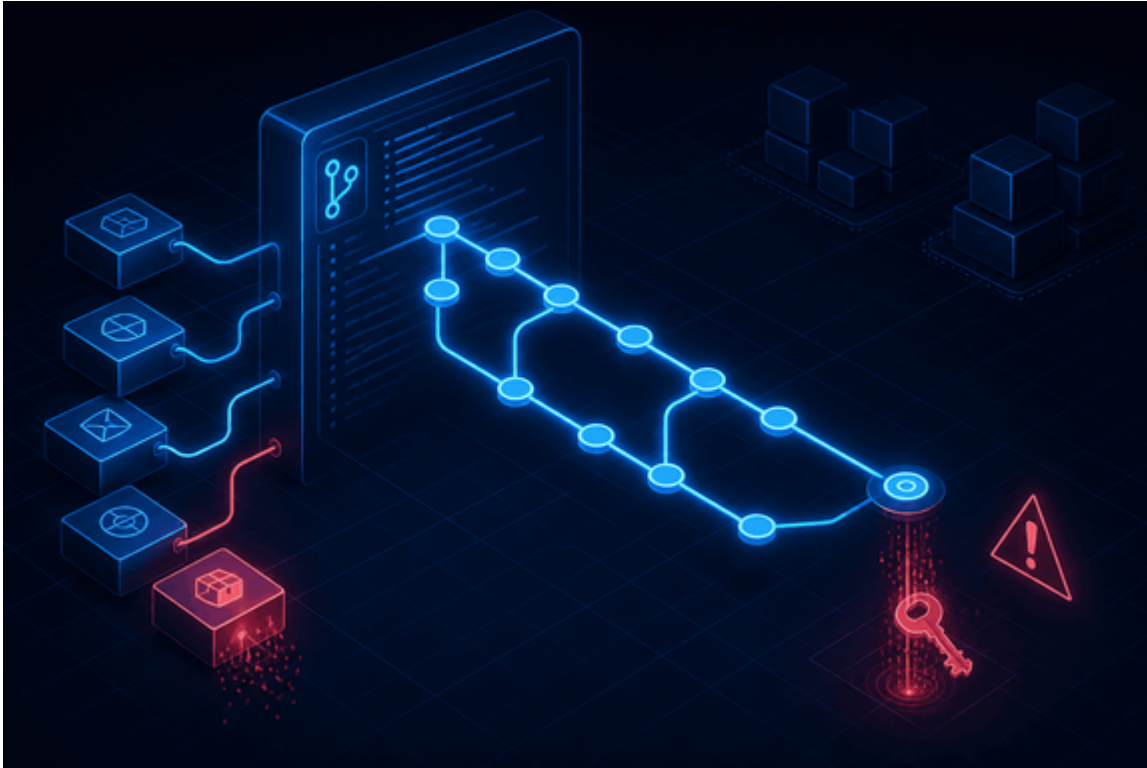


Code Repositories and Third-Party Code: Where Your Risk Actually Lives

Fri, Aug 21, 10:00 AM EDT · <https://scottslab.io/posts/code-repositories-and-third-party-code>



TL;DR

A secret pushed to a public repo is compromised the moment it lands, and deleting the commit does not undo it. Git keeps history, so the only fix is rotation. Borrowed code is the other half: event-stream, dependency confusion and the xz-utils backdoor were all attacks on the supply chain rather than on anyone's application. You own the risk of every line that runs in your name, whether you wrote it, borrowed it, or published it by accident.

Two of the biggest sources of risk in modern software aren't your code at all: they're where you keep it, and whose code you borrowed.

The public/private line is a one-way door

A repository is unambiguously good engineering: version control, an audit log of who changed what, easy reuse. The security wrinkle is the public/private boundary, and the thing to understand is that crossing it is **not reversible**.

Public repositories are scraped continuously by automation that exists specifically to harvest credentials. A committed cloud key is typically found and used in **minutes, not days**. That's long

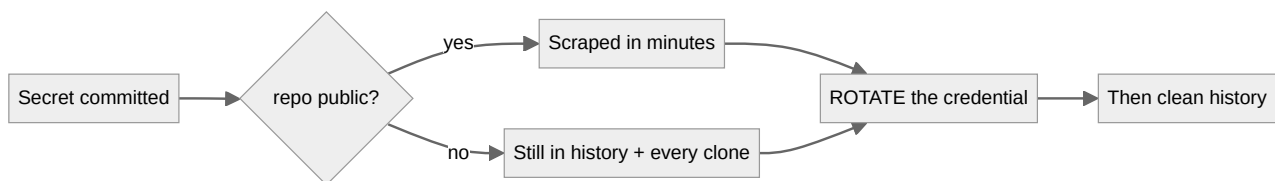
before a human notices. There is no window in which you quietly delete it and get away with it.

And this is the part people get wrong: **deleting the commit does not remove the secret**. Git keeps history. The blob is still reachable by hash, still in every clone anyone made, still in every fork, and, if the repo was ever public, very likely in a third-party mirror you can't reach. Rewriting history helps future clones and does nothing about the copies already taken.

So the only correct response to a leaked credential is **rotate it**. Treat the key as burned the moment it lands, revoke it, and *then* clean the history if you like. A team that spends the first hour rewriting git history instead of revoking the key has spent the hour the attacker needed.

The controls that actually work sit *before* the push: **push protection**, which blocks a commit containing a credential pattern from ever leaving the developer's machine, and secret scanning across history so you learn about the ones already there. Both are cheap and both beat detection after the fact.

The related control is **code integrity measurement**: a cryptographic hash proving the artefact you shipped is the one that was reviewed and approved, so nothing was swapped between approval and release.



The public/private line is a one-way door

Borrowed code is code you own

Libraries and SDKs let you move fast on code you didn't write. You also inherit every flaw in it. That much is obvious. What's less obvious is that the dependency tree is now an *attack surface in its own right*, targeted deliberately. Three shapes are worth knowing because they're structurally different:

Compromising a real package. In 2018 the widely-used `event-stream` npm package was handed over to a new maintainer who added a dependency carrying obfuscated code that targeted the Copay wallet application. It harvested keys from any account holding more than 100 BTC or 1,000 Bitcoin Cash. Nobody was breached by a bug. The maintainer changed.

Dependency confusion. In 2021 Alex Birsan showed that if your build resolves package names from both a public registry and a private one, publishing a *public* package with the same name as your *internal* one, at a higher version, can make the build fetch the attacker's. It worked against Apple, Microsoft and PayPal, and it needed no vulnerability at all: just how resolution order works.

The long game. The xz-utils backdoor (CVE-2024-3094, 2024) was a multi-year effort in which a contributor built trust, gained maintainer rights, and slipped a backdoor into a compression library in a path that reached sshd. It was found by an engineer investigating a *half-second* login slowdown. It was very nearly in every major Linux distribution.

The lesson from all three is the same: **your dependency review cannot be a one-time check at adoption**, because the package that was safe when you chose it can change hands afterwards.

What to actually do about it

Pin and use a lockfile. A lockfile records the exact resolved versions and their hashes for the whole tree, including transitive dependencies. Without one, "the same build" isn't the same build.

Verify integrity, not just version. Hashes in the lockfile mean a tampered artefact fails to install rather than silently succeeding.

Know your transitive tree. Nobody is compromised by the twelve dependencies they chose; they're compromised by the eight hundred those twelve pulled in. Software Composition Analysis is how you answer "am I running it?" in minutes rather than three days of grepping.

Configure your resolver explicitly. Dependency confusion is a *configuration* bug. Scope your internal packages and make sure private names cannot be satisfied from a public registry.

Apply the same testing rigour to borrowed code. "A vendor wrote it" is not a security assessment.

The model

You own the risk of every line that runs in your name, whether you wrote it, borrowed it, or accidentally published it.

Repository hygiene keeps your secrets from becoming someone else's. Dependency tracking keeps someone else's bug from becoming your breach. Neither is glamorous. Both are where real incidents actually start: not in clever exploits against code you wrote, but in a key someone pasted and a package someone trusted.