# PKI and Secrets with Vault

as analyzed and reported here

p.pires@travelaudience.com

# Agenda

- Problem and Solution
- Vault 101
- Deploy Vault
- Authenticate as an operator
- Setup a Public-Key Infrastructure
- Disaster recovery
- `kubernetes-vault-client`
- Example: `time-server`
- Questions?

# Problem and Solution

# The problem

At Travel Audience, we need to centrally manage and encrypt sensitive information such as database passwords or third-party service credentials.

These pieces of sensitive information, from now on referred to as *secrets*, must be made available to (but not just) applications running in our Kubernetes clusters, in a secure and controlled way.

We also need to establish a public-key infrastructure (PKI) capable of issuing both client and server TLS certificates used for establishing trust between our applications (mutual authentication (mTLS)).

# The problem

At first glance, these needs could be easily addressed by using Kubernetes secrets and a tool like **`openssl`** to help establish a PKI.

However, this simple approach has a few important limitations:

1. Secret objects are bound to one namespace. They can only be referenced by pods in that same namespace.
2. Kubernetes offers little to no control over *who* can access one or more secrets.
3. Can’t share secrets between applications running in different clusters.
4. Kubernetes does not (yet) support encryption of secrets at rest.

# Acceptance criteria for a solution

The chosen solution must:

- enable authentication of applications, trust establishment between applications, and secure secret management;
- be usable by applications running in different Kubernetes clusters or in different GCP projects - for as long as the GCP organization is the same (hint: Cloud IAM);
- be secure and tightly integrated with both GCP services and Kubernetes.

# The solution

After an exhaustive evaluation of a few secret management tools at our disposal, we decided to build our solution on top of Hashicorp Vault.

Besides the fact that Vault delivers pretty much all we are looking for, this decision was also motivated by:

- Vault is OSS
- Vault's security model is mature
- Broad adoption (big and active community)

# Vault 101

# Vault

Vault is a tool for managing secrets that “*secures, stores, and tightly controls access to tokens, passwords, certificates, API keys and other secrets in modern computing*”.

As described, Vault stores existing secrets such as database passwords and also dynamically generates new ones such as certificates used for establishing mTLS.

# Vault: writing and reading a secret

```
$ vault write secret/postgresql username="myapp" password="s4f3#p455w0rd!"

Success! Data written to: secret/postgresql

$ vault read secret/postgresql

Key             Value

---             -----

password        s4f3#p455w0rd

username        myapp
```

# Vault: Secret backends

Vault supports several different *secret backends*—components that store and generate secrets. To deliver the proposed solution we will make heavy use of the **`kv`** and **`pki`** [types of] secret backends.

The **`kv`** backend stores and returns arbitrary strings (such as `s4f3#p455w0rd!` in the previous example). A secret managed by a `kv` backend secret may have multiple fields (such as `username` and `password` in the previous example). After all, kv stands for key-value, hence it's a map.

The **`pki`** backend generates and manages private keys and certificates used for establishing TLS. It supports several different configuration parameters.

# Vault: Secret backends

Before being used, *secret backends* must be ***mounted*** at a given path. In the context of Vault, ***mounting*** a *secret backend* means creating a new, isolated instance of the backend.

By default Vault mounts an instance of the `kv` backend at `secret/`, and that is what we have used in the previous example. All other *secret backends* must be explicitly mounted and configured.

There may be any number of *secret backends* mounted at any given time, and it is possible to have multiple instances of the same *secret backend* mounted at different paths and using different configurations.

**Unmount** is a destructive operation!

# Vault: Secret backends

```
$ vault mount -path supersecret kv

Successfully mounted 'kv' at 'supersecret'!

$ vault write supersecret/foo bar=baz

Success! Data written to: supersecret/foo

$ vault read -field="bar" supersecret/foo

baz

$ vault read -field="bar" secret/foo

No value found at secret/foo
```

# Vault: Authentication

Vault features an HTTP API through which every single request flows. This API is used by both operators and applications in order to store and retrieve secrets and/or certificates.

Vault's authentication model is based on the concept of ***tokens***. Each token has an associated set of ***policies*** and may or may not have a defined ***time-to-live***.

A token may be obtained directly from Vault or by using an external authentication provider. Vault has first-class support for Kubernetes and Cloud IAM as authentication providers, making it easy to provide Kubernetes/GCP applications with tokens. We will focus on Kubernetes alone!

# Vault: Authentication

Vault's support for external authentication providers is implemented in the form of ***auth backends***—a component that performs authentication against an external source and returns Vault auth tokens associated with a set of policies on success.

*Auth backends* support configuration parameters such as the set of policies or the time-to-live to associate with the tokens.

More advanced backends support the concept of ***role***—a set of constraints on the entities that may authenticate with the backend. These backends usually allow for configuring policies and TTLs on a *per-role* basis.

# Vault: Authentication

In order to be used, an *auth backend* must be ***mounted*** at a given path. For instance, the command

```
$ vault auth-enable -path vault-training kubernetes
```

mounts an *auth backend* of type **`kubernetes`** at `auth/vault-training`. This *auth backend* may then be configured as necessary.

There may be any number of *auth backends* mounted at any time, and it is possible to mount multiple instances of the same *auth backend* at different paths. Each instance of an *auth backend* has its own configuration.

# Vault: Authentication

Some backends are more suitable for user authentication—such as `github` and `ldap`—while some others are more suitable for application authentication—such as `kubernetes`. Some others, like `gcp`, are suitable for both since they leverage on abstractions such as service accounts.

Some of the auth backends allow for defining multiple ***roles***. A ***role*** is a mapping between a specific external entity, such as a particular Kubernetes service account, and a set of Vault policies. We'll have a deeper look at roles in a short while.

# Vault: Authorization

While authentication may be delegated to an external provider such as Cloud IAM or Kubernetes, authorization is fully managed within Vault using ***policies***—sets of rules which control what paths in the API can be accessed, and how.

For instance, a policy may specify that the bearer of a given token has read-only access to `secret/postgresql`:

```
path "secret/postgresql" { capabilities = ["read"] }
```

Any token associated with this policy will be able to read—but not modify—the `postgresql` secret.

# Vault: Authorization

There are two pre-defined policies in Vault—`root` and `default`. The former allows full access to the Vault API while the latter allows only very basic functionality that all users should be allowed such as looking-up or revoking their own token.

User-defined policies work on a ***deny-by-default*** basis, and can be made as permissive as desired. For instance, and assuming a `kv` secret backend is mounted at `bidder`, this policy allows full control over the contents of the backend

```
path "bidder/*" {
    capabilities = ["create", "read", "update", "delete", "list"]
}
```

We'll refer to this policy as `bidder-rw` in the next slides.

# Vault: Authorization

| | `bidder-rw` | `root` |
|---|---|---|
| Read secrets on `secret/` | × | ✓ |
| Create secrets on `bidder/` | ✓ | ✓ |
| Update secrets on `bidder/` | ✓ | ✓ |
| Read secrets on `bidder/` | ✓ | ✓ |
| Delete a secret on `bidder/` | ✓ | ✓ |
| List secrets on `bidder/` | ✓ | ✓ |
| Unmount `bidder/` | × | ✓ |

# Vault: Authorization

As mentioned before, some auth backends like `gcp` and `kubernetes` may be configured to map specific external entities into a set of policies.

For instance, we may create a `bidder-secret-provisioner` Cloud IAM service account and instruct the `gcp` backend to map it to the `bidder-rw` policy:

```
$ vault write /auth/gcp/roles/bidder-secret-provisioner \

    type="iam" project_id="bidder-152710" ttl="1h" \

    policies="bidder-rw" \

    bound_service_accounts="bidder-secret-provisioner@bidder-152710.iam.gserviceaccount.com"
```

This set of rules define the `bidder-secret-provisioner` ***role***.

# Vault: Authentication and Authorization

The `bidder-secret-provisioner` role makes it easy for the `bidder` team to manage their own secrets autonomously. All that is necessary is a ***private key*** for `bidder-secret-provisioner@bidder-152710.iam.gserviceaccount.com`, which may be retrieved from the GCP console or using `gcloud`.

A member of the `bidder` team wanting to provision secrets needs only to login against the `bidder-secret-provisioner` role using a signed JWT token. If login is successful, the `gcp` backend will return a Vault authentication token associated with the `bidder-rw` policy and with a TTL of one hour, as configured. The provisioner then `vault auths` with the auth token and starts provisioning secrets.

# Vault: Authentication and Authorization

If this proves to be a useful strategy, we may agree on having permissive policies such as `bidder-rw` and corresponding roles on the `gcp` backend on a *per-team* basis, so that every team may provision their secrets autonomously.

It should be noted, however, that this represents a ***compromise*** between security and ease of use. For instance, an ill-intentioned member from a different team with sufficient Cloud IAM permissions on the organization can impersonate the `bidder-secret-provisioner@bidder-152710.iam.gserviceaccount.com` service account and modify or destroy secrets on the `bidder/` secret backend.

In an ideal scenario only operators should be given such permissive policies.

# Vault: A word on “roles”

Now that we’ve seen what a *role* represents in the context of an *auth backend*, we should also mention that some *secret backends*—most notably the `pki` backend—also support the concept of *role*. These are completely different concepts.

A role in the context of the `pki` backend represents a set of rules over what domain names, IPs and certificate types (client or server) are acceptable to be requested from the backend. These roles have nothing to do with authentication or authorization—only with the properties of generated certificates.

# Vault: Storage and Encryption

Vault relies on an external *storage backend* such as `etcd` to store data and state. All data is encrypted using the ***Advanced Encryption Standard*** (AES) algorithm before being stored and cannot be decrypted by brute-force or similar attacks.

At present, there is no known practical attack that would allow someone to read data encrypted by AES without knowledge of the encryption key in use.

*Storage backends* also play a key role on high-availability. We will use etcd as the storage backend for our solution, due to its reliability, its support for HA and its ease of setup in Kubernetes (etcd-operator).

# Vault: Initialization

Before using Vault, we need to initialize it, namely:

1. Generate the ***encryption key*** - to be used by Vault as the primary encryption key used to encrypt everything, except itself.
2. Generate the ***master key*** - to be used by Vault to encrypt the ***encryption key***. This key is never stored. Instead, and by default, it is split into five parts called the ***unseal keys***, to be distributed among trusted individuals (not stored by Vault). Also by default, a set of three *unseal keys* is needed for the reconstruction of the *master key*. By splitting and distributing the *master key* Vault ensures no single point of failure exists regarding data.

# Vault: Initialization

During initialization, Vault also generates a “special” authentication token called the ***initial root token***. This is the token associated with the `root` policy—thus allowing for full control over Vault—and is special in the sense that it never expires and allows for creating other “root tokens” with no expiration date.

This token can be extremely useful in development and testing scenarios but should be handled with care in production scenarios. In fact, it is a recommended action to revoke the initial root token as soon as possible after it is issued.

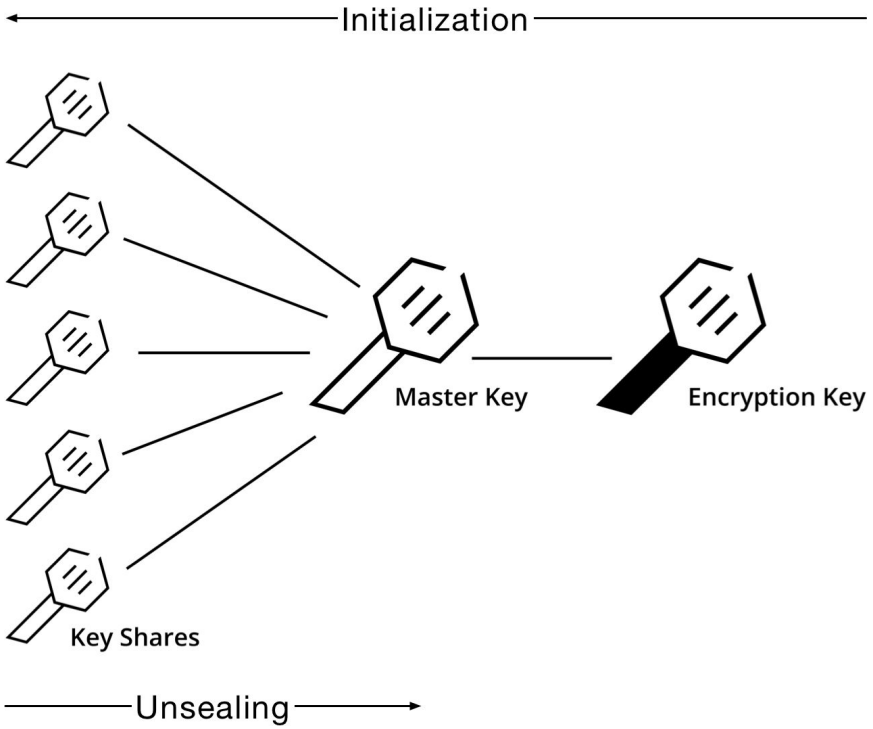
# Vault: Unsealing

After initialization—as well as when it restarts—Vault enters the ***sealed*** state. This means that Vault has forgotten about the *master key* and cannot decrypt the *encryption key*—thus being unable to encrypt or decrypt data.

In order to encrypt or decrypt data Vault must be ***unsealed***. The ***unsealing*** process consists, by default, in providing Vault with three out of the five *unseal keys*, thus allowing Vault to reconstruct the *master key*. Vault is then able to decrypt the stored *encryption key* and start servicing requests.

Again, Vault does not store the *master key* or the *unseal keys*. If these are somehow lost all data stored by Vault is lost as well.

# Vault: Initialization and Unsealing



# Vault: Sealing

The reverse of the *unsealing* process is called ***sealing***.

This procedure instructs Vault to forget about the *master key* so that no further requests can be serviced. It is an *emergency procedure* and should be adopted whenever an intrusion is detected to minimize the risk of exposure.

After Vault is sealed a new *unsealing* process is required.

# Vault & Google Cloud KMS

As an additional security measure, and since we will deploy Vault on top of GCP, Google Cloud KMS will be used to safely store the unseal keys.

Cloud KMS is a key management service that allows for encrypting and decrypting data using strong algorithms. Being tightly integrated with Cloud IAM, adequate permissions must be established so that only trusted individuals can unseal Vault, even in a disaster scenario where all but one trusted individuals are unavailable.

# Vault & Google Cloud KMS

Central to Cloud KMS are the concepts of ***keyring*** and ***key***—respectively, a named grouping of keys similar to a keychain and a named object representing a cryptographic key.

*Keyrings* hold multiple keys and may be manipulated by individuals or service accounts with *Cloud KMS Admin* permissions on the organization or the project (that runs Vault).

*Keys* hold the actual cryptographic material used to encrypt and decrypt data. Data may be decrypted by individuals with *Cloud KMS CryptoKey Decrypter* permission on the organization or the project. This permission should be given only to trusted individuals as it may compromise the security of the solution.

# Authentication with Kubernetes

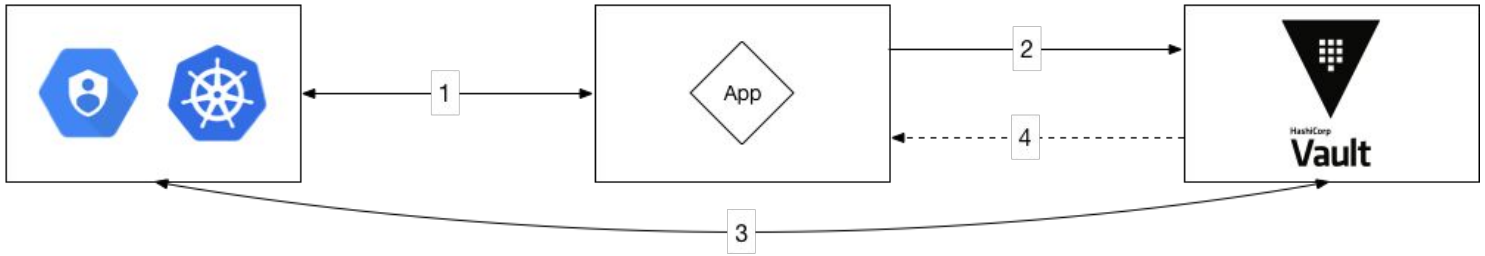
# Authentication with Kubernetes

As already mentioned, Vault has first-class support for authenticating applications running in Kubernetes clusters (Kubernetes service-accounts), making it easy to provide them with Vault authentication tokens.

To receive a Vault authentication token, applications have to login against a cluster-specific **`kubernetes`** *auth backend* using the *service account token* injected by Kubernetes in the pod they are running in.

When the *auth backend* receives a login request, it verifies the authenticity of the service account token against the Kubernetes API and, if successful, returns a Vault token to the application.

# Authentication with Kubernetes



# Authentication with Kubernetes

If the Kubernetes cluster, where a given application is running, has RBAC enabled it is necessary to grant access to the TokenReview API for every combination of namespace and service account in use by creating a `ClusterRoleBinding` that includes said combinations.

For example, to authenticate with Vault using the `default` service account in namespaces `nsone` and `nstwo` it is necessary to create the following `ClusterRoleBinding`:

# Authentication with Kubernetes

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tokenreview-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: default
  namespace: nsone
- kind: ServiceAccount
  name: default
  namespace: nstwo
```

# Authentication with Cloud IAM

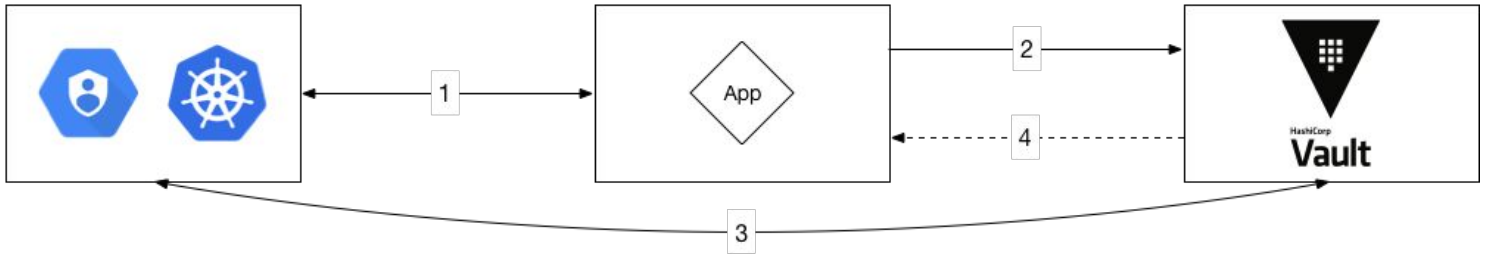
# Authentication with Cloud IAM

Vault also supports authenticating applications using Cloud IAM service accounts. To receive a Vault authentication token, applications have to login against a **`gcp`** *auth backend* using a *signed JWT token* obtained from the Cloud IAM API.

When the *auth backend* receives a login request it verifies the authenticity of the token against the Cloud IAM API and, if successful, returns a Vault token.

This flow differs from the Kubernetes-specific flow in that a *service account private key* must be provided to the application and adequate permissions must be set on every GCP project. As such, Kubernetes auth is preferred over Cloud IAM auth for new applications and deployments.

# Authentication with Cloud IAM



# Exposing Vault

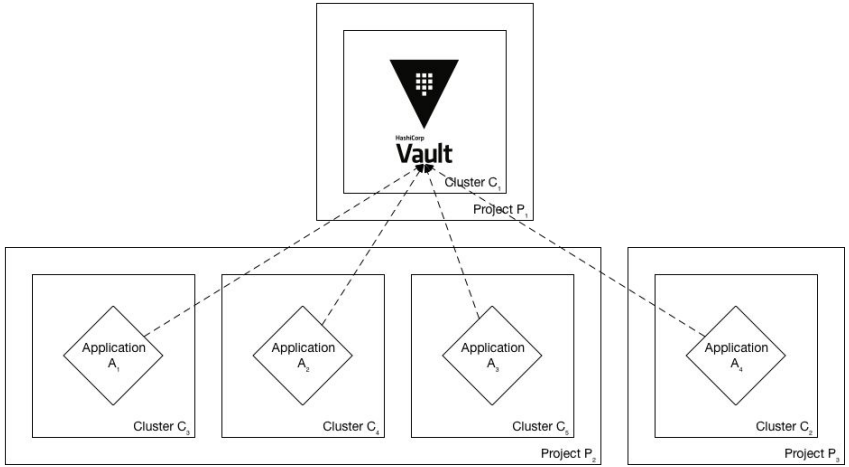
# Exposing Vault

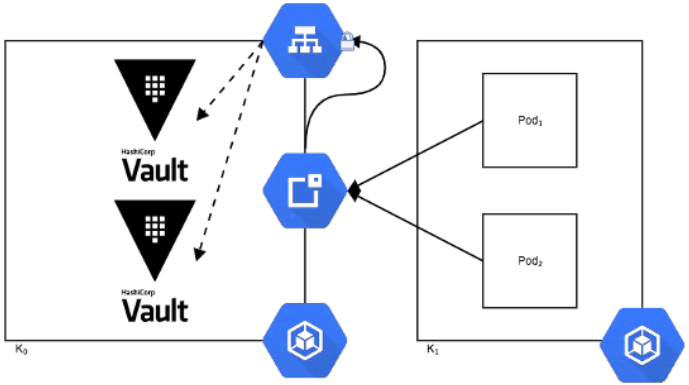
The final part of our solution involves exposing Vault to multiple Kubernetes clusters.

We have proposed that Vault is exposed via a public, TLS-enabled GCLB associated with a given global IP address.

This allow us to keep complexity to a minimum while not compromising on the security and cost of the solution.

# Exposing Vault



# Deployment pre-requisites

# Google Cloud Platform: Requirements

- A GCP project:
  - Google Cloud Key Management Service (KMS) API enabled
- Google Cloud IAM roles:
  - Cloud KMS > Cloud KMS Admin
  - Cloud KMS > Cloud KMS CryptoKey Encrypter/Decrypter
  - Container > Container Engine Admin
  - Compute Engine > Compute Network Admin
- A working GKE cluster:
  - Kubernetes v1.8.4
  - Legacy authorization disabled
  - Read/Write permissions for storage

# Setup Google Cloud KMS

As mentioned before one will use Google Cloud KMS to add an extra layer of security to the Vault deployment.

To do that one will create a Google Cloud KMS ***keyring*** (`vault`) and two Google Cloud KMS ***keys*** (`etcd` and `init`) with which to encrypt and decrypt data.

The `etcd` key will be used to encrypt the certificates and private keys used to secure etcd—as these can't be managed by Vault—and the `init` key will encrypt data generated during initialization—the unseal keys and the initial root token.

# Setup Google Cloud KMS (DONE!)

```
$ gcloud kms keyrings create vault \
    --location global
$ gcloud kms keys create etcd \
    --location global --keyring vault --purpose encryption
$ gcloud kms keys create init \
    --location global --keyring vault --purpose encryption
```

# Setup RBAC on GKE

Due to a known issue with RBAC on GKE one must grant themselves the `cluster-admin` role manually before proceeding.

# Setup RBAC on GKE

```
$ MY_GCLOUD_USER=$(gcloud info | grep Account | awk -F'[][]' '{print $2}')

$ kubectl create clusterrolebinding \

    demo-cluster-admin \

    --clusterrole=cluster-admin \

    --user=${MY_GCLOUD_USER}

clusterrolebinding "demo-cluster-admin" created
```

# Deploy Vault

# Create the `vault` namespace

To better group and manage the components of the solution it is recommended to create a dedicated `vault` namespace in the cluster.

# Create the `vault` namespace

```
$ kubectl create namespace vault

namespace "vault" created
```

# Deploy `etcd-operator`

`etcd-operator` will be responsible for managing the `etcd` cluster that Vault will use as storage backend.

`etcd-backup-operator` and `etcd-restore-operator` will, in turn, handle tasks such as periodic backups and disaster recovery.

Make sure to have cloned https://github.com/travelaudience/kubernetes-vault.

# Deploy `etcd-operator`

```
$ kubectl create -f ./etcd-operator/etcd-operator-bundle.yaml
clusterrole "etcd-operator" created
serviceaccount "etcd-operator" created
clusterrolebinding "etcd-operator" created
deployment "etcd-operator" created
deployment "etcd-backup-operator" created
deployment "etcd-restore-operator" created
service "etcd-restore-operator" created
```

# Verify `etcd-operator`

At this point it is a good idea to check whether the deployments succeeded.

# Verify `etcd-operator`

```
$ kubectl -n vault get pod

NAME                                    READY     STATUS    RESTARTS   AGE

etcd-operator-5876bdb586-dj6dc          1/1       Running   0          1m

etcd-backup-operator-5876bdb586-t2jjf   1/1       Running   0          1m

etcd-restore-operator-5876bdb586-dk455  1/1       Running   0          1m
```

# Verify `etcd-operator`

```
$ ETCD_OPERATOR_POD_NAME=$(kubectl -n vault get pod \
    | grep etcd-operator \
    | awk 'NR==1' \
    | awk '{print $1}')
$ kubectl -n vault logs -f "${ETCD_OPERATOR_POD_NAME}"
(...)
```

# Verify `etcd-operator`

```
$ ETCD_BACKUP_OPERATOR_POD_NAME=$(kubectl -n vault get pod \
    | grep etcd-backup-operator \
    | awk 'NR==1' \
    | awk '{print $1}')
$ kubectl -n vault logs -f "${ETCD_BACKUP_OPERATOR_POD_NAME}"
(...)
```

# Verify `etcd-operator`

```
$ ETCD_RESTORE_OPERATOR_POD_NAME=$(kubectl -n vault get pod \
    | grep etcd-restore-operator \
    | awk 'NR==1' \
    | awk '{print $1}')
$ kubectl -n vault logs -f "${ETCD_RESTORE_OPERATOR_POD_NAME}"
(...)
```

# Create TLS certificates and secrets for `etcd`

Before proceeding, one needs to generate TLS certificates to secure communications within and to the `etcd` cluster.

Even though the `etcd` cluster won't be exposed to outside of the Kubernetes cluster it is highly recommended to adopt this additional security measure.

# Create TLS certificates for `etcd`

```
$ ./tls/create-etcd-certs.sh

2017/11/25 10:08:12 [INFO] generating a new CA key and certificate from CSR

(...)
```

In this step, certificates are generated by *cfssl* and then encrypted with Cloud KMS.

**ATTENTION:** During my tests, *cfssl* would seldomly crash. If it happens, re-run the command above.

# Create TLS secrets (containing certificates) for `etcd`

```
$ ./tls/create-etcd-secrets.sh

secret "etcd-peer-tls" created

secret "etcd-server-tls" created

secret "etcd-operator-tls" created

secret "vault-etcd-tls" created
```

In this step, certificates are decrypted with Cloud KMS and stored as Kubernetes secrets.

# Deploy the `etcd` cluster

**ATTENTION:** Before deploying *etcd*, let's remove the generated certificates from disk:

```
$ git clean -xfd
```

**ATTENTION:** We will talk about disaster-recovery soon but as soon as these certificates are gone, new ones will be needed for rebuilding the PKI infra.

Now that `etcd-operator` and the necessary Kubernetes secrets are adequately setup, it is time to create the `etcd` cluster.

The `etcd` cluster will be comprised of three nodes running `etcd 3.1.10`.

# Deploy the `etcd` cluster

```
$ kubectl create -f etcd/etcd-etcdcluster.yaml

etcdcluster "etcd" created
```

# Deploy the `etcd` cluster

Before proceeding any further, one must check whether the `etcd` cluster deployment succeeded by inspecting pods in the `vault` namespace.

# Verify the `etcd` cluster

```
$ kubectl -n vault get pod

NAME                                   READY     STATUS    RESTARTS   AGE
etcd-0000                              1/1       Running   0          1m
etcd-0001                              1/1       Running   0          1m
etcd-0002                              1/1       Running   0          1m
etcd-operator-5876bdb586-dj6dc         1/1       Running   0          7m
etcd-backup-operator-5876bdb586-t2jjf  1/1       Running   0          7m
etcd-restore-operator-5876bdb586-dk455 1/1       Running   0          7m
```

# Deploy Vault

FYI, we decided to go with 2 pods for HA, in *active-failover* mode.

Vault's deployment has to be split in three parts. In the first, one creates the Vault `StatefulSet`, which runs two (2) Vault pods that are ***uninitialized*** and ***sealed***.

These two Vault instances will not accept any requests except for the ones required for the initial configuration process.

# Deploy Vault

```
$ kubectl create -f vault/nginx-configmap.yaml
configmap "vault" created
$ kubectl create -f vault/vault-configmap.yaml
configmap "vault" created
$ kubectl create -f vault/vault-serviceaccount.yaml
serviceaccount "vault" created
$ kubectl create -f vault/vault-service.yaml
service "vault" created
$ kubectl create -f vault/vault-statefulset.yaml
statefulset "vault" created
```

# Verify Vault

Before proceeding any further one must check whether the Vault deployment succeeded by inspecting pods in the `vault` namespace.

# Verify Vault

```
$ kubectl -n vault get pod | awk 'NR==1 || /vault/'
NAME                                    READY     STATUS    RESTARTS   AGE
vault-0                                 1/2       Running   0          1m
vault-1                                 1/2       Running   0          1m

$ kubectl -n vault logs vault-0 vault
(...)
2017/11/21 15:48:13.382731 [INFO ] core: security barrier not initialized
2017/11/21 15:48:18.383671 [INFO ] core: security barrier not initialized
2017/11/21 15:48:23.381753 [INFO ] core: security barrier not initialized
```

# Initialize Vault

Vault must now be ***initialized***, and both instances must be ***unsealed***.

As the Vault pods are not accessible from outside the cluster at this time, one needs to establish port-forwarding to one’s local workstation.

# Initialize `vault`

```
$ kubectl -n vault port-forward \
    vault-0 18200:8200
Forwarding from 127.0.0.1:18200 -> 8200
Forwarding from [::1]:18200 -> 8200
```

```
$ export VAULT_ADDR="http://127.0.0.1:18200"
$ vault init | gcloud kms encrypt \
    --plaintext-file - \
    --ciphertext-file vault-init.kms \
    --keyring vault \
    --key init \
    --location global
```

# Unseal `vault-0`

```
$ kubectl -n vault port-forward \
    vault-0 18200:8200
Forwarding from 127.0.0.1:18200 -> 8200
Forwarding from [::1]:18200 -> 8200
```

```
$ export VAULT_ADDR="http://127.0.0.1:18200"
$ for i in {1..3}; do \
    vault unseal "$(gcloud kms decrypt \
        --plaintext-file - \
        --ciphertext-file vault-init.kms \
        --keyring vault \
        --key init \
        --location global \
        | awk "NR==${i}" \
        | awk -F ": " '{print $2}')"
done
```

# Unseal `vault-0`

```
$ kubectl -n vault get pod | awk 'NR==1 || /vault/'
NAME                                    READY     STATUS    RESTARTS   AGE
vault-0                                 2/2       Running   0          5m
vault-1                                 1/2       Running   0          5m

$ kubectl -n vault logs vault-0 vault
(...)
2017/11/25 15:52:15.382731 [INFO ] core: vault is unsealed
(...)
```

# Unseal `vault-1`

**ATTENTION:** Since we're port-forwarding to each Vault pod, we now need to bind a different port.

# Unseal `vault-1`

```
$ kubectl -n vault port-forward \
    vault-1 28200:8200
Forwarding from 127.0.0.1:28200 -> 8200
Forwarding from [::1]:28200 -> 8200
```

```
$ export VAULT_ADDR="http://127.0.0.1:28200"
$ for i in {1..3}; do \
    vault unseal "$(gcloud kms decrypt \
        --plaintext-file - \
        --ciphertext-file vault-init.kms \
        --keyring vault \
        --key init \
        --location global \
        | awk "NR==${i}" \
        | awk -F ": " '{print $2}')"
done
```

# Unseal `vault-1`

```
$ kubectl -n vault get pod | awk 'NR==1 || /vault/'
NAME                                    READY     STATUS    RESTARTS   AGE
vault-0                                 2/2       Running   0          7m
vault-1                                 1/2       Running   0          7m

$ kubectl -n vault logs vault-1 vault
(...)
2017/11/25 15:54:17.382731 [INFO ] core: vault is unsealed
2017/11/25 15:54:17.382731 [INFO ] core: entering standby mode
(...)
```

# Vault and High-Availability

The second Vault pod (`vault-1`) will operate as a ***standby*** instance — it will accept requests and forward them to the ***active*** instance (in this case, `vault-0`).

In all cases, and to optimize traffic, Kubernetes will mark standby instances as *not ready* so that any requests are directed only to the active instance.

# Revoke Vault’s initial root token

Before proceeding, and as a security measure, one should revoke Vault’s initial root token — a “superuser” token generated during the initialization process.

In practice one should ***always*** revoke a root token when it is no longer needed. This guarantees that root tokens are as short-lived as possible.

# Revoke Vault’s initial root token

```
$ kubectl -n vault port-forward \
    vault-0 18200:8200
Forwarding from 127.0.0.1:18200 -> 8200
Forwarding from [::1]:18200 -> 8200
```

```
$ export VAULT_ADDR="http://127.0.0.1:18200"
$ vault auth "$(gcloud kms decrypt \
    --plaintext-file - \
    --ciphertext-file vault-init.kms \
    --keyring vault \
    --key init \
    --location global \
    | awk "NR==6" \
    | awk -F ": " '{print $2}')"
Successfully authenticated! You are now logged in.
(...)
$ vault token-revoke -self
Success! Token revoked if it existed.
```

# Expose Vault

One will now expose Vault to outside the cluster, so that applications running in other clusters can access it. To do this one needs to create a `vault` global static IP (DONE!) in Google Cloud Platform and attach a Google Cloud Load-Balancer to it.

After the `vault` IP address is created, one must configure the DNS of the domain one is going to use to expose Vault (in this case, **`vault.travelaudience.com`**) so that it points at this IP address (DONE!)

# Expose vault

```
$ gcloud compute addresses create core-vault --global
Created [https://www.googleapis.com/compute/v1/projects/<project-name>/global/addresses/vault].
$ gcloud compute addresses describe core-vault --global
address: X.X.X.X
creationTimestamp: '2017-11-25T06:25:39.628-07:00'
description: ''
(...)
$ dig @8.8.8.8 vault.travelaudience.com A
(...)
vault.travelaudience.com.   299   IN    A     X.X.X.X
(...)
```

# Expose vault

```
$ kubectl create -f vault/vault-api-service.yaml

service "vault" created

$ kubectl create -f vault/vault-api-ingress.yaml

ingress "vault" created

$ kubectl create -f kube-lego/kube-lego-bundle.yaml

namespace "kube-lego" created

configmap "kube-lego" created

deployment "kube-lego" created
```

# Break… because GCLB takes a while

# Expose vault

```
$ curl -s https://vault.travelaudience.com/v1/sys/health | jq
{
    "initialized": true,
    "sealed": false,
    "standby": false,
    "server_time_utc": 1512140698,
    "version": "0.9.0",
    "cluster_name": "vault-cluster-ccd9fb3e",
    "cluster_id": "d6b9eeec-71fa-21d7-f3aa-f202a4415f4f"
}
```

# Authenticate as an operator

# Generate a new root token

During the deployment process one has revoked the initial root token generated by Vault. This was made as an additional security measure.

To operate Vault one needs to be able to obtain new root tokens whenever necessary. This is achieved using the `vault generate-root` command.

# Generate a new root token

```
$ export VAULT_ADDR="https://vault.travelaudience.com"

$ vault generate-root -genotp

OTP: <otp>
```

# Generate a new root token

```
$ vault generate-root -otp="<otp>" \
        "$(gcloud kms decrypt --plaintext-file - --ciphertext-file vault-init.kms \
        --keyring vault --key init --location global \
        | awk "NR==1" | awk -F ": " '{print $2}')"
Nonce: <nonce>
Started: true
Generate Root Progress: 1
Required Keys: 3
Complete: false
```

# Generate a new root token

```
$ vault generate-root -otp="<otp>" -nonce="<nonce>" \
        "$(gcloud kms decrypt --plaintext-file - --ciphertext-file vault-init.kms \
        --keyring vault --key init --location global \
        | awk "NR==2" | awk -F ": " '{print $2}')"
Nonce: <nonce>
Started: true
Generate Root Progress: 2
Required Keys: 3
Complete: false
```

# Generate a new root token

```
$ vault generate-root -otp="<otp>" -nonce="<nonce>" \
        "$(gcloud kms decrypt --plaintext-file - --ciphertext-file vault-init.kms \
        --keyring vault --key init --location global \
        | awk "NR==3" | awk -F ": " '{print $2}')"
Nonce: <nonce>
Started: true
Generate Root Progress: 3
Required Keys: 3
Complete: true
Encoded root token: <encoded-root-token>
```

# Generate a new root token

```
$ vault generate-root -otp="<otp>" -decode="<encoded-root-token>"

Root token: <root-token>
```

# Authenticate with & revoke the root token

In the previous step one has generated a root token with which to authenticate. After authenticating using `vault auth` one may perform any operation.

It is **extremely important** to **revoke** this root token once the operation that required its generation is **finished**.

# Authenticate with & revoke the root token

```
$ vault auth

Successfully authenticated! You are now logged in.

token: <root-token>

token_duration: 0

token_policies: [root]

$ vault token-revoke -self

Success! Token revoked if it existed.
```

# Setup a Public-Key Infrastructure

# Setup the Root CA

To align with Vault's recommendations and with the industry's best practices one will provision two certificate authorities: a ***root certificate authority*** and an ***intermediate certificate authority***.

The root CA will have a lifespan of approximately **100 years**, being only used to sign the intermediate CA's certificates. It is this shorter-lived intermediate CA that will sign the certificates required by applications.

# Setup the Root CA

```
$ vault mount -path root-ca pki

Successfully mounted 'pki' at 'root-ca'!

$ vault mount-tune -max-lease-ttl 876000h root-ca

Successfully tuned mount 'root-ca'!

$ vault write -field=certificate root-ca/root/generate/internal \

     common_name="travelaudience GmbH Root CA" \

     ttl="876000h" > root-ca.crt
```

# Configure the Root CA

```
$ vault write root-ca/config/urls \

    issuing_certificates="https://vault.travelaudience.com/v1/root-ca/ca" \

    crl_distribution_points="https://vault.travelaudience.com/v1/root-ca/crl"

Success! Data written to: root-ca/config/urls
```

# Setup the Intermediate CA

Now one is ready to setup the Intermediate CA. This CA's lifespan will be of approximately ten years and it will be responsible for signing the certificates requested by applications.

In order to setup this CA one first instructs Vault to generate a ***certificate signing request (CSR)*** that will be signed by the root CA. Then, one uploads the signed certificate chained with the root CA's certificate so that Vault can distribute it.

# Setup the Intermediate CA

```
$ vault mount -path=intermediate-ca pki

Successfully mounted 'pki' at 'intermediate-ca'!

$ vault mount-tune -max-lease-ttl 87600h intermediate-ca

Successfully tuned mount 'intermediate-ca'!

$ vault write -field=csr intermediate-ca/intermediate/generate/internal \

    common_name="travelaudience GmbH Ltd. Intermediate CA" > intermediate-ca.csr
```

# Sign the Intermediate CA’s Certificate

```
$ vault write -field=certificate root-ca/root/sign-intermediate \

    csr=@intermediate-ca.csr ttl="87600h" > intermediate-ca.crt
```

# Upload the signed Intermediate CA’s certificate

```
$ cat intermediate-ca.crt <(echo) root-ca.crt > intermediate-chain.crt

$ vault write intermediate-ca/intermediate/set-signed \

    certificate=@intermediate-chain.crt
```

# Disaster recovery

# Disaster recovery

From the moment it is unsealed, Vault is completely stateless. This means that all information about the state of the system is stored in its storage backend, or in this case, `etcd`.

Therefore, backing up the Vault deployment is a matter of backing up the underlying `etcd` cluster.

# Disaster recovery

`etcd-backup-operator` and `etcd-restore-operator` provide the basic functionality one needs to backup and restore the etcd cluster.

Backups are written *on-demand* to a Cloud Storage (GCS) bucket from where they may be later retrieved to re-create the cluster.

# Create a GCS bucket

```
$ gsutil mb -c coldline -l eu -p <project-id> gs://<bucket-name>

Creating gs://<bucket-name>/...
```

# Create an `etcd` backup

Backups of the `etcd` cluster are created using the `EtcdBackup` custom resource. The creation of an `EtcdBackup` triggers the creation of a backup to the Google Cloud Storage bucket.

The hexadecimal value of the current **`etcd` revision** is included in the filename, which is created under the `vault/etcd` path.

# Create an `etcd` backup

```
apiVersion: etcd.database.coreos.com/v1beta2
kind: EtcdBackup
metadata:
  name: example-etcd-backup
  namespace: vault
spec:
  clusterName: etcd
  gcs:
    bucketName: <bucket-name>
  operatorSecret: etcd-operator-tls
 storageType: GCS
```

```
$ kubectl create -f example-etcd-backup.yaml
etcdbackup "example-etcd-backup" created
```

# Verify the `etcd` backup was successful

```
$ gsutil ls -lh -p <project-id> gs://<bucket-name>/vault/etcd/

    7.09 MiB  2017-12-03T08:00:05Z  gs://<bucket-name>/vault/etcd/3.1.10_<revision>_etcd.backup

TOTAL: 1 objects, 7430176 bytes (7.09 MiB)
```

# Simulate disaster

Disaster may come in any form, being it an accidental delete of the Kubernetes cluster or even GCP failure.

For demonstration purposes, and for the sake of simplicity, one will simply delete the `etcd` cluster “manually” in order to simulate total loss.

# Simulate disaster

```
$ kubectl -n vault delete etcdcluster etcd

etcdcluster "etcd" deleted

$ kubectl -n vault get pod | awk 'NR == 1 || /etcd/'

NAME                                    READY     STATUS    RESTARTS   AGE

etcd-backup-operator-75cd7c69c6-7j7fs   1/1       Running   0          1h

etcd-operator-6c7bdfc7b4-6skkm          1/1       Running   0          1h

etcd-restore-operator-5d76cb87f5-9tlzl  1/1       Running   0          1h
```

# Simulate disaster

In the unlikely event of Vault deployment surviving `etcd` cluster failure, the Vault pods will eventually enter the `CrashLoopBackoff` state since they can't reach the storage backend. When etcd becomes available, Vault resumes operation in **sealed** mode.

# Simulate disaster

```
$ kubectl -n vault get pod | awk 'NR == 1 || /vault/'

NAME                READY    STATUS             RESTARTS   AGE

vault-0             1/1      CrashLoopBackOff   5          1h

vault-1             1/1      CrashLoopBackOff   5          1h

$ kubectl -n vault logs vault-0 vault

Error initializing storage of type etcd: (...) dial tcp: lookup

etcd-client.vault.svc.cluster.local on 10.15.240.10:53: no such host
```

# Restore the `etcd` cluster

Much like creating a backup, restoring the `etcd` cluster is a matter of creating an `EtcdRestore` custom resource pointing at the latest backup stored in the bucket.

The `EtcdRestore` spec will contain parts of the original `EtcdCluster` spec — cf. `etcd/etcd-etcdcluster.yaml` — so that the restored cluster is identical.

# Restore the `etcd` cluster

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdRestore"
metadata:
  name: etcd
  namespace: vault
spec:
  clusterSpec:
    size: 3
    version: 3.1.10
    TLS:
      static:
        member:
          peerSecret: etcd-peer-tls
          serverSecret: etcd-server-tls
        operatorSecret: etcd-operator-tls
  gcs:
    path: gs://<bucket-name>/vault/etcd/3.1.10_<revision>_etcd.backup
```

```
$ kubectl create -f etcd-etcdrestore.yaml

etcdrestore "etcd" created
```

# Verify that the `etcd` cluster has been re-created

After a few minutes the `etcd` cluster will be up and running again.

One may check progress either by inspecting the `vault` namespace or by watching the logs of the `etcd-operator` pod.

# Verify that the `etcd` cluster has been re-created

```
$ kubectl -n vault get etcdcluster
NAME      AGE
etcd      4m
$ kubectl -n vault get pod | awk 'NR==1 || /etcd-[0-9]+/'
NAME                                  READY     STATUS    RESTARTS   AGE
etcd-0000                             1/1       Running   0          4m
etcd-0001                             1/1       Running   0          4m
etcd-0002                             1/1       Running   0          4m
```

# Unseal `vault-0` and `vault-1`

Since the Vault pods crashed and were restarted, one must now unseal `vault-0` and `vault-1` again so that they may serve requests.

The unsealing procedure is the same as the one previously described. Once it is performed one may check whether data has in fact been restored by inspecting, for instance, the `root-ca/config/urls` path.

# Verify that data has been restored

```
$ vault read root-ca/config/urls

Key                        Value

---                        -----

crl_distribution_points  [https://vault.travelaudience.com/v1/root-ca/crl]

issuing_certificates     [https://vault.travelaudience.com/v1/root-ca/ca]

ocsp_servers             []
```

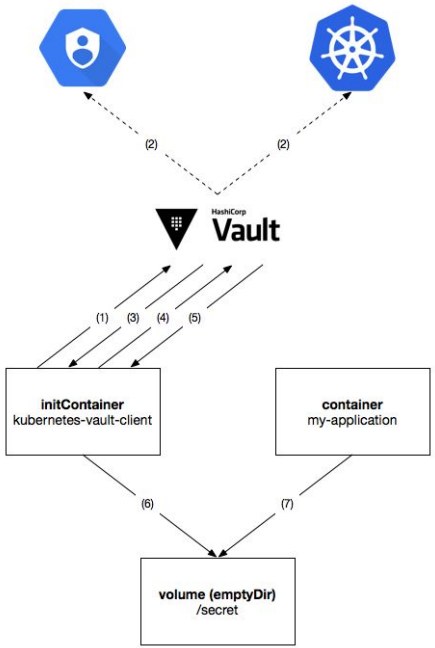
# kubernetes-vault-client

# kubernetes-vault-client

kubernetes-vault-client helps Kubernetes applications access Vault secrets in a secure way. It runs as an ***init container*** that automatically authenticates with Vault and dumps requested secrets and certificates as files to a shared volume so that they can be used by other containers in the pod.

kubernetes-vault-client supports authenticating against both the gcp and kubernetes auth backends. Using the kubernetes auth backend is preferable in almost all situations since it is much easier to configure and integrates tightly with the Kubernetes concepts of ServiceAccount and Namespace.

# kubernetes-vault-client

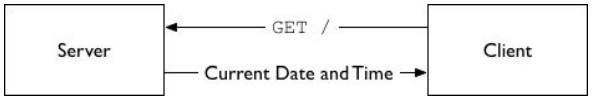


# Example: `time-server`

# Example: `time-server`

One will now walk through the steps necessary to transform an insecure web application — one that stores credentials in the code and does not use TLS — into a secure application that reads secrets and TLS certificates from Vault.

The application used as an example is a simple HTTP time server that responds to `GET /` requests with the current date and time.



# Example: `time-server`

The code for this example lives at

https://github.com/travelaudience/kubernetes-vault-example

The code and policies in the repo are intentionally very restrictive (and thus more secure). They serve as a reference of what things should look like in a production scenario.

However, and in order to ease things during this training, we will use slightly different, more permissive policies.

# Provision the `time-server` credentials as a secret

```
$ vault write secret/time-server-credentials \
    username="time-client" \
    password="safe#passw0rd!"
Success! Data written to: secret/time-server-credentials
```

# Create PKI role for `time-server`

```
$ vault write intermediate-ca/roles/time-server \

  allowed_domains="time-server.default.svc.cluster.local,time-server.default,time-server" \

  allow_subdomains="false" \

  allow_bare_domains="true" \

  max_ttl="2160h" \

  client_flag="false"

Success! Data written to: intermediate-ca/roles/time-server
```

# Create PKI role for `time-client`

```
$ vault write intermediate-ca/roles/time-client \
    allowed_domains="time-client" \
    allow_subdomains="false" \
    allow_bare_domains="true" \
    max_ttl="2160h" \
    server_flag="false"
Success! Data written to: intermediate-ca/roles/time-client
```

# Create the `time-server` policy

```
$ cat <<EOF > time-server.hcl

path "secret/time-server-credentials" { capabilities = ["read"] }

path "intermediate-ca/issue/time-client" { capabilities = ["create","update"] }

path "intermediate-ca/issue/time-server" { capabilities = ["create","update"] }

EOF

$ vault write sys/policy/time-server policy=@time-server.hcl

Success! Data written to: sys/policy/time-server
```

# Enable the kubernetes authentication backend

```
$ KUBERNETES_HOST=$(gcloud container clusters describe gke-1 \
    --format json --project training-235717 \
    | jq -r .endpoint)
$ KUBERNETES_CA_CERT=$(gcloud container clusters describe gke-1 \
    --format json --project training-235717 \
    | jq -r .masterAuth.clusterCaCertificate | base64 -D)
```

# Enable the kubernetes authentication backend

```
$ vault auth-enable --path "training-235717-gke-1" kubernetes

Successfully enabled 'gcp' at 'training-235717-gke-1'!

$ vault write auth/training-235717-gke-1/config \

    kubernetes_host="https://${KUBERNETES_HOST}" \

    kubernetes_ca_cert="${KUBERNETES_CA_CERT}"

Success! Data written to: auth/training-235717-gke-1/config
```

# Configure RBAC to allow authentication

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tokenreview-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:auth-delegator
subjects:
- kind: ServiceAccount
  name: default
  namespace: default
```

```
$ gcloud container clusters get-credentials \
    gke-1 \
    --project training-235717
$ kubectl create -f tokenreview-binding.yaml
clusterrolebinding "tokenreview-binding" created
```

# Create the `time-server` role in the auth backend

```
$ vault write auth/training-235717-gke-1/role/time-server \

    bound_service_account_names="default" \

    bound_service_account_namespaces="default" \

    policies="default,time-server" \

    period="60s"

Success! Data written to: auth/training-235717-gke-1/role/time-server
```

# Initial example

Code

# Initial example

```
$ kubectl create -f 01-initial-example/01-time-server.yaml
statefulset "time-server" created
service "time-server" created
$ kubectl create -f 01-initial-example/02-time-client.yaml
statefulset "time-client" created
$ kubectl get pod
NAME            READY     STATUS    RESTARTS   AGE
time-client-0   1/1       Running   0          1m
time-server-0   1/1       Running   0          1m
```

# Initial example

```
$ kubectl logs time-server-0

2017/12/06 12:33:33 got time request from 'time-client'

2017/12/06 12:33:38 got time request from 'time-client'

$ kubectl logs time-client-0

2017/12/06 12:33:33 got time from server: 2017-12-06T12:33:33Z

2017/12/06 12:33:38 got time from server: 2017-12-06T12:33:38Z
```

# Initial example (cleanup)

```
$ kubectl delete statefulset time-client
$ kubectl delete statefulset time-server
$ kubectl delete service time-server
```

# Using secrets from Vault

Code

# Using secrets from Vault

```
$ kubectl create -f 02-using-secrets-from-vault/01-configmap.yaml
configmap "kubernetes-vault-client-02" created
$ kubectl create -f 02-using-secrets-from-vault/02-time-server.yaml
statefulset "time-server" created
service "time-server" created
$ kubectl create -f 02-using-secrets-from-vault/03-time-client.yaml
statefulset "time-client" created
$ kubectl get pod
(...)
```

# Using secrets from Vault

```
$ kubectl logs time-server-0 kubernetes-vault-client

time="2017-12-06T13:02:39Z" level=info msg="vault: auth successful"

time="2017-12-06T13:02:40Z" level=info msg="initC: dump successful"

$ kubectl logs time-client-0 kubernetes-vault-client

time="2017-12-06T13:03:22Z" level=info msg="vault: auth successful"

time="2017-12-06T13:03:23Z" level=info msg="initC: dump successful"

$ kubectl exec time-server-0 -- cat /secret/username

time-client

$ kubectl exec time-server-0 -- cat /secret/password

safe#passw0rd!
```

# Using secrets from Vault (cleanup)

```
$ kubectl delete statefulset time-client

$ kubectl delete statefulset time-server

$ kubectl delete service time-server

$ kubectl delete configmap kubernetes-vault-client-02
```

# Using certificates from Vault

Code

# Using certificates from Vault

```
$ kubectl create -f 03-using-certificates-from-vault/01-configmap.yaml
configmap "kubernetes-vault-client-03" created
$ kubectl create -f 03-using-certificates-from-vault/02-time-server.yaml
statefulset "time-server" created
service "time-server" created
$ kubectl create -f 03-using-certificates-from-vault/03-time-client.yaml
statefulset "time-client" created
$ kubectl get pod
(...)
```

# Using certificates from Vault

```
$ kubectl logs time-server-0 kubernetes-vault-client

time="2017-12-06T13:44:04Z" level=info msg="vault: auth successful"

time="2017-12-06T13:44:04Z" level=info msg="initC: dump successful"

$ kubectl logs time-client-0 kubernetes-vault-client

time="2017-12-06T13:44:06Z" level=info msg="vault: auth successful"

time="2017-12-06T13:44:06Z" level=info msg="initC: dump successful"

$ kubectl exec time-server-0 -- ls /secret

chain.pem

crt.pem

key.pem
```

# Using secrets from Vault (cleanup)

```
$ kubectl delete statefulset time-client

$ kubectl delete statefulset time-server

$ kubectl delete service time-server

$ kubectl delete configmap kubernetes-vault-client-03
```

Questions?