. Trust is the foundation of any overall strategy to help secure your organization.

- Trust is accumulated by addressing code at the five primary build layers and using cryptographic tools to digitally attest to security assessments made along the build timeline.
- Every action taken in a pipeline should be used to increase the trust or fail-fast; either result provides valuable feedback about application security.
- Establishing a validated chain of trust requires the use of immutable artifacts that have been built using a series of layered security tools to remove known vulnerabilities and bad software coding practices.
- Trust accumulates as the artifact is promoted through progressive environments.
- Policy-based tooling is used to accept artifacts into or reject artifacts from an environment based on the presence or absence of accumulated trust.
- Full automation is critical to orchestrating and securing a CI/CD pipeline.

Google has a wide variety of tools to address all five primary build layers using many popular software languages and frameworks.