Proposal: Secure the Public Go Module Ecosystem with the Go Notary

Russ Cox
Filippo Valsorda

Last updated: March 4, 2019.

golang.org/design/25530-notary

Discussion at golang.org/issue/25530.

Abstract

We propose to secure the public Go module ecosystem by introducing a new server, the Go notary, which serves what is in effect a `go.sum` file listing all publicly-available Go modules. The `go` command will use this service to fill in gaps in its own local `go.sum` files, such as during `go get -u`. This ensures that unexpected code changes cannot be introduced when first adding a dependency to a module or when upgrading a dependency.

Background

When you run `go get rsc.io/quote@v1.5.2`, `go get` first fetches `https://rsc.io/quote?go-get=1` and looks for `<meta>` tags. It finds

```
<meta name="go-import"
      content="rsc.io/quote git https://github.com/rsc/quote">
```

which tells the code is in a Git repository on `github.com`. Next it runs `git clone https://github.com/rsc/quote` to fetch the Git repository and then extracts the file tree from the `v1.5.2` tag, producing the actual module archive.

Historically, `go get` has always simply assumed that it was downloading the right code. An attacker able to intercept the connection to `rsc.io` or `github.com` (or an attacker able to break into one of those systems, or a malicious module author) would be able to cause `go get` to download different code tomorrow, and `go get` would not notice.

There are many challenges in using software dependencies safely, and much more vetting should typically be done before taking on a new dependency, but no amount of vetting is worth anything if the code you download and vet today differs from the code you or a collaborator downloads tomorrow for the “same” module version. We must be able to authenticate whether a particular download is correct.

For our purposes, “correct” for a particular module version download is defined as the same code everyone else downloads. This definition ensures reproducibility of builds and makes vetting of specific module versions meaningful, without needing to attribute specific archives to specific authors, and without introducing new potential points of compromise like per-author keys. (Also, even the author of a module should not be able to change the bits associated with a specific version from one day to the next.)

Being able to authenticate a particular module version download effectively moves code hosting servers like `rsc.io` and `github.com` out of the trusted computing base of the Go module ecosystem. With module authentication, those servers could cause availability problems by not serving a module version anymore, but they cannot substitute different code. The introduction of Go module proxies (see `go help goproxy`) introduces yet another way for an attacker to intercept module downloads; module authentication eliminates the need to trust those proxies as well, moving them outside trusted computing base.

See the Go blog post “Go Modules in 2019” for additional background.

Module Authentication with `go.sum`

Go 1.11’s preview of Go modules introduced the `go.sum` file, which is maintained automatically by the `go` command in the root of a module tree and contains cryptographic checksums for the content of each dependency of that module. If a module’s source file tree is obtained unmodified, then the `go.sum` file allows authenticating all dependencies needed for a build of that module. It ensures that tomorrow’s builds will use the same exact code for dependencies that today’s builds did. Tomorrow’s downloads are authenticated by `go.sum`.

On the other hand, today’s downloads—the ones that add or update dependencies in the first place—are not authenticated. When a dependency is first added to a module, or when a dependency is upgraded to a newer version, there is no entry for it in `go.sum`, and the `go` command today blindly trusts that it downloads the correct code. Then it records the hash of that code into `go.sum` to ensure that code doesn’t change tomorrow. But that doesn’t help the initial download. The model is similar to SSH’s “trust on first use,” and while that approach is an improvement over “trust every time,” it’s still not ideal, especially since developers typically download new module versions far more often than they connect to new, unknown SSH servers.

We are concerned primarily with authenticating downloads of publicly-available module versions. We assume that the private servers hosting private module source code are already within the trusted computing base of the developers using that code. In contrast, a developer who wants to use `rsc.io/quote` should not be required to trust that `rsc.io` is properly secured. This trust becomes particularly problematic when summed over all dependencies.

What we need is an easily-accessed `go.sum` file listing every publicly-available module version. But we don’t want to blindly trust a downloaded `go.sum` file, since that would become the next attractive target for an attacker.

Transparent Logs

The Certificate Transparency project is based on a data structure called a _transparent *log*. The transparent log is hosted on a server and made accessible to clients for random access, but clients are still able to verify that a particular log record really is in the log and also that the server never removes any log record from the log. Separately, third-party auditors can iterate over the log checking that the entries themselves are accurate. These two properties combined mean that a client can use records from the log, confident that those records will remain available in the log for auditors to double-check and report invalid or suspicious entries. Clients and auditors can also compare observations to ensure that the server is showing the same data to everyone involved.

That is, the log server is not trusted to store the log properly, nor is it trusted to put the right records into the log. Instead, clients and auditors interact skeptically with the server, able to verify for themselves in each interaction that the server really is behaving correctly.

For details about the data structure, see Russ Cox's blog post, "Transparent Logs for Skeptical Clients." For a high-level overview of Certificate Transparency along with additional motivation and context, see Ben Laurie's ACM Queue article, "Certificate Transparency: Public, verifiable, append-only logs."

The use of a transparent log for module hashes aligns with a broader trend of using transparent logs to enable detection of misbehavior by partially trusted systems, what the Trillian team calls "General Transparency."

Proposal

We propose to publish the `go.sum` lines for all publicly-available Go modules in a transparent log, served by a new server called the Go notary. When a publicly-available module is not yet listed in the main module's `go.sum` file, the `go` command will fetch the relevant `go.sum` lines from the notary instead of trusting the initial download to be correct.

Notary Server

The Go notary will run at `https://notary.golang.org/` and serve the following endpoints:

- `/latest` will serve a signed tree size and hash for the latest log.
- `/lookup/M@V` will serve the log record number for the entry about module M version V, followed by the data for the record. If the module version is not yet recorded in the log, the notary will try to fetch it before replying. Note that the data should never be used without first authenticating it against a signed tree hash.
- `/record/R` will serve the data for record number R.
- `/tile/H/L/K[.p/W]` will serve a log tile. The optional `.p/W` suffix indicates a partial log tile with only `W` hashes.

Proxying a Notary

A module proxy can also proxy requests to the notary. The general proxy URL form is `<proxyURL>/notary/<notaryURL>`. If `GOPROXY=https://proxy.site` then the latest signed tree would be fetched using `https://proxy.site/notary/notary.golang.org/latest`. Including the full notary URL allows a transition to a new notary log, such as `notary.golang.org/v2`.

Before accessing any notary URL using a proxy, the proxy client should first fetch `<proxyURL>/notary/supported`. If that request returns a successful (HTTP 200) response, then the proxy supports proxying notary requests. In that case, the client should use the proxied notary only, never falling back to a direct connection to the notary. If the `/notary/supported` check fails with a “not found” (HTTP 404) response, the proxy is unwilling to proxy the notary, and the client should connect directly to the notary. Any other response is treated as the notary being unavailable.

A corporate proxy may want to ensure that clients never make any direct notary connections (for example, for privacy; see the “Rationale” section below). The optional `/notary/supported` endpoint, along with proxying actual notary requests, lets such a proxy ensure that a `go` command using the proxy never makes a direct connection to notary.golang.org. But simpler proxies may wish to focus on serving only modules and not notary data—in particular, module-only proxies can be served from entirely static file systems, with no special infrastructure at all. Such proxies can respoond with an HTTP 404 to the `/notary/supported` endpoint, so that clients will connect to the notary directly.

`go` command client

The `go` command is the primary consumer of the notary’s published log. The `go` command will verify the log as it uses it, ensuring that every record it reads is actually in the log and that no observed log ever drops a record from an earlier observed log.

The `go` command will store the notary’s public key in `$GOROOT/lib/notary/notary.cfg`. That file will also contain the default starting signed tree size and tree hash, updated with each major release.

The `go` command will then cache the latest signed tree size and tree hash in `$GOPATH/pkg/notary/notary.golang.org/latest`. It will cache tiles in `$GOPATH/pkg/mod/download/cache/notary/notary.golang.org/tile/H/L/K[.W]`. These two different locations let `go clean -modcache` delete any cached tiles as well, but no `go` command (only a manual `rm -rf $GOPATH/pkg`) will wipe out the memory of the latest observed tree size and hash. If the `go` command ever does observe a pair of inconsistent signed tree sizes and hashes, it will complain loudly on standard error and fail the build.

The `go` command must be configured to know which modules are publicly available and therefore can be verified by the notary, versus those that are closed source and must not be verified, especially since that would transmit potentially private import paths over the network to the notary `/lookup` endpoint. A few new environment variables control this configuration. (See the `go env -w` proposal for a way to manage these variables more easily.)

- `GOPROXY=https://proxy.site/path` sets the Go module proxy to use, as before.
- `GONOPROXY=prefix1,prefix2,prefix3` sets a list of module path prefixes, possibly containing globs, that should not be proxied. For example:

  ```
  GONOPROXY=*.corp.google.com,rsc.io/private
  ```

  will bypass the proxy for the modules foo.corp.google.com, foo.corp.google.com/bar, rsc.io/private, and rsc.io/private/bar, though not rsc.io/privateer (the patterns are path prefixes, not string prefixes).

- `GONOVERIFY=prefix1,prefix2,prefix3` sets a list of module path prefixes, again possibly containing globs, that should not be verified using the notary.

  We expect that corporate environments may fetch all modules, public and private, through an internal proxy; `GONOVERIFY` allows them to disable notary-based verification of internal modules while still verifying public modules. Therefore, `GONOVERIFY` must not imply `GONOPROXY`.

  We also expect that other users may prefer to connect directly to source origins but still want verification of open source modules or proxying of the notary itself; `GONOPROXY` allows them to arrange that and therefore must not imply `GONOVERIFY`.

The notary not being able to report `go.sum` lines for a module version is a hard failure: any private modules must be explicitly listed in `$GONOVERIFY`. (Otherwise an attacker could block traffic to the notary and make all module versions appear to verify.) The notary can be disabled entirely with `GONOVERIFY=*`. The command `go get -insecure` will report but not stop after notary failures.

Rationale

The motivation for authenticating module downloads is covered in the background section above. Note that we want to authenticate modules obtained both from direct connections to code-hosting servers and from module proxies.

Two topics are worth further discussion: first, having a single notary service for the entire Go ecosystem, and second, the privacy implications of a notary.

One Notary

The Go team at Google will run the Go notary as a service to the Go ecosystem, similar to running `godoc.org` and `golang.org`. There is no plan to allow use of alternate notaries, which would add complexity and potentially reduce the overall security of the system, allowing different users to be attacked by compromising different notaries.

We originally considered having multiple notaries signing individual `go.sum` entries and requiring the `go` command to collect signatures from a quorum of notaries before accepting an entry. That design depended on the uptime of multiple services and could still be compromised undetectably by compromising enough notaries. That is, that design would blindly trust a quorum of notaries.

The design presented here uses the transparent log eliminates blind trust in a quorum of notaries and instead uses a “trust but verify” model with a single notary. In this design, the notary’s published `go.sum` lines are accepted by the `go` command client, but the published lines are also verifiably preserved for auditing by any interested third party. In fact, we hope that proxies run by various organizations in the Go community will serve as auditors and double-check Go notary log entries as part of their ordinary operation. Another useful service that could be enabled by the notary is a notification service to alert authors about new versions of their own modules.

Privacy

Contacting the Go notary to authenticate a new dependency requires sending the module path and version to the notary. There are two potential privacy concerns. First, a misconfigured `go` command might send the names of private module paths (for example, `rsc.io/private/secret-plan`) to the notary. The notary would try to fetch the module and fail, but the path would have been exposed in the network traffic. Second, even using only public modules, there might be a concern that contacting the notary at all would expose information about how popular particular modules are in a particular organization (or at least in a particular client IP block).

The design addresses these two privacy concerns in two ways: with both a lightweight, partial solution for each and a heavier, complete solution.

The lightweight, partial solution for a misconfigured `go` command that asks the notary about a non-public module is to make it fail as loudly as possible. If the `go` command is configured to ask the notary about a particular module, and the notary cannot return information about that module, the download fails and the `go` command stops. This ensures both that all public modules are in fact authenticated and also that any misconfiguration must be corrected (by setting `$GONOVERIFY` to avoid the notary for those private modules) in order to achieve a successful build. This way, the frequency of misconfiguration should be minimized.

The lightweight, partial solution for exposing information about module usage is to only contact the notary when there is not already an entry in `go.sum`. If a module version is already listed in `go.sum`, it is assumed to be correct, with no notary interaction. This allows authentication of previously-downloaded private modules and also ensures that only the first use of a new module version is exposed to the notary.

These lightweight solutions are meant to make the notary usable out of the box for most Go developers. If there are additional lightweight solutions that can be adopted to further reduce privacy concerns, we would be happy to consider them.

The heavier, complete solution for notary privacy concerns is for developers to put their usage behind a proxy, such as a local Athens instance or JFrog's GoCenter, assuming those proxies add support for proxying and caching the Go notary service endpoints. (Those endpoints are designed to be highly cacheable for exactly this reason, and a proxy with a full copy of the notary log doesn't have to leak any information about what modules are in use, at the cost of maintaining its own index to answer lookup requests.) We anticipate that there will be many proxies available for use in the Go ecosystem. Part of the motivation for the Go notary is to allow the use of any available proxy to download modules, without any reduction in security. Developers can then use any proxy they are comfortable using, or run their own.

Compatibility

The introduction of the notary does not have any compatibility concerns at the command or language level. However, proxies that serve modified copies of public modules will be incompatible with the notary and stop being usable. This is by design: such proxies are indistinguishable from man-in-the-middle attacks.

Implementation

The Go team at Google is working on a production implementation of both a Go module proxy and the Go notary, as we described in the blog post "Go Modules in 2019."

We will publish a notary client as part of the `go` command, as well as an example notary implementation. We intend to ship support for the notary, enabled by default, in Go 1.13.

Russ Cox will lead the `go` command integration and has posted a stack of changes in golang.org/x/exp/notary.