The CAW driver

The CAW driver is our native certificate authority driver. In essence, it is a shell script encompassing ordinary OpenSSL certificate authority. This has many pros:

- If you can do it in openssl.cnf, you can do it in CAW too.
- Supported on any Linux/UNIX platform which has openssl, bash and coreutils. Also supported on MinGW.
- Readable script that is easy to debug and fix. Anyone can do it.
- You can migrate almost any existing CA into the CAW.
  - Migrate certificates, keys(optional) and CSRs. Each of those files have to be named by serial number of the corresponding certificate.
  - Construct openssl certificate db (plain text file of given format). If you have the data, this can be done with a few hours of scripting.
- Native support for PKCS#11 crypto tokens.
- Also, we give out almost totally preconfigured CAW instance and installation instructions.
- Also, the deployment is simply unpacking CAW into a folder. Need another CA instance? Unpack CAW into another folder.

It also has some cons:

- There are plenty of bad openssl.cnf tutorials on the internet. Seriously. We are fighting it by heavily commenting the CAW-supplied configuration but that is basically all we can do about it. 😕
- OpenSSL integration with PKCS#11 tokens is not something that "just works". It may be version-, token- or distribution-dependent. But if you make it work with plain openssl, you can easily integrate it into CAW.
- The CA is not concurrent. The CAW handles it by pessimistic locking.

The CAW script

CAW is a shell wrapper above the OpenSSL-based certificate authority (abbreviated: OSSL CA). It allows you to use the OSSL CA in a similar way the EasyRSA does. CAW is primarily created as a CA backend for the CzechIdM Certificate Authority module but it is possible to extend/incorporate it somewhere else. It also provides an user-friendly CA implementation which can be used right away from the command line.

For the list of capabilities and input/output formats, please refer to the CAW shell script. Simply run CAW to get the usage screen where you can find everything you will ever need. :)

```
./caw
Unknown command '' specified.
Usage: ./caw command [--param1 value1 --param2 value2 ...]
...
COMMAND create-key-and-cert - generates new private key, CSR and signs a
certificate
OUTPUT
                Success: Serial number of the issued certificate written
```

```
onto STDOUT. Return code 0.
                Error: Error message on STDERR. Return code 1.
PARAMETERS
--country        countryName. Mandatory.
--state          stateOrProvinceName. Mandatory.
--locality       localityName. Mandatory.
--org            organizationName. Mandatory.
--ou             organizationalUnitName. Mandatory.
--cn             commonName. Mandatory.
--mail           emailAddress. Mandatory.
--pass           private key passphrase. Mandatory.
... and so on ...
```

In its core, CAW uses a well-known OSSL CA all with its **openssl.cnf** file and such. Therefore every configuration which can be specified in **openssl.cnf** can be made available in the CAW. CAW makes use of **openssl.cnf** as often as possible (i.e. with defaults for the **openssl req** command) and very often invokes openssl using **-batch** argument.

But beware, CAW has also its own configuration file **caw_settings.source**. This file contains some options that need to be in sync with options in **openssl.cnf**. So if you are fiddling with **openssl.cnf**, always also check **caw_settings.source**.

Additional information can be found in one of those three places:

- In the CAW usage page. Simply invoke **./caw**.
- As a comment in the **caw_settings.source** file.
- As a comment in the CAW script itself (i. e. **authors**, **changelog**, **TODOs**).

Core functions

- Self-contained CA
  - CAW does not depend on the global **/etc/openssl.cnf**, it brings its own **openssl.cnf** along. That means, your CA is completely separated from others.
  - You can run different CAs just by giving each its own folder.
  - Installation is merely unpacking a tarball and generating CA certificate and starting serial number.
- Handling concurrency problems
  - OpenSSL CA (**openssl ca …**) must not be invoked in multiple instances at the same time. CAW uses lockfiles to prevent that.
- Private key and CSR private storage
  - CAW archives all files it has created, including users' private keys and CSRs. Private keys all always AES-encrypted by a password which the end-user specifies. Therefore even the CA does not have an access to the private key.
- Support for hardware tokens using PKCS11
  - If your OpenSSL version can support your hardware token, you can use it in CAW. Only thing you need to do is to configure it in **openssl.cnf** and **caw_settings.source** (and there is already a template for that).
- Unix-like style of invocation
  - Everything that goes into the CAW is a **command line argument**. Everything that goes out of the CAW is either a output of successful operation (on **STDOUT**) or an error (on

    STDERR).
  - Return code for successful operation is **0**, for error it is **1**.
  - All information going to/from CAW is in **printable form**, large data mainly as PEM or base64-encoded.
- Usable as a root or intermediary CA
  - CAW allows additional certificates to be added to the chain when downloading a particular cetificate. This can be used to supply all parts of the certificate chain from the issuing CA to the root CA. If no such additional certificates are specified, CAW acts as a root CA.
- Private key and certificate creation
  - When user supplies just the **SubjectDN components**, the private key, CSR and certificate are automatically generated. You can validate the SubjectDN components by regex-based engine.
- CSR signing
  - User-provided CSR is checked for sufficient **signature algorithm** and for **SubjectDN components**. CAW makes sure the comparison is text-based (by regexes) and does not care about datatypes. This effectively solves usability problems with OpenSSL's **policy_match** in heterogenous environments.
- Certificate prolongation
  - Because CAW stores CSRs, it is possible to prolong certificate by reissuing it with new validity period. All that is needed is certificate's serial number.
  - CAW will not allow the expired/revoked certificate to be reissued.
  - This is not a big security risk because for access the CSRs, an attacker must have access to the machine running the CA. If he does, you have much bigger problem than some accessible CSR.
- Certificate revocation
  - Revoke certificate just by specifying its serial number and revocation reason.
- Certificate database querying
  - CAW provides basic interface to query certificate database and for certificate validation.
- Certificate bundling
  - When the issued certificate is requested for download, user can specify if he wants it to be bundled with private key and/or whole certificate chain.
  - When requesting the bundle with private key, user must specify the password he has given during the certificate creation.
- CRL issuing and publishing
  - CAW enables you to create CRLs just by calling **./caw create-crl**. Another user can then publish the CRL into public destination by calling **./caw publish-crl**.
- Housekeeping tasks
  - CAW takes care of orphan files and similar things that can happen during the life of CA. All those tasks are done by **./caw housekeep**.

Installation

1. Create separate user for your authority. **Ensure that no other user can read the home directory.** This happens for example on the (open)SuSE where each home is granted to the *users* group which encompasses all users.

   ```
   [root@ca ~]# useradd -r -m -s /bin/bash authority1
   ```

2. Move the CAW directory to the user's home.

   ```
   [root@ca ~]# mv caw.tgz /home/authority1/
   ```

```
[root@ca authority1]# tar xzf caw.tgz
[root@ca authority1]# ll
total 28
drwxr-xr-x 4 1000 users  4096 Aug 24 14:36 caw
-rw-r--r-- 1 root root  24563 Aug 24 15:16 caw.tgz
[root@ca authority1]# chown -Rf authority1:authority1 caw/
[root@ca authority1]# chmod 750 caw
```

3. Ensure that **caw** script is runnable.

```
[root@ca authority1]# cd caw/
[root@ca caw]# chmod 750 caw
```

4. Create new starting serial number. This number can be, say, *01* but there is a caveat attached to this - OpenSSL then works with 8bit serial mode (**this is potentially dangerous**). Better way is to create truly random 128bit serial number as the example shows.

```
[root@ca caw]# cd ca/
[root@ca ca]# openssl rand -hex 16 > serial
```

5. If you want to use serial number prefixes, now it is the time to set it up. **If you don't know what it is, you can safely skip this step.** Suppose the random serial was *b1676557ad077ef7144c227d16a55025*. Then we simply edit the starting serial to have our desired prefix (say, *aaaccc000*). Total length of the serial number must remain 128b, so our new serial will be *aaaccc000d077ef7144c227d16a55025*. Write it into the **serial** file. We can, possibly, overflow to the prefix *aaaccc001* so be aware of it - our "prefix" is not a real prefix. It is merely a cleverly chosen starting number.
6. Generate your CA certificate the usual way. For how to set it up with PKCS11 crypto token, see the end of this document. **Select the appropriate private key size.** Also, there are x509v3 certificate extensions which are handy to have in the authority's certificate. Default are in **ca_crt.extensions** file. Edit them (and command line parameters in the example below) according to your needs.

```
[root@ca ca]# su - authority1
[authority1@ca ~]$ cd caw/ca/
[authority1@ca ca]$ pwd
/home/authority1/caw/ca
[authority1@ca ca]$ openssl genrsa -out private/ca.key 2048
[authority1@ca ca]$ chmod 400 private/ca.key
[authority1@ca ca]$ openssl req -new -in private/ca.key -out ca.csr -
key private/ca.key
[authority1@ca ca]$ openssl x509 -req -in ca.csr -signkey
private/ca.key -days 1000 -out ca.crt -sha256 -extfile
../ca_crt.extensions
[authority1@ca ca]$ rm ca.csr
```

7. CAW folder comes with a number of empty directories. Although needless now, they are automatically used by the **caw** script for managing the authority. Do not delete them.
8. Check the **ca_openssl.cnf** configuration file and configure your authority. This file is an ordinary openssl.cnf, but is realized in local variant. This enables multiple caw-based CAs to coexist one along the other just by separating their directories. The ca_openssl.cnf contains preconfigured CA so only small adjustments should be necessary. Follow the comments in the

file itself. The most important things to set up are:
   1. Enable (configure) the PKCS11 engine or disable it entirely (comment it out).
   2. Configure **default_days** and **default_crl_days** and other certificate-related settings.
   3. Configure parameters in the **req** stanza - those are used for generating new keys and requests.
   4. Configure certificate extensions in the **issued_cert_ext** stanza. Do not forget to set up the **crlDistributionPoints** correctly.

9. Check the **caw_settings.source** configuration file and configure it accordingly. The tricky part there is that some settings have to have the same values as in **ca_openssl.cnf**. Again, the comments in the file will help you. The most important things to set up are:
   1. Configure **CA_OSSL_ENGINE_PARAM** if you want to use PKCS11 token. Set it to an empty string if you store the CA's private key in the file.
   2. Configure the **CA_OSSL_ROOT_CHAIN** if your CAW authority is not the root authority.
   3. Configure the **SubjectDN** and **CSR** validation prefixes. This also lets you set a basic policy for user's passphrase complexity. Also, set allowed signature algorithms.
10. Generate the empty CRL file.

    ```
    [authority1@ca caw]$ ./caw create-crl
    ```

11. You should be good to go. Follow the examples and try to obtain your first certificate.

Examples

1. Create private key and certificate

   ```
   [root@ca ~]# ./caw create-key-and-cert --country CZ --state "Czech
   Republic" --locality Prague --org BCV --ou TEST --cn user.test.bcv --
   pass demodemo
   0C0774BACDF2CA2A52BEEF68A0F1D411
   ```

2. Prolong certificate

   ```
   [root@ca ~]# ./caw prolong-cert --serial
   0C0774BACDF2CA2A52BEEF68A0F1D411
   0C0774BACDF2CA2A52BEEF68A0F1D412
   ```

3. Download (private key,certificate,certificate chain) bundle from the CA

   ```
   [root@ca ~]# ./caw get-cert --serial 0C0774BACDF2CA2A52BEEF68A0F1D411 -
   -with-pkey --pass demodemo --with-chain
   MIIKoQIBAzCCCmcGCSqGSIb3DQEHAaCCClgEggpUMIIKUDCCBQcGCSqGSIb3DQEHBqCCBPg
   wggT0
   ...
   FbAM6nS5jJYQ4s4VKDElMCMGCSqGSIb3DQEJFTEWBBRGj5/LUBZtcz/k+N96L7RzdleanDA
   xMCEw
   CQYFKw4DAhoFAAQUCqImx0Un2qmtSACpEWD4i2ivunMECFJnEuzDIEtHAgIIAA==
   ```

4. Revoke a certificate

   ```
   [root@ca ~]# ./caw revoke-cert --serial
   0C0774BACDF2CA2A52BEEF68A0F1D411 --reason keyCompromise
   ```

5. Refresh the CRL

```
[root@ca ~]# ./caw create-crl
```