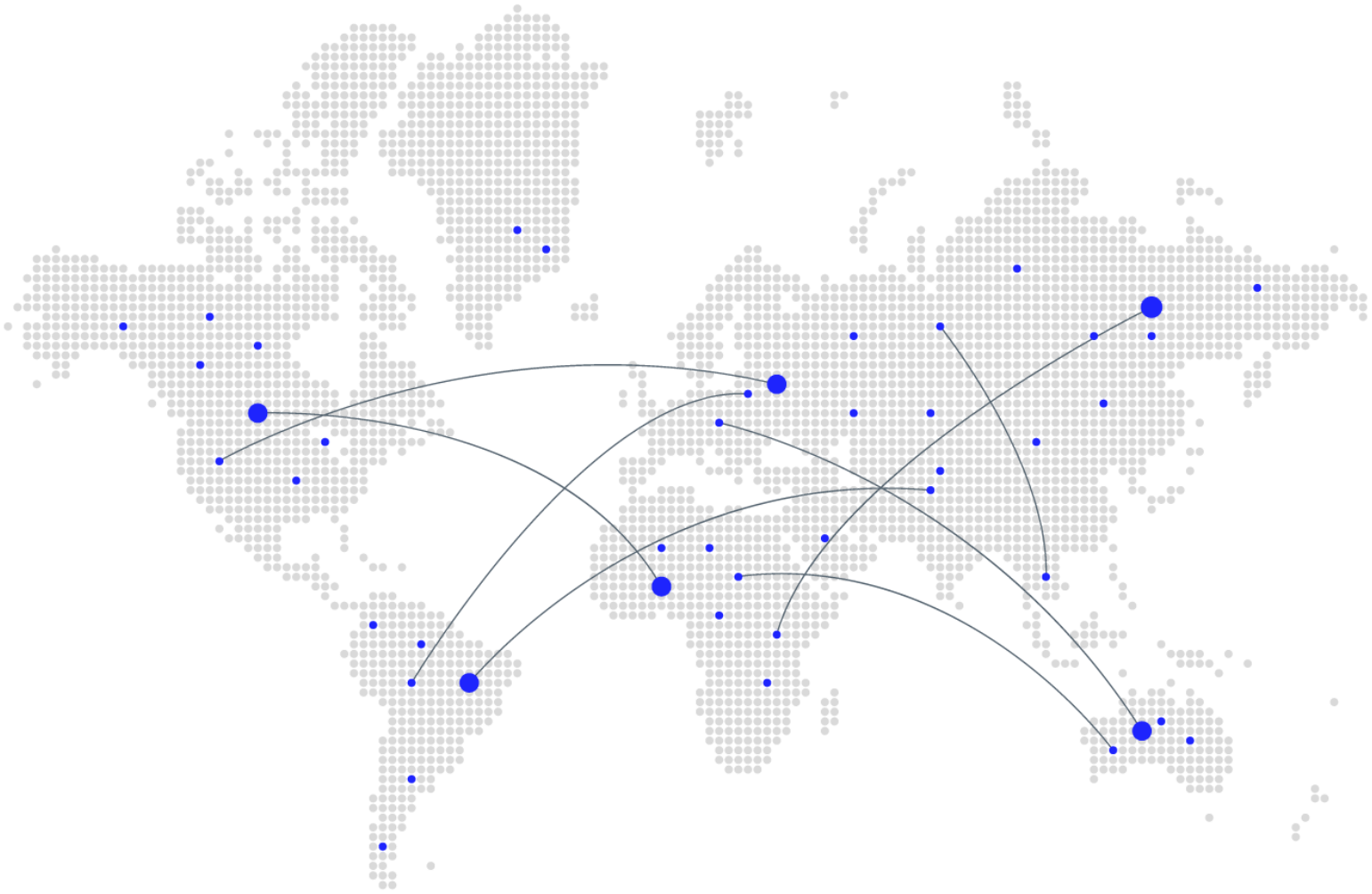
A Software Supply Chain Security Audit of Git



Published in collaboration with

OSTIF.org GitLab Chainguard

Overview

OSTIF asked staff from Chainguard and GitLab to audit the software supply chain security practices of git, the popular open-source source code management project. The audit team attempted to apply the SLSA framework, described below, to git/git (the upstream git project) and to git-for-windows.[1]

The team ultimately concluded that SLSA was an inappropriate framework for evaluating git/git because git/git releases source code directly and not a built artifact such as a binary. The report does nonetheless offer some thoughts on the software supply chain security practices of git/git, namely (1) that the project's vibrant community offers protection against software supply chain attacks even though its software supply chain security practices aren't machine-readable or machine-verifiable in the manner envisioned by frameworks like SLSA and (2) that git/git project does not sign its tags to avoid recursive dependence on itself but does build trust through the use of an uploader tool (kup) that only allows strongly authenticated identities to upload artifacts. Another sub-section also describes how downstream consumers such as git-for-windows are able to build sufficient trust in git's source code.

The team did apply the SLSA framework to git-for-windows. The results are in the "Audit of git-for-windows" section. The audit found that git/git practices do not hinder the software supply chain security practices of the downstream consumer of git/git, in this case, git-for-windows.

What is SLSA?

Supply-chain Levels for Software Artifacts (SLSA, pronounced *salsa*) is a framework for software supply chain integrity. With roots in Google's internal practices and now housed under the umbrella of the Open Source Security Foundation, SLSA defines levels of software supply chain security and a set of practices to achieve these levels.

Version 0.1 of SLSA (at the time of writing, there is active progress towards a 1.0 specification) emphasizes a set of software supply chain security practices that deal with source code, the build process, and provenance. Further information on the 0.1 SLSA specification can be found here: https://slsa.dev/spec/v0.1/requirements

[1] As part of this audit's proceedings, we held written and oral interviews with Randall Becker, a contributor to git and Managing Director of Nexbridge, Inc, Johannes Schindelin, the git-for-windows maintainer, and a number of other git community members. We are thankful for their time and thoughtful insights.

To be sure, SLSA does not cover all aspects of software supply chain security (notably vulnerability management among dependencies) and the framework is admittedly a work-in-progres. That said, the growing momentum behind SLSA and our belief that the application of this framework to the git supply chain could expose the strengths and weaknesses of SLSA motivated us to employ SLSA.

Audit of git/git

git/git is this section's shorthand for the main git project.

Why Is SLSA Not Appropriate for Auditing git/git?

The way that git/git "releases" artifacts make SLSA an inappropriate framework for auditing git/git.

SLSA levels, strictly speaking, apply to built software artifacts. For instance, the provenance document associated with SLSA practices would traditionally document the source code URL, the builder, and the build commands used to create the artifacts. git/git does not build and release software artifacts, at least not in the traditional sense of compiling a binary artifact derived from source code. Instead, git/git, with each tag, releases a signed tarball, not a binary or other built artifact. This tarball is effectively a clone of the source code at a particular tag; therefore a git/git "release" is nothing more than a signature stating that an authorized person chose to cut a tag. Consequently, there are no build artifacts to audit using the framework of SLSA.

Notes on the Strengths and Limits of Software Supply Chain Security Practices in git/git

While SLSA was inappropriate for evaluating git/git, we nevertheless offer some observations gathered in the process of conducting interviews, interacting with the git contributors, and reviewing software artifacts.

Trusting Merges to the git Source Code

The strongest protection against the possibility of a software supply chain attack against git/git arguably lies in the vibrant community of contributors and direct consumers of git. There is a dedicated community of individuals that interact regularly through email, IRC, and other means and that stewards over git/git. Their practices are transparent and

ensure that the spirit of SLSA, especially the source code aspects of SLSA, are satisfied. One might argue that the security of git/git is human-readable; even outsiders can quickly perceive the protective strength of the contributor community.

One limitation is arguably that the software supply chain security practices of git/git are not machine-readable. SLSA especially emphasizes the creation of machine-readable and machine-verifiable documents for protecting the integrity of a software project. But this emphasis on machine-readability is new; git/git predates SLSA. For instance, SLSA emphasizes two trusted persons reviewing all changes to the source code. git/git undoubtedly satisfies this through a vibrant email list culture, though formal and directly machine-readable verification is not possible.

Defining the authoritative clone

The git project does not sign commits or tags[2]. In other projects, a signed tag is a guarantee that the signing identity recognizes all commits before the tag as genuine, vetted changes merged to the source. However, a signed tag for a git/git commit must be signed and verified using git/git itself. This self-signing poses a “who watches the watchmen?” trust problem: any compromise to git/git can impact the trust of the signed git/git tags.

Because there are no signed tags, git/git requires a way to define which clone is the authoritative clone of the repository. Because git/git is designed to be a distributed source control system, any user can clone the source, add commits and tag a new release from their fork. The trust model therefore needs to be external to the source.

As mentioned, the main artifact of a git release is a signed tarball containing the compressed source code at the tagged commit. The source file is uploaded to kernel.org using the *kernel.org uploader* (kup). Kup guarantees that only allowed users, identified by pre-registered ssh keys, can upload new tarballs.

Using the signed tarballs, any downstream consumer can either build git from trusted sources or verify the fork they are working on came from an unmodified mirror that matches the authoritative tree. Downstream, derivative projects can build on the verified tree to add additional levels of trust and secure builds as their needs require them. Let’s see two examples.

---

2 https://github.com/git/git/blob/2b4f5a4e4bb102ac8d967cea653ed753b608193c/Documentation/SubmittingPatches#L406-L411

How git's Release Practices Impact Downstream Consumers

Git-for-Windows

git-for-windows is effectively a distribution of git with some added patches that builds several git binaries and installers. Patches are added and the fork is periodically rebased. There are also build pipelines that compile the various packages git-for-windows releases, some of which bundle third party software, including git's own dependencies but also packages such as bash and curl.

As we'll see in the full SLSA audit of git for windows, git's release process provides enough trust to build a relatively complex distribution mechanism such as git-for-windows.

These two projects showcase how downstream consumers can build enough trust in git's source code, benefiting from the project's strong review process and contributor community. As mentioned earlier, some aspects of the release process make it hard to perform automated verifications or audits, but the core trust to build from the published sources is strong enough. git/git release practices themselves also do not impair others from building more sophisticated release processes.

Audit of git-for-windows

git-for-windows (https://gitforwindows.org/) allows Windows users to take advantage of git. We selected this project given its direct consumption of git/git and therefore its resemblance to a number of other projects that repackage git/git in particular package managers. Additionally, unlike git/git, git-for-windows does build a binary artifact and is an appropriate candidate for a SLSA audit.

SLSA Audit

The git-for-windows SLSA audit resulted in table 1 below. Each category of requirements (source, build, provenance, and contents of provenance) are logically separated. The SLSA level associated with each control (i.e., each row) can be found in

the columns. The green check marks indicated evidence of compliance with a control; red boxes indicate the audit team's inability to find evidence of compliance with a control.

	SLSA Levels			
Source Requirements	1	2	3	4
Version controlled	✓			
Verified history	✓	✓	✓	
Retained indefinitely	✓	✓	✓	✓
Two-person reviewed				
Build Requirements	1	2	3	4
Scripted build	✓			
Build service	✓	✓		
Build as code	✓	✓	✓	
Ephemeral environment	✓	✓	✓	
Isolated	✓	✓	✓	
Parameterless				
Hermetic				
Reproducible				
Provenance	**1**	**2**	**3**	**4**
Available				
Authenticated				
Service generated				
Non-falsifiable				
Dependencies complete				
Contents of Provenance	**1**	**2**	**3**	**4**
Identifies artifact				
Identifies builder				
Identifies build instructions				
Identifies source code				
Identifies entry point				
Includes all build parameters				
Includes all transitive dependencies				
Includes reproducible info				
Includes metadata				

Table 1. SLSA Audit Results for git-for-windows

Note: A checkmark inside a green box indicates the existence of the practice. A red box without the checkmark indicates that the audit team did not find evidence of this practice.

In short, git-for-windows achieves SLSA level 3 for source and build but does not explicitly produce a provenance document, which means that all provenance-related controls are not satisfied.

It's worth noting an important caveat about the provenance results in table 1. The audit's finding that there is no provenance hinges on the fact that the Azure pipelines release process used by the project does not emit a formal provenance document. Releases are not, however, completely untraceable. The audit log provided by Azure Pipelines contains useful data about how artifacts were built including the source code build point, data to locate the build as code configuration, execution parameters and others. Compiling this data into a provenance attestation could provide downstream consumers a machine readable, verifiable record of how artifacts were assembled.