Go
HashiCorp's Vault

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

In the Fall 2017 issue, we examined using Go to set up TLS encryption between our gls service and gls client. To recap, Go has three strong libraries which we used: the crypto/x509 library and the crypto/rsa libraries provided us with a means of generating the certificate, and the crypto/tls library provided us with a way of wrapping the gls communications with encryption.

In this issue, we're going to look to another tool to handle our certificate and key generation: HashiCorp's Vault (https://www.vaultproject.io/).

Vault, which is written in Go, has a Go client library for it. Not surprisingly, this library is not in the standard Go library. This will give us a chance to get a taste of the new Golang dependency management tool: dep (https://github.com/golang/dep).

We're going to use the existing code from last issue's article, but we've added a new file: `certs/generate_certs_vault.go` to do our certificate generation. You can get this code from https://github.com/cmceniry/login-glss, or by running:

```
shell$ go get -u github.com/cmceniry/login-glss
```

Using an External Secret Store

Organizations are under increasing pressure from regulatory entities to ensure proper handling of secrets. They need to be able to demonstrate a proper chain of custody and limited exposure of those secrets. They need to be able to show when a secret was accessed and by whom.

This ends up involving a significant amount of overhead and having a large impact on code and configuration processes. The secret cannot be kept with other configuration data, even though it is critical to the application or service being able to run.

What are some of these secrets? A few examples are:

- Passwords for service accounts to access databases
- Salts or shared secrets for message hashing and signatures
- Shared secrets for encryption channels
- TLS keys or the passphrases to TLS keys

Imagine having to go to every location where an application is running and manually putting a password or passphrase in place. The application code and the rest of its configuration are already there, but you still can't start the application without having these secrets. Even in a well-run organization, this can cause critical delays to service delivery or restoration. And that this is not well auditable is just as bad. You have to rely on people filling out sign in/sign out forms for retrieving the password or passphrase.

Vault makes it easier and faster to handle the secret and to keep that handling auditable. It is a network service that can share or issue secrets. The application or application server authenticates itself to Vault and receives an access token in response. The application then uses this token to retrieve any secrets it needs. Vault is maintaining the audit log for when that secret was created, modified, or retrieved.

Vault is designed to have multiple pluggable back ends. A back end handles a particular type of secret—such as a generic or database password. In the case of the database password, Vault can perform an action to create the credentials on the fly when it is asked for the password. This allows limited-use passwords and other precautions to reduce risk.

We're going to use Vault to generate our keys and certificates using the Vault Go library. But before we do that, we need to set up Vault and prepare it to be a certificate authority.

Getting Started: Vault

To get set up with Vault for this exercise, we're going to:

- Install Vault
- Start the Vault server
- Set our authentication credentials for Vault
- Configure Vault with our certificate authority back end
- Have Vault generate a key and certificate for our certificate authority
- Configure a role to use our certificate authority

Vault is a combined network server and client in one binary. You can download the binary for several platforms from the project's Web site: https://www.vaultproject.io/downloads.html.

To keep this article brief, we're going to cut a few corners when starting the server—namely, start it in `dev` mode. This leaves out the certificates for encrypting the communication with `vault`, and shortcuts the authentication phase by using a predefined token available in `dev` mode. Vault will keep everything in memory so this is definitely not a permanent installation. In a production deployment, you would want to examine both of these areas more closely.

Start `vault` with the `-dev` and `-dev-root-token-id=mytoken` and send it to the background.

```
shell$ ./vault server -dev -dev-root-token-id=mytoken &
[1] 13625
shell$ ==> Vault server configuration:
                     Cgo: disabled
         Cluster Address: https://127.0.0.1:8201
              Listener 1: tcp (addr: "127.0.0.1:8200", cluster address:
"127.0.0.1:8201", tls: "disabled")
               Log Level: info
        Mlock: supported: false, enabled: false
        Redirect Address: http://127.0.0.1:8200
                 Storage: inmem
                 Version: Vault v0.8.3
             Version Sha: 6b29fb2b7f70ed538ee2b3c057335d706b6d4e36

==> WARNING: Dev mode is enabled!
```

Next, we'll want to set up a few environment variables. `VAULT_ADDR` sets the connection point for Vault. `VAULT_TOKEN` sets the authentication token to use.

```
shell$ export VAULT_ADDR=http://127.0.0.1:8200
shell$ export VAULT_TOKEN=mytoken
```

Now we can set up the back end in Vault. In this case, we're going to use the `pki` back end, which will be our certificate authority. We want to make it available inside of Vault at a known location—we'll use `myca`. Back ends are set up with the `mount` command.

```
    shell$ ./vault mount -path=myca pki
    2017/09/23 21:22:55.762379 [INFO ] core: successful mount:
path=myca/ type=pki
    Successfully mounted 'pki' at 'myca'!
```

Now we need to have Vault generate the key and certificate for our certificate authority. Vault uses a generic interface to the back ends—namely, you can perform write (Create/Update), read (Read), and delete (Delete) operations on paths inside of Vault. When a back end is mounted, it exposes child paths underneath the mount path. You can perform CRUD operations on these child paths as appropriate for the back end. The paths and their usages for the `pki` back end can be found at https://www.Vaultproject.io /api/secret/pki/index.html. We're going to start by issuing a write to the path for "Generate Root." For this path, we have to specify the common name that will be stamped on the CA's certificate.

```
    shell$ ./vault write myca/root/generate/internal common_
name="My CA"
    Key           Value
   _        -----
    certificate    -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
    expiration     1508992653
    issuing_ca     -----BEGIN CERTIFICATE-----
    ...
    -----END CERTIFICATE-----
    serial_number   54:64:79:74:5c:b8:a1:6a:66:0c:88:6e:eb:bb:
40:1e:46:4b:d4:43
```

Vault responds with a generic response as well—key-value pairs. For this path, it responds with the CA's certificate, its expiration date represented in UNIX Epoch time, and the serial number. The certificate shows up twice—once as itself and once as its own `issuing_ca`. The second time has more to do with the structure of Vault's internal code for generating certificates—this will show up later as well. To mirror the last issue, replace the certs/CA.crt file with the certificate PEM block from above.

The last piece of setup in the Vault is that we need to configure roles inside of our `myca` path. These roles are both specific to the `pki` back end and specific to this instance of it. They represent the options for what configurations—namely, the common name, and type—can be put on to the issued certificates. Since we're already using the powerful root token, we're going to generate powerful roles which can mint certificates with any name, but we'll keep separate roles for generating server certificates and client certificates.

```
    shell$ ./vault write myca/roles/powerserver allow_any
_name=true \
        enforce_hostnames=false \
        server_flag=true \
        client_flag=false
    Success! Data written to: myca/roles/powerserver
    shell$ ./vault write myca/roles/powerclient allow_any
_name=true \
        enforce_hostnames=false \
        server_flag=false \
        client_flag=true
    Success! Data written to: myca/roles/powerclient
```

Getting Started: Vault Client Library

To get ready to use the client, we're going to:

- Add a reference client to our project code
- Initialize `dep` and let it pull down our code dependencies

`dep` looks at your code to decide what it needs to pull in. To start to use `dep`, we're going to add the Vault client as an `import` in `certs/generate_certs_vault.go`. Since the package name `api` is a bit too generic, we're going to specify that the qualified identifier of the package name is going to be `vaultapi`.

```
    import (
        vaultapi "github.com/hashicorp/vault/api"
```

Now we let `dep` do the hard part. For demonstration purposes, I'm using the verbose flag; otherwise, `dep` is very quiet.

```
    shell$ dep init -v
    Root project is "github.com/cmceniry/login-glss"
     3 transitively valid internal packages
     2 external packages imported from 2 projects
    (0)   ✓ select (root)
    (1) ? attempt github.com/kelseyhightower/gls with 1 pkgs; 1
versions to try
    (1)     try github.com/kelseyhightower/gls@master
    (1) ✓ select github.com/kelseyhightower/gls@master w/1 pkgs
    (2) ? attempt github.com/hashicorp/vault with 1 pkgs; 76
versions to try
    (2)     try github.com/hashicorp/vault@v0.8.3
    ...
      Locking in master (42a06e0) for direct dep github.com
/kelseyhightower/gls
    ...
      Locking in v0.8.3 (6b29fb2) for direct dep github.com
/hashicorp/vault
      Locking in master (68e816d) for transitive dep github.com
/hashicorp/hcl
```

What is `init` doing?

From our project, it starts by examining every .go file and looking at their `import` statements. From that it starts to build out a list of dependencies and pulls those in. It then does this same examination of the dependencies' `import` statements and iterates. When it looks at a dependency, it looks for any versioning information that that dependency may give—this usually shows up as semantic versioning (`v$major.$minor.$patch`)-based Git tags.

`dep` creates a `Gopkg.toml` file if one does not already exist. The `toml` file is used to specify any version constraints on the dependencies. After `dep` has collected all of the dependency and version information, and any constraints from the `toml` file, it attempts to solve finding the appropriate version of every dependency.

Once solved, `dep` creates a `Gopkg.lock` file, and pulls down any missing dependencies. The `lock` file is the version information which `dep` has picked for the current dependency solution. `dep` uses the `vendor` pattern for storing dependencies. When it pulls down a dependency, it stores that dependency in the vendor directory of this project. This allows the build to be specific to this particular project and not conflate items in the src directory of your `$GOPATH`. That way, if you are working with multiple projects that have conflicting dependencies, you can keep those separate rather than rebuilding your `$GOPATH/src` all of the time.

After `init` is done, it's good to take a look at what it has gathered. The `status` subcommand provides the current dependency solution and also takes a look at the upstream repositories to provide the latest version information.

```
shell$ dep status | cut -b1-80
PROJECT                          CONSTRAINT     VERSION        REVISION LATE
github.com/fatih/structs         *              v1.0           a720dfa  a720
github.com/golang/snappy         *              branch master  553a641  553a
github.com/hashicorp/errwrap     *              branch master  7554cd9  7554
github.com/hashicorp/go-cleanhttp *             branch master  3573b8b  3573
github.com/hashicorp/go-multierror *            branch master  83588e7  8358
github.com/hashicorp/go-rootcerts *             branch master  6bb64b3  6bb6
github.com/hashicorp/hcl         *              branch master  68e816d  68e8
github.com/hashicorp/vault       ^0.8.3         v0.8.3         6b29fb2  6b29
github.com/kelseyhightower/gls   branch master  branch master  42a06e0  42a0
github.com/mitchellh/go-homedir  *              branch master  b8bc1bf  b8bc
github.com/mitchellh/mapstructure *             branch master  d0303fe  d030
github.com/sethgrid/pester       *              branch master  0af5bab  0af5
golang.org/x/net                 *              branch master  0744d00  0744
golang.org/x/text                *              branch master  1cbadb4  1cba
```

While we won't need it in this exercise, you can always re-solve and bring your dependencies up-to-date with `dep ensure`.

Now that we have our dependencies, we can build out certificate generator.

Generating Keys and Certificates with the Vault API

To start off, our main program needs to set up our Vault connection. The Vault API does this in two parts—initializing the configuration and using that configuration to create a client struct. As with working with the command-line client, we need to explicitly set the Vault address as appropriate.

```
config := vaultapi.DefaultConfig()
config.Address = "http://127.0.0.1:8200"
c, err := vaultapi.NewClient(config)
```

Unlike before, where we set the authentication token as part of our environment, the token is set on the client.

```
c.SetToken("mytoken")
```

Now we can ask Vault to issue a key and certificate. We're going to start with the glssd server certificate.

First, we call `Logical()` to indicate that we're going to be accessing a Vault back end. If we wanted to perform administrative functions, such as `mount`, to Vault instead of data functions, we can use the `Sys()` function to get access there.

As with the command line interface, the `Write` call used a generic interaction with Vault. In the logical subsystem, we call a similar `Write` to the `myca/issue/powerserver` endpoint to have Vault issue us a key and certificate. In addition to the path, we have to supply some data—`common_name` and `ttl` for the expiration to use. How this data is transmitted is where the generic interaction comes into play. The `Write` call of the Vault API has the same signature regardless of what back end is in use. To allow for different back-end forms, it has to rely on loose type checking—the kind you find with the empty interface. And to allow for multiple data parameters, requests to `Write` take data in the form of a map where the map key is the data name as a string, and the map element is the data value as an empty interface.

```
s, err := c.Logical().Write(
  "myca/issue/powerserver",
  map[string]interface{}{
    "common_name": "localhost",
    "ttl":        "1h",
})
```

Vault responds with a Secret struct. There are several parts to it, but the part we care about is in the `Data` field. Much in the same way that the input data was in the generic `map[string]interface{}`, the `Data` field is also a `map[string]interface{}`. We can access the returned keys and assert their type to what we know they are. In particular, for the `pki` back end's issue commands, we get back the `private_key` key and the `certificate`. We take these and assert them to strings. Strings easily cast to byte slices which are what the `ioutil.WriteFile` func needs to save them out to disk.

```
ioutil.WriteFile(
  "certs/server.key",
  []byte(s.Data["private_key"].(string)),
  0444,
)
ioutil.WriteFile(
  "certs/server.crt",
  []byte(s.Data["certificate"].(string)),
  0444,
)
```

Next we do the same actions for the client certificate. This time, we also have to use a different role because that is the role we used which will issue certificates with the client usage set on them.

```
s, err = c.Logical().Write(
  "myca/issue/powerclient",
  map[string]interface{}{
    "common_name": "glss Client A",
    "ttl":        "1h",
  },
)
...
```

```
ioutil.WriteFile(
  "certs/client.key",
  []byte(s.Data["private_key"].(string)),
  0444,
)
ioutil.WriteFile(
  "certs/client.crt",
  []byte(s.Data["certificate"].(string)),
  0444,
)
```

A word of warning: we're being fast and loose with the conversion from the empty interface of the response to something we can write to a file. This means that any issues that crop up here will result in Go panics. It would be advisable to add conversion error checking to this code before building off of it. Or you could use the github.com/mitchellh/mapstructure library, which provides a way to convert loose data into structs much like you would with the encoding/json and encoding/xml libraries.

With that complete, we can run it to generate the new keys and certificates:

```
shell$ go run certs/generate_certs_vault.go
Success!
```

Since we have a drop in replacement for the `generate_certs.go` method, we can run the same commands as we did last issue, and verify that we're still working with the Vault-issued certificates:

```
shell$ ./glssd &
[1] 32659
shell$ 2017/09/23 23:15:31 Starting glsd..
shell$ ./glss .
2017/09/23 23:15:34 user="glss Client A" connect
drwxr-xr-x         442 Sep 23 22:39 .
drwxr-xr-x         510 Sep 23 23:14 .git
-rw-r--r-       42  Sep 23 22:39 .gitignore
-rw-r--r-     2776  Sep 23 22:08 Gopkg.lock
-rw-r--r-      687  Sep 23 22:08 Gopkg.toml
....
```

Application Changes When Using an External Secret Store

To extend the example here, instead of doing a drop-in replacement for the key and certificate generator, you can imagine that the glssd program itself would contact Vault and get a new key and certificate every time it started up. This is interesting for several reasons:

- We know when a specific certificate was issued to a specific client and can track and audit that.
- We can set the certificate lifetimes relatively low, increase key rotation, and decrease the impact timeframe of a leaked key.
- We can apply deployment automation to our environment without having to worry about dirtying our source control systems with keys.

Some of the above is very much dependent on the way the application authenticates to the Vault system, but that will have to wait for a future article. Regardless, the benefits are intriguing.

I hope this article has convinced you that it is relatively straightforward to retrieve items from external secret stores. I hope you take the time to see what they can do for you to help improve the overall security stance in your code and at your organization.