Zero-Trust Supply Chains

Dan Lorenc
June 26th 2021

Introduction

Supply-chain security and technology have lagged behind network and service security for a number of reasons, **and it's time to fix that!** Zero-trust technologies have dramatically improved and simplified other forms of enterprise infrastructure, but haven't been applied to supply-chain security yet. This document explores how zero-trust could be applied to build systems, through the Sigstore and SPIFFE/SPIRE projects.

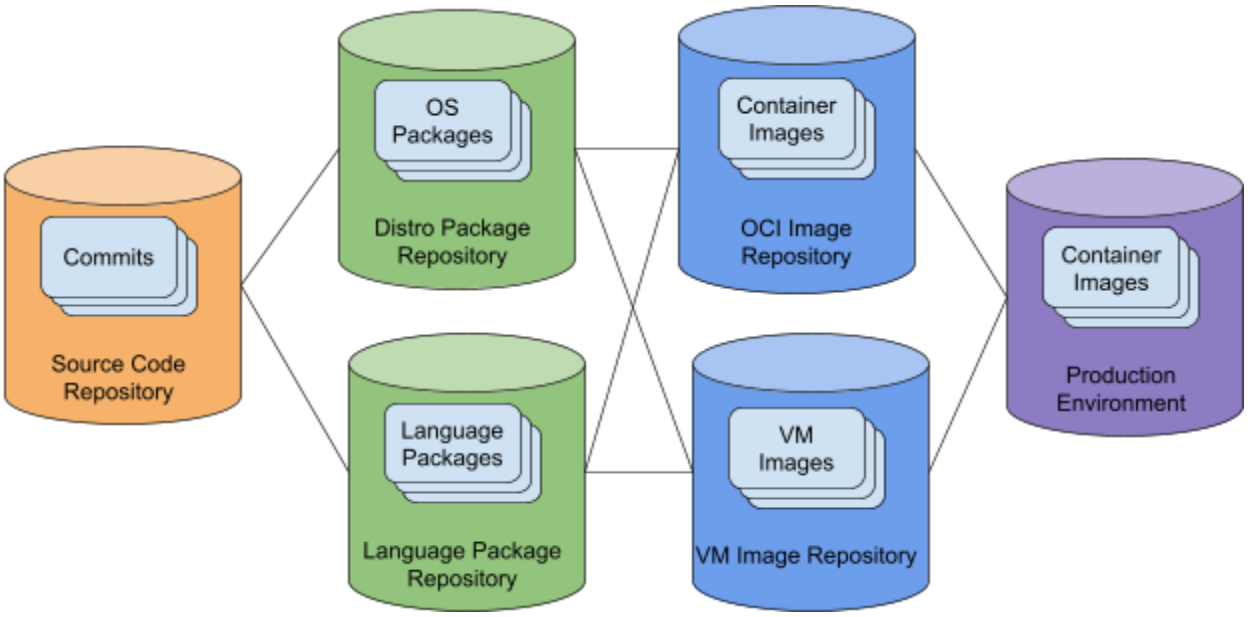
TektonCD is used as an example build system and In-Toto as an example provenance format. Other systems and formats would work as well.

Zero Trust

Traditionally, network and workload security relied on trusted "perimeters". Firewalls, internal networks and physical security provided defense against attackers by keeping them out. This type of architecture is simple and effective when all assets are in one place, the firewall doesn't need many holes and all hardware is on the same physical network.

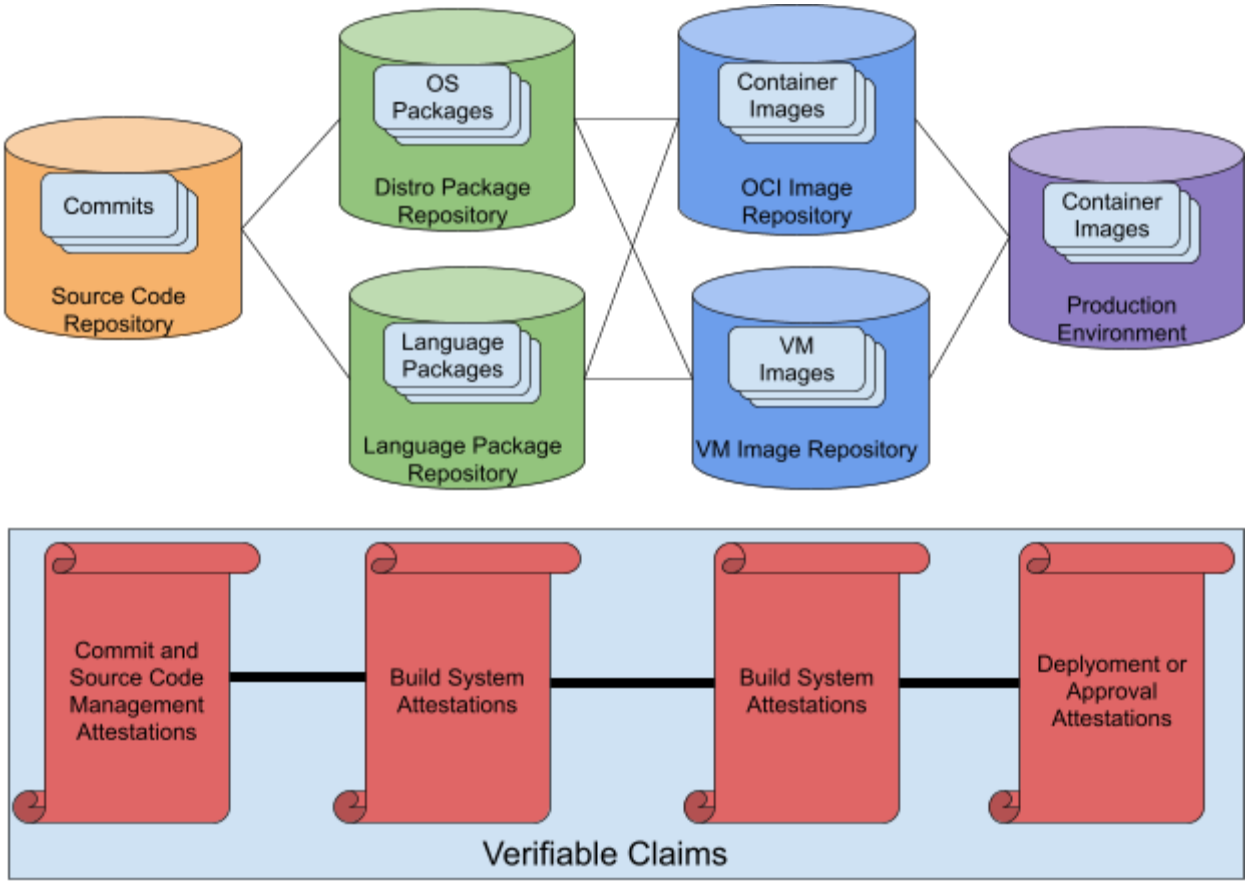
Unfortunately, this isn't true anymore. The workplace is distributed. Devices are mobile and environments are ephemeral. Enter zero-trust security. Zero-trust architecture focuses on protecting *assets*, not *perimeters*. Services authenticate users against hardware instead of network endpoints. Users authenticate with multi-factor-authentication (FIDO2, etc.) and devices authenticate with hardware-roots-of-trust (TPM2, etc.). The end result is a system focused on fine-grained access control. Instead of trusting every device on a network, you control exactly which users and systems have access to which services.

In a supply-chain, artifacts travel along a series of *repositories* (source code or binary artifact) as they are transformed from initial commit to running artifact in a production environment and trust is typically managed at the *repository-level*. An organization trusts an SCM for their first-party code, and a series of artifact managers (language, container) for their third-party code. Stricter companies may require all artifacts to be mirrored into a fully-trusted first-party repository. In these scenarios, trust moves internal but remains at the *repository* level.



In typical open source supply-chains, a compromise in **any one** of these systems is enough to attack the final system. There are typically many more separate systems than depicted above, and many are run for free by open-source communities with low or zero budgets.

A **Zero Trust Supply Chain** moves artifact repositories **out** of the Trusted Compute Base. Individuals and build systems attest to source code and artifacts directly. These attestations form a verifiable chain from its origin (developer or system) to final, deployed production artifact. Artifact metadata (including rich provenance) is digitally-signed with PKI support. Signed metadata files, or *attestations,* are stored and accessible in a global transparency log.



Instead of trusting the systems to serve accurate package metadata, we verify each claim in the chain itself against the actors or systems that attested to them. **This will likely be uncomfortable at first, because it will force organizations to fully-enumerate and confront everything they trust.**

SPIFFE/SPIRE

The SPIFFE (Secure Production Identity Framework for Everyone) is a framework and a set of standards for identifying and securing communications between application services following zero-trust principles (and the SPIRE is an open source implementation of SPIFFE), growing in popularity across the public and private clouds. Similar to the widely-deployed Web PKI, SPIFFE allows for hierarchical, federated identity mapping and trust. Instead of using centrally-trusted CAs, SPIFFE allows for each organization to provision their own trust domains, and to selectively trust other domains with standard x509 PKI. The SPIFFE protocol defines how these x509 certificates and JWTs are structured and validated to allow for fine-grained trust mapping.

Unfortunately these short-lived credentials cannot be directly used to cryptographically sign software artifacts because of subtle but important differences in the trust models of artifacts, mostly related to Key Management, Expiration and Revocation policies. The **core difference** is that SPIFFE keys are used and checked **at the same time**, while code signing keys must be

valid when they are used, but the signatures produced are checked **much later.** (See the Background section for a discussion and explanation of these differences.)

Sigstore - Supply Chain Transparency

Sigstore applies a new approach for trust in supply chains based on transparency logs. Sigstore provides a set of tooling and services to allow organizations to publish verifiable metadata (provenance) about artifacts that are publicly distributed. This metadata is stored on a transparency log (Rekor) where it can be audited, discovered and used in policy engines by consumers. **By placing all of its own operations on transparency logs, the sigstore service itself does not need to be trusted** - it is independently auditable and cryptographically verifiable.

Users generate very short-lived (<20 min) code signing credentials and receive code signing certificates from sigstore through an OIDC authentication challenge. They use those keys to sign artifacts and then destroy them immediately after. The initial certificate and signature are time stamped and then placed on transparency logs, which help detect key compromise and provide a transparent timestamp authority. **Sigstore provides tooling for a zero-trust supply-chain today for publicly distributed artifacts.**

Public-Private Boundaries

Sigstore's OIDC-based Code Signing PKI is great for individuals operating with public identities (rather than internal, corporate ones). Sigstore includes a system for developers to refresh their own credentials, which can be used to create long-term signatures for artifacts. This system could even be extended using the ACME-protocol or the OIDC device flow for build systems that operate publicly (on a hostname like actions.github.com, or cloudbuild.googleapis.com). **But this system does not work well for internal, automated build environments that do not expose a public endpoint.** There is no endpoint to do an ACME challenge against, and there is no individual to complete an OIDC flow.

Further, these build environments do not even have a **public identity** to authenticate as. What name would my Tekton cluster sign artifacts as? How would a consumer of my images even know this? Pre-shared secrets in an ACME flow, or long-lived service account tokens would solve the interactive challenge problem, but not the identity one.

Enter SPIFFE/SPIRE

These were discussed briefly in the initial section on zero-trust networks, but ruled out as a solution because the short-lived credentials are not appropriate for artifact signing. Fortunately, we can combine these systems and specifications with the Sigstore Binary Transparency Log (Rekor) and the Sigstore Certificate Authority/PKI (Fulcio, which includes its own Certificate

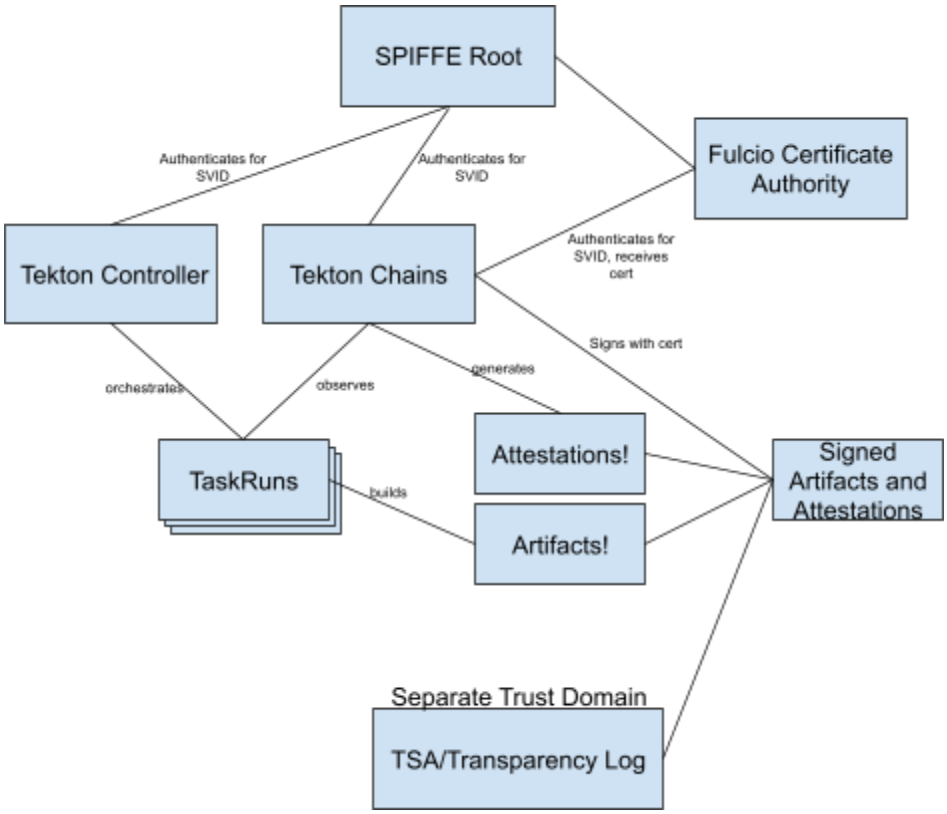
Transparency Log) to get the best of both worlds: a **Federated, Verifiable, Zero-Trust Supply Chain**.

The SPIFFE protocol provides a specification for a federated identity system based on Trust Domains. Each Trust Domain looks roughly like an internal x509 PKI with a set of root certificates. These trust roots can issue identity tokens, called SPIFFE Verifiable Identity Documents (SVID), that are in cryptographically-separate, globally-identifiable namespaces.

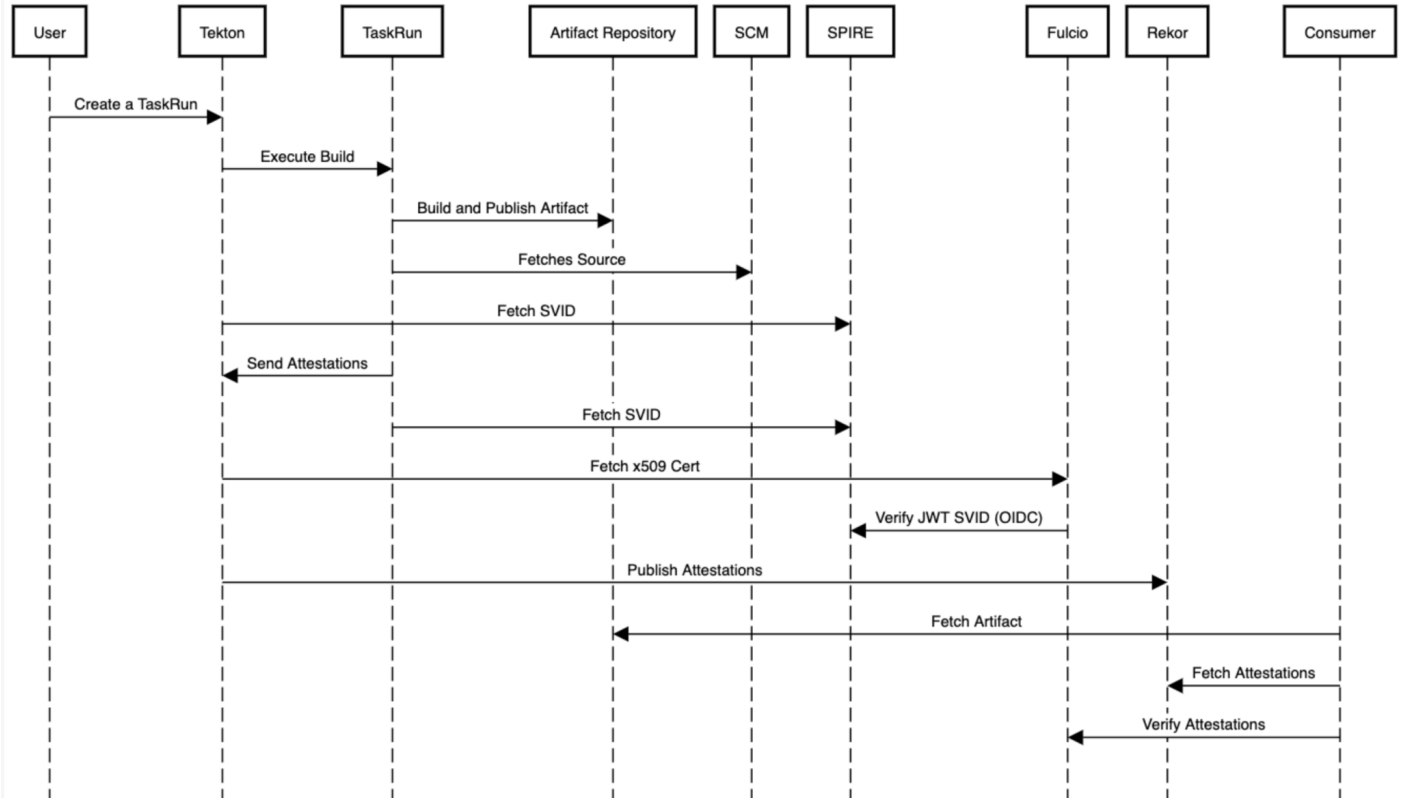
Organizations can bridge Trust Domains (by configuring Trust Root federation, or x509 Certificates) into whatever topologies they need. These topologies can even span organizations (again via federation). This protocol and specification allows organizations to grant meaningful names and tokens to internal systems that can be trusted and verified by external organizations.

Putting it all together

In addition to OIDC-Challenges, the Sigstore Code Signing CA can issue certificates based on SPIFFE tokens. This system would bind public keys to SVIDs rather than email addresses, providing a zero-trust identity system. An organization using SPIFFE could run the OSS Fulcio server internally to grant short-lived Code Signing certificates to workloads from SVID tokens. Using TektonCD for the build system, here's what that might look like:

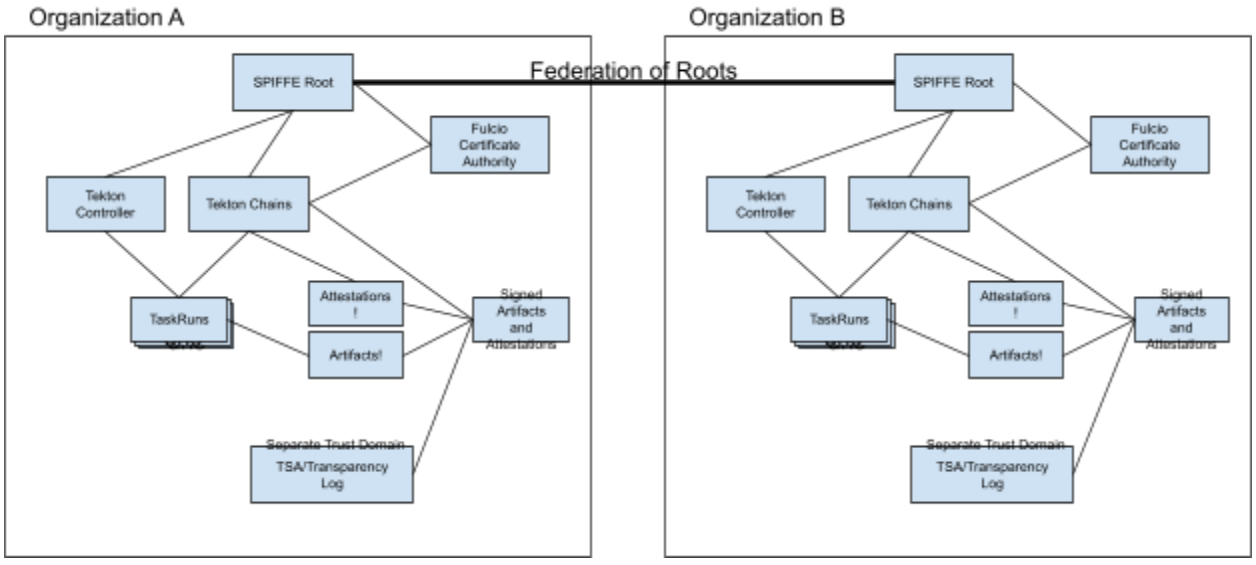


As a process diagram:



This model makes sense for purely internal artifacts, where the binaries and metadata never need to leave an organization. The timestamping could happen via a third-party timestamping authority (TSA) or an internal TSA in a separate trust-domain. A transparency log could be used, but a TSA is simpler and probably fine for internal use-cases.

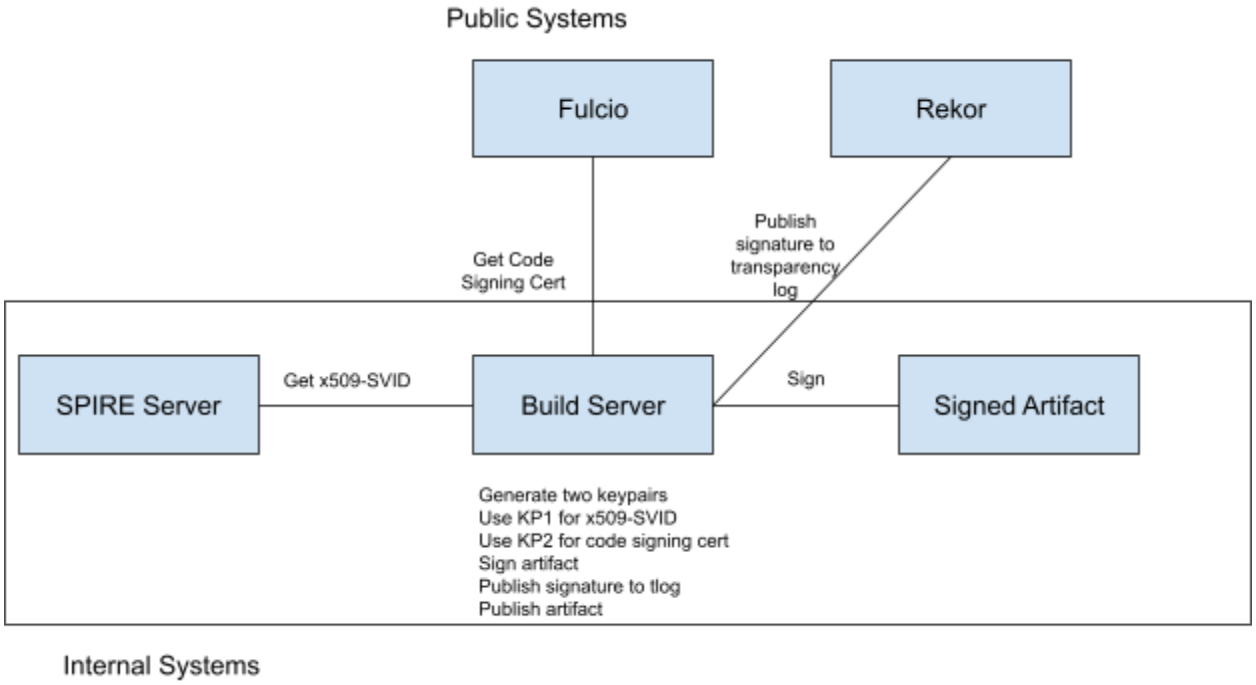
This could be easily extended to multiple organizations that exchange artifacts or metadata like this, by trusting each other's SPIFFE roots:



This model would make sense for organizations that distribute artifacts exclusively to each other, in one direction or both. The TSA would likely be third-party in this model. A transparency log could be used instead.

For publicly distributed artifacts, organizations have a few options. An internal build service can request short-lived certificates from the public Fulcio instance directly. The organization operating the build service can register with the public Fulcio by uploading their root trust domain certificate and the SPIFFE ID corresponding to the workload for the build server, something like `spiffe://prod.acme.com/build`. The build server can make API requests directly to the public Fulcio instance and authenticate with its SPIFFE bundle. The certificate issued by Fulcio can be verified up to the public Fulcio root and will be in the public certificate transparency log, enabling easy verification for end users.

This model looks like:

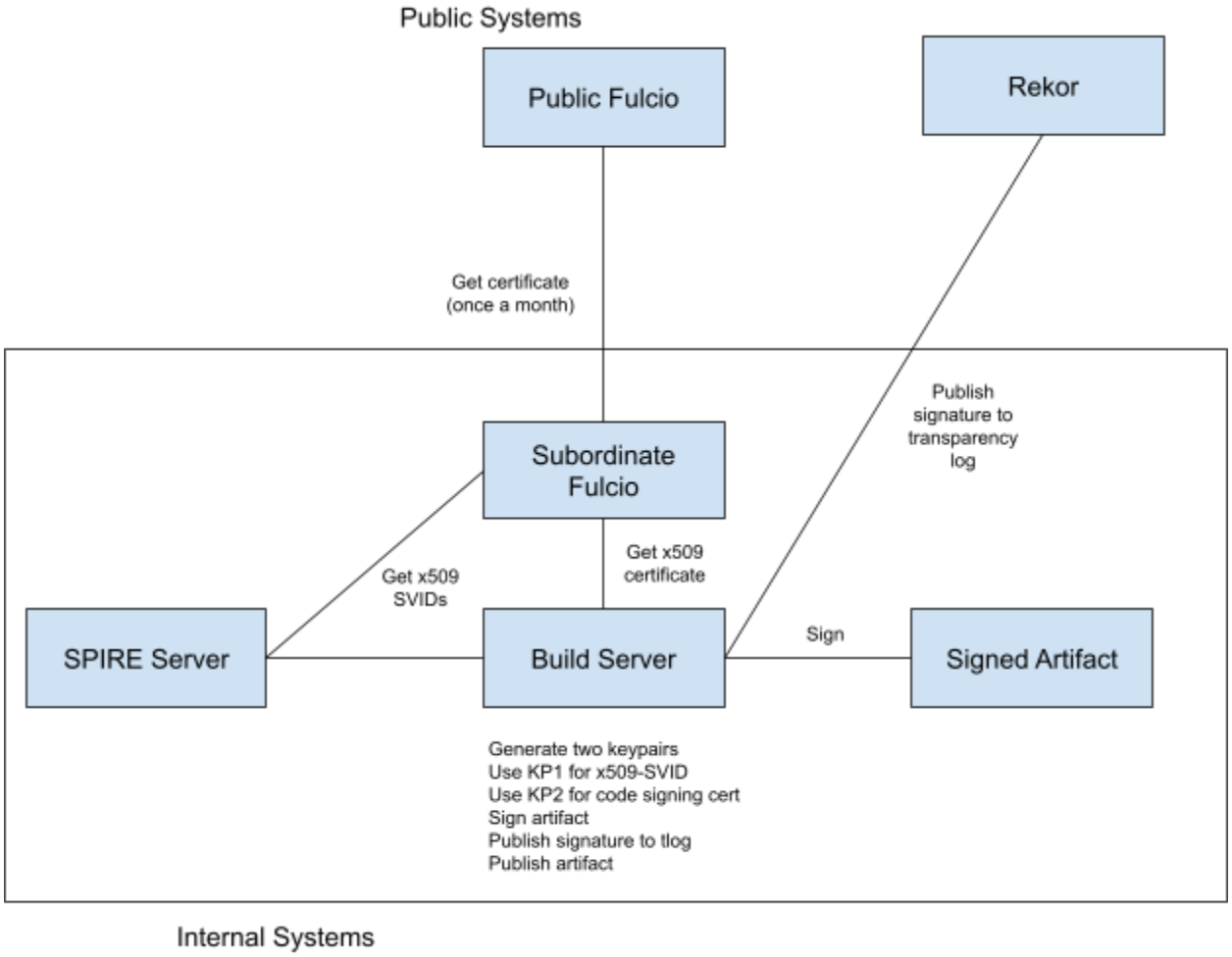


This model is simple to operate, but it requires the build service to have external network access to the production Fulcio instance. An alternate approach for organizations who wish to avoid the build-time dependency on the public instance, or who wish to avoid external network access is to run a subordinate Fulcio server.

The organization can register this subordinate server (again through a SPIFFE root key and ID) with the public instance. The public instance can grant a medium-term (1 month?) code signing CA certificate that can be used to issue actual code signing certificates. This root certificate is

used by the subordinate Fulcio instance, and must be renewed each month. This certificate could also be strengthened by an ACME challenge against a DNS entry corresponding to the hostname. This "verifies" the SPIFFE root namespace by piggybacking off of Web PKI.

This flow looks like:



With this flow, artifacts can still be verified against the public Fulcio certificate through a chain. Leaf certificates no longer have to be in a transparency log, as the organization issuing them is in control of their issuance, but this may still be desirable. Actual artifact signatures themselves would still need to be time stamped somehow. For public artifacts, this should still happen via a public binary transparency log or third-party TSA. This means the build system cannot operate **completely** offline if it wishes to sign artifacts that will be distributed publicly.

Implementation Plan

Phase 1

- Add SPIRE support into Tekton Chains
    - Instead of a fixed key, the Chains controller will generate short-lived keypairs and retrieve SPIRE credentials from the UNIX host socket.
- Add a SPIRE support to Fulcio using ACME and the SPIRE OIDC Federation model.
    - Fulcio will issue certificates based on OIDC-tokens signed by the SPIRE OIDC endpoint, to a SVID.
    - **Note:** the SPIFFE Trust Domain for each SVID must be known to Fulcio for this to be meaningful.
        - Fulcio will need to know the OIDC ./well-known URI and the Trust domain. These should match, and can be validated.
        - This could be configured manually by sending PRs containing a SPIFFE root certificate and corresponding SPIFFE namespace to the public GitHub repo, or using an ACME challenge
- Add Fulcio-SPIRE support into Tekton Chains
    - Combine the previous two items - Chains can fetch short-lived credentials from the SPIRE socket, and exchange these for a short-lived Code Signing certificate from Fulcio.
    - The signature should be published to Rekor

Tekton Chains can now sign things (artifacts AND Provenance Attestations) without any long-lived keys or user interaction!

Phase 2

Improve support for private Fulcio instances.

- Make it easy to deploy a disconnected Fulcio instance and TSA.
- Allow configuration of Tekton Chains against a private Fucio server instead of the public one.

Tekton Chains can now sign internal artifacts (AND Provenance Attestations) without any external network access or long-lived keys.

Phase 3

Subordinate Fulcio instances.

- Add support for Fulcio to grant subordinate **CA certificates** based on SPIFFE/ACME.
- Build a "certbot" equivalent to run internally and keep these CA certificates live by completing the challenges every month.
- Add support for organizations to register for subordinate certificates.
    - This could again be done via a PR to a GitHub repo with a SPIFFE namespace and root certificate.

Tekton Chains can now sign internal artifacts (AND Provenance Attestations) without a build-time dependency on Fulcio. Signatures must still be time-stamped or entered into the transparency log to be trusted outside of a single trust domain.

Further Work

Provenance/In-Toto

This document mostly talked about direct signatures over artifacts. These are in common use today, but other systems like In-Toto exist to allow for making more powerful claims (attestations) about how an artifact was produced. The Tekton Chains system is designed to observe and capture all the data necessary to produce rich In-Toto Attestations, and this work is already under way.

Rekor (Sigstore's Binary Transparency Log) can also be extended to support *verifiable* storage, discovery, and querying over these In-Toto Attestations, providing Attestation Transparency instead of just Signature Transparency. The rest of this document should directly apply to signatures over Attestations as well as artifacts.

TUF

For artifacts that are meant to be distributed publicly via an update system, TUF is a natural fit. The systems proposed in this document could easily be extended to support the generation, publication, and signing of TUF metadata in addition to artifacts (just like In-Toto provenance). TUF's key management techniques could easily be applied to further protect the root keys used by SPIFFE Trust Domains, and TUF metadata could be published with an artifact to protect versioning metadata against other attacks on update systems.

Build identity

Right now Tekton (and Chains) rely on the security of the underlying Kubernetes control plane to correctly observe the builds they orchestrate. An attacker who can manipulate objects in the Kubernetes data store (etcd) or the Control Plane itself (via misconfigured RBAC) might be able to trick Tekton into signing malicious artifacts or generating incorrect provenance. This can be strengthened by further integrating Tekton and SPIFFE.

In addition to issuing the Tekton Control Plane a SPIFFE ID, Tekton could issue SPIFFE IDs to each build, and make these credentials available within the Run. These credentials could be used for the build process to produce cryptographic proofs that can be used to reduce the trust needed in the Kubernetes Control Plane itself.

These SVIDs could also be used with OIDC to authenticate externally, reducing the need for bearer tokens and long-lived secrets in builds. This is similar to GKE workload identity bound to

Kubernetes Service accounts, but these credentials could be validated in a federated system or by third parties.

Hardware Root of Trust

Users can configure the attestation mechanisms for each build's SVID, increasing trust even further. Specifically, SPIRE can be configured to attest workloads to a hardware root using a TPM. The build system (orchestrator), build executor (TaskRun), and observer/signer (Chains) can all produce these attestations, and verifiers can chain them all together to tie all the individual operations of a build back to the hardware and operating system configuration they originated from.

Background

Supply Chains - Artifacts vs. People

Supply-chain security is very similar to network security and identity management. Instead of authenticating people or systems, we need to authenticate *artifacts*. There are multiple systems in use today, but all rely on cryptographic signatures and some form of PKI. Existing PKI systems for artifact signing include distributed systems like PGP's Web of Trust or TOFU models, centralized systems like Apple and the Play Store's code signing systems, as well as hierarchical CA systems like Authenticode. Newer technologies like transparency logs are also starting to serve a role, with usage in the Go modules ecosystem as well other Binary transparency implementations.

Authenticating an artifact is similar to authenticating a system or an individual in many ways, but it also differs in a few key ways that mean the existing technologies don't map perfectly. The biggest difference is that artifacts are not "alive". **An artifact is like a rock**. It cannot vouch for, or attest to its own identity like a person or system can. A person can vouch for their own identity by entering a password, touching a token or even meeting in person to show a government ID. A system can vouch for its identity by responding to a challenge (like in ACME) or using a hardware root-of-trust with something like a TPM. If you ask an artifact who it is, it just **sits there like a rock.**

This might sound silly, but has large implications on the way we need to design authentication systems. Cryptographic authentication systems usually rely on one party possessing a secret, and using this secret to complete challenges that prove they still have possession of this secret. If this secret is lost or stolen, the protocol fails and the user must recover their identity in some way. Well-designed systems acknowledge that secrets cannot be kept secret forever, so they include some form of automatic rotation or expiration.

This means users or systems must periodically obtain or generate new secrets to use to attest to their identity. Each protocol specifies exactly how this works (ACME for Web PKI, cookie,

session or JWT expiration for user logins, etc.), but the key problem is that they require some active form of interaction from the user/system. **Artifacts are just rocks** sitting on the ground, so they can't do this.

Existing Approaches

This next section explains some of the different approaches for long-term artifact trust.

Head in the sand

One common approach to artifact trust is to assume you (unlike everyone else in the history of software) will be able to protect your private key. You generate secret material to digitally sign an artifact, and assume that it will not be lost or stolen. PGP assumes keys last forever, other PKIs set slightly more reasonable lifetimes on the order of several years. In these systems, an artifact is "trusted" as long as the secret used to sign it is still trusted.

This works out well if keys can actually be stored securely and the expected lifetime of artifacts is much shorter than the lifetime of the keys. Unfortunately these two requirements are opposed - key lifetime is directly correlated to the risk of losing or leaking a key, yet the keys need to be valid for long periods of time to allow artifacts to run for long periods of time.

The loss of a key is also catastrophic in these systems. The standard technique is to publish a revocation list or invalid keys, but this means ALL artifacts signed by the leaked key are now untrusted. Partial recovery means re-signing ALL artifacts with the new key, which is difficult or impossible. This is hard to recover from and also unnecessary. Key compromise or loss can usually be tied to a rough point in time - you might know it was secure until sometime last week. Why revoke anything before then?

Timestamp Authorities

Another solution is to introduce a third-party notary or timestamping system. This notary records each signature event, and stamps that at a specific point in time. This lets us keep track of when each artifact was signed (and cryptographically prove that), and revoke only the artifacts that have been signed after the key was lost. This makes recovery easier, but introduces the need for a third-party timestamping authority. Access to a third party like this is operationally complex and undesirable in many internal build systems, so adoption has been limited. **It also requires trusting a third-party, which means we haven't really achieved zero trust - we've only moved it around.** Using multiple independent TSAs, or taking it one step further and using Time Transparency (like Roughtime, or Rekor's model) can help here.

TUF

The Update Framework is another approach to supply-chain security that addresses the revocation/timestamping problem. TUF addresses a lot of other threat models related to update systems (in addition to artifact trust) so the comparison is a bit difficult. Instead of using

expiration/revocation on certificates and keys, the TUF model relies on expiring signatures. Each TUF metadata payload contains an expiration date - the same key (or a different key) can be used to extend this date by publishing another set of metadata. TUF allows for key revocation and rotation through quorum signatures - as long as a majority of keys are still valid, old ones can be rotated out without the need to re-sign artifacts.

Timestamp authorities and TUF do not solve the initial trust problem though. Organizations and people still need some form of PKI to know which public keys (or TUF). We still need a way for organizations to trust artifacts from each other without the need to poke holes in firewalls or trust third-parties.