Proposal: Secure the Public Go Module Ecosystem

Russ Cox
Filippo Valsorda

Last updated: April 24, 2019.

golang.org/design/25530-sumdb

Discussion at golang.org/issue/25530.

Abstract

We propose to secure the public Go module ecosystem by introducing a new server, the Go checksum database, 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.

The original name for the Go checksum database was "the Go notary," but we have stopped using that name to avoid confusion with the CNCF Notary project, itself written in Go, not to mention the Apple Notary.

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 it 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 checksum database. 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 checksum database instead of trusting the initial download to be correct.

Checksum Database

The Go checksum database will run at `https://sum.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 (that is, the `go.sum` lines for module M version V) and a signed tree hash for a tree that contains 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 the signed tree hash and authenticating the signed tree hash against the client's timeline of signed tree hashes.
- `/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. Clients must fall back to fetching the full tile if a partial tile is not found. The record data for the leaf hashes in `/tile/H/0/K[.p/W]` are served as `/tile/H/data/K[.p/W]` (with a literal `data` path element).

Clients are expected to use `/lookup` and `/tile/H/L/...` during normal operations, while auditors will want to use `/latest` and `/tile/H/data/...`. A special `go` command may also fetch `/latest` to force incorporation of that signed tree head into the local timeline.

Proxying a Checksum Database

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

Before accessing any checksum database URL using a proxy, the proxy client should first fetch `<proxyURL>/sumdb/<sumdb-name>/supported`. If that request returns a successful (HTTP 200)

response, then the proxy supports proxying checksum database requests. In that case, the client should use the proxied access method only, never falling back to a direct connection to the database. If the `/sumdb/<sumdb-name>/supported` check fails with a "not found" (HTTP 404) or "gone" (HTTP 410) response, the proxy is unwilling to proxy the checksum database, and the client should connect directly to the database. Any other response is treated as the database being unavailable.

A corporate proxy may want to ensure that clients never make any direct database connections (for example, for privacy; see the "Rationale" section below). The optional `/sumdb/supported` endpoint, along with proxying actual database requests, lets such a proxy ensure that a `go` command using the proxy never makes a direct connection to sum.golang.org. But simpler proxies may wish to focus on serving only modules and not checksum data—in particular, module-only proxies can be served from entirely static file systems, with no special infrastructure at all. Such proxies can respond with an HTTP 404 or HTTP 410 to the `/sumdb/supported` endpoint, so that clients will connect to the database directly.

`go` command client

The `go` command is the primary consumer of the database'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 refer to `$GOSUMDB` to find the name and public key of the Go checksum database. That variable will default to the `sum.golang.org` server.

The `go` command will cache the latest signed tree size and tree hash in `$GOPATH/pkg/sumdb/<sumdb-name>/latest`. It will cache lookup results and tiles in `$GOPATH/pkg/mod/download/cache/sumdb/<sumdb-name>/lookup/path@version` and `$GOPATH/pkg/mod/download/cache/sumdb/<sumdb-name>/tile/H/L/K[.W]`. (More generally, `https://<sumdb-URL>` is cached in `$GOPATH/pkg/mod/download/cache/sumdb/<sumdb-URL>`.) This way, `go clean -modcache` deletes cached lookup results and tiles but not the latest signed tree hash, which should be preserved for detection of timeline inconsistency. 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 looked up in the checksum database, versus those that are closed source and must not be looked up, especially since that would transmit potentially private import paths over the network to the database `/lookup` endpoint. A few new environment variables control this configuration. (See the `go env -w` proposal, now available in the Go 1.13 development branch, 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).

- `GOSUMDB=<sumdb-key>` sets the Go checksum database to use, where `<sumdb-key>` is a verifier key as defined in package note.
- `GONOSUMDB=prefix1,prefix2,prefix3` sets a list of module path prefixes, again possibly containing globs, that should not be looked up using the database.

  We expect that corporate environments may fetch all modules, public and private, through an internal proxy; `GONOSUMDB` allows them to disable checksum database lookups for internal modules while still verifying public modules. Therefore, `GONOSUMDB` 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 database itself; `GONOPROXY` allows them to arrange that and therefore must not imply `GONOSUMDB`.

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

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 database server for the entire Go ecosystem, and second, the privacy implications of a database server.

Security

The Go team at Google will run the Go checksum database as a service to the Go ecosystem, similar to running `godoc.org` and `golang.org`. It is important that the service be secure. Our thinking about the security design of the database has evolved over time, and it is useful to outline the evolution that led to the current design.

The simplest possible approach, which we never seriously considered, is to have one trusted server that issues a signed certificate for each module version. The drawback of this approach is that a compromised server can be used to sign a certificate for a compromised module version, and then that compromised module version and certificate can be served to a target victim without easy detection.

One way to address this weakness is strength in numbers: have, say, N=3 or N=5 organizations run independent servers, gather certificates from all of them, and accept a module version as valid when, say, (N+1)/2 certificates agree. The two drawbacks of this approach are that it is significantly more expensive and still provides no detection of actual attacks. The payoff from targeted replacement of source code could be high enough to justify silently compromising (N+1)/2 notaries and then making very selective use of the certificates. So our focus turned to detection of compromise.

Requiring a checksum database to log a `go.sum` entry in a transparent log before accepting it does raise the likelihood of detection. If the compromised `go.sum` entry is stored in the actual log, an auditor can find it. And if the compromised `go.sum` entry is served in a forked, victim-specific log, the

server must always serve that forked log to the victim, and only to the victim, or else the `go` command's consistency checks will fail loudly, and with enough information to cryptographically prove the compromise of the server.

An ecosystem with multiple proxies run by different organizations makes a successful “forked log” attack even harder: the attacker would have to not only compromise the database, it would also have to compromise each possible proxy the victim might use and arrange to identify the victim well enough to always serve the forked log to the victim and to never serve it to any non-victim.

The serving of the transparent log in tile form helps caching and proxying but also makes victim identification that much harder. When using Certificate Transparency's proof endpoints, the proof requests might be arranged to carry enough material to identify a victim, for example by only ever serving an even log sizes to the victim and odd log sizes to others and then adjusting the log-size-specific proofs accordingly. But complete tile fetches expose no information about the cached log size, making it that much harder to serve modified tiles only to the victim.

We hope that proxies run by various organizations in the Go community will also serve as auditors and double-check Go checksum database log entries as part of their ordinary operation. (Another useful service that could be enabled by the database is a notification service to alert authors about new versions of their own modules.)

As described earlier, users who want to ensure their own compromise requires compromising multiple organizations can use Google’s checksum database and a different organization’s proxy to access it.

Generalizing that approach, the usual way to further improve detection of fork attacks is to add gossip, so that different users can check whether they are seeing different logs. In effect, the proxy protocol already supports this, so that any available proxy that proxies the database can be a gossip source. If we add a `go fetch-latest-chccksum-log-from-goproxy` (obviously not the final name) and

```
GOPROXY=https://other.proxy/ go fetch-latest-checksum-log-from-goproxy
```

succeeds, then the client and other.proxy are seeing the same log.

Compared to the original scenario of a single checksum database with no transparent log, the use of a single transparent log and the ability to proxy the database and gossip improves detection of attacks so much that there is little incremental security benefit to adding the complexity of multiple notaries. At some point in the future, it might make sense for the Go ecosystem to support using multiple databases, but to begin with we have opted for the simpler (but still reasonably secure) ecosystem design of a single database.

Privacy

Contacting the Go checksum database to authenticate a new dependency requires sending the module path and version to the database server.

The database server will of course need to publish a privacy policy, and it should be written as clearly as the Google Public DNS Privacy Policy and be sure to include information about log retention windows. That policy is still under development. But the privacy policy only matters for data the database receives. The design of the database protocol and usage is meant to minimize what the `go` command even sends.

There are two main privacy concerns: exposing the text of private modules paths to the database, and exposing usage information for public modules to the databas.

Private Module Paths

The first main privacy concern is that a misconfigured `go` command could send the text of a private module path (for example, `secret-machine.rsc.io/private/secret-plan`) to the database. The database will try to resolve the module, triggering a DNS lookup for `secret-machine.rsc.io` and, if that resolves, an HTTPS fetch for the longer URL. Even if the database then discards that path immediately upon failure, it has still been sent over the network.

Such misconfiguration must not go unnoticed. For this reason (and also to avoid downgrade attacks), if the database cannot return information about a module, the download fails loudly 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 `$GONOSUMDB` to avoid the database for those private modules) in order to achieve a successful build. This way, the frequency of misconfiguration-induced database lookups should be minimized. Misconfigurations fail; they will be noticed and fixed.

One possibility to further reduce exposure of private module path text is to provide additional ways to set `$GONOSUMDB`, although it is not clear what those should be. A top-level module's source code repository is an attractive place to want to store configuration such as `$GONOSUMDB` and `$GOPROXY`, but then that configuration changes depending on which version of the repo is checked out, which would cause interesting behavior when testing old versions, whether by hand or using tools like `git bisect`.

(The nice thing about environment variables is that most corporate computer management systems already provide ways to preset environment variables.)

Private Module SHA256s

Another possibility to reduce exposure is to support and use by default an alternate lookup `/lookup/SHA256(module)@version`, which sends the SHA256 hash of the module path instead of the module path instead. If the database was already aware of that module path, it would recognize the SHA256 and perform the lookup, even potentially fetching a new version of the module. If a misconfigured `go` command sends the SHA256 of a private module path, that is far less information.

The SHA256 scheme does require, however, that the first use of a public module be accompanied by some operation that sends its module path text to the database, so that the database can update its inverse-SHA256 index. That operation—for now, let's call it `go notify <modulepath>`—would need to be run just once ever across the whole Go ecosystem for each module path. Most likely the author would do it, perhaps as part of the still-hypothetical `go release` command, or else the first user of the module would need to do it (perhaps thinking carefully about being the first-ever user of the module!).

A modification of the SHA256 scheme might be to send a truncated hash, designed to produce K-anonymity, but this would cause significant expense: if the database identified K public modules with the truncated hash, it would have to look up the given version tag for all K of them before returning an answer. This seems needlessly expensive and of little practical benefit. (An attacker might even create a long list of module paths that collide with a popular module, just to slow down requests.)

The SHA256 + `go notify` scheme is not part of this proposal today, but we are considering adding it, with full hashes, not truncated ones.

Public Module Usage Information

The second main privacy concern is that even developers who use only public modules would expose information about their module usage habits by requesting new `go.sum` lines from the database.

Remember that the `go` command only contacts the database in order to find new lines to add to `go.sum`. When `go.sum` is up-to-date, as it is during ordinary development, the database is never contacted. That is, the database is only involved at all when adding a new dependency or changing the version of an existing one. That significantly reduces the amount of usage information being sent to the database in the first place.

Note also that even `go get -u` does not request information about every dependency from the database: it only requests information about dependencies with updates available.

The `go` command will also cache database lookup results (reauthenticating them against cached tiles at each use), so that using a single computer to upgrade the version of a particular dependency used by N different modules will result in only one database lookup, not N. That further reduces the strength of any usage signal.

One possible way to even further reduce the usage signal observable by the database might be to use a truncated hash for K-anonymity, as described in the previous section, but the efficiency problems described earlier still apply. Also, even if any particular fetch downloaded information for K different module paths, the likely-very-lopsided popularity distribution might make it easy to guess which module path a typical client was really looking for, especially combined with version information. Truncated hashes appear to cost more than the benefit they would bring.

The complete solution for not exposing either private module path text or public module usage information is to us a proxy or a bulk download.

Privacy by Proxy

A complete solution for database privacy concerns is to for developers to access the database only through a proxy, such as a local Athens instance or JFrog Artifactory instance, assuming those proxies add support for proxying and caching the Go database service endpoints.

The proxy can be configured with a list of private module patterns, so that even requests from a misconfigured `go` command never not make it past the proxy. The database endpoints are designed for cacheability, so that a proxy can avoid making any request more than once. Requests for new versions of modules would still need to be relayed to the database.

We anticipate that there will be many proxies available for use in the Go ecosystem. Part of the motivation for the Go checksum database 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.

Privacy by Bulk Download

What little usage signal leaks from a proxy that aggressively caches database queries can be removed entirely by instead downloading the entire checksum database and answering requests using the local copy. We estimate that the Go ecosystem has around 3 million module versions. At an estimated footprint of 200 bytes per module version, a much larger, complete checksum database of even 100 million module versions would still only be 20 GB. Bandwidth can be exchanged for complete anonymity by downloading the full database once and thereafter updating it incrementally (easy, since it is append-only). Any queries can be answered using only the local copy, ensuring that neither private module

only). Any queries can be answered using only the local copy, ensuring that neither private module paths nor public module usage is exposed. The cost of this approach is the need for a clients to download the entire database despite only needing an ever-smaller fraction of it. (Today, assuming only a 3-million-entry database, a module with even 100 dependencies would be downloading 30,000 times more database than it actually needs. As the Go ecosystem grows, so too does the overhead factor.)

Downloading the entire database might be a good strategy for a corporate proxy, however.

Privacy in CI/CD Systems

A question was raised about privacy of database operations especially in CI/CD systems. We expect that a CI/CD system would *never* contact the database.

First, in typical usage, you only push code to a CI/CD system after first at least building (and hopefully also testing!) any changes locally. Building any changes locally will update `go.mod` and `go.sum` as needed, and then the `go.sum` pushed to the CI/CD system will be up-to-date. The database is only involved when adding to `go.sum`.

Second, module-aware CI/CD systems should already be using `-mod=readonly`, to fail on out-of-date `go.mod` files instead of silently updating them. We will ensure that `-mod=readonly` also fails on out-of-date `go.sum` files if it does not already (#30667).

Compatibility

The introduction of the checksum database 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 new checks 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 checksum database, as we described in the blog post "Go Modules in 2019."

We will publish a checksum database client as part of the `go` command, as well as an example database implementation. We intend to ship support for the checksum database, 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.