Google Cloud

White paper
January 2022

Delivering software securely

Table of Contents

Chapter

01 Understanding the software supply chain

Chapter

02 Industry-wide standards and frameworks

Table of Contents

Chapter

03 Managed services for each stage

Chapter

04 Getting started

Chapter

05 Further reading

Chapter

01 Understanding the software supply chain

Current security landscape

Speed and time to market are top priorities for organizations everywhere that build software and applications to meet their customers' needs. These strategic imperatives have been the driving force behind the tremendous growth of containers as the platform of choice. Over the past year, having reaped the benefits of containers, which include faster time to market, higher availability, improved security, better scalability, and reduced costs, many of these organizations have started thinking about the serverless approach as well.

While software solutions have reduced the time it takes to deliver a new feature or even a new product, many of the existing security practices are unable to keep up with the increase in velocity, leading to one of three problems:

Developers are slowed down by existing processes, resulting in delays

Security and operations teams make compromises that open the organization up to threats

Development teams work around existing security processes to meet deadlines, making them vulnerable

The last few years have seen a slew of security breaches classified as “software supply chain” attacks.

Log4Shell was a dangerous vulnerability in Apache Log4j software identified in December 2021. Flagged with the maximum CVSS score of 10, this vulnerability was particularly devastating because of the popularity of Log4j, a Java-based logging framework. Two things contributed to the severity: first, it was very easy to exploit and allowed for full remote code execution, and second, it was often many layers deep in the dependency tree and therefore easily missed.

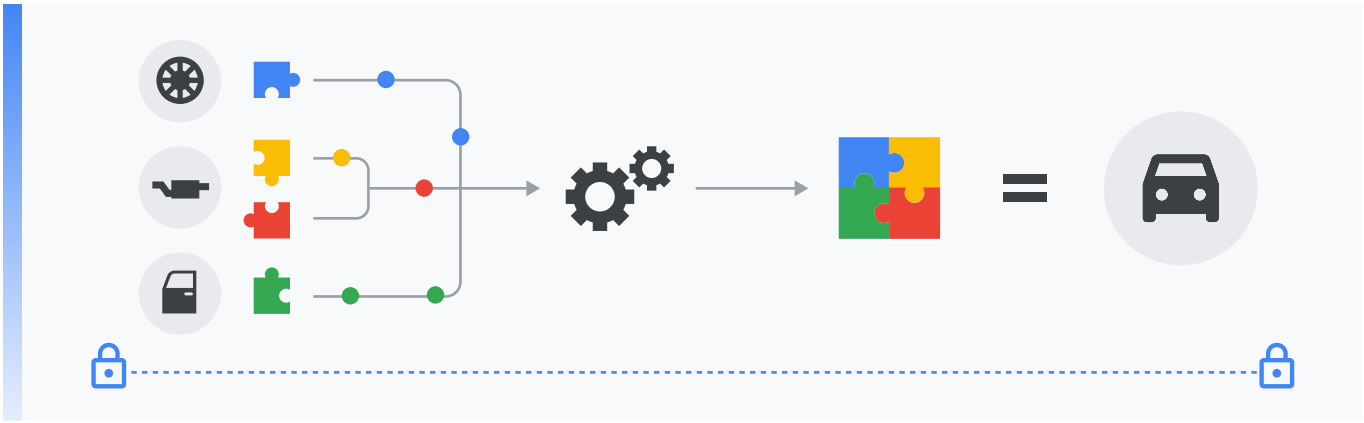
Solarwinds, an IT management software firm, was attacked by nation-state actors who injected malicious code into official builds of open-source software in use at the company. This malicious update was pushed to 18,000 customers, including the U.S. Treasury and Commerce Departments.

Kaseya, another IT management software provider, was attacked via a zero-day vulnerability that compromised the Kaseya VSA server and sent a malicious script to deliver ransomware that encrypted all files on the affected systems.

The urgent need to respond to these and other similar incidents led the White House to release an Executive Order in May 2021 requiring organizations that do business with the federal government to maintain certain standards of software security.

The software supply chain

In many ways, the term “software supply chain” is very appropriate: the processes that go into creating a software supply chain are very similar to those that go into manufacturing a car.

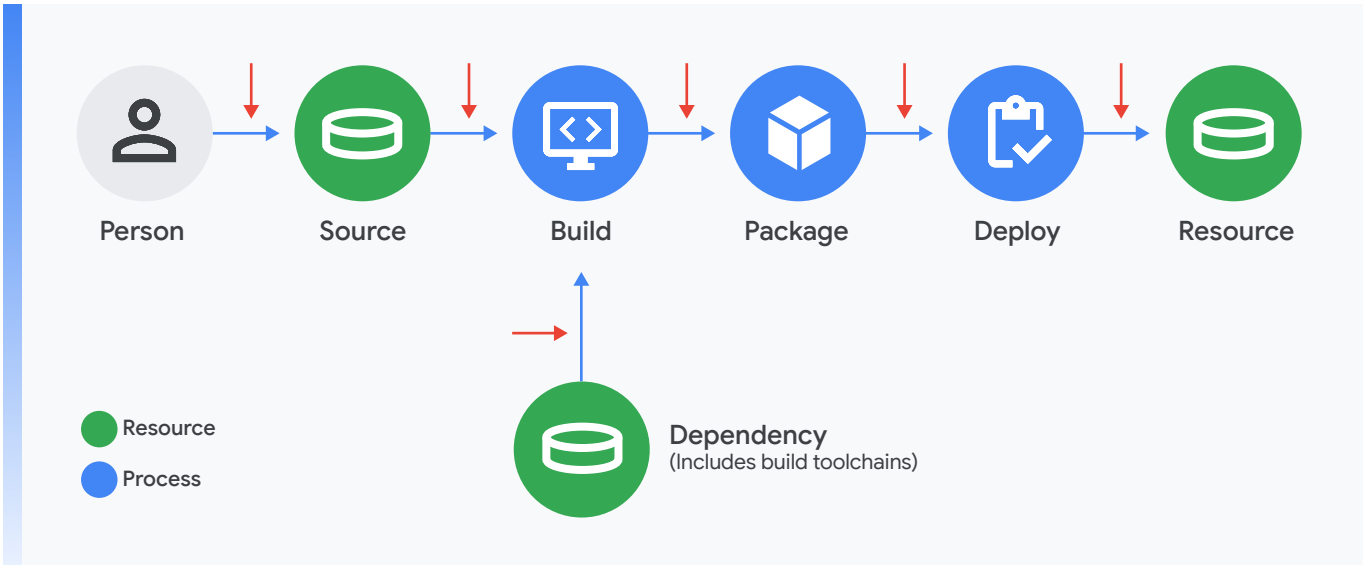


A car manufacturer obtains various off-the-shelf parts, manufactures its own proprietary components, and then puts them all together using a heavily automated process. The manufacturer ensures the security of its operations by making sure each third-party component comes from a trusted source. First-party components are tested extensively to ensure they don't have security issues. And finally, assembly is carried out through a trusted process that results in finished cars.

The software supply chain is similar in many ways. A software manufacturer obtains third-party components often open source in nature that perform specific functions and develops its own software, which is its core intellectual property. The code is then run through a build process that combines these components into deployable artifacts, which are then deployed into production.

The weak link in the chain

It only takes one unsecured link to breach the software supply chain

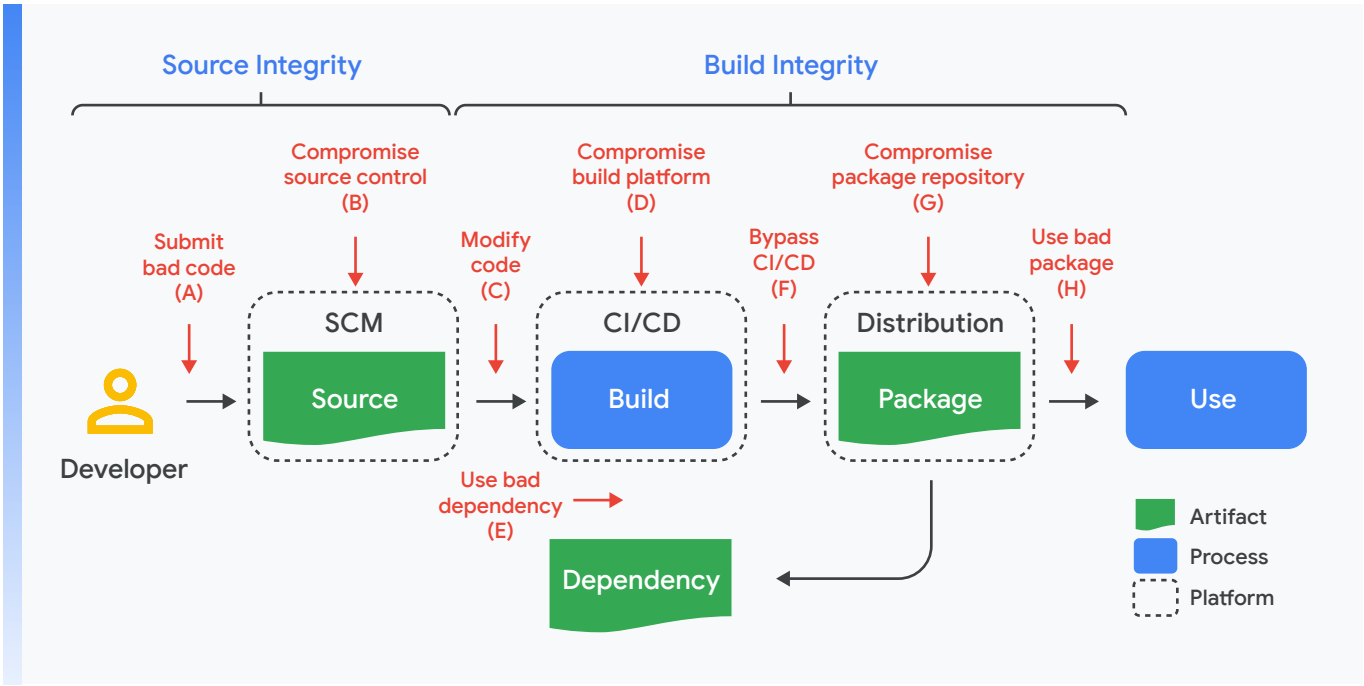


As in the case of the high-profile attacks last year, each of the steps in the process can lead to a weakness attackers can exploit.

For example, the average npm package has 12 direct dependencies and about 300 indirect dependencies. In addition, we know that nearly 40% of all published npm packages depend on code with known vulnerabilities.

Those vulnerabilities might not actually make the code unsafe—for instance, the vulnerability might be in a part of the library that's never actually used. But these vulnerabilities still must be checked.

The scale of this problem is monumental. If even one of these vulnerabilities were to go unpatched, it could provide an opportunity for bad actors to gain entry to your software supply chain.



Here are a few examples of attacks that leveraged vulnerabilities in each of these stages depicted in the diagram above.

	Threat	Known example
A	Submit bad code to the source repository	Linux hypocrite commits: Researcher attempted to intentionally introduce vulnerabilities into the Linux kernel via patches on the mailing list.
B	Compromise source control platform	PHP: Attacker compromised PHP's self-hosted git server and injected two malicious commits.
C	Build with official process but from code not matching source control	Webmin: Attacker modified the build infrastructure to use source files not matching source control.
D	Compromise build platform	SolarWinds: Attacker compromised the build platform and installed an implant to inject malicious behavior during each build.
E	Use bad dependency (i.e., A-H, recursively)	event-stream: Attacker added an innocuous dependency and then updated the dependency to add malicious behavior. The update did not match the code submitted to GitHub (i.e., attack F).
F	Upload an artifact that was not built by the CI/CD system	CodeCov: Attacker used leaked credentials to upload a malicious artifact to a GCS bucket, from which users download directly.
G	Compromise package repository	Attacks on Package Mirrors: Researcher ran mirrors for several popular package repositories, which could have been used to serve malicious packages.
H	Trick consumer into using bad package	Browserify typosquatting: Attacker uploaded a malicious package with a similar name as the original.

Making the chain stronger: Google Cloud's thought leadership in open source

At Google, we have been building global-scale applications for decades. Over time we have open sourced many of our internal projects to help increase developer velocity. At the same time, we developed various internal processes to help secure the software experience.

Here are some of the efforts we are involved in to make software supply chains stronger everywhere.

Increased Investment – We announced in August 2020 that we will invest $10 billion over the next five years to strengthen cybersecurity, including expanding zero-trust programs, helping secure the software supply chain, and enhancing open-source security.

Supply-chain Levels for Software Artifacts (SLSA) – SLSA is an end-to-end framework for supply chain integrity. It is an open-source equivalent of many of the processes we have been implementing internally at Google. SLSA provides an auditable provenance of what was built and how.

DevOps Research and Assessment (DORA) – Our DORA team conducted a seven-year research program, validating a number of technical, process, measurement, and cultural capabilities that drive higher software delivery and organizational performance.

Open Source Security Foundation – We co-founded Open Source Security Foundation in 2019, a cross-industry forum on supply chain security.

Allstar – Allstar is a GitHub App installed on organizations or repositories to set and enforce security policies. This allows for continuous enforcement of security best practices for GitHub projects.

Open Source Scorecards – Scorecards use evaluation metrics like well-defined security policy, code review process, and continuous test coverage with fuzzing and static code analysis tools to provide a risk score for open-source projects.

We believe two things are necessary to overcome the problem of software supply chain security:

1. Industry-wide standards and frameworks.
2. Managed services that implement these standards using principles of least privilege in a zero-trust architecture. A zero-trust architecture is one in which no person, device, or network enjoys inherent trust; instead, all trust, which allows access to information, must be earned.

Let's look into these one by one:

Chapter

02 Industry-wide standards and frameworks

To understand the principles that go into securing the software supply chain, let's start with SLSA.

In its current state, SLSA is a set of incrementally adoptable security guidelines being established by industry consensus. In its final form, SLSA will differ from a list of best practices in its enforceability: it will support the automatic creation of auditable metadata that can be fed into policy engines to give "SLSA certification" to a particular package or build platform.

SLSA is designed to be incremental and actionable and to provide security benefits at every step. Once an artifact qualifies at the highest level, consumers can have confidence that it has not been tampered with and can be securely traced back to its source—something that is difficult, if not impossible, to do with most software today.

SLSA consists of four levels, with SLSA 4 representing the ideal end state. The lower levels represent incremental milestones with corresponding incremental integrity guarantees. The requirements are currently defined as follows:

SLSA 1 requires that the build process be fully scripted/automated and generate provenance. Provenance is metadata about how an artifact was built, including the build process, top-level source, and dependencies. Knowing the provenance allows software consumers to make risk-based security decisions. Though provenance at SLSA 1 does not protect against tampering, it offers a basic level of code source identification and may aid in vulnerability management.

SLSA 2 requires using version control and a hosted build service that generates authenticated provenance. These additional requirements give the consumer greater confidence in the origin of the software. At this level, the provenance prevents tampering to the extent that the build service is trusted. SLSA 2 also provides an easy upgrade path to SLSA 3.

SLSA 3 further requires that the source and build platforms meet specific standards to guarantee the auditability of the source and the integrity of the provenance, respectively. SLSA 3 provides much stronger protections against tampering than earlier levels by preventing specific classes of threats, such as cross-build contamination.

SLSA 4 is currently the highest level, requiring two-person review of all changes and a hermetic, reproducible build process. Two-person review is an industry best practice for catching mistakes and deterring bad behavior. Hermetic builds guarantee that the provenance's list of dependencies is complete. Reproducible builds, though not strictly required, provide many auditability and reliability benefits. Overall, SLSA 4 gives the consumer a high degree of confidence that the software has not been tampered with. More details on these proposed levels can be found in the GitHub repository, including the corresponding Source and Build/ Provenance requirements.

The software supply chain can be broken down into five distinct phases: code, build, package, deploy, and run. We will address each of these phases in terms of our approach to security.

Chapter

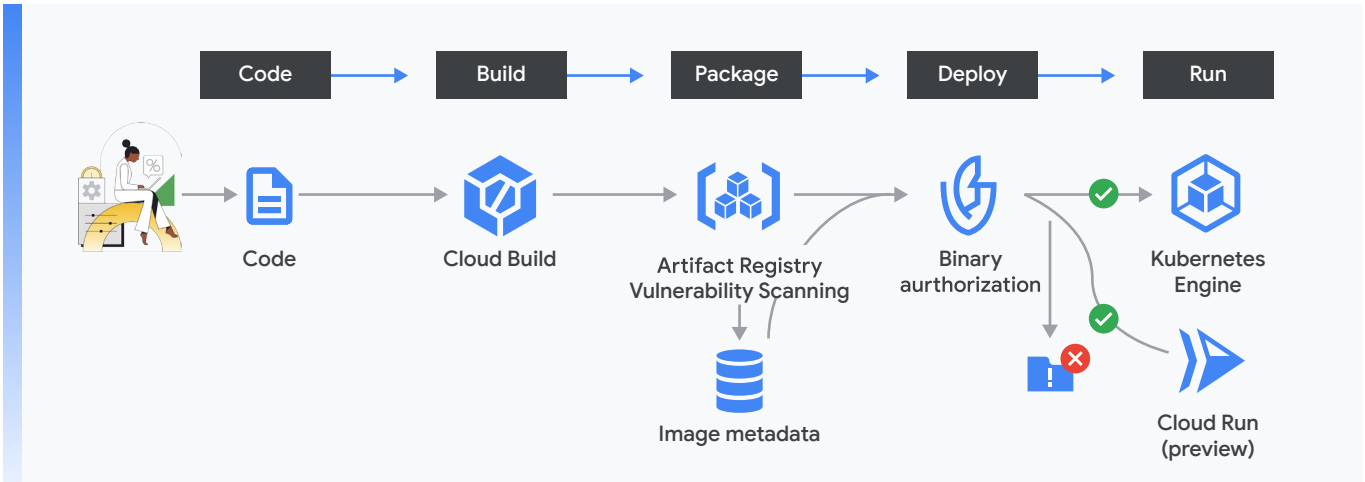
03 Managed services for each stage

At Google Cloud, we provide fully managed tools, from code and build to deploy and run, with the above standards and best practices implemented by default.

Securing your software supply chain requires establishing, verifying, and maintaining a chain of trust that establishes the provenance of your code and ensures that what you're running in production is what you intended. At Google, we accomplish this via attestations that are generated and checked throughout the software development and deployment process, enabling a level of ambient security through things like code review, verified code provenance, and policy enforcement. Together, these processes help us minimize software supply chain risk while improving developer productivity.

At the base, we have common secure infrastructure services like identity and access management and audit logging. Next, we secure your software supply chain with a way to define, check, and enforce attestations across your software lifecycle.

Let's look more closely at how to achieve ambient security in your development process through policies and provenance on Google Cloud.



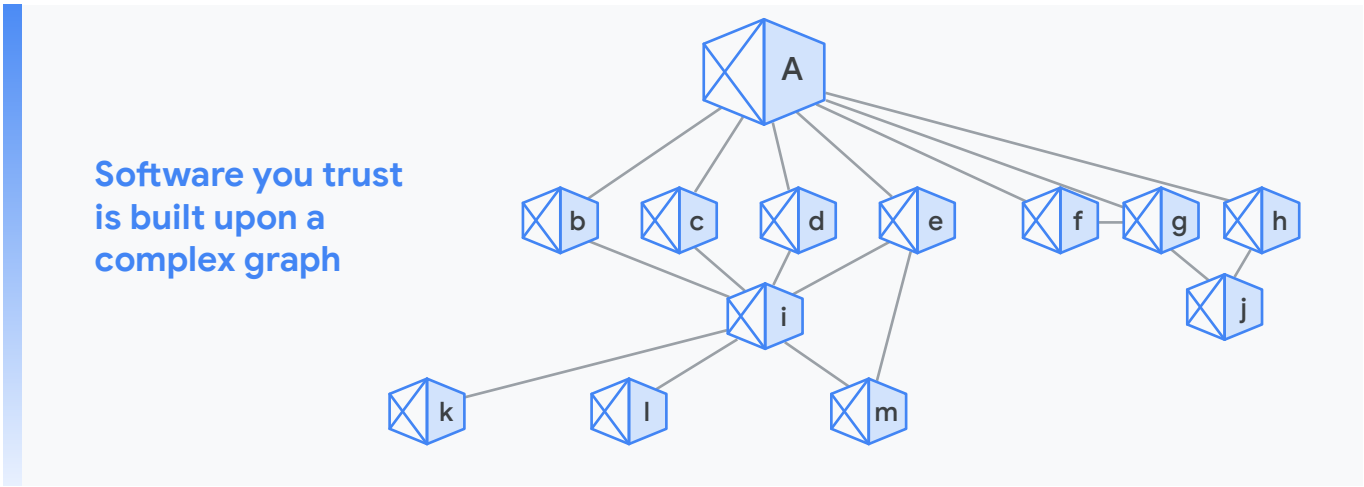
Phase 1: Code

Securing your software supply chain begins when your developers start designing your application and writing code. This includes both first-party software as well as open source components, each of which comes with its own challenges.

Open source software and dependencies

Open source empowers developers to build things faster so organizations can be more nimble and productive. But open source software is not perfect by any means, and while our industry depends on it, we often have very little insight into its dependencies and the varying levels of risk that come with it. For most enterprises, the risk primarily results from either vulnerabilities or licenses.

The open source software, packages, base images, and other artifacts you depend on form the foundation of your "chain of trust."



For example, consider that your organization is building software "a." This diagram shows the chain of trust; in other words, the number of implicit dependencies in your project. In the diagram, "b" through "h" are direct dependencies and "i" through "m" are indirect dependencies.

Now consider that there is a vulnerability deep down in the dependency tree. The problem can show up across many components very quickly. Moreover, dependencies change quite frequently: on an average day, 40,000 npm packages see a change in their dependencies.

Open Source Insights is a tool built by Google Cloud that provides a transitive dependency graph so you can see your dependencies and their dependencies, all down the dependency tree. Open Source Insights is continuously updated with security advisories, licensing information, and other security data across multiple languages all in one place. When used in conjunction with Open Source Scorecards, which provide a risk score for open source projects, Open Source Insights helps your developers make better choices across the millions of open source packages available.

To address this concern, it is key to focus on the dependencies as code. As these dependencies move toward the end of the supply chain, it's harder to inspect them. To secure your dependencies, we recommend starting with the supply:

- Use tools like Open Source Insights and OSS Scorecards to get a better understanding of your dependencies.
- Scan and verify all code, packages, and base images through an automated process that is a key part of your workflow.
- Control how people access these dependencies. It is critical to tightly control the repositories for both first-party and open source code, with constraints around thorough code review and audit requirements.

We will cover the build and deploy processes in more detail further on, but it's also important to verify the provenance of the build, leverage a secure build environment, and ensure that the images are signed and subsequently validated at deploy time.

There are also a number of safe coding practices developers can employ:

- Automate testing
- Use memory-safe software languages
- Mandate code reviews
- Ensure commit authenticity
- Identify malicious code early
- Avoid exposing sensitive information
- Require logging and build output
- Leverage license management

Phase 2: Build

The next step in securing your software supply chain involves establishing a secure build environment at scale. The build process in essence starts with importing your source code in potentially one of many languages from a repository and executing builds to meet the specifications laid out in your config files.

Cloud providers like Google give you access to an up-to-date managed build environment that lets you build images at any scale you need.

As you go through the build process, there are a number of things to think about:

- Are your secrets secure during the build process and beyond?
- Who has access to your build environments?
- What about relatively new attack vectors or exfiltration risks?

To develop a secure build environment, you should start with secrets. They are critical and relatively easy to secure. Start by ensuring that your secrets are never plaintext and as far as possible not part of your build. Instead you should ensure they are encrypted and your builds are parameterized to refer to external secrets to use as needed. This also simplifies periodic rotation of secrets and minimizes the impact of any leaks.

The next step is to set up permissions for your build. There are various users and service accounts involved in your build process, for instance, some users may need to be able to manage secrets, while others may need to manage the build process by adding or modifying steps, and still others may just need to view logs.

As you do this, it is important to follow these best practices:

- The most important is the principle of least privilege. Implement finegrain permissions to give users and service accounts the precise permissions they need to effectively do their jobs.
- Make sure you know how users and service accounts interact and have a clear understanding of the chain of responsibility from setting up a build to executing it to the downstream effects of the build.

Next, as you scale up this process, establish boundaries around your build to the extent possible and then use automation to scale up through config as code and parameterization. This allows you to audit any changes to your build process effectively. In addition, make sure you meet compliance needs through approval gating for sensitive builds and deployments, pull requests for infrastructure changes, and regular human-driven reviews of audit logs.

Finally, make sure the **network** suits your needs. In most cases, it is best to host your own source code in private networks behind firewalls. Google Cloud gives you access to features like Cloud Build Private Pools, a locked down serverless build environment within your own private network perimeter, and features like VPC Service Controls to prevent exfiltration of your intellectual property.

Binary Authorization

While IAM is both a must-have and a logical starting point, it is not foolproof. Leaky credentials represent a serious security risk, so to reduce your reliance on IAM you can switch to an attestation-based system that's less error-prone. Google uses a system called binary authorization, which allows only trusted workloads to be deployed.

The binary authorization service establishes, verifies, and maintains a chain of trust via attestations and policy checks throughout the process. Essentially, binary authorization generates cryptographic signatures—attestations—as code and other artifacts move toward production, and then before deployment these attestations are checked based on policies.

When using Google Cloud Build, a set of attestations is captured and added to your overall chain of trust. For example, attestations are generated for which tasks were run, what build tools and processes were used, and so on. Notably, Cloud Build helps you achieve SLSA Level 1 by capturing the source of the build configuration, which can be used to validate that the build was scripted. Scripted builds are more secure than manual builds and are required for SLSA Level 1. In addition, your build's provenance and other attestations can be looked up using the container image digest, which creates a unique signature for any image, and is also required for SLSA Level 1.

Phase 3: Package

Once your build is complete, you have a container image that is almost ready for production. It is essential to have a secure location to store your images that can prevent tampering of existing images and uploads of unauthorized images. Your package manager would likely need to have images for both first-party and open-source builds as well as language packages your applications use.

Google Cloud's Artifact Registry provides you with such a repository. Artifact Registry is a single place for your organization to manage both container images as well as language packages (such as Maven and npm). It is fully integrated with Google Cloud's tooling and runtimes and comes with support for native artifact protocols. This makes it simple to integrate with your CI/CD tooling as you work to set up automated pipelines.

Similar to the build step, it is essential to ensure access permissions to Artifact Registry are well thought through and follow the principles of least privilege. Beyond restricting unauthorized access, the package repository can provide a lot more value. Artifact Registry for instance includes vulnerability scanning to scan your images and ensure they are safe to deploy. This service scans images against a constantly refreshed and updated vulnerability database to evaluate against new threats and can alert you when a vulnerability is found.

This step generates additional metadata, including an attestation for whether an artifact's vulnerability results meet certain security thresholds. This information is then stored in our analysis service, which structures and organizes the artifact's metadata, making it readily accessible to binary authorization. You can use this to automatically prevent risky images from being deployed to Google Kubernetes Engine (GKE).

Phase 4 & 5: Deploy and run

The final two phases of the software security supply chain are deploy and run. While these are two separate steps, it makes sense to think about them together as a way to ensure that only authorized builds make it to production.

At Google, we've developed best practices for determining what kind of builds should be authorized. This starts with ensuring the integrity of your supply chain so that it produces only artifacts that you can trust. Next, it includes vulnerability management as part of the software delivery lifecycle. Finally, we put those two pieces together to enforce workflows based on policies for integrity and vulnerability scanning.

When you get to this stage, you have already been through the code, build, and package phases; attestations captured along the supply chain can be verified for authenticity by binary authorization. In enforcement mode, an image is deployed only when the attestations meet your organization's policies, and in audit mode, policy violations are logged and trigger alerts. You can also use binary authorization to restrict builds from running unless they were built using the sanctioned Cloud Build process. Binary authorization ensures that only properly reviewed and authorized code gets deployed.

Deploying your images to a trusted runtime environment is essential. Our managed Kubernetes platform, GKE, takes a security-first approach to containers.

GKE takes care of much of the cluster security concerns you need to care about. Automatic cluster upgrades allow you to keep your Kubernetes patched and up-to-date automatically using release channels. Secure boot, shielded nodes, and integrity checks

ensure that your node's kernel and cluster components haven't been modified and are running what you intend and that malicious nodes can't join your cluster. Finally, confidential computing allows you to run clusters with nodes whose memory is encrypted so that data can be kept confidential even while it's being processed. Couple that with data encryption while at rest and in transit over the network, and GKE provides a very secure, private, and confidential environment to run your containerized workloads.

Beyond this, GKE also enables better security for your applications through certificate management for your load balancers, workload identity, and advanced network capabilities with a powerful way to configure and secure the ingress into your cluster. GKE also offers sandboxed environments to run untrusted applications while protecting the rest of your workloads.

With GKE Autopilot, GKE's security best practices and features are automatically implemented, further reducing the attack surface and minimizing misconfiguration that can lead to security issues.

Of course, the need for verification doesn't stop at deployment. Binary authorization also supports continuous validation, enabling continued conformance to the defined policy even after deployment. If a running application falls out of conformance with an existing or newly added policy, an alert is created and logged, giving you confidence that what you're running in production is exactly what you intended.

Vulnerability management

Along with ensuring integrity, another aspect of supply chain security is ensuring that any vulnerabilities are found quickly and patched. Attackers have evolved to actively insert vulnerabilities into upstream projects. Vulnerability management and defect detection should be incorporated throughout all stages of the software delivery lifecycle.

Once the code is ready for deployment, use a CI/CD pipeline and take advantage of the many tools available to do a comprehensive scan of the source code and the generated artifacts. These tools include static analyzers, fuzzing tools, and various types of vulnerability scanners.

After you've deployed your workload to production, and while it's running in production and serving your users, it's necessary to monitor emerging threats and have plans for taking immediate remediation action.

Conclusion

To recap, securing a software supply chain is all about taking best practices like SLSA and using trusted managed services that help you implement these best practices.

It is essential to

- Start with your code and dependencies and ensure you can trust them.
- Protect your build system and use attestations to verify all necessary build steps were followed.
- Make sure all your packages and artifacts are trusted and cannot be tampered with.
- Enforce controls over who can deploy what and maintain an audit trail. Use binary authorization to validate attestations for every artifact you want to deploy.
- Run your applications in a trusted environment and ensure no one can tamper with them while they are running. Keep a watch for any newly discovered vulnerabilities so you can protect your deployment.

At Google, we build in best practices for each step along this journey into our product portfolio so you have a trusted foundation to build upon.

Chapter

04 Getting started

Ready to start securing your software supply chain? Just to be clear, where you begin is largely arbitrary—there's no one action that's going to secure your entire supply chain, and there's no one action that's more important than any other when it comes to total supply chain security. That said, here are four recommendations for getting started.

1. Patch your software

If you've deployed code into your production environment with known vulnerabilities, you've done the attacker's job for them. From that point it doesn't matter how well you've secured your software supply chain because they've already got a way in. So, patching is critical.

2. Take control of what's running in your environment

Once you're on top of patching, you want to be able to control your software supply chain itself. This starts with being able to assert that what you're running really came from your build tools or trusted repositories. That level of control helps prevent both purposeful attacks and inadvertent errors, like in the case of a developer who deployed something they didn't realize was unsafe. This gives you a strong foundation for adding tools like click tests and binary authorization.

3. Ensure third-party vendor packages are secure

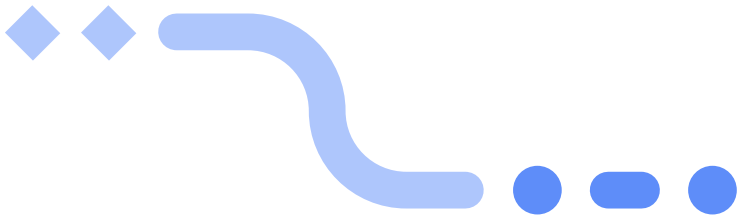
An emerging issue in supply chain security is the frequency with which vendors' software is being compromised to provide a conduit for ransomware or unauthorized access into their target customer deployments. The third-party vendor packages you run in your environment—for instance, system management products, network management products, or security products—often have high degrees of privilege. We suggest asking those vendors to go beyond their boilerplate security statements to provide a degree of assurance about the packages you're using. You might ask them what their SLSA level of conformance is or whether they're in scope of the requirements in the recent Executive Order.

4. Have a private copy of your source code

If you're using open source software, don't use a version you pull directly off the Internet to build from. Instead, have a private copy you keep patched so you have a clean place to start with for every build and can know with 100% confidence where the source code came from.

Chapter

05 Further reading



DevOps best practices

1. Six years of the State of DevOps Report, a set of articles with in-depth information on the capabilities that predict software delivery performance, and a quick check to help you find out how you're doing and how to get better.
2. Google Cloud 2021 Accelerate State of DevOps Report
3. Google Cloud whitepaper: Re-architecting to cloud native: an evolutionary approach to increasing developer productivity at scale

Securing the software supply chain

1. Google Cloud blog: What is zero trust identity security?
2. Google Security blog: Introducing SLSA, an end-to-end framework for supply chain integrity
3. Google Cloud whitepaper: Shifting left on security: Securing software supply chains

Ready to take your next steps?

To find out more about how Google Cloud can help secure your software supply chain and your business, just get in touch.

Talk to an expert