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 `ca\_openssl.cnf`. So if you are fiddling with `ca\_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 `ca\_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 lockdir 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 `ca\_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. **If you do not use PKCS11 crypto token (that is, you want to use `ca.key`)**, deconfigure PKCS11 engine template according to this howto.
11. Generate the empty CRL file.

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

12. 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
```

Example of CAW driver configuration in Appliance

1. Create a new folder. It is possible to add more authorities to this folder. Each authority has its own caw folder, caw script and own configuration caw_settings.source and ca_openssl.cnf.In this example, use only one CA.

```
[root@ca ~]# mkdir /data/volumes/czechidm/cert-authority
```

2. Unzip caw driver to caw directory. Caw driver can be downloaded from our git repository:CAW.

```
[root@ca ~]# unzip caw-master.zip -d /data/volumes/czechidm/cert-
authority/
[root@ca ~]# mv /data/volumes/czechidm/cert-authority/caw-master
/data/volumes/czechidm/cert-authority/caw
```

3. In this example we received from customer already generated CA certificate - private key public key customerCa.key, public key customerCa.pem and configuration file customerCa.conf.Copy a files to caw directory and replace our default test ca.

```
[root@ca ~]# cp customerCa.key /data/volumes/czechidm/cert-
authority/caw/ca/private/ca.key
[root@ca ~]# cp customerCa.pem /data/volumes/czechidm/cert-
authority/caw/ca/ca.pem
```

4. We use random random 128bit serial number.

```
[root@ca ~]# cd /data/volumes/czechidm/cert-authority/caw/ca/
[root@ca ca]# openssl rand -hex 16 > serial
```

5. It is also necessary to merge customerCa.conf file with the caw configuration file ca_openssl.cnf and caw_settings.source. The ca_openssl.cnf and caw_settings.source contains preconfigured CA. Follow the comments in the files and edit files by customerCa.conf.
6. Set a correct permision and owner. .

```
[root@ca czechidm]# chown -Rf 999:998 cert-authority/
[root@ca czechidm]# chmod 400 cert-authority/caw/ca/private/ca.key
[root@ca czechidm]# chmod 750 cert-authority/
[root@ca czechidm]# cd cert-authority/caw/
[root@ca caw]# chmod 750 caw
```

7. It is necessary to directory cert-authority mount into a CzechIdM container, because the caw script must be executable by IdM. To file /data/registry/node-active-config/docker-compose-czechidm.yml add:

```
 - type: bind
        source: /data/volumes/czechidm/cert-authority
        target: /opt/cert-authority
        read_only: false
```

8. An important part of ca is the CRL file, which must be generated regularly. CAW creates CRL by calling ./caw create-crl. Create the .service unit that will generate CRL file. Create new file in /usr/lib/systemd/system/iam-crl-refresh.service

```
[Unit]
Description=CRL refreshing
After=network.target docker.service
[Service]
Type=simple
ExecStart=/usr/bin/docker exec czechidm /opt/cert-authority/caw/caw
create-crl
```

9. Create a .timer unit file which actually schedules the .service unit you just created. Create it in the same location as the .service file. The service is started every hour.

```
[Unit]
Description=CzechIdM refresh CRL
After=network.target docker.service
[Timer]
OnCalendar=*-*-* *:00:00
[Install]
WantedBy=multi-user.target
```

10. Enable new service and timer:

```
[root@ca czechidm]# systemctl enable iam-crl-refresh.service iam-crl-
refresh.timer
```

11. The crl has to be available via a web proxy. First, you must mount the file in the Web Proxy container. to file /data/registry/node-active-config/docker-compose-web-proxy.yml add:

```
 - type: bind
        source: /data/volumes/czechidm/cert-authority/caw/ca/crl/ca.crl
        target: /etc/nginx/crl/ca.crl
        read_only: true
```

12. To make the file available from web proxy it is necessary to modify a file /data/volumes/web-proxy/config/reverse_proxy.conf and add:

```
-   location /crl/ca.crl {
        root /etc/nginx/;
    }
```

    Then the crl file is available at URL:https://iam.appliance.tld/crl/ca.crl

13. The next step is to configure our crl module ca directly in IdM. Instructions for configuration in IdM are here: crt module tutorial

Deconfiguring default PKCS11 engine

By default, CAW comes with a preconfigured template for the PKCS11-enabled crypto engine. In some deployments, this is not necessary and the goal is to have CA's private key and certificate stored on the machine's filesystem. This short howto will show you how to swap preconfigured PKCS11 engine for the more usual setup.

The `ca\_openssl.cnf` file has several sections, referred as "stanzas". Stanza starts with `[stanza name]` and ends when another stanza starts.

To deconfigure the PKCS11 template:

1. Edit the `ca\_openssl.cnf` file.
    1. Comment out `openssl\_conf = openssl\_init` at the top of the file.
    2. Comment out whole `[openssl\_init]` stanza.
    3. Comment out whole `[engine\_section]` stanza.
    4. Comment out whole `[pkcs11\_engine]` stanza.
    5. In the `[CA\_default]` stanza, uncomment the `private\_key = …` line. Set it to a path to your private key or create private key in the `$dir/private/ca.key` path. The key has to have at most `600` (`-rw\-\-\-\-\-\-`) privileges.
2. Edit the `caw\_settings.source` file.
    1. Set the `CA\_OSSL\_ENGINE\_PARAM="…"` to `CA\_OSSL\_ENGINE\_PARAM=""` (empty variable).

Configuring CAW with the PKCS11 crypto token

DISCLAIMER: This howto is flawed in terms of overall security. Therein, we describe how to generate private key and then store it in the PKCS11 token. The token we use is SoftHSM, which is not a real physical token. To make it truly secure, you have to (at least):

- Use real hardware crypto token.
- Generate CA private key directly on the token (via openssl, p11-tool or such).
- The private key pin (and security pin) really should not be 1234 (or 123456 respectively). Use reasonably strong pins.

If you modify the example below according to those remarks, then your setup should be secure enough.

1. First, we generate the CA private key. This is not secure, but it is quicker for demonstration purposes. For more info, see the disclaimer.

```
[root@ca ~]# openssl genrsa -out ca.key 4096
```

2. Create the CA CSR.

```
[root@ca ~]# openssl req -new -key ca.key -out ca.csr -sha512
```

3. Create the CA certificate with appropriate extensions.

```
[root@ca ~]# openssl x509 -req -days 3650 -in ca.csr -signkey ca.key -
out ca.crt -sha512 -extfile ca_extensions.txt
```

4. SoftHSM needs the key to be in PKCS8 format. The default generated from OpenSSL is PKCS5 so we have to convert it. Also, using `-nocrypt` option is a bad idea. But, again, we should be generating private key directly on the token.

```
[root@ca ~]# openssl pkcs8 -in ca.key -topk8 -out ca_pk8.key -nocrypt
```

5. Suppose the SoftHSM is installed and configured. The SoftHSM's default data dir is in /var so you have to run it as root or reconfigure the data directory location - see the next section on how to configure SoftHSM to run under non-root user. Anyway, we first init the token and then import the CA private key.

```
[root@ca ~]# softhsm2-util --init-token --slot 0 --label ca --so-pin
123456 --pin 1234
The token has been initialized.
[root@ca ~]# softhsm2-util --pin 1234 --so-pin 123456 --import
ca_pk8.key --label ca --id A1B2 --no-public-key --slot 0
The key pair has been imported.
```

6. We have to link our token with openssl and test the setup. If everything goes right, we get the output similar to this one:

```
[root@ca ~]# openssl engine -t dynamic -pre
SO_PATH:/usr/lib64/openssl/engines/pkcs11.so -pre ID:pkcs11 -pre
LIST_ADD:1 -pre LOAD -pre MODULE_PATH:/usr/lib64/pkcs11/libsofthsm2.so
-pre VERBOSE -pre PIN:1234
(dynamic) Dynamic engine loading support
Success: SO_PATH:/usr/lib64/openssl/engines/pkcs11.so
Success: ID:pkcs11
Success: LIST_ADD:1
Success: LOAD
Success: MODULE_PATH:/usr/lib64/pkcs11/libsofthsm2.so
Success: PIN:1234
Loaded: (pkcs11) pkcs11 engine
  [ available ]
```

7. Now create the user certificate request the usual way and try to sign it with the CA. We have stored ours in user.csr. We will also need an identifier of the PKSC11 token, which we can get by invoking `softhsm2-util –show-slots`. Certificate signing should look basically like this (we used openssl shell in this snippet):

```
[root@ca usercert]# openssl
OpenSSL> engine -t dynamic -pre
SO_PATH:/usr/lib64/openssl/engines/pkcs11.so -pre ID:pkcs11 -pre
LIST_ADD:1 -pre LOAD -pre MODULE_PATH:/usr/lib64/pkcs11/libsofthsm2.so
-pre VERBOSE -pre PIN:1234
(dynamic) Dynamic engine loading support
Success: SO_PATH:/usr/lib64/openssl/engines/pkcs11.so
```

```
Success: ID:pkcs11
Success: LIST_ADD:1
Success: LOAD
Success: MODULE_PATH:/usr/lib64/pkcs11/libsofthsm2.so
Success: VERBOSE
Success: PIN:1234
Loaded: (pkcs11) pkcs11 engine
PKCS#11: Initializing the engine
Found 2 slots
  [ available ]
OpenSSL> x509 -req -engine pkcs11 -CAkeyform engine -CAkey slot_0-
label_ca -CA ca.crt -days 1000 -set_serial 15 -sha256 -in user.csr -out
user.crt
PKCS#11: Initializing the engine
Found 2 slots
engine "pkcs11" set.
Signature ok
subject=/C=CZ/ST=Czech Republic/L=Prague/O=BCV/OU=TEST/CN=user.test.bcv
Getting CA Private Key
Loading private key "slot_0-label_ca"
Looking in slot 0 for key: label=ca
[0] SoftHSM slot 0             login             (ca)
[1] SoftHSM slot 1             uninitialized, login  (no label)
Found slot:  SoftHSM slot 0
Found token: ca
Found 0 certificate:
No private keys found.
Loading private key "slot_0-label_ca"
Looking in slot 0 for key: label=ca
[0] SoftHSM slot 0             login             (ca)
[1] SoftHSM slot 1             uninitialized, login  (no label)
Found slot:  SoftHSM slot 0
Found token: ca
Found 0 certificate:
Found 1 private key:
1 P  id=ffffffa2ffffffb2 label=ca
^C
```

8. Then we can verify the signing was correct:

```
[root@ca usercert]# openssl verify -CAfile ca.crt user.crt
user.crt: OK
```

9. This tells us that openssl integration with the PKCS11 engine is working. To configure it in CAW, edit the `[engine]` section in the `ca_openssl.cnf` and supply the values you just used on the command line.
10. The last step is to configure `CA\_OSSL\_ENGINE\_PARAM` variable in the `caw\_settings.source` file. There is already template for that, the only value that needs changing should be an identifier of the PKCS11 slot.

Using SoftHSM under non-root user

SoftHSM is not a real crypto token. It is very handy during development but not meant for high-security production use.

We are installing SoftHSM on CentOS 7 operating system. Then, we will configure it to run under authority1 OS user.

1. Add the EPEL repository and install necessary packages.

```
[root@ca ~]# yum install epel-release
[root@ca ~]# yum install softhsm opensc engine_pkcs11
```

2. Check if there is a system-wide configuration file.

```
[root@ca ~]# cat /etc/softhsm2.conf
# SoftHSM v2 configuration file
directories.tokendir = /var/lib/softhsm/tokens/
objectstore.backend = file
# ERROR, WARNING, INFO, DEBUG
log.level = INFO
```

3. We have two options:
   - Change the system-wide configuration to point to a directory the `authority1` user can access.
   - Create local configuration file. This is better in cases where we want to have separate SoftHSM token storages.
4. We go for the second option. First, we change to the authority1 user and create our directory structure and config file.

```
[root@ca ~]# su - authority1
[authority1@ca ~]$ pwd
/home/authority1
[authority1@ca ~]$ mkdir -pv softhsm/tokens
mkdir: created directory ‘softhsm’
mkdir: created directory ‘softhsm/tokens’
[authority1@ca ~]$ chmod 750 softhsm
[authority1@ca ~]$ vim softhsm/softhsm2.conf
[authority1@ca ~]$ cat softhsm/softhsm2.conf
directories.tokendir = /home/authority1/softhsm/tokens/
objectstore.backend = file
# ERROR, WARNING, INFO, DEBUG
log.level = INFO
```

5. SoftHSM is managed with the `softhsm2-util` program. This program reads environment variable SOFTHSM2_CONF. When the variable is not defined, it uses configuration `/etc/softhsm2.conf`. We will export the variable with proper path to config file. **If you modified the system-wide config, you can skip this step.**

```
[authority1@ca ~]$ export
```

```
SOFTHSM2_CONF=/home/authority1/softhsm/softhsm2.conf
```

6. Now we simply call the `softhsm2-util` and initialize a new token. **Please DO NOT use PINs like 1234 or 123456 in the real world scenarios.**

```
[authority1@ca ~]$ softhsm2-util --init-token --slot 0 --label
test_slot --pin 1234 --so-pin 123456
```

7. We can check, that new token object was created in the configured backend store.

```
[authority1@ca ~]$ find /home/authority1/softhsm/tokens/ -type f
/home/authority1/softhsm/tokens/ee48c4b2-
b28e-8c2d-7921-59236dda0866/token.lock
/home/authority1/softhsm/tokens/ee48c4b2-
b28e-8c2d-7921-59236dda0866/generation
/home/authority1/softhsm/tokens/ee48c4b2-
b28e-8c2d-7921-59236dda0866/token.object
```

8. SoftHSM is working! To use such configuration with CAW, edit the `caw\_settings.source` and append the `export SOFTHSM2_CONF…` line to it. This will make CAW use the configured store. **If you modified the system-wide config, you can skip this step.**

From:
https://wiki.czechidm.com/ - **CzechIdM Identity Manager**

Permanent link:
https://wiki.czechidm.com/tutorial/adm/caw_driver

Last update: **2022/04/12 08:39**