Kubernetes: Up & Integrated — Secrets & Configuration

This is the second in a four part series on how we at Qubit built our production ready Kubernetes (k8s) environments. For those of us with pre-existing configuration management workflows, moving to k8s can present some challenges. We will briefly describe Qubit's previous configuration management process, where it caused problems and what we wanted to preserve. After this we will present a detailed overview of the workflow we have built. This will be a long post, you have been warned!

The tools we will discuss here are not open-source at this time. We hope to release something in the near future. The workflow is also not perfect. We will discuss where the weaknesses lie and how things may change.

As with the previous article we will assume a reasonable working knowledge of Kubernetes. We will also be discussing some issues involving HashCorp's Vault. We'll discuss relevant Vault topics briefly but will not attempt to give an exhaustive treatment.

The Ghost of Config Past

Qubit's previous configuration management process was built around Puppet and hiera-eyaml. Any configuration containing sensitive material, such as passwords, authentication tokens and private keys, are stored in a central Puppet Git repository. The repository contains a public key that can encrypt data which can than be decrypted by the Puppet master server. Nodes within the cluster infrastructure retrieve configuration from the Puppet master, the master decrypts any encrypted data, passing fully rendered configuration to the cluster node. The container orchestration infrastructure on each node (Apache Mesos) can then mount the rendered content into containers as they start.

Many of the workloads running within the clusters are Node.js tasks using the node-config package for configuration. We leverage the merging behaviour of both hiera-eyaml and node-config. Applications can ship per-environment settings in e.g. `staging.yaml` , hiera then merges an unencryped per-application YAML section with the decrypted per-application settings, and renders this into a `local.yaml` file. The practical result is that developers can place a section of their configuration, as YAML, into our Puppet repository, and have that presented to their application in a secure fashion.

The developer workflow is as follows:

1. Create or update an application's configuration requirements.
2. Create a new section in the Puppet repository with the encrypted version of a secret.
3. Raise a pull-request to merge the new configuration section.
4. Wait for Puppet to deploy the secrets to the cluster nodes. In our case: one run per-hour per-node.
5. Once merged, deploy the new version of the application.

This process meets many of our security needs:

- Secrets are not checked-in to any source repository in plain text.
- Plain text secrets are at rest in a relatively minimal set of locations (on the root partition of each cluster node).
- Plain text secrets are only visible to the intended recipient application.

Many of you will have similar workflows. With hindsight and experience it is easy to be critical of this workflow. We'll discuss the practical problems we experienced, but it is important to state that for many organisations this process is perfectly adequate. Many of the issues we'll discuss below could be fixed by other means, without moving away from Puppet. We could certainly have stuck to this workflow with our move to k8s, for many people that will certainly be the right thing to do. That being said, here is a brief list of issues we faced regularly:

- Pull-request review by the Infrastructure team becomes a bottleneck for developers.
- Unnoticed errors in configuration easily broke Puppet runs for all applications, and all related infrastructure.
- Feedback on when Puppet has finally deployed configuration to the cluster is very poor and of limited visibility. Either Puppet runs were forced, which required Infrastructure team action, or the developer had to wait for "about an hour" before deploying.

- Deploying an application before Puppet had deployed configuration had unexpected results. Docker creates a local directory if asked to mount a file into a container if no such file exists on disk. This results in Puppet runs failing when a directory exists where a file should have been. This required Infrastructure team involvement to fix. The application would also deploy, but fail when attempting to read a directory.

As Qubit's development teams scaled up, occasional irritating misbehaviour turned into serious problems.

What Does "Good" Look Like?

In the previous section we have discussed some of the practical issues we faced with our Puppet based configuration process. It is worth taking a step back and trying to understand how these issues impacted the Big Picture.

- Developers had to be aware of the behaviour of a tool they would never actually run or use directly (Puppet).
- Developers had to work with an extra code repository.
- The Infrastructure team had to be involved with many new deploys, or emergency deploys, and all secret changes (or any changes to any of the configuration managed by Puppet).

As mentioned in the previous article in this series, we aim to have development teams have complete ownership of everything needed by their application from one single code repository. Ideally this should include:

- Code, tests and CI configuration.
- Configuration needed for deploying the application.
- Run-time configuration.
- Metrics and alerting configuration.

It is worth mentioning at this point that Qubit do not (at this time) have what other organisations might call an Operations team (variously known as 'SRE team', or the outright oxymoron of 'DevOps team'). Development teams are responsible for the day-to-day running of their applications. Qubit's Infrastructure team is focused on building, managing and maintaining the underlying infrastructure and providing the tools for the developers to use that infrastructure. This allows the number of services and associated development teams to scale independently of the Infrastructure team.

A Brave New World

Kubernetes presented a new challenge. One of our goals was to make deployments agnostic of the underlying cloud provider. We could have built k8s infrastructure via Puppet in both GCE and AWS, we had already done this for Mesos. Our existing Puppet infrastructure would have needed work to support GCE. The presence of GKE made this approach less attractive. Kops had also provided a straightforward approach to delivering a solid k8s infrastructure on AWS without this effort. The idea of no longer requiring Puppet to build the underlying cluster nodes was too attractive to pass up, especially when combined with the issues presented above.

The subject of presenting secrets to containers is generally known as Secret Injection. Fortunately for us, people had already outlined approaches to Secret Injection within k8s. Whilst none of the existing tools completely covered our use case, there was enough prior-art to give us a head start.

We had settled on HashiCorp's Vault for storage of secrets. Vault is an incredibly powerful tool. While there is a wealth of introductory documentation on Vault, there is considerably less by way of detailed discussion of production ready deployment. All requests to retrieve secrets from Vault require a Vault Token with suitable assigned policies for reading a given secret. Getting an initial token for a service to talk to Vault can seem like a 'chicken and egg' situation. Various mechanisms can be used to get an initial Vault token (see Authentication Backends), but the most relevant to our use case all require some other process to already have a Vault Token. This 'Token Bootstrapping' represents a real challenge to production Vault deployments.

Once you have logged in to Vault, you still have to access and use your secrets. We considered requiring developers to talk to Vault directly. This would have been the easiest option for the Infrastructure team, and arguably given the best end result in terms of Vault integration. In reality the code changes required to some 200 micro-services ruled this out. Vault is not an easy tool to use, especially when compared with YAML configuration files loaded via node-conf. Simply retrieving secrets is not the only issue, for some secrets the Vault Token must be periodically refreshed.

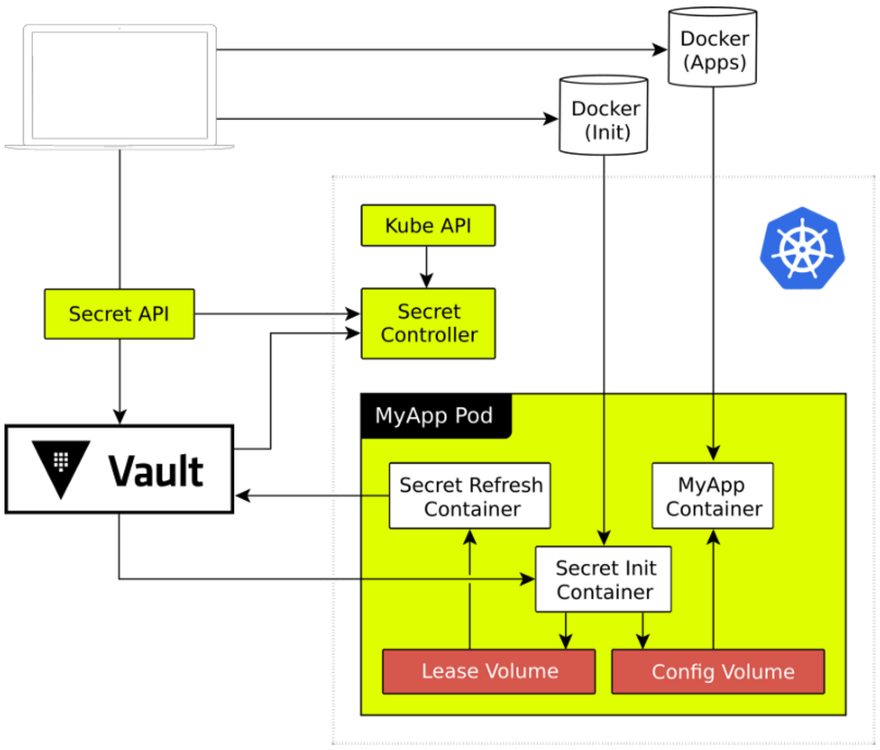
Our final goals were:

- Minimal / No code alteration for existing applications.
- Minimal configuration changes.
- Configuration from the application's source repository.
- Configuration available immediately upon deployment.
- Minimal exposure of plain text secret content.
- Allow leveraging advanced Vault concepts (such as managed PKI, and managed lifetime secrets for AWS).

Our eventual approach was inspired by three existing projects:

- https://github.com/kelseyhightower/vault-controller
- https://github.com/Boostport/kubernetes-vault
- https://github.com/kelseyhightower/confd

Since you have been so patient, and we still have a long road ahead, we'll break up proceeding with the basic block diagram of what we eventually produced.



Token Bootstrap

We elected to use Vault's AppRole feature. With AppRole a micro-service is assigned to a given AppRole in Vault. A micro-service instance must log in to the Vault AppRole to obtain a Vault Token before it can begin reading secrets. In order to log in to the AppRole, an instance of a micro-service requires two pieces of information:

- AppRole RoleID, by default this is a UUID. There is only one RoleID for a given AppRole.
- AppRole SecretID, a UUID that is retrieved from Vault by another process which is allowed to issue SecretIDs for a given AppRole. In our configuration this SecretID is unique for each new running instance, may only be used to log in to Vault once, and is only valid for a limited time after being issued.

Getting two pieces of information to our embryonic micro-service instance might seem harder than the problem we originally had of getting a single Vault Token into our container. We seem to have made our situation worse. There is a very good reason for taking this approach. These two bits of information can be delivered by different means, and neither of the processes responsible for either part of that delivery ever have enough information to actually log into the AppRole and retrieve the applications secrets. Our embryonic micro-service instance can combine these two IDs and use them to log into Vault, the resulting token grants it the power to read secrets. Since the SecretID may only be used once we can also detect potential misuse. If a service is unable to log in to Vault with a given RoleID and SecretID then only one of two things could have happened. It either took them too long, or the SecretID has already been used. It is possible, via Vault's Audit infrastructure, to spot and alert on the latter case.

This is excellent, but we still have a Token Bootstrapping problem. In order to get the RoleID and SecretIDs for an AppRole, whatever is requesting them must, in turn, have a valid Vault token. In our case, the RoleID will be retrieved by the process that is deploying the application (usually a CI tool or developer laptop), and the SecretID will be retrieved by an in-cluster Vault Controller. We could pick any other Vault authentication method here. The RoleID could be retrieved by a user authenticated via the GitHub authentication mechanism, whilst the SecretID process could be authenticated by an EC2 or GCP machine credential.

There is a further complication here that ruled out these simple authentication methods. Vault Policies are not very dynamic things at present. We want each AppRole to be limited to reading secret for just its associated micro-service. Out of the box Vault would require the developer to be able to create a policy for their application, and create the associated AppRole. This gives the developer carte-blanche as to the content of that policy. Whilst we do trust our developers, this would require them to understand Vault policies and would be very prone to error.

Since we already have a central authentication mechanism based on JWTs (discussed in our previous article), we elected to leverage it. We have built a small Secret API that sits in front of Vault and is responsible for managing AppRoles and policies. The deployment process can request the RoleID for a given application from the Secret API. After verifying the request the Secret API creates a set of policies for the application, along with an AppRole. Similarly, access to SecretIDs is controlled via this service. Pushing these parts of the Vault flow through an API also allows us to use our existing authorisation mechanism based on our internal Google Groups.

The End Result

Here is a brief overview of each of the components in the diagram above:

- Vault: Stores secrets at rest and enforces access policies.
- SecretAPI: Manages Vault AppRoles and policies, and is responsible for authorising access to Vault AppRole details. It is also used for setting secrets and maintaining a version history of secrets.
- Docker (Init): A Docker Registry containing a set of Init Containers for applications.
- Docker (Apps): A Docker Registry containing the main application containers. This is currently the same registry as the Init registry, this will be discussed further in the Future Work section.
- Secret Controller: Runs within each Kubernetes cluster. This is responsible for retrieving SecretIDs for new Pods.
- Secret Init Container: This is a container added to each Pod needing to receive configuration.

- Secret Refresh Cotainer: Optional container that can be used to manage Vault leases.
- Config Volume: A tmpfs in-memory volume shared between the Init Container and the main Application Container. This stores the final configuration for the application.
- Lease Volume: A tmpfs in-memory volume shared between the Init Container and the Refresh Container. This stores any Vault Leases that will need to be refreshed.

The final system is probably best described via a worked example. We will describe the key components in the process as they become relevant.

We will start with an application. We have a source code repository with the following content:

```
/Dockerfile             - to build out final application
container
/deployer.yml           - our deployment configuration file
/config/staging.yml     - non-secret node-config file for
staging
/config/production.yml - non-secret node-config file for
production
/config/local.yml.tmpl - Go template that will become
local.yml
/...                    - the rest of our application code
```

Developers, if so authorised, can set specific secrets within Vault using a command line utility that talks to the Secret API.

When the application is deployed:

1. The main application's Docker container is built and pushed to the application Docker repository.
2. The deployment tool contacts the Secret API to retrieve the RoleID for this application, specifying the destination cluster's details.
3. If the user running the deployment is suitably authorised the Secret API server ensures the necessary Vault AppRole and policies exists and returns the RoleID and AppRole name to the user.

The Vault AppRoles and Policies

When an application is first deployed we create four items in Vault. First we create the AppRole itself. This controls the set of permissions that will be given to the final Pod that retrieves the information. An example might be as follows:

```
auth/approle/role/aws.stg.myapp
```

```
period                  300
policies                [default secret-api/read-
aws.stg.myapp]
secret_id_num_uses      1
secret_id_ttl           300
token_max_ttl           300
token_num_uses          0
token_ttl               300
```

This is an AppRole for instances of `myapp` running in the `stg` (staging) cluster within `aws` (Amazon Web Services). The tokens issued to the final application will be refreshable tokens and will be given the `secret-api/read-aws.stg.myapp` policy. We also create that policy:

```
policy secret-api/read-aws.stg.myapp:
```

```
path "secret/myapp/*" { capabilities = ["read", "list"] }
path "secret/global/*" { capabilities = ["read", "list"] }
path "secret/project/aws/*" { capabilities = ["read",
"list"] }
path "secret/cluster/aws/stg/*" { capabilities = ["read",
"list"] }
path "pki/cluster/stg/*" { capabilities = ["read", "list"] }
path "pki/cluster/aws/stg/issue" { capabilities = ["create",
"update"] }
path "aws/creds/aws.stg.myapp" { capabilities = ["read",
"list"] }
```

This policy allows the application to:

- read and list various static secrets, some specific to it, others shared among applications
- read and create certificates in a cluster wide PKI
- create AWS credentials specific to this applications role. These credentials are limited to the lifetime of the application.

We also create a second policy. This will be used by the Secret Contoller later on in the process.

```
policy secret-api/getsecretid-aws.stg.myapp:

path "auth/approle/role/aws.stg.myapp/secret-id" {
                capabilities = ["create", "update"]
        }
```

We also create a Vault Token Role (this is separate from an AppRole), associated with this policy:

```
auth/token/roles/getsecretid-aws.stg.myapp

allowed_policies        [secret-api/getsecretid-
aws.stg.myapp]
disallowed_policies     [root]
explicit_max_ttl        0
name                    getsecretid-aws.stg.myapp
orphan                  false
path_suffix
period                  0
renewable               true
```

We will explain this policy briefly when we get to the Secret Contoller.

We continue with our deployment:

5. The deployment tool takes the AppRole information returned in the previous step and builds an InitContainer that will be added to the Pod.

The Init Container

Kubernetes allows Pods to have Init Containers. These containers are run before other in the Pod. They must run, and finally exit cleanly before the Pod start-up is allowed to continue.

Our Init Container includes 3 things:

- A binary capable of logging in to a Vault AppRole, and rendering Go Template files that include secret lookups into Vault.
- The AppRole RoleID information. This is the first half of our AppRole login details.

- The set of configuration files from the `/config` directory of the original source code.

Once the Init Container is pushed to the Docker registry, the deployment continues:

6. The deployment tool creates a set of k8s Deployments in the cluster. The deployments include the Vault AppRole name as an annotation on any Pod specification that wishes to engage in the Configuration process.

7. Kubernetes starts a Pod.

8. The Init Container starts. After reading in the RoleID information it begins listening on a known port.

9. The Secret Controller watches for new Pods. Upon finding one with suitable annotations, and with a running InitContainer listening on the known port, it begins its work.

The Secret Controller

The secret controller runs in each cluster. It has a Google Cloud Service Account specific to that cluster which permits it to retrieve SecretIDs on behalf of applications started in that cluster.

When it finds a new Pod wanting configuration, it contacts the SecretAPI and asks for a Vault Token for reading SecretIDs for the given application. Once the SecretAPI approves this request, a new Vault token with the `secret-api/getsecretid-aws.stg.myapp` policy is issued (via the previously mentioned Token Role). The Secret Controller then contacts Vault directly and retrieves a new SecretID for the required AppRole. This is then passed to the waiting Init Container via a HTTP POST. Ideally this would be a HTTPS POST, however bootstrapping a certificate here is tricky. Since the SecretID is of no use without the RoleID, and we can detect abuse of leaked SecretIDs, we do not consider this to be a major issue. Potentially the forthcoming Istio project could help here, or the Kubernetes Certificate API could be used.

The Secret Controller's work is now done, and the deployment of the Pod continues inside the Init Container.

10. The Init Container now has the AppRole RoleID that was baked into the container image and the SecretID that has been given to it by the

Controller. It proceeds to log in to the Vault AppRole. All things being well, it will receive a Vault Token with the `secret-api/read-aws.stg.myapp` policy mentioned above.

11. The Init Container now copies any files in the configuration data baked into its image into the shared configuration volume. During this process any file with a `.tmpl` extension is assumed to be a Go Template. The template handling code contains a rich set of functions (including the Sprig). Several functions are included for interacting with Vault. The following is an illustrative subset:

- `lookup` : This looks up a given secret name in the application specific area of the Vault tree for this application.
- `awsCredsEnvVars` : Retrieves a set of AWS credentials for this process and renders them as sh shell environment variables.
- `dbUsername` , `dbPassword` : These can be used to request database credentials from Vault.
- `tlsKey` , `tlsCert` , `tlsCA` : These generate a TLS key and certificate that can be used for mutual authentication within the cluster. The certificates are long lived and a largely intended for providing TLS to any loadbalancer ingress.
- `rawRead` , `rawWrite` : These permit arbitrary reads/writes to Vault paths. Since the policy limits the application to specific paths, allowing arbitrary read/write is fine.

After template rendering the resulting content is written to the shared configuration directory (with the `.tmpl` extension stripped). The end result is a set of complete configuration files with the relevant secrets for this application, stored on a temporary filesystem readable by the main application container.

The `db*` and `aws*` functions are worth a little further comment. It is important to note here that the credentials the application will get are specific to this running instance of the application and are not shared with other instances. When the application exits these credentials are revoked and destroyed. The creation of these credentials is audited by Vault. We can, for instance, track an action in AWS down to a specific running instance of an application. A practical upshot of this is that AWS and Database credential management is significantly simplified. We no longer need to roll keys. This can be trivially achieved by a rolling restart of the application (or a fresh deployment).

The final act of our Init Container relates to these temporary credentials. If requested, the Init Container will additionally write the Token and Lease information for any per-process credentials into a second temporary filesystem.

If all goes well, our Init Container exits cleanly and our deployment continues. If it does not (due to missing secrets, unavailable Vault server, or general network flakiness), the Init Container will eventually be restarted and the process will continue again from Step 8. We shall continue on, assuming all is well:

12. At this point our main application container starts. It can read its configuration from the shared configuration volume, access required services and begin its work. In the basic case we are now done. If we are using credentials we are leasing from Vault, one further action is required.

13. Our Kubernetes Pod may include a Secret Refresh Sidecar. If so, it will be started alongside our main application container. It will periodically refresh our Vault token, ensuring that our per application credentials are kept valid. Upon termination it will revoke the Vault token, thus destroying any associated credentials.

Profit!…

Our application is successfully deployed. Our secrets are in in-memory storage, only visible to the application we desired to have them. Each component on the way has had minimal access to secret related material. By generating temporary per-process credentials we have also achieved something that would be considerably harder to achieve with our original Puppet configuration.

Loss?

It would be dishonest not to point out some of the weak points in this process:

- The Secret API has considerable access to Vault. Vault has no support for subclassing or templating of policies at this time, so I consider this largely unavoidable unless you are willing to have an Operations team that can take responsibility for create the required Vault elements.
- None of this plays particularly well with Kubernetes Secrets or ConfigMaps. This is less of a problem for internal services, where

we generate the k8s objects ourselves, but becomes more awkward when leveraging 3rd party helm charts.

- Config from environment variables is still possible but cannot be configured on the container. A shell must be present inside the container, and suitable shell content rendered to populate the environment. Since this limits visibility of the environment variables from outside the Pod, this can be seen as a reasonable trade-off.
- The Refresh Sidecar needs further work. Kubernetes will not destroy a Pod if an individual Sidecar crashes. Since the crashing sidecar may have revoked the Vault token, we really need to destroy the entire Pod and start again. This could be done from the Secret Controller. Further experience needs to be gathered.
- Kubernetes secrets are still very much a work in progress. Vault integration is on the horizon and it is quite possible that much of this process will be rendered irrelevant in the next 6 months.
- Clearly this process is non-trivial. In reality the volume of code amounts to no more than about 1000 lines spread over 4 processes (the Secret API, Init Container, Refresh Container and Controller). We believe that the end-user experience is no more complicated than the original tooling (with the possible exception of the Go Templates). Time will tell.

Future Work

We are fairly comfortable with the flow as is. It has proved very reliable and adaptable. The delay on container start imposed by the Init Container is quite minimal, usually dwarfed by the time to download the main container. A typical Init Container run time is in the order of 1 or 2 seconds.

One area for potential enhancement is the build of the Init Container. The deployer in the above process builds the Init Container and pushes it to a common Docker Registry. The content of the container is just the configuration files and templates that they already have access to. However, they do not strictly need the RoleID. One option we are considering is the have the SecretAPI actually build the Init Container and pass back an image name that can then be included in the Pod. If the Docker Init Registry is only readable by the target k8s cluster, and writable by the Secret API, then the RoleID would never be readable by the process deploying the Pod. Whilst the RoleID does not present a major attack vector, minimising access to it seems like a reasonable option.

Conclusion

If you have gotten this far, then we thank you for taking the time to read this and hope it has been informative. Kubernetes and Vault are excellent tools and can work together well, but proper integration is a non-trivial task.

We hope to release the bulk of the tooling used by the process in the following months. Similar, simpler, tools already exist and may be suitable for simpler requirements. Much of the design has been motivated by our multi-cluster configuration and should probably be avoided if you have that option. Ultimately we hope that this post, and the others to follow will be useful in discussing some of the more difficult aspects of production Kubernetes use, and help feed into more general solution that can benefit the whole community.

If you fancy tackling engineering challenges like this, check out the Qubit careers page—we are hiring in a wide variety roles and locations.