jwt-go

A go (or 'golang' for search engine friendliness) implementation of JSON Web Tokens

NEW VERSION COMING: There have been a lot of improvements suggested since the version 3.0.0 released in 2016. I'm working now on cutting two different releases: 3.2.0 will contain any non-breaking changes or enhancements. 4.0.0 will follow shortly which will include breaking changes. See the 4.0.0 milestone to get an idea of what's coming. If you have other ideas, or would like to participate in 4.0.0, now's the time. If you depend on this library and don't want to be interrupted, I recommend you use your dependency mangement tool to pin to version 3.

SECURITY NOTICE: Some older versions of Go have a security issue in the cryotp/elliptic. Recommendation is to upgrade to at least 1.8.3. See issue #216 for more detail.

SECURITY NOTICE: It's important that you validate the `alg` presented is what you expect. This library attempts to make it easy to do the right thing by requiring key types match the expected alg, but you should take the extra step to verify it in your usage. See the examples provided.

What the heck is a JWT?

JWT.io has a great introduction to JSON Web Tokens.

In short, it's a signed JSON object that does something useful (for example, authentication). It's commonly used for `Bearer` tokens in Oauth 2. A token is made of three parts, separated by `.`'s. The first two parts are JSON objects, that have been base64url encoded. The last part is the signature, encoded the same way.

The first part is called the header. It contains the necessary information for verifying the last part, the signature. For example, which encryption method was used for signing and what key was used.

The part in the middle is the interesting bit. It's called the Claims and contains the actual stuff you care about. Refer to the RFC for information about reserved keys and the proper way to add your own.

What's in the box?

This library supports the parsing and verification as well as the generation and signing of JWTs. Current supported signing algorithms are HMAC SHA, RSA, RSA-PSS, and ECDSA, though hooks are present for adding your own.

Examples

See the project documentation for examples of usage:

- Simple example of parsing and validating a token
- Simple example of building and signing a token
- Directory of Examples

Extensions

This library publishes all the necessary components for adding your own signing methods. Simply implement the `SigningMethod` interface and register a factory method using `RegisterSigningMethod`.

Here's an example of an extension that integrates with multiple Google Cloud Platform signing tools (AppEngine, IAM API, Cloud KMS): https://github.com/someone1/gcp-jwt-go

Compliance

This library was last reviewed to comply with RTF 7519 dated May 2015 with a few notable differences:

- In order to protect against accidental use of Unsecured JWTs, tokens using `alg=none` will only be accepted if the constant `jwt.UnsafeAllowNoneSignatureType` is provided as the key.

Project Status & Versioning

This library is considered production ready. Feedback and feature requests are appreciated. The API should be considered stable. There should be very few backwards-incompatible changes outside of major version updates (and only with good reason).

This project uses Semantic Versioning 2.0.0. Accepted pull requests will land on `master`. Periodically, versions will be tagged from `master`. You can find all the releases on the project releases page.

While we try to make it obvious when we make breaking changes, there isn't a great mechanism for pushing announcements out to users. You may want to use this alternative package include: `gopkg.in/dgrijalva/jwt-go.v3`. It will do the right thing WRT semantic versioning.

BREAKING CHANGES:*

- Version 3.0.0 includes *a lot* of changes from the 2.x line, including a few that break the API. We've tried to break as few things as possible, so there should just be a few type signature changes. A full list of breaking changes is available in `VERSION_HISTORY.md`. See `MIGRATION_GUIDE.md` for more information on updating your code.

Usage Tips

Signing vs Encryption

A token is simply a JSON object that is signed by its author. this tells you exactly two things about the data:

- The author of the token was in the possession of the signing secret
- The data has not been modified since it was signed

It's important to know that JWT does not provide encryption, which means anyone who has access to the token can read its contents. If you need to protect (encrypt) the data, there is a companion spec, `JWE`, that provides this functionality. JWE is currently outside the scope of this library.

Choosing a Signing Method

There are several signing methods available, and you should probably take the time to learn about the various options before choosing one. The principal design decision is most likely going to be symmetric vs asymmetric.

Symmetric signing methods, such as HSA, use only a single secret. This is probably the simplest signing method to use since any `[]byte` can be used as a valid secret. They are also slightly computationally faster to use, though this rarely is enough to matter. Symmetric signing methods work the best when both producers and consumers of tokens are trusted, or even the same system. Since the same secret is used to both sign and validate tokens, you can't easily distribute the key for validation.

Asymmetric signing methods, such as RSA, use different keys for signing and verifying tokens. This makes it possible to produce tokens with a private key, and allow any consumer to access the public key for verification.

Signing Methods and Key Types

Each signing method expects a different object type for its signing keys. See the package documentation for details. Here are the most common ones:

- The HMAC signing method (`HS256` , `HS384` , `HS512`) expect `[]byte` values for signing and validation
- The RSA signing method (`RS256` , `RS384` , `RS512`) expect `*rsa.PrivateKey` for signing and `*rsa.PublicKey` for validation
- The ECDSA signing method (`ES256` , `ES384` , `ES512`) expect `*ecdsa.PrivateKey` for signing and `*ecdsa.PublicKey` for validation

JWT and OAuth

It's worth mentioning that OAuth and JWT are not the same thing. A JWT token is simply a signed JSON object. It can be used anywhere such a thing is useful. There is some confusion, though, as JWT is the most common type of bearer token used in OAuth2 authentication.

Without going too far down the rabbit hole, here's a description of the interaction of these technologies:

- OAuth is a protocol for allowing an identity provider to be separate from the service a user is logging in to. For example, whenever you use Facebook to log into a different service (Yelp, Spotify, etc), you are using OAuth.
- OAuth defines several options for passing around authentication data. One popular method is called a "bearer token". A bearer token is simply a string that *should* only be held by an authenticated user. Thus, simply presenting this token proves your identity. You can probably derive from here why a JWT might make a good bearer token.
- Because bearer tokens are used for authentication, it's important they're kept secret. This is why transactions that use bearer tokens typically happen over SSL.

Troubleshooting

This library uses descriptive error messages whenever possible. If you are not getting the expected result, have a look at the errors. The most common place people get stuck is providing the correct type of key to the parser. See the above section on signing methods and key types.

More

Documentation can be found on godoc.org.

The command line utility included in this project (cmd/jwt) provides a straightforward example of token creation and parsing as well as a useful tool for debugging your own integration. You'll also find several implementation examples in the documentation.