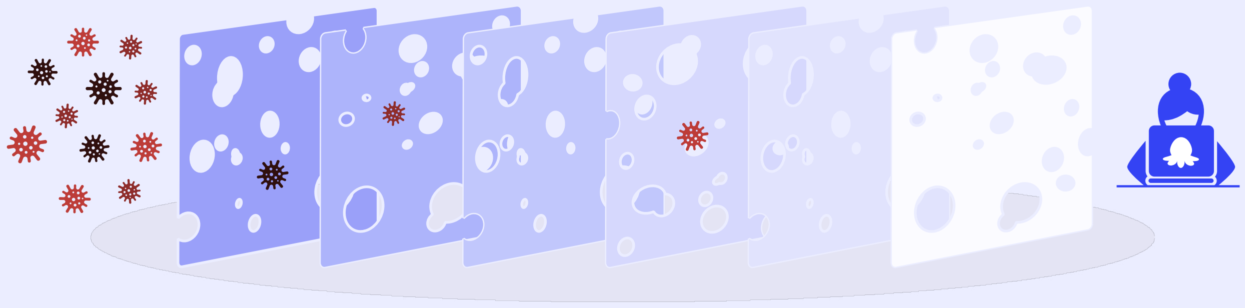
chainguard

Chainguard's Hardening Guide

At Chainguard, our approach to hardening matches modern high-reliability practices (e.g. Google Borg): start by accepting the inevitability that systems will fail, and seek to minimize correlated failures. Or as the NSA put it in its AI System Security guidance: Adopt a [zero trust] mindset, which assumes a breach is inevitable or has already occurred.

We accept that individual security controls are fallible (e.g. 0-days happen), and so we embrace a defense-in-depth model where we apply complementary and overlapping security controls, such that when a single layer fails the other layers provide a measure of redundant protection.

The Swiss Cheese Cybersecurity Defense-in-Depth Model

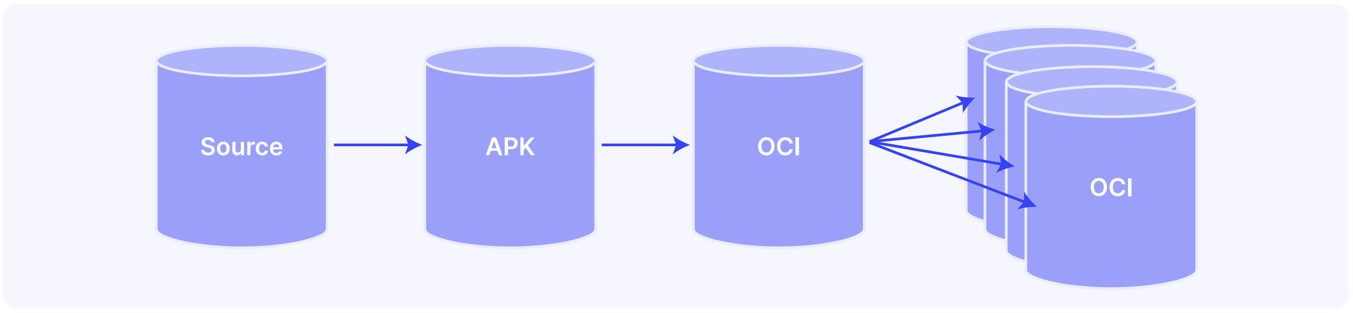


Each intervention (layer) has imperfections (holes). Multiple layers improve success.

This document walks through a high-level snapshot of the layers of ~~cheese~~ defense Chainguard currently has in place, and some of our forward-thinking around how/where we plan to improve and add layers.

High-level process segmentation

We own every step of our image production pipeline end-to-end. Starting with how we fetch source code from upstream git repositories and turn it into APK packages. Then assembling those APK packages into OCI images. Finally distributing those OCI images to our customers based on their entitlements.



To ensure integrity as we traverse these states, our systems rely heavily on cryptographic hashing and signatures:

- Our packaging pipelines always start with a source fetch based on an immutable reference, which ranges from SHA1 (e.g. git commit) to SHA512. The resulting APKs are signed (in a trusted control plane) with a strong signing key.
- Our image builds verify these APK signatures when assembling those APKs into OCI-compliant container images. The resulting OCI images are signed (and attested, more below) using the Sigstore project.
- Our signed OCI images are then distributed to our customers' private repositories in accordance with their entitlements. Upon delivery, we will optionally send a CloudEvents webhook to customer subscription endpoints. The CloudEvent webhook carries an Open ID Connect (OIDC) token authorizing the event as from Chainguard, for that particular customer, and including a cryptographic hash of the event payload in a custom OIDC claim (if there's a stronger event-based authentication scheme, we haven't seen it!).

In short: we take cryptographic verification of artifacts traversing our supply chain extremely seriously.

Let's take a deeper look into each of these hops along the way, and how we layer defenses into the process at every step.

Source

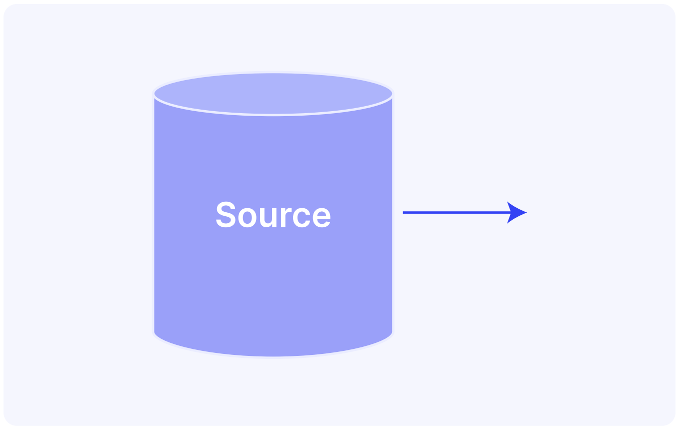
So where does the initial source definition come from, and how does Chainguard make controlled updates to this definition?

Today, the initial source code checksum is determined by the Chainguard developer creating the initial package definition. That developer also generally configures a small declarative block which tells our automation how to identify when new versions of the upstream software package become available, which serves two purposes:

1. It allows our automation to confirm the original source checksum matches expectations, and
2. It allows our automation to update the checksums for new versions as they are released.

To track software updates our automation ties into several systems including the GitHub API, Release Monitoring, and End-of-Life.

Regardless of the source of the changes, package updates all flow through our Pull Request process, and any package definition changes are reviewed by a Wolfi maintainer (see The Human Element below), a dry-run of the package build is performed (see Source to Package below), and a battery of checks are performed that surface additional signal to the reviewer (see Package below).



As an aside, you may have noticed that the only checksum in the above flow, which isn't signed, is the source code! We noticed this too, which is why Chainguard created (and subsequently donated to Sigstore) the gitsign project to help close this gap. However, it is still relatively early days and very few projects are signing their release tags using the strong identity-based signing of Sigstore.

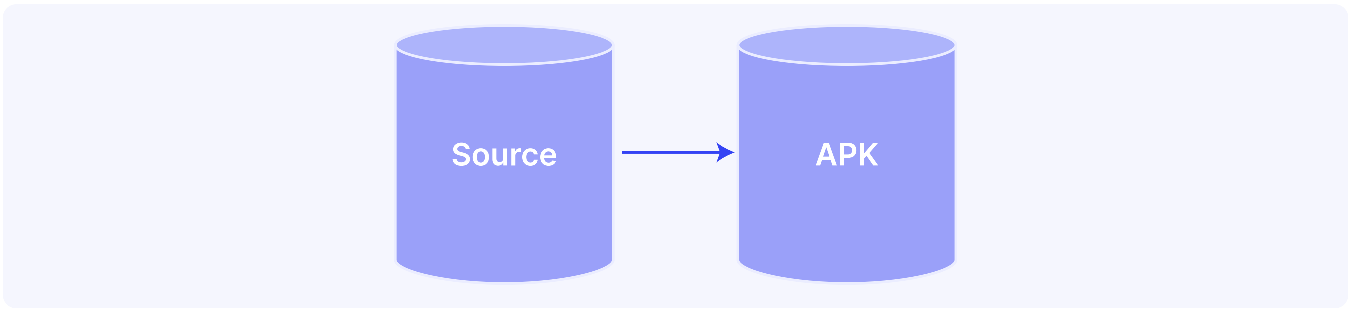
Future Directions:

We will likely start to leverage projects like the OpenSSF scorecards project (started by our very own CPO Kim Lewandowski) as an indicator of upstream projects' healthiness, to further arm code reviewers about places that may need additional scrutiny.

As `gitsign` gains traction with projects, we may start to validate that certain projects' releases are signed by the appropriate release automation. Think of it somewhat like admission control in Kubernetes, but for source commits making their way into Wolfi.

Explore ways to incentivize upstream maintainers to embrace higher security standards (like the above), and establish enough of a sense of identity to start to mitigate potential "insider risk" threats (see The Human Element).

Source to Package



The process of turning an “immutable reference” to an upstream project into a set of APK packages is defined via one of our melange configurations. Melange allows the package’s author to run an upstream project’s build process and then carve up the resulting build artifacts into a number of sub-packages.

On minimal packages: We generally try to slice up the resulting APK packages in accordance with the granularity with which the constituent pieces might be independently consumed. As an example a project may have a CLI (`foo`), a shared library (`libfoo`), man pages (`foo-docs`), and headers (`foo-dev`). This fine slicing ensures that when it is time to assemble an image from a set of our packages, that the transitive closure only pulls in what is necessary for a truly minimal image.

On hardened packages: As part of our hardening process, we pass a number of flags to various toolchains to increase security. One example of where we go above and beyond what most other linux distributions do is passing `-D_FORTIFY_SOURCE=3` to our C++ toolchains. We are also investing in memory-safe alternatives for certain packages, which we give a higher priority so that the APK solver prefers them to less safe variants.

On quality control: In order to shift quality control to the “left” our package definitions also carry a place where package authors can define tests. These tests must pass before a new version of the package is merged and released.

On build-process security: As part of living our mantra of treating our development platform like a production system, we heavily “eat our own dogfood” (aka dogfooding). Package builds happen within ephemeral containers that are defined as build-time dependencies of the final package (in the melange configurations). By dogfooding, we enable ourselves to tap into our own value-prop in ways that give us an unparalleled level of ephemeral build environment hardening:

- Our ability to produce minimal containers means these build environments have a smaller surface than virtually any other build environment.
- The few packages that get included benefit from our Low/No CVE SLA.

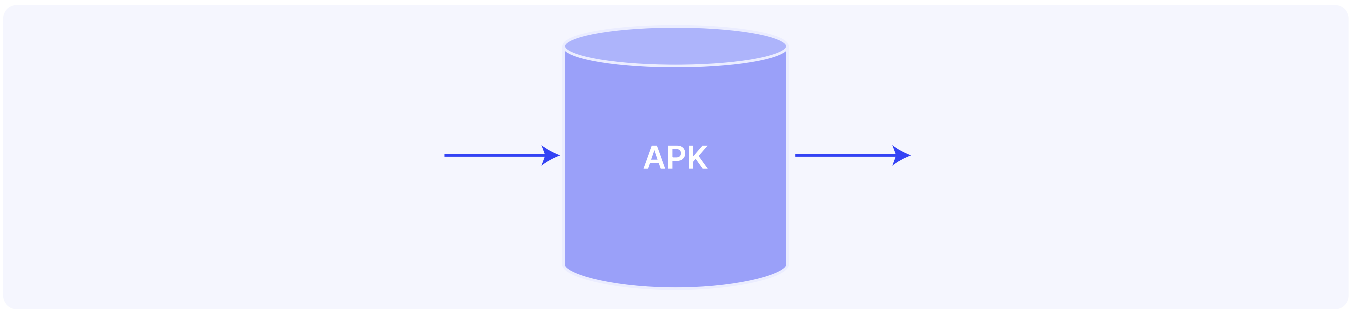
In addition to this, we have a disciplined separation of our build’s “trusted supervisor” orchestration layer from the “untrusted guest” execution layer to ensure a strong separation between privilege/credentials and the execution of any untrusted / unreviewed code.

Future Directions:

As we continue to invest in our build infrastructure to increasingly leverage the elasticity of the cloud, we will also continue to invest more heavily into build process isolation and observability (e.g. tracking file/network activities for anomaly detection).

Another direction we continue to think about is carrying toolchain patches that upstream won't accept to further harden our downstream builds and layer in additional security.

Package



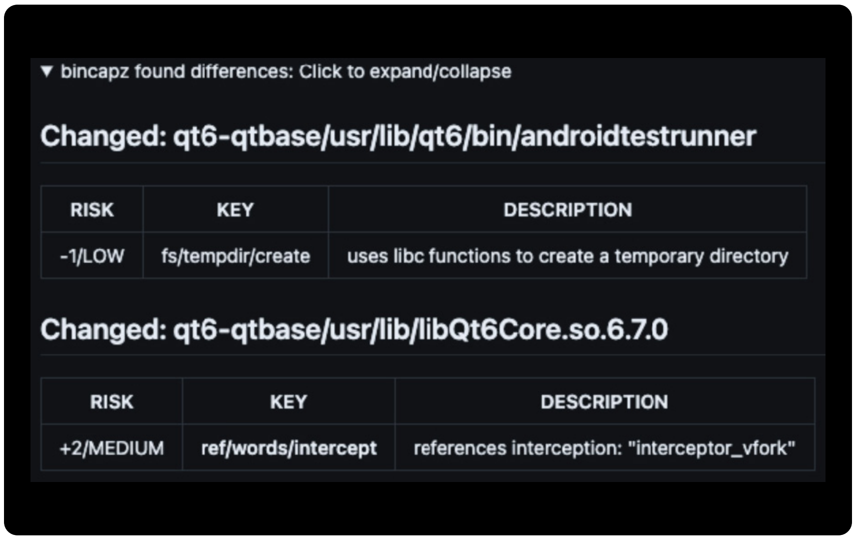
There are two main states that a package exists in that are worth discussion:

1. A throw-away build that is performed pre-submit, and
2. A release build that is produced and signed post-submit.

We run a battery of checks across packages in both of these states. The throw-away package builds are analyzed as part of the pull-request workflow and any results are surfaced to the author and reviewer (given the context, these are often presented as a diff vs. a previous release build of the package). The release builds are continuously analyzed to ensure that as our tooling improves we can flag any issues we learned about. The set of checks performed pre and post merge are generally very overlapping to avoid only discovering things after a package has merged (think: the former is a "shifting left" of the latter). The main purpose of the latter is to catch things that only come to light after things have been built (e.g. a newly discovered CVE).

On Low/No CVE SLA: one of the key analyses we perform is to run CVE scans over our packages. This ensures that new package builds and updates to existing packages satisfy our SLA before they are merged. We continuously monitor our release package builds to measure our SLA and trigger CVE remediation as new vulnerabilities are discovered.

On malware scanning: In addition to CVEs a common request we get from our customers is around scanning for malware. To fill this gap, Chainguard has a tool called `bincapz` that includes over 12k YARA rules (many from YARA forge) for both detecting malware and attempting to classify the syscalls performed by applications. As package updates come in, we use `bincapz` to analyze the binaries included in the package and enable the code reviewer to assess whether the upstream software potentially includes changes in capabilities. For instance, here it flagged an update to Qt that introduced the symbol `interceptor_vfork` to detect when it is running under asan/msan:



▼ bincapz found differences: Click to expand/collapse

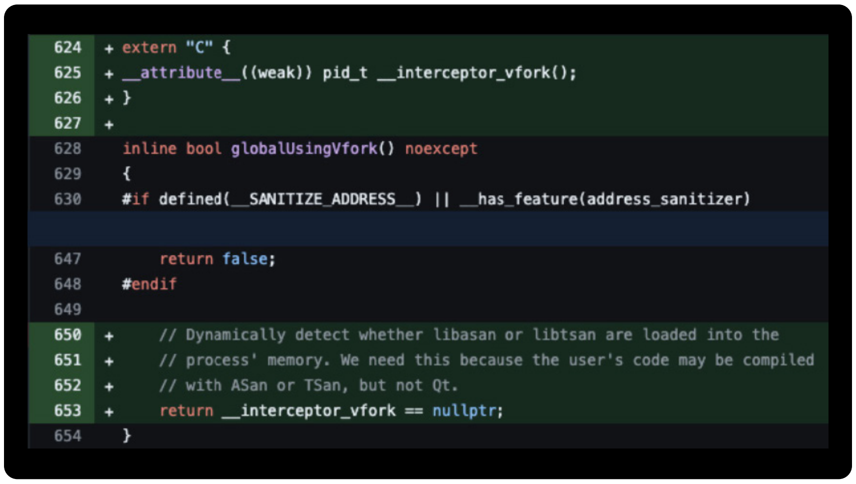
Changed: qt6-qtbase/usr/lib/qt6/bin/androidtestrunner

RISK	KEY	DESCRIPTION
-1/LOW	fs/tempdir/create	uses libc functions to create a temporary directory

Changed: qt6-qtbase/usr/lib/libQt6Core.so.6.7.0

RISK	KEY	DESCRIPTION
+2/MEDIUM	ref/words/intercept	references interception: "interceptor_vfork"

You can clearly see the introduction of this symbol in the upstream source diff:



```
+ extern "C" {
+ __attribute__((weak)) pid_t __interceptor_vfork();
+ }
+
  inline bool globalUsingVfork() noexcept
  {
  #if defined(__SANITIZE_ADDRESS__) || __has_feature(address_sanitizer)

      return false;
  #endif

+     // Dynamically detect whether libasan or libtsan are loaded into the
+     // process' memory. We need this because the user's code may be compiled
+     // with ASan or TSan, but not Qt.
+     return __interceptor_vfork == nullptr;
  }
```

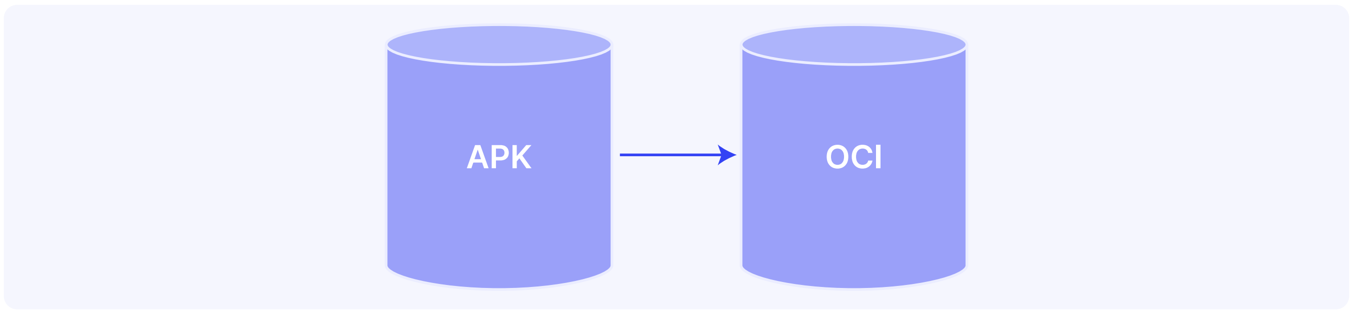
Once a Wolfi maintainer (see The Human Element) has reviewed package changes and the assorted signals extracted by our automation (above), it is merged and a new release build of the package is performed (see Source to Package) and then the final package is signed and added to our APK repository. If a maintainer is required to make any manual changes to the package update (e.g. to fix a broken build or patch CVEs), then a different reviewer must approve those edits before it can be merged.

Future Directions:

We plan to continue to refine our new `bincapz` integration to improve signal/noise, add continuous monitoring, make its results extremely actionable for the general reviewer pool, PR-blocking for sufficiently high-signal checks, and continue to expand its capabilities (no pun intended).

Similar to `bincapz` this is an area where we will continue to invest R&D to find new innovative ways to expand our capacity to automate and elevate reviews, and add layers to our "Defense in Depth" strategy.

Package to Image



We are continuously rebuilding our full image directory. We trigger builds immediately on image definition changes, and frequently throughout the day to pick up any package changes (see previous sections).

On Chainguard differentiation: Our image build process is driven by Chainguard's highly differentiated homegrown tooling apko, which is purpose-built to produce OCI images from a declarative configuration. It is designed to include the minimum transitive closure of dependencies from some core list of packages (this is a key facet of why finely slicing packages is important, see Source to Package). Most Chainguard images are defined such that this "core list of packages" is a single application package, so literally everything included into the final image is either a result of a dependency the application has, or because we have inadequately sliced a package (which we can fix as we become aware of them, see Source to Package). In addition to minimizing the number of packages, the vast majority of Chainguard images default to a `nonroot` user.

Chainguard's tooling for producing minimal images is unparalleled in its capacity to produce an enormous breadth of minimized application and base images, while also remaining functional and fully compatible with SCA tooling.

Due to this minimalism, our images accumulate CVEs more slowly than alternatives, and are less susceptible to "living off the land attacks", which makes this a valuable layer of a "Defense in Depth" security posture.

On signing and attesting: However, we don't just YOLO our images into the wild, there is a whole process we go through prior to tagging new images!

Once we have assembled an image as described above, we publish it by digest to our registry. Our release automation then signs the image so that users can verify an image came from our release process. We also attest to several important properties:

- We attest the SBOMs of our images
- We attest the SLSA provenance of our images
- We attest the "locked" image configuration used to produce each image

On bitwise reproducibility: Next we verify that we can reproduce a bitwise-equivalent image from the locked configuration using a parallel toolchain. This reproducibility makes our image provenance auditable by our users. Since we confirm this reproducibility with a separate `apko` toolchain, it would require an attacker to compromise both form factors of our `apko` toolchains to evade detection.

On image testing: From there we move on to more testing to qualify that the produced images work as intended. For complex containerized applications, which consist of several distinct containers, we test them as a single unit (e.g. kyverno, cert-manager, etc). We will generally try to test things in form factors consistent with how our customers are consuming things. For example, images that run on Kubernetes are deployed with their accepted Helm chart on ephemeral clusters and functionally validated. For images representing applications with well known conformance benchmarks, we ensure these benchmarks are run periodically and validated. One example is the class of Java images, which are benchmarked against the JCK.

On compliance benchmarks: We have many customers in regulated industries (Federal, Finance, Health Care, Transportation, Critical Infrastructure), which impose assorted hardening requirements on the software they deploy. To satisfy our customers' compliance requirements, the produced images are then benchmarked against popular compliance frameworks, currently our focus is on: STIGs and CIS Benchmarks (reach out with requests). This added benchmark validation is "another layer of cheese" relevant specifically for compliance auditors and organizations looking to pass audits (e.g. FedRAMP).

The availability of benchmarks varies by image, but a representative benchmark is chosen to implement. As a baseline, all images are benchmarked against the General Purpose Operating System (GPOS) Security Requirements Guide (SRG), using the underlying "OS" as the system in question; this ensures general evidence is available for comment requirements such as FIPS. For images where a dedicated SRG is present (ie: PostgreSQL), an additional SRG is implemented alongside the GPOS SRG.

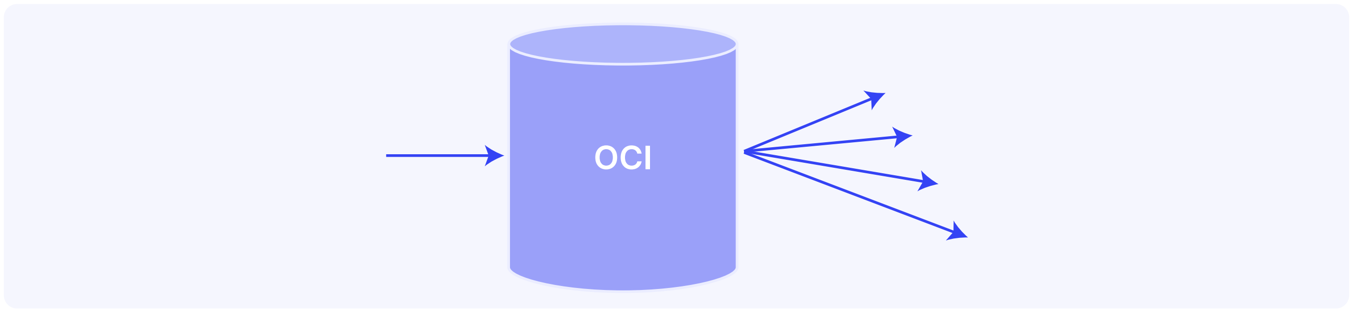
On going live: Only after the image is published, signed, attested, and has passed all of these qualifications we will tag things. For our developer images, this makes them available immediately. For our customers, stay tuned for Image to Customer.

Future Directions:

We will continue to invest in both our macro (all images) and micro (per-image) quality controls to ensure that our images meet the highest quality and security standards.

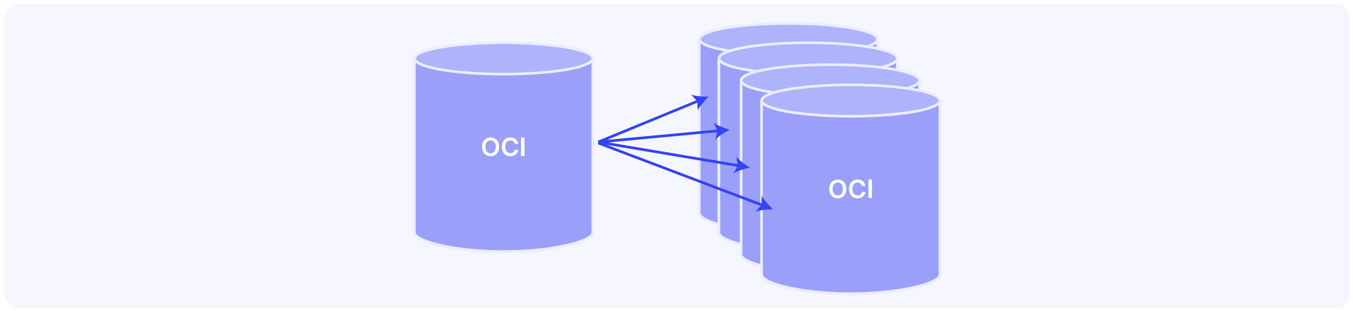
We will continue to expand our compliance benchmarking based on the needs of our customers (e.g. beyond just Federal benchmarks).
We also plan to deepen our Federal compliance story by developing a dedicated SRG for Chainguard Images.

Image



This stage is very similar to Package, so I will avoid the repetition, but we run an assortment of analyses on our images to ensure we are satisfying our SLAs and ensuring our images are of high quality.

Image to Customer



As images are staged to our private image repository `cgr.dev/chainguard-private` our "catalog syncer" looks up customers that are entitled to each new image and the new image is distributed to customers' private repositories, e.g. `cgr.dev/mattmoor.dev`.

As mentioned in High-level process segmentation, customers can configure real-time CloudEvent notifications so that as their entitled images are delivered to their Chainguard repositories, they can take action. We have samples demonstrating how to mirror these images to ECR or GCR based on these authenticated events, and this is even how our integration with DockerHub works. Users can also use something like this to kick-off their own downstream qualification process.

Future Directions:

One capability I have long wanted to build is leveraging our OIDC Federation capabilities to build first-class support for mirroring images to customer's cloud provider registries, so that they do not need to run their own integration.

Underlying Production Infrastructure

We take our production practices extremely seriously, and (as mentioned in Source to Package) we treat our development platform with the same level of production rigor. This includes practices we have talked about at length (and much more):

- Ephemerality: short-lived credentials, ephemeral infrastructure.
- Minimalism: least-privilege, minimal hardened images, network segmentation.
- Immutability: pinning commits, pinning image digests, read-only filesystems.

In our pursuit of production excellence, we tend to not even let platform limitations stand in our way, e.g. we created our own Security Token Service for Github called Octo STS to enable ourselves to leverage short-lived credentials brokered via trust relationships over long-lived credentials prone to leaks.

We have also adopted novel real-time auditing practices that are very complementary to least privilege in our layered "Defense in Depth" strategy (stay tuned for more details here).

One of my favorite anecdotes about our zero trust architecture: Suppose that hypothetically our entire production serving infrastructure were to become compromised. Let's take it even further: suppose the cloud platform we are running on is fully compromised…

If our users are verifying that the signatures of the images they are receiving from us were produced by our release automation and alerting when those signatures don't match, then the only real attack the attacker can mount without immediate detection is a simple replay attack (and even this can be mitigated with "build horizon" policies around the age of images).

This is one of the reasons I am such a fan of cryptographic content-addressing and signing: your blob storage could be fully compromised and the worst case scenarios are Denial-of-Service or Replay attacks.

Future Directions:

We are constantly looking for ways to raise the bar with respect to production security to make our "stack of cheese" as impenetrable as possible. We already have a backlog of innovations we are in the process of writing up, so stay tuned.

The Human Element

Just like we employ defense in depth to ensure the final image is hardened, Chainguard employs similar procedures to secure ourselves. Chainguard uses multiple techniques in order to detect and alert on attacks targeting our systems and employees, including:

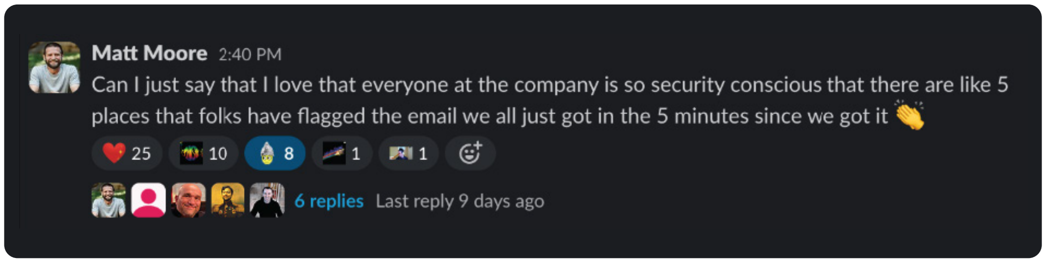
- Credential exfiltration (e.g. running the wrong docker image as root, and it phones home with a pile of credentials and your bitcoin wallet, or simply weak / recycled passwords).
- Social Engineering (e.g. phishing, cyber bullying, etc)
- Insider Threats (e.g. Edward Snowden, The Shadow Brokers, Jia Tan)

On Chainguard insider threats: All full-time Chainguardians undergo a background check at hiring (where countries allow). As a simple function of payroll, we have identification on file and bank account information for all Chainguardians. A majority of full-time Chainguardians generally also meet up 2-4 times a year at partial- or wholecompany summits.

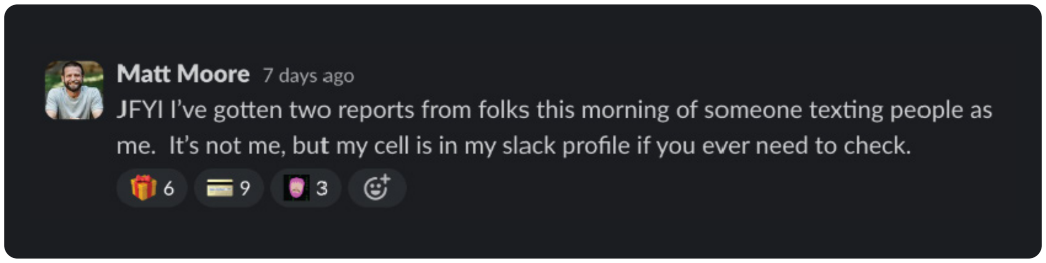
We also have layers of protection against a single employee being able to unilaterally take adverse actions without at least triggering an alert. These include things such as code review requirements, and a requirement to explicitly elevate user permissions (aka `sudo`) to interact directly with production environments (which is only available to our oncall groups). We have also been deploying a form of continuous auditing to our production environments, which causes all anomalous (even when authorized) access to monitored resources to immediately get flagged for review.

On social engineering: All full-time Chainguardians undergo annual security awareness training.

As a fun recent anecdote, one of our healthcare providers sent out an email having to due with taxes that looked enough like phishing that it immediately prompted a half dozen different folks to raise the alarm within moments of it happening:


Matt Moore 2:40 PM
Can I just say that I love that everyone at the company is so security conscious that there are like 5 places that folks have flagged the email we all just got in the 5 minutes since we got it 👏
❤️ 25 10 8 1 1
6 replies Last reply 9 days ago

... days later, the team sprung into action again to dodge some actual social engineering:


Matt Moore 7 days ago
JFYI I've gotten two reports from folks this morning of someone texting people as me. It's not me, but my cell is in my slack profile if you ever need to check.
🎁 6 💳 9 3

I still haven't gotten my gift cards yet... ;)

On credential exfiltration: All critical systems used by Chainguard utilize SSO and require phishing-resistant MFA using FIDO2 security keys. In cases where longer-lived credentials are persisted on the local filesystem (e.g. GCP / Chainguard refresh tokens), the tokens have a maximum lifespan of no more than 24 hours, and are revocable. For interactions with GitHub, Chainguard has banned the use of Classic Personal Access Tokens (PATs), requires approval for Fine-Grained PATs (phasing out), has banned Deploy Keys, and encourages the use of hardware security keys for SSH-based Git access.

Future Directions:

As mentioned in Source, invest in ways to assess and mitigate upstream "insider risk" factors (e.g. OpenSSF scorecards, gitsign).

Require the use of hardware security keys for use with GitHub to eliminate this last bastion of GitHub credential exfiltration risk.

Make the use of `gitsign` a required check across the Chainguard orgs to ensure we have a strong sense of contributor identity.

Explore the requirement of SSO for the Wolfi organization (potentially complicated by a future desire for community membership).

Expand the use of Cloud Workstations (or equivalent) for development to partition employee devices from source code that lives in minimal, ephemeral, CVE-free Wolfi-based environments.

Deepen our App/Device trust requirements for interacting with key Chainguard platforms (GitHub, GCP, Salesforce, etc).

Putting it all together

With our end-to-end ownership of this delivery pipeline (including a great deal of its tooling), we are able to gain tremendous visibility into the software supply chain servicing our customers. One of the analogies folks love is the "food recall" analogy. How do you track down everyone that got a shipment of bad red onions?

With our deep visibility into the software composition of our images based on trusted provenance, combined with usage data (who is pulling what images), during a recent high-profile software vulnerability we were able to determine every single affected customer with a single (albeit complex) query, and ensure that they were notified and received updated images extremely quickly.

So (to mix my food metaphors) while we are deeply committed to making it incredibly difficult for any bad "red onions" to make it through our gauntlet of "swiss cheese", we are also committed to building the tools we need to recall anything that inevitably makes it through.