# Automated Certificate Issuance with Hashicorp Vault

Slides and walkthrough available on GitHub

# What makes certificates so special (compared to regular keys and passwords)?

# A certificate is a proof of your identity.

In computer terms, it means a certificate authority vouches that you have a **private** key.

Unlike a password, that private key is known only by the owner and never shared.

# What are certificate used for?

Certificates prove your identity to a third party that shares a common trustee with you.

You don't need prior arrangement, but you must

- Trust the authority that issued the certificate
- Be able to confirme the certificate is authentic

Futur entities will be able to authenticate you if they decide to use the same trustee

# Examples of certificates

## High trust

# Examples of certificates

## Low trust

# Issuing certificates

# Old-school certificate process

1. Generate a key pair
    - Public that everyone will see
    - Private that **only the owner** will see
2. Generate a certificate signing request (CSR)
    - Prove that you own the private key, without sending it
3. Send the CSR to the certificate authority
4. Wait for the Certificate Authority to issue a certificate
    - The CA will sign your public key
5. Configure your web server to use the private key and certificate
6. Goto 1 every year or so… Don't forget!

# Who/What should generate the key ?

Private key **can be exported** unless it is on a smartcard

- If you can use the key, you can export it (iSECPartner's jailbreak)
- Using smartcards is not pratical unless you are sitting next to the server

Private key reuse is a handy Wireshark hack

- Most CA don't check for that
- It lowers security

Vault can do both, and you can mix them at will

- Generate the private key and certificate at once
- Sign a certificate request

Pick the one that works best for every circomstance

# Prepare the PKI backend

## Mount the backend

```
#Nothing to install, just enable the PKI secret backend

vault secrets enable --path=issuer pki
```

# Prepare the PKI backend

Mount the backend

```
#Nothing to install, just enable the PKI secret backend

vault secrets enable --path=issuer pki
```

Set the URL that will be put in the issued certificates

```
#Enable the PKI engine on path `issuer`

vault secrets enable --path=issuer pki
```

# Prepare the PKI backend

Mount the backend

```
#Nothing to install, just enable the PKI secret backend

vault secrets enable --path=issuer pki
```

Set the URL that will be put in the issued certificates

```
#Enable the PKI engine on path `issuer`

vault secrets enable --path=issuer pki
```

Vault doesn't know how it can be reached on the network, tell it

```
#Set the OCSP and CRL protocol URL

vault write issuer/config/urls \
   issuing_certificates="http://localhost:8200/v1/issuer/ca" \
   crl_distribution_points="http://localhost:8200/v1/issuer/crl" \
   ocsp_servers="http://localhost:8200/v1/issuer/ocsp"
```

# Link Vault to your corporate PKI

Your Public Key Infrastructure (PKI) looks like this:

- A Root Certificate Authority (CA) is used only to sign the Issuing CA's certificates
- Issuers are the ones actually giving out certificates

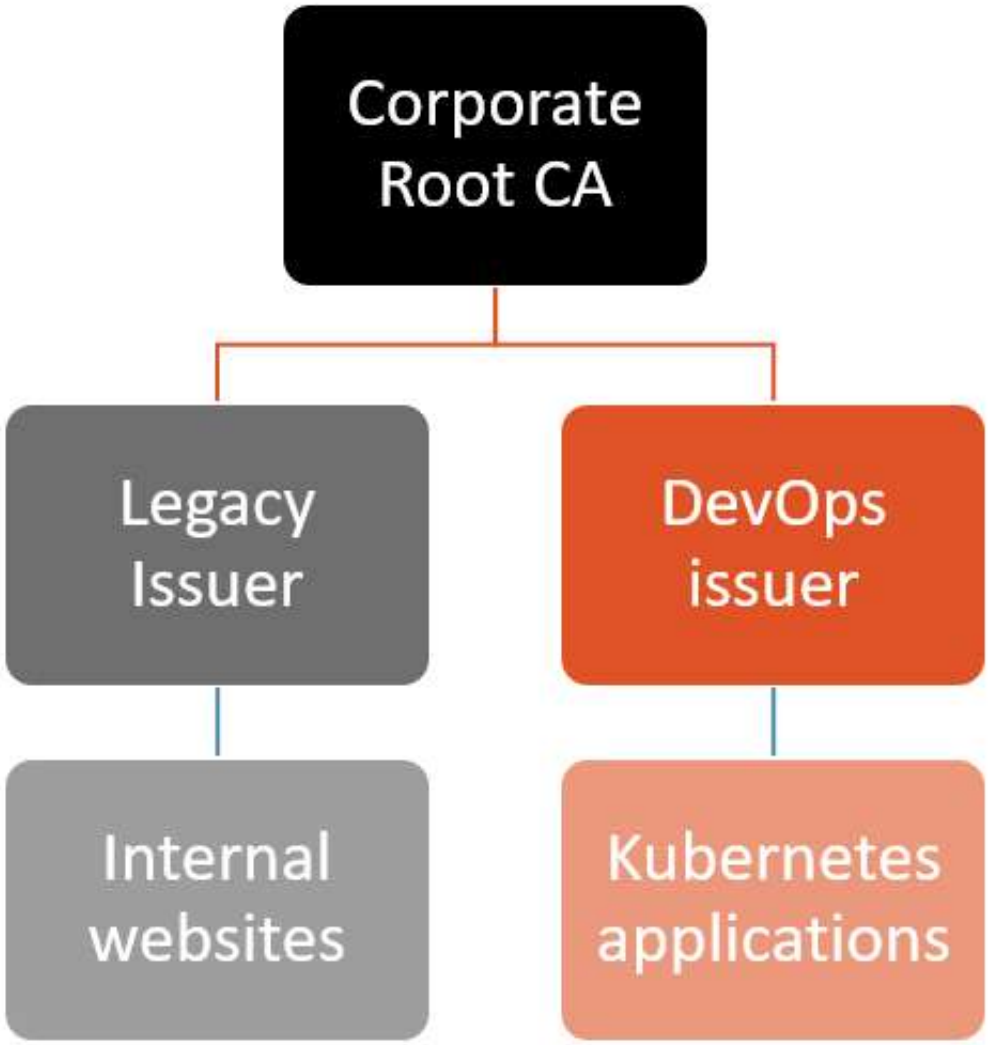
Vault will create a CSR, for your PKI CA to sign

- Old school, long lived certificate, not automated
- Root CA are usually offline and revived every 2 years, plan accordingly

Vault will be another issuer of your PKI

- You will need to push the certificate to your corporate desktops

# Link Vault to your corporate PKI



# Create key pair and Certificate Signing Request

Vault's private key can be

- Imported: You are tranfering from a legacy PKI, like openssl
- Generated: Private key will never leave Vault

To have vault generate (and keep) its private key:

```
#Using internal in this command means the private key never leaves Vault
#Gives the name devops-issuer.paralint.lab to your issuer

vault write --field=csr issuer/intermediate/generate/internal \
   common_name=devops-issuer.paralint.lab | tee devops-request.csr
```

The certificate signing request is saved in the file `devops-request.csr`

# Give Vault its certificate

Have the certificate request signed by your certificate authority

- Remember, root CA are offline most of the time

# Give Vault its certificate

Have the certificate request signed by your certificate authority

- Remember, root CA are offline most of the time

Add the certificate to Vault

```
#Complete the issuer base configuration

vault write issuer/intermediate/set-signed certificate=@devops-cert.pem
```

# Define your certificate template (aka Role)

About one template per use case

- HTTPS for internal applications
- Email signature
- User authentication

You can restrict many aspect of the certificate you issue with a given tempalte

- Hostnames
- Validity period
- Key usage
- Many more documented online

# Example certificate template (aka Role)

This would be the content of `devops-role.json` managed "as-code"

```
{
  "allow_any_name": false,
  "allow_bare_domains": false,
  "allow_glob_domains": true,
  "allow_ip_sans": false,
  "allow_localhost": false,
  "allow_subdomains": false,
  "allowed_domains": [ "app*.cloud.paralint.lab" ],
  "allowed_other_sans" : "email;UTF-8:*@paralint.lab",
  "country": [ "CA" ],
  "enforce_hostnames": true,
  "ext_key_usage": [ "ServerAuth" ],
  "key_bits": 2048,
  "key_type": "rsa",
  "key_usage": [ "DigitalSignature", "KeyAgreement", "KeyEncipherment" ],
  "locality": [ "Montreal" ],
  "max_ttl": "72h",
  "organization": [ "Hashicorp User Group" ],
  "ou": [ "Kubernetes DevOps" ],
  "postal_code": [ "H3B 2E3" ],
  "province": [ "Quebec" ],
  "server_flag": true,
  "ttl": "24h"
}
```

# Add the template to Vault

Just POST the JSON file to the endpoint name:

```
vault write issuer/roles/devops @devops-role.json
```

This operation should be restricted to the security team

# Issue a certificate using that template

You must be authenticated Post a JSON request to Vault and the get the goods right back

```
#This translate to a POST request to Vault
vault write issuer/issue/devops @- << EOF
{
   "ttl" : "48h",
   "common_name": "apptastic.cloud.paralint.lab",
   "alt_names":   "app5678.cloud.paralint.lab, app9999.cloud.paralint.lab"
}
EOF
```

How to use it is up to you

- Your platform knows how to automate this
- It might require to use the `sign` endpoint

# Renew a certificate

You don't really renew a certificate, you get a new one

Should you revoke the certificate you are replacing?

- Certificates are short lived
- What happens if you restore a backup?

# Certificate revocation

The hard part about revocation, is when to do it

You just post the certificate serial number to Vault API

You will likely have this endpoint restricted

# Permissions

Use Vault path based ACL, like you would for anything else

| Task | Performed by | Vault path |
|---|---|---|
| Mount the PKI backend | Vault Root token | `sys/mount` |
| Create/Update a certificate template | Security administrator | `pki/roles/:name` |
| Issue a certificate | Infrastructure | `pki/issue/:name`<br>`pki/sign/:name` |
| Revoke a certificate | Security Administrator | `pki/revoke/` |

In this talk, the default mount name "`pki`" was replaced by "`issuer`"

# Integrating with Kubernetes cert-manager

Define an `Issuer` resource

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Issuer
metadata:
  name: vault-issuer
  namespace: default
spec:
  vault:
    path: issuer/sign/devops
    server: https://vault.paralint.lab
    caBundle: <base64 encoded caBundle PEM file>
    auth:
      appRole:
        path: approle
        roleId: "291b9d21-8ff5-..."
        secretRef:
          name: cert-manager-vault-approle
          key: secretId
```

# Automation

Define a `Certificate` resource that uses your `Issuer` resource:

```
apiVersion: certmanager.k8s.io/v1alpha1
kind: Certificate
metadata:
  name: apptastic
  namespace: default
spec:
  secretName: apptastic-tls
  issuerRef:
    name: vault-issuer
  commonName: apptastic.cloud.paralint.lab
  dnsNames:
  - app5678.cloud.paralint.lab
  - app9999.cloud.paralint.lab
```

Cert-manager will store the create certificate in the `apptastic-tls` Kubernetes secret