Automatically generate PKI certificates with Vault

2017-08-25 wdijkerman

A while a go I wrote an item on how to setup a secure Vault with Consul as backend and its time to do something with Vault again. With this blogpost we will setup Vault with the PKI backend. With the PKI backend we can generate or revoke short lived ssl certificates with Vault.

The goal with this blogpost is that we create intermediate CA certificate, configure Vault and generate certificates via the cmd line and via the API. The reason we use intermediate CA certificate is that if something might happen with the certificate/key, its much easier to revoke it and recreate a new intermediate certificate. If this would happen with the actual ROOT CA, you'll have a troubles and work to fix it again. So keep the ROOT CA files on a safe place!

Preparations

We will create an intermediate certificate that Vault will be using to create and sign certificate requests. We have to create a new key and the certificate needs to be signed by the ROOT CA. First we create the key:

```
openssl genrsa -out private/intermediate_ca.key.pem 4096
```

And now we need to create a certificate signing request:

```
openssl req -config intermediate/openssl.cnf -new -sha256 \
-key private/intermediate_ca.key.pem -out \
csr/intermediate_ca.csr.pem
```

We have to make sure that we fill in the same information as the original CA, but in this case we use a slightly different Organisation Unit name so we know/verify that a certificate is signed by this intermediate CA instance. Once we filled in all data, we have to sign it with the ROOT CA to create the actual certificate:

```
openssl ca -keyfile private/cakey.pem -cert \
dj-wasabi.local.pem -extensions v3_ca -notext -md \
sha256 -in csr/intermediate_ca.csr.pem -out \
certs/intermediate_ca.crt.pem
Using configuration from /etc/pki/tls/openssl.cnf
Check that the request matches the signature
Signature ok
Certificate Details:
        Serial Number: 18268543712502854739 (0xfd86e7b7336db453)
        Validity
            Not Before: Aug 23 13:56:08 2017 GMT
            Not After : Aug 21 13:56:08 2027 GMT
        Subject:
            countryName               = NL
            stateOrProvinceName       = Utrecht
            organizationName          = dj-wasabi
            organizationalUnitName    = Vault CA
            commonName                = dj-wasabi.local
            emailAddress              = ikben@werner-dijkerman.nl
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                93:46:3D:69:24:32:C7:11:C4:B7:27:66:89:67:FB:1F:8E:1B:50:97
            X509v3 Authority Key Identifier:
                keyid:60:63:7E:0F:54:5E:7D:A5:37:A8:6F:BD:27:BF:73:15:56:B2:89:31

            X509v3 Basic Constraints:
                CA:TRUE
Certificate is to be certified until Aug 21 13:56:08 2027 GMT (3650 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]y
Write out database with 1 new entries
Data Base Update
```

With the -keyfile and -cert we provide the key and crt file of the root CA to sign the new intermediate ssl certificate. Ok, 10 years might be a little bit to long, but this is just for my local environment and my setup probably won't last that long. 🙂

We are almost done with the preparations and one thing we need to do before we go configuring Vault. We have to combine both the CA certificates and the intermediate private key into a single file, before we can upload it to Vault.

```
cat certs/intermediate_ca.crt.pem dj-wasabi.local.pem \
private/intermediate_ca.key.pem > certs/ca_bundle.pem
```

First we print the contents of the newly created crt file, then the ROOT ca crt file and as last the intermediate private key and place that all in a single file called *ca_bundle.pem*.

Vault

Now we are ready to continue with the Vault part. We open a terminal to the host/container running Vault and before we can do somehting, we have to authenticate ourself first. I use the root token for authenticating:

```
export VAULT_TOKEN=<_my_root_token_>
```

The **pki** backend is disabled at default so we have to enabled it before we can use it. You can enable it multiple times, each enabled backend can be used for a specific domain. In this post we only use one domain, but lets pretend we need to create a lot more after this so we don't use "defaults" in paths and naming.

We will mount the **pki** plugin for the *dj-wasabi.local* domain, so lets use the path: **dj-wasabi**. We give it a small description and then we specify the **pki** backend and then hit enter.

```
vault mount -path=dj-wasabi -description="dj-wasabi Vault CA" pki
```

There are some more options we don't use for now with this example but maybe you want some more control for it, you can see them by executing the command: *vault mount –help*.
We can verify that we have mounted the **pki** backend by executing the *vault mounts* command:

```
bash-4.3$ vault mounts
Path        Type       Accessor            Plugin  Default TTL  Max TTL    Force No Cache  Replication Behavior  Description
cubbyhole/  cubbyhole  cubbyhole_2540c354  n/a     n/a          n/a        false           local                 per-token private secret storage
dj-wasabi/  pki        pki_6e5dc562        n/a     system       system     false           replicated            dj-wasabi Vault CA
secret/     generic    generic_fb0527dd    n/a     system       system     false           replicated            generic secret storage
```

```
sys/        system     system_347beff9     n/a     n/a          n/a        false           replicated            system endpoints used for control, policy and debugging
```

Now its time to upload the intermediate bundle file. I have temporarily placed the file in the config directory of Vault (Its a host mount, so it was easier to copy the file to the container) and now we have to upload it to our *dj-wasabi* backend. We have to upload our ca bundle file into the path we earlier used to mount the **pki** backend: *<mount_path>/config/ca*, in my case it is *dj-wasabi/config/ca*:

```
vault write dj-wasabi/config/ca \
pem_bundle="@/vault/config/ca_bundle.pem"
Success! Data written to: dj-wasabi/config/ca
```

If you get an error now, it probably means something went wrong with either creating the ca bundle file or validating the intermediate certificate.

Now we need to set some correct urls. These urls are placed in the certificates that are generated and that allows browsers/applications to do some validations. We will set the following urls:

- **issuing_certificates**: The endpoint on which browsers/3rd party tools can request information about the CA;
- **crl_distribution_points**: The endpoint on which the Certification Revocation List is available. This is a list with revoked Certificates;
- **ocsp_servers**: The url on which the OCSP service is available. OCSP Stands for Online Certificate Status Protocol and is used to determine the state of the Certificate. You can see it as a better version of the Certificate Revocation List;

Lets configure the urls:

```
vault write dj-wasabi/config/urls \
issuing_certificates="https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/ca" \
crl_distribution_points="https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/crl" \
ocsp_servers="https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/ocsp"
Success! Data written to: dj-wasabi/config/urls
```

We will come later on the blog post about this. 🙂

Before we can generate certificates, we need to create a role in Vault. With this role we map a name to a policy. This policy describes the configuration that is needed for generating the certificates. For example we have to configure on which domain we need create the certificates, can we create sub domains and most important, what is the ttl of a certificate.

```
vault write dj-wasabi/roles/dj-wasabi-dot-local allowed_domains="dj-wasabi.local" allow_subdomains="true" max_ttl="72h"
Success! Data written to: dj-wasabi/roles/dj-wasabi-dot-local
```

We are all set now, so lets create a certificate.

We specify the just created role and at minimum we have to provide the common_name (In this case *small-test.dj-wasabi.local*). You can find here all the options you can give when generating a certificate. The command looks like this:

```
vault write dj-wasabi/issue/dj-wasabi-dot-local common_name=small-test.dj-wasabi.local
Key                 Value
---                 -----
ca_chain            [-----BEGIN CERTIFICATE-----
MIIFtTCCA52gAwIBAgIJAP2G57czbbRTMA0GCSqGSIb3DQEBCwUAMFcxCzAJBgNV
...
-----END CERTIFICATE-----
issuing_ca          -----BEGIN CERTIFICATE-----
MIIFtTCCA52gAwIBAgIJAP2G57czbbRTMA0GCSqGSIb3DQEBCwUAMFcxCzAJBgNV
...
```

```
-----END CERTIFICATE-----
private_key         -----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEAsFSmpBCFN945+Chyz/YqsB2a/T73kdst4v7qm2ZLK50RxCj0
...
-----END RSA PRIVATE KEY-----
private_key_type    rsa
serial_number       03:f2:bb:f5:27:16:81:20:76:0d:91:6f:fd:10:05:2d:a6:e1:59:e3
```

The command provides a lot of information and I have removed some of it to not full a whole page with unreadable data. It provides you all the data you'll need to create a service that needs ssl certificates. As you see, it provides the *certificate* and the *private_key*, but also the *ca_chain*.

API

Lets generate a SSL certificate via the API.

```
curl -XPOST -k -H 'X-Vault-Token: <_my_root_token_>' \
-d '{"common_name": "blog.dj-wasabi.local"}' \
https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/issue/dj-wasabi-dot-local
```

We do an *POST*, and as a minimum we only provide the common_name (In this case *blog.dj-wasabi.local*). We use the *X-Vault-Token* which in my case is the ROOT Token as a header and we post it to the url *https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/issue/dj-wasabi-dot-local* url. If you remember, the dj-wasabi-dot-local is the name of the role, so this role has the correct ttl etc.

Lets execute it and once the certificate is created, a lot of output is returned in json format:

```
curl -XPOST -k -H 'X-Vault-Token: <_my_root_token_>' \
-d '{"common_name": "blog.dj-wasabi.local"}' \
https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/issue/dj-wasabi-dot-local
{"request_id":"e1d0f686-d0d8-d1d8-d7ab-428c7322229b","lease_id":"","renewable":false,"lease_duration":0,"data":{"ca_chain":["-----BEGIN CERTIFICATE-----asas-----END CERTIFICA
```

Again I removed a lot of unreadable data from the example. Again you'll see the *private_key*, *certificate* and the *ca_chain* which can be used with a service like nginx.

Lets do an overview of all certificates stored in our Vault:

```
curl -XGET -H 'X-Vault-Token: <_my_root_token_>' \
--request LIST https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/certs
{"request_id":"fb5e7060-0d02-211a-ae25-50507a334706","lease_id":"","renewable":false,"lease_duration":0,"data":{"keys":["03-f2-bb-f5-27-16-81-20-76-0d-91-6f-fd-10-05-2d-a6-e1
```

We see that there are 2 certificates stored in the Vault, the "keys" has 2 values. These keys are the *Serial Numbers* of the certificates. We have to use this *Serial Number* if we want to revoke it or we just want to get the certificate. An example of getting the certificate:

```
curl -XGET -H 'X-Vault-Token: df80e726-d3f0-8344-3782-fec19fe7a745' \
https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/cert/11-42-ba-66-94-b4-c9-5c-e5-1a-77-da-76-2e-57-5d-b5-64-f5-c3
{"request_id":"ae6e63f9-c04e-ac4c-d8a8-254347284771","lease_id":"","renewable":false,"lease_duration":0,"data":{"certificate":"-----BEGIN CERTIFICATE-----asasas-----END CERTI
```

Again I removed some data from the example. You can only get the certificate, not the private key. I've copied the contents of the certificate in a file called *blog.dj-wasabi.local.crt* on my Mac, so when I run the *openssl x509* command, it will show some information about this certificate:

```
openssl x509 -in blog.dj-wasabi.local.crt -noout -text
```

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            11:42:ba:66:94:b4:c9:5c:e5:1a:77:da:76:2e:57:5d:b5:64:f5:c3
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=NL, ST=Utrecht, O=dj-wasabi, OU=Vault CA, CN=dj-wasabi.local/emailAddress=ikben@werner-dijkerman.nl
        Validity
            Not Before: Aug 23 16:51:36 2017 GMT
            Not After : Aug 26 16:52:05 2017 GMT
        Subject: CN=blog.dj-wasabi.local
 ...
            Authority Information Access:
                OCSP - URI:https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/ocsp
                CA Issuers - URI:https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/ca

            X509v3 Subject Alternative Name:
                DNS:blog.dj-wasabi.local
            X509v3 CRL Distribution Points:

                Full Name:
                  URI:https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/crl
 ...
```

The output shows that the certificate is only valid (Validity) for 3 days (72 hours). If you take a look at the "Authority Information Access", you'll see the urls (*OCSP* and the *CA Issuers*) we have set earlier. And a little bit further we see the *CRL Distribution Points*, an url we also have set with the set urls command.

Keep in mind: Only during the generation of the certificate, the private key is returned. If you did loose the private key, then revoke the certificate and generate a new one.

As last command in this blogpost we do a revoke of an certificate. We have to do an *POST* and sent the *serial_number* to the revoke endpoint.

```
curl -XPOST -k -H 'X-Vault-Token: <_my_root_token_>' \
-d '{"serial_number":"03-f2-bb-f5-27-16-81-20-76-0d-91-6f-fd-10-05-2d-a6-e1-59-e3"}' \
https://vault.service.dj-wasabi.local:8200/v1/dj-wasabi/revoke
{"request_id":"ea8a7132-231f-7075-f42b-f81b272cc9cd","lease_id":"","renewable":false,"lease_duration":0,"data":{"revocation_time":1503506236,"revocation_time_rfc3339":"2017-0
```

It returns a json output with a key named *revocation_time*. This is the time since epoch when the certificate is revoked, 0 if the certificate isn't revoked.

So, that was it! Have fun!