Setting up CFSSL

November 13, 2017 @ 21:20:00 / Amos

I have smartOS server at home with few (dozen) virtual machines, and was thinking that centralized PKI server would be a good thing, so I tried to create my own solution in go, and it worked, in a basic mode, but anything more would be a lot of work... So, my attention was moved to CloudFare SSL toolkit (https://blog.cloudflare.com/introducing-cfssl/) which, at first, looked complicated and too big for my needs... But, in the end I gave it a try.

My use case scenario is that I need multiple intermediate CA certificates, each of which can create multiple server certificates, which in return can create user certificates... So user certificate will be valid only on server that was generated by, and no other... Also, if server certificate is compromised, when I revoke it, all it's client certificates are automagically invalid!

First you need to install **cfssl**, and since it's written in Go, you need working installation of Go 1.6 or greater (just follow their instructions (https://github.com/cloudflare/cfssl#installation)). Also, what it doesn't say is that you also need a gcc compiler (I was bitten by this because I have created basic smartOS instance which didn't have development tools...).

After you installed it, and it's working, begins fun part of using it...

Usage of **cfssl** is based around config and certificate data in JSON file.

Sample *rootCA.csr.json*

```
{
  "CN": "Root CA",
  "key": {
    "algo": "ecdsa",
    "size": 256
  },
  "ca": {
    "expiry": "262800h",
    "pathlen": 2
  },
  "names": [
    {
      "C": "US",
      "L": "San Francisco",
      "OU": "Dropsonde Certificate Authority",
      "ST": "California"
    }
  ]
}
```

`CN` field is CommonName
`key` field is definition for private key. Algorithm and key size if specified here
`ca` field is for attributes specific for certificate that will be CA (certificate authority (here is pathlen set to 2 because I wan't to use intermediate CA))
`names` field contains *Country, Location, OrganizationalUnit* and *State* fields.

And now generate rootCA with:

```
mkdir rootCA
cfssl genkey -initca rootCA.csr.json |cfssljson -bare rootCA/rootCA
```

and you can inspect it with

```
openssl x509 -in rootCA/rootCA.pem -text -noout
```

So, now you have root CA which can issue intermediate CA which in turn can issue host/server certificates that can sign user/client certificates...

For intermediate CA, and every other certificates, we need *config.json*...

Sample *config.json*

```
{
    "signing": {
        "default": {
            "expiry": "43800h"
        },
        "profiles": {
          "intermediate": {
              "expiry": "43800h",
              "usages": [
                  "signing",
                  "key encipherment",
                  "cert sign",
                  "crl sign"
              ],
              "ca_constraint": {
                  "is_ca": true,
                  "max_path_len":1
              }
          },
          "server": {
                "expiry": "43800h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "server auth",
                    "cert sign",
                    "crl sign"
                ]
            },
            "client": {
                "expiry": "43800h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "client auth",
                    "email protection"
                ]
            }
        }
    }
}
```

Fields are pretty much self-explanatory, and profiles are used for different options and usages for signing different types of certificates...

Sample *intermediateCA.csr.json*

```
{
  "CN": "Intermediate CA",
  "hosts": [
    ""
  ],
  "key": {
    "algo": "ecdsa",
    "size": 256
  },
  "names": [
    {
      "C": "US",
      "L": "San Francisco",
      "OU": "Dropsonde Certificate Authority",
      "ST": "California"
    }
  ]
}
```

It should be almost same as *rootCA.csr.json*, but `CN` field **must** be different. Also, there is no `ca` field here.

And now we can generate intermediate CA:

```
mkdir intermediateCA
cfssl gencert -ca=rootCA/rootCA.pem -ca-key=rootCA/rootCA-key.pem -config=config.json -profile=intermediate intermediateCA.csr.json |cfssljson -bare i
ntermediateCA/intermediateCA
```

and you can inspect it with

```
openssl x509 -in intermediateCA/intermediateCA.pem -text -noout
```

You should notice that `pathlen` field has decreased, and now it's 1.

So, now you should remove *rootCA* private key, and keep it in a safe, offline place, since we don't use it any more (that is, we use it only to sign/revoke intermediate certificates). In fact, by *best practices*, it should be created in offline, non networked machine, and kept locked...

Next in line is server/host certificate.

Sample *server.csr.json*

```
{
    "CN": "example.net",
    "hosts": [
        "example.net",
        "www.example.net"
    ],
    "key": {
        "algo": "ecdsa",
        "size": 256
    },
    "names": [
        {
            "C": "US",
            "ST": "CA",
            "L": "San Francisco"
        }
    ]
}
```

For server certificate, there are few key fiels.
`CN` is primary server address/hostname, and in `hosts` field you need to specify alternate hostnames.

Server certificate is signed with intermediateCA and *not* rootCA!

Command to generate server certificate is almost the same as one for intermediate certificate, but we select different singing profile from *config.json*

```
mkdir server
cfssl gencert -ca=intermediateCA/intermediateCA.pem -ca-key=intermediateCA/intermediateCA-key.pem -config=config.json -profile=server server.csr.json
 |cfssljson -bare server/server
```

once again, you can inspect it with

```
openssl x509 -in server/server.pem -text -noout
```

If you now try to verify certificate chain of server certificate with

```
openssl verify -CAfile intermediateCA/intermediateCA.pem server/server.pem
```

it will fail!

And that is to be expected, because we don't have all certificates up to rootCA.

So, if you create chain

```
cat rootCA/rootCA.pem intermediateCA/intermediateCA.pem > chainCA.pem
```

and run

```
openssl verify -CAfile chainCA.pem server/server.pem
```

you will get result **OK**, since in chain.pem we have intermediateCA *and* rootCA.
We will need to create similar file for client certificate...

All that is left is to create user certificate.

Sample *user1.csr.json*

```
{
    "CN": "user1",
    "hosts": [
        ""
    ],
    "key": {
        "algo": "ecdsa",
        "size": 256
    },
    "names": [
        {
            "C": "US",
            "ST": "CA",
            "L": "San Francisco"
        }
    ]
}
```

`hosts` field is not needed, and `CN` is field in which you most likely wan't to put user's username.

This certificate is signed with servers certificate on which we want to use it.

```
mkdir user1

cfssl gencert -ca=server/server.pem -ca-key=server/server-key.pem -config=config.json -profile=client user1.csr.json |cfssljson -bare user1/user1
```

And we are using last profile defined in *config.json*, named *client*.

Inspect certificate

```
openssl x509 -in user1/user1.pem -text -noout
```

And to verify it

```
cat rootCA/rootCA.pem intermediateCA/intermediateCA.pem server/server.pem > chain.pem
openssl verify -CAfile chain.pem server/server.pem
```

And you should get OK result.

That's it! Next will be setting up cfssl API server for creating and signing certificates...