Encrypting data is important, both in transit and at rest. By far the most popular method of in-transit encryption is SSL/TLS. That sad truth is, except for our public facing web sites, most administrators rarely use it unless they have to. Many companies only run their own CA for VPN's or LDAP infrastructure, and they tend to use old solutions like Easy-RSA (https://github.com/OpenVPN/easy-rsa).

Hashicorp's Vault (https://www.vaultproject.io/) burst onto the scene last year and has taken secrets management to the next level. One underrated capability of Vault is to act as a Certificate Authority (CA) via the PKI secrets backend. The docs (https://www.vaultproject.io/docs/secrets/pki/index.html) are a little thin for helping people get going, so I wanted to provide a complete walkthrough to help people explore this exciting capability of Vault.

In this tutorial we'll:

Setup a Vault Server

Create a Root CA for our organization

Create an Intermediate CA for our organization

Create TLS Keys and Certificates for a web server

Test the certificate using NGINX

I want to clarify at the outset that this is a proof-of-concept walkthrough and doesn't necessarily constitute good or best practices. Our focus here is on the basics of utilizing the PKI backends for our purposes. A real-world deployment of Vault should be setup in HA mode, be protected with TLS itself, utilize non-root tokens and policies, and the TTL's associated with your CA's and Certs should be carefully considered depending on your deployment.

The most exciting aspect of Vault as a CA is that your end-points requiring protection can request certs and keys directly from Vault whenever they wish. This means that you can set really low TTL's on your certificates and simply update them from cron on a regular basis. That functionality however is something I'll cover in a separate blog, but be assured that it is that capability that takes Vault from a nifty alternative to EasyRSA or CFSSL (https://github.com/cloudflare/cfssl) to a mind-blowing game changer for how we manage TLS.

Starting Vault

After downloading Vault (https://www.vaultproject.io/downloads.html), I'm going to create a directory into which it'll store secrets and configuration. We'll create a basic configuration file and start the server:

```
$ mkdir vault && cd vault
$ vi vault.hcl
disable_mlock  = true

listener "tcp" {
  address = "0.0.0.0:8200"
  tls_disable = 1
}

backend "file" {
  path = "/home/benr/vault/secrets"
}
$ vault server -config=vault.hcl
==> Vault server configuration:

                 Backend: file
              Listener 1: tcp (addr: "0.0.0.0:8200", tls: "disabled")
```

```
           Log Level: info
               Mlock: supported: true, enabled: false
             Version: Vault v0.6.0

==> Vault server started! Log data will stream in below:
```

In another terminal we'll initialize and unseal the Vault for use:

```
$ export VAULT_ADDR='http://127.0.0.1:8200'
$ vault init
Unseal Key 1: 811538b33c90d6f558b0296e12dc0023fc4086f5cbc424a2a3766d52dd52d7cf01
Unseal Key 2: 3eb6e073249168bdef779cf0e47f5c02baf7a2260e3d531073ae40862916029302
Unseal Key 3: b99b0b9a16f2f88ab83f87ddab4e6f483b15f288889bc9d1b9b2d154ad14ac8f03
Unseal Key 4: c24260a579914e78f78123b7a83fc96ebd16434980fc5a003f24bd5e2ecf7fa804
Unseal Key 5: 456f8b4c4bf2de4fa0c9389ae70efa243cf413e7065ac0c1f5382c8caacdd1b405
Initial Root Token: d194e2e3-6483-aa23-9bf2-f1bb31b0edbb
...

$ vault unseal 811538b33c90d6f558b0296e12dc0023fc4086f5cbc424a2a3766d52dd52d7cf01
$ vault unseal 3eb6e073249168bdef779cf0e47f5c02baf7a2260e3d531073ae40862916029302
$ vault unseal b99b0b9a16f2f88ab83f87ddab4e6f483b15f288889bc9d1b9b2d154ad14ac8f03
Sealed: false
Key Shares: 5
Key Threshold: 3
Unseal Progress: 0
$ vault auth
Token (will be hidden): d194e2e3-6483-aa23-9bf2-f1bb31b0edbb
Successfully authenticated! You are now logged in.
token: d194e2e3-6483-aa23-9bf2-f1bb31b0edbb
token_duration: 0
token_policies: [root]
```

Great! Vault is up and unsealed and ready to use.

Creating a Root CA

Within Vault, secrets are managed by "backends". To use a backend it must be mounted. When you get started with Vault this seems very odd, but there turns out to be a good reason. Backends can be mounted multiple times with different paths. This is extremely important when we do PKI because each PKI backend can only represent a single CA! Therefore, we'll be mounting the PKI backend twice, once for the Root CA and one more for the Intermediate CA. In this way, you can support as many CA's as you wish on a single Vault server, keeping them completely distinct.

So, we begin by mounting a PKI backend for our "cuddletech" Root CA. When we mount it, we'll provide a "path" (used for accessing the specific backend), description, and maximum lease TTL:

```
$ vault mount -path=cuddletech -description="Cuddletech Root CA" -max-lease-ttl=87600h pki
Successfully mounted 'pki' at 'cuddletech'!
$ vault mounts
Path         Type       Default TTL  Max TTL    Description
cubbyhole/   cubbyhole  n/a          n/a        per-token private secret storage
cuddletech/  pki        system       315360000  Cuddletech Root CA
secret/      generic    system       system     generic secret storage
sys/         system     n/a          n/a        system endpoints used for control, policy and debugging
```

Now we're ready to actually create our CA Certificate and Key!

```
$ vault write cuddletech/root/generate/internal \
> common_name="Cuddletech Root CA" \
> ttl=87600h \
> key_bits=4096 \
> exclude_cn_from_sans=true
Key             Value
---             -----
certificate     -----BEGIN CERTIFICATE-----
MIIFKzCCAxOgAwIBAgIUDXiI3GDzP2IbQ9IatFSCv9Pq/lgwDQYJKoZIhvcNAQEL
BQAwHTEbMBkGA1UEAxMSQ3VkZGxldGVjaCBSb290IENBMB4XDTE2MDcwOTA4MTIz
..
axscmLdVE2HTB87W1H77iKKN8n9Xne//LUidxVX0Kg==
-----END CERTIFICATE-----
expiration      1783411981
issuing_ca      -----BEGIN CERTIFICATE-----
MIIFKzCCAxOgAwIBAgIUDXiI3GDzP2IbQ9IatFSCv9Pq/lgwDQYJKoZIhvcNAQEL
BQAwHTEbMBkGA1UEAxMSQ3VkZGxldGVjaCBSb290IENBMB4XDTE2MDcwOTA4MTIz
...
axscmLdVE2HTB87W1H77iKKN8n9Xne//LUidxVX0Kg==
-----END CERTIFICATE-----
serial_number   0d:78:88:dc:60:f3:3f:62:1b:43:d2:1a:b4:54:82:bf:d3:ea:fe:58
```

Excellent! We have a CA! Lets look at the certificate to verify. We'll pull this certificate via *curl* and pipe the PEM into *openssl* (the *vault* CLI has a bug the makes it preferable to use curl in this case):

```
$ curl -s http://localhost:8200/v1/cuddletech/ca/pem | openssl x509 -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0d:78:88:dc:60:f3:3f:62:1b:43:d2:1a:b4:54:82:bf:d3:ea:fe:58
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=Cuddletech Root CA
        Validity
            Not Before: Jul  9 08:12:31 2016 GMT
            Not After : Jul  7 08:13:01 2026 GMT
        Subject: CN=Cuddletech Root CA
...
```

Great! The last thing for us to do is to properly configure the URL's Vault will use for accessing the CA and CRL URLs:

```
$ vault write cuddletech/config/urls issuing_certificates="http://10.0.0.22:8200/v1/cuddletech
Success! Data written to: cuddletech/config/urls
```

The CA is ready. On to our intermediate CA!

Creating an Intermediate CA

Creating the Intermediate CA is similar to that of the Root CA, with the big difference being that instead of creating a Cert and Key in one action, we'll create a key and CSR, then sign that CSR by the Root before putting the resulting Cert back into the intermediate. So we'll be working with a second PKI backend but switching back into the first just to sign the CSR.

So, again, create a new backend for the intermediate. We'll call this the Ops Intermediate CA:

```
$ vault mount -path=cuddletech_ops -description="Cuddletech Ops Intermediate CA" -max-lease-ttl=26280h pki
Successfully mounted 'pki' at 'cuddletech_ops'!

$ vault mounts
Path             Type       Default TTL  Max TTL    Description
cubbyhole/       cubbyhole  n/a          n/a        per-token private secret storage
cuddletech/      pki        system       315360000  Cuddletech Root CA
cuddletech_ops/  pki        system       94608000   Cuddletech Ops Intermediate CA
```

Next we generate an Intermediate CSR:

```
$ vault write cuddletech_ops/intermediate/generate/internal \
>  common_name="Cuddletech Operations Intermediate CA"
>  ttl=26280h \
>  key_bits=4096 \
>  exclude_cn_from_sans=true
Key     Value
---     -----
csr     -----BEGIN CERTIFICATE REQUEST-----
MIICuDCCAaACAQAwMDEuMCwGA1UEAxMlQ3VkZGxldGVjaCBPcGVyYXRpb25zIElu
dGVybWVkaWF0ZSBDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALt8
...
hD8cpHTXqjKExYWKc/rQDgjw9+RNDdb45xszDagrgFgNPqI9i0fNh9jViMmjUiTc
PQTZS4XxIoRrx1/xVHJ4Qm++ntLPVCvzjMZafg==
-----END CERTIFICATE REQUEST-----
```

We'll cut and paste that CSR into a new file **cuddletech_ops.csr**. The reason we output the file here is so we can get it out of one backend and into another and then back out.

```
$ vault write cuddletech/root/sign-intermediate \
>  csr=@cuddletech_ops.csr \
>  common_name="Cuddletech Ops Intermediate CA" \
>  ttl=8760h
Key             Value
---             -----
certificate     -----BEGIN CERTIFICATE-----
MIIEZDCCAkygAwIBAgIUHuIhRF3tYtfoZiAFdjcCtQpMR+cwDQYJKoZIhvcNAQEL
BQAwHTEbMBkGA1UEAxMSQ3VkZGxldGVjaCBSb290IENBMB4XDTE2MDcwOTA4Mjkz
...
UtI2b/AamAqf340eRKmSdEh4WypB4JR+t259YA45w2j4mS+rxREycEk4YosR/vUs
```

```
jekMiq57yNq7h8eOTrnOulJxazbVrYGb
-----END CERTIFICATE-----
expiration      1470645002
issuing_ca      -----BEGIN CERTIFICATE-----
MIIFKzCCAxOgAwIBAgIUDXiI3GDzP2IbQ9IatFSCv9Pq/lgwDQYJKoZIhvcNAQEL
BQAwHTEbMBkGA1UEAxMSQ3VkZGxldGVjaCBSb290IENBMB4XDTE2MDcwOTA4MTIz
..
1FRGlwHUg+6IIZBVIapzivLc6pAvLFPxQlQvT5CNHPk91zwyNQ9ZX2PzatdajUnd
axscmLdVE2HTB87W1H77iKKN8n9Xne//LUidxVX0Kg==
-----END CERTIFICATE-----
serial_number   1e:e2:21:44:5d:ed:62:d7:e8:66:20:05:76:37:02:b5:0a:4c:47:e7
```

Now that we have a Root CA signed cert, we'll need to cut-n-paste this certificate into a file we'll name **cuddletech_ops.crt** and then import it into our Intermediate CA backend:

```
$ vault write cuddletech_ops/intermediate/set-signed \
> certificate=@cuddletech_ops.crt
Success! Data written to: cuddletech_ops/intermediate/set-signed
```

Awesome! Lets verify:

```
$ curl -s http://localhost:8200/v1/cuddletech_ops/ca/pem | openssl x509 -text | head -20
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            76:12:53:41:be:18:98:2c:a1:51:4a:f8:f0:bd:b4:a3:44:7e:74:59
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=Cuddletech Root CA
        Validity
            Not Before: Jul  9 09:23:39 2016 GMT
            Not After : Jul  9 09:24:09 2017 GMT
        Subject: CN=Cuddletech Ops Intermediate CA
 ...
```

The last thing we need to do is set the CA & CRL URL's for accessing the CA:

```
$ vault write cuddletech_ops/config/urls \
> issuing_certificates="http://10.0.0.22:8200/v1/cuddletech_ops/ca" \
> crl_distribution_points="http://10.0.0.22:8200/v1/cuddletech_ops/crl"
Success! Data written to: cuddletech_ops/config/urls
```

Requesting a Certificate for a Web Server

Now that our CA's are configured, we'll want to issue certificates. Doing so requires 2 steps. First we create a **role** which defines constraints around the certificates we generate, such as key type and strength, types of certificates allowed, etc. Secondly we'll actually request a certificate using the role.

We'll create a role named "web_server" on our Intermediate CA, which issues 2048 bit keys with a maximum TTL of 1 year and allows any name.

```
$ vault write cuddletech_ops/roles/web_server \
> key_bits=2048 \
> max_ttl=8760h \
> allow_any_name=true
Success! Data written to: cuddletech_ops/roles/web_server
```

Now we can use that role to issue a cert!

```
$ vault write cuddletech_ops/issue/web_server \
> common_name="ssl_test.cuddletech.com" \
> ip_sans="172.17.0.2" \
> ttl=720h
> format=pem
Key                     Value
---                     -----
lease_id                cuddletech_ops/issue/web_server/e03318f2-d005-8196-4ed5-a42f9cd55238
lease_duration          2591999
lease_renewable         false
certificate             -----BEGIN CERTIFICATE-----
MIIE7jCCAtagAwIBAgIUN+vXFuIf42v1SW+mDROUVAm+lUMwDQYJKoZIhvcNAQEL
BQAwKTEnMCUGA1UEAxMeQ3VkZGxldGVjaCBPcHMgSW50ZXJtZWRpYXRlIENBMB4X
DTE2MDcwOTA5MzE1N1oXDTE2MDgwODA5MzIyN1owIjEgMB4GA1UEAwwXc3NsX3Rl
...
issuing_ca              -----BEGIN CERTIFICATE-----
MIIF5DCCA8ygAwIBAgIUdhJTQb4YmCyhUUr48L20o0R+dFkwDQYJKoZIhvcNAQEL
...
```

```
private_key          -----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEApBabDpPZIloRQUpro3tQEls0FEFvsvfraQzQJLD2dicSPZ2s
CqYyT8OXMclrapG7KKTYp79AaTW8LgNg3WvCzoMGDfhLL9m0QomzrMDzoW8Q7iQO
1MV4f6JXjGMbOMMXatKQlO32fLZln8m+/yJ3pOW0S6uatFzZ/N3+ed+gDuUc7eAO
...
private_key_type     rsa
serial_number        37:eb:d7:16:e2:1f:e3:6b:f5:49:6f:a6:0d:13:94:54:09:be:95:43
```

Since we're using NGINX, we'll put the *certificate* and *issuing_ca* certs into a single file named **ssl_test.cuddletech.com.crt**. We'll put the *private_key* into **ssl_test.cuddletech.com.key**.

Now we're ready to setup NGINX!

Testing our Cert with NGINX

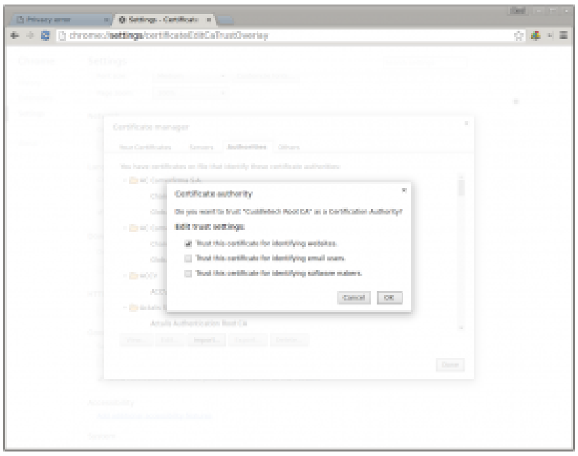
To test our certificate we'll use NGINX. Remember that NGINX expects the CA certificate to be appended into the same file as the ssl_certificate (server cert first, then CA cert afterwards). Copy the cert and key files to NGINX and start it up.

```
server {
    listen              443 ssl;
    server_name         ssl_test.cuddletech.com;
    ssl_certificate     ssl_test.cuddletech.com.crt;
    ssl_certificate_key ssl_test.cuddletech.com.key;
    ssl_protocols       TLSv1 TLSv1.1 TLSv1.2;
    ssl_ciphers         HIGH:!aNULL:!MD5;


    location / {
      root   /usr/share/nginx/html;
      index  index.html index.htm;
    }
}
```

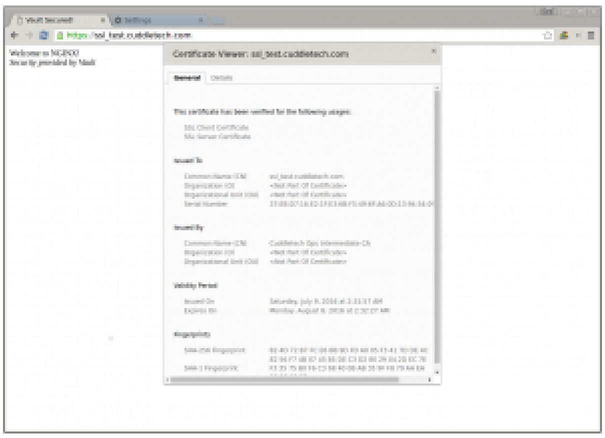
Once NGINX is up, point your browser at it and you'll get the old familiar "Your connection is not secure" warning. All we need to do is extract our Root CA certificate and import it into our browser thanks to chains of trust! To export the root CA:

```
$ curl -s http://localhost:8200/v1/cuddletech/ca/pem > cuddletech_ca.pem
```



(http://cuddletech.com/wordpress/wp-content/uploads/2016/07/vault-tls-2-1.png)

Once imported into your browsers Authority list you'll be flying high with your Vault powered internal TLS!



(http://cuddletech.com/wordpress/wp-content/uploads/2016/07/vault-tls-1-1.png)