RFC 769 Implemented: Package container build dependencies as Alpine packages

Author: Will Dollman
Date: 2023-01-05
Status: Implemented
Decider:
Input providers: Feroz Abdul Salam, JH Chabran, Keegan Carruthers-Smith
Approvers (please review by EOD 2023-01-20): Feroz Abdul Salam, JH Chabran, Diego Comas, Rafal Gajdulewicz, Keegan Carruthers-Smith
Approvals: Diego Comas Keegan Carruthers-Smith Feroz Abdul Salam
Team: Security

Summary

By packaging container build dependencies, we will increase security and assurance for the contents of our container images, make builds faster and more reliable, and make it easier for developers to add and modify existing dependencies.

Dependencies: any tool, library or binary that we're fetching and installing manually in our container build process that ends up as a part of the final artifact we're shipping to customers.

Background

In addition to Sourcegraph binaries, we frequently need to include other binary dependencies in our published container images. These include tools such as comby, p4-fusion, and ctags.
These dependencies are installed using a variety of different methods, such as copying binaries from other container images, fetching binaries from a third party, or building a binary from source when container images are built.

Sometimes, these dependencies are available as Alpine packages but we do not use them because we need a very specific version or additional build flags. Therefore, we build the binaries on our own.

This RFC is focused on creating a common, safe way to use third-party dependencies in container images, rather than changing the way we build and deploy existing internal tooling.

For example, scip-ruby has its own Bazel-based build process and would not fall in scope of this RFC.

This RFC has a companion Notebook which illustrates the issues and proposed solution using code snippets from our Dockerfiles and other build scripts.

Problem

The way we fetch and build our dependencies has a number of issues. Some of these have an impact on the security of the container images that we ship to customers, while others impact the build process itself.

Reliance on third-party sources

If a dependency's source is unavailable at build time, or if the resource has been removed, the build will fail. This makes our builds more fragile, and reliant on third-party services.
While outages on large providers such as GitHub are uncommon they do occur and have broken our build pipeline in the past. Dependencies hosted on these providers could be removed without warning, which would break all builds until the issue is resolved.

Missing dependency verification

Dependencies are currently fetched from third-parties without verifying a checksum or signature. This means that a dependency's maintainer, or a malicious actor with access to their infrastructure, could modify the dependency at any time either by changing functionality or inserting malicious code. We would be unable to detect such a modification, and it would be included in our container images.

Multiple ways of fetching dependencies

Having multiple ways of fetching dependencies makes them more difficult to reason about and adds complexity to the build process. Currently, we fetch dependencies by fetching binaries from third parties, copying binaries from other container images, or fetching and building from source.

This also makes it more difficult to reuse dependencies across Dockerfiles, as consideration needs to be given to multi-layer builds, runtime dependencies, and keeping dependencies at the same version.

Slower container image builds

Building a dependency from source is significantly slower than simply using a compiled binary. Several dependencies are built from source, and in the case of p4-fusion we employ a manual build-and-cache system due to the length of time the compilation takes.

While container image layer caching should help avoid binaries being rebuilt unnecessarily, in practice Buildkite logs indicate that binaries are frequently being built from scratch during CI builds.

Runtime dependency management

Runtime dependencies of these components (such as libstdc++ required by p4-fusion) need to be identified and installed manually.

Proposal

We should build and package these dependencies as Alpine packages (APKs). These packages can be built once using CI, stored in a package repository, and installed using apk (a tool used to install and manage APKs).

Adopting packages for dependencies will have a wide range of benefits that address the issues with our existing system:

- Dependencies only need to be fetched from their source once when the package is built, making container image builds more reliable and protecting us from disappearing dependencies.
- Dependencies can be verified using checksums, which can be matched against the official releases. Once a dependency is packaged, it will not be possible for modifications to affect our build process.
- We will have a single way of managing dependencies, making it easier to reuse packages between images. Most engineers are already familiar with apk tools.
- All binaries will be compiled when the package is built, so any container image builds that currently include a binary compilation step will run much faster.
- Runtime dependencies can be specified at the package level, and in many cases are detected automatically.

To build dependencies as APKs, I propose we use Chainguard's Melange build tool. This is an open source tool that allows packages to be built using declarative pipelines, in addition to supporting package signing and software bill of materials (SBOM) tooling. Providing a full SBOM to customers will give them a full view of everything inside our images, and initial discussions with customers indicate that this could speed up the lengthy auditing process new releases go through.

We are in discussions with Chainguard about adopting their Wolfi container OS (see WIP RFC 768), and Melange integrates with their container image build tools. APK packages are supported by both Alpine and Wolfi, so Wolfi adoption is not a prerequisite for adopting dependency packaging.

Melange's reproducible builds make it a good fit with Bazel, and Chainguard have previously worked with the contractor we are collaborating with to drive Bazel adoption. Bazel support would be a nice-to-have in the future, but is not a blocker to adoption.

We will also need to host a package repository, which consists of the packages and an index file. Chainguard can provide this service, so we can consider whether to build or buy here. My preference is to buy in order to move quickly here as there is no lock-in. We can consider building at a later stage; the only complexity here is around generating and signing the index file.

Progress So Far

As part of work to create a proof of concept instance of Sourcegraph running on Wolfi, I've so far packaged the majority of our dependencies using Melange. This results in significantly cleaner Dockerfiles.

Being able to package dependencies with complex build processes such as p4-fusion provides confidence that we will be able to use this tooling to package any other future dependencies we require.

Definition of success

- We will have more confidence in the contents and security of our container image builds
- All existing Dockerfile dependencies will be packaged using Melange, and installed using apk
- Buildkite CI container image builds will be faster and more reliable