Equip Your CA With HSM For <50 Euros

With the recent breaches of Certificate Authorities (Comodo, Diginotar) I wanted to take a closer look at the security of my own Certificate Authority (CA). This CA is used for identification and authentication of servers, clients and users in my home network.

What you will learn…

- How to use smart cards to store private keys
- The use of OpenSSL with a different crypto provider

What you should know…

- Your way around a FreeBSD system
- The basics of a PKI
- Why you need a Certificate Authority

The keys for signing certificates are typically stored in text files on the file system. Access to the disk will reveal the keys. An attacker can make a copy and start issuing certificates outside my control. The primary design requirement is therefore that private keys will never be accessible in plain text.

Hardware Security Module

Enter HSM, hardware designed to keep the private key private. They come in various forms, from an appliance / PCI card to a USB token or smart card. PCI cards are typically encased in metal with features like automatic key deletion upon physical tampering. They are basically a dedicated computer running HSM software and using the PCI bus for interfacing with the host computer. Appliances are typically these PCI cards in an enclosure providing networking and rudimentary user control. HSMs can also have the form of a single chip. These do not have their own power supply (for deleting keys), but are still a small computer running HSM software. They can be embedded on smart cards, in USB tokens or integrated in other systems like the TPM chip on your motherboard.

Design Choices

For my HSM I decided to use smart cards, because they are cheap, readily available and easy to experiment with. The computer hardware is generic FreeBSD supported platform (i386 in my case, as I am working on putting it on an ALIX / NanoBSD installation). Attached is a Feitian SCR310 smart card reader (ftsafe-r310) and a Feitian PKI card (FTCOS / PK-01C). This reader is cheap, but any supported reader is fine (see *http://pcsclite.alioth.debian.org/ccid/section.html* for the complete list). This card was primarily chosen for its price and availability. It has lots of storage memory available (64k), but a limitation of 2048 bits for RSA keysize. Please do your own research on the types of readers and cards to fit your requirements.

For my production system, I will move to a Gemalto USB TR reader. It has an adapter so it can be mounted in a 3.5” floppy bay.

Figure 1. *Cards*

The software is a clean FreeBSD 8.2 with the following ports:

- `/usr/ports/devel/libccid` – interface for USB and serial smart card readers
- `/usr/ports/security/opensc` – tools for smart card management (in PC/SC mode)
- `/usr/ports/security/engine-pkcs11` – engine for PKCS11 support in OpenSSL
- `/usr/ports/security/openssl` – tools for certificate management (in this case)

Due to the many, many dependencies, it will take some time to install. All default settings for the ports are fine.

There is a number of different CA designs possible. From very flat (the root CA issues all certificates directly) to very hierarchical (various sub and sub-sub CA's issue certificates on behalf of the root CA). My design has one root CA with a sub CA per functional area. This is done for security reasons. First of all, the root CA only has to sign the sub CA's. The use and exposure of the root CA's private key is therefore very limited. Next to that, the server sub CA never issues client certificates, so the VPN concentrator has to trust only the clients sub CA when validating certificates. Client certificates issued by the (possibly hacked) server- or user sub CA are not accepted.

Testing The Setup

After the installation of the software, it is time to plugin the reader and test the setup. Insert the card in the reader and run:

```
# opensc-tool --list-readers

 # Detected readers (pcsc)
```

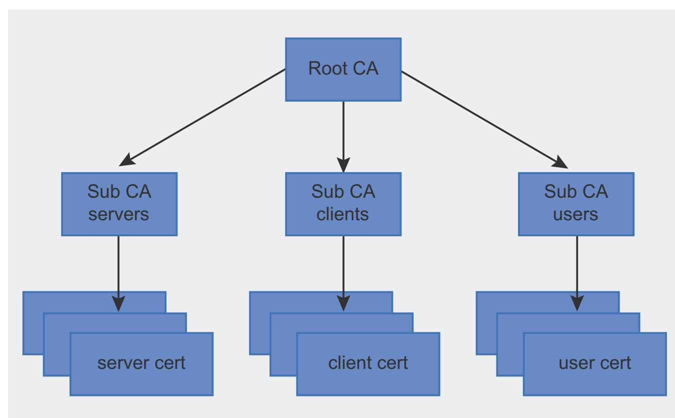

Figure 2. *CA-design*

```
Nr.  Card  Features  Name
0    Yes             Feitian SCR310 00 00
```

The reader is recognized by the driver. Let's initialize the card by formatting it with a PKCS15 structure. PKCS15 is a cryptographic token information format standard originally designed by RSA. ISO 7816-15 now manages the card-related parts of this standard.

```
# pkcs15-init --erase-card
# pkcs15-init --create-pkcs15 \
  --profile pkcs15+onepin \
  --auth-id 01 \
  --pin 0000 \
  --puk 123456 \
  --label „ewak.net PKI"
```

This specific card supports only one user-PIN, so the pkcs15+onepin profile is used. More advanced cards can offer more users and role separation.

The PIN for the user (auth-id 01) is set to 0000. If you want to reset the PIN, you will need the PUK, which is set to 123456. The label is just a name to identify the card.

The card now has a PKCS15 structure and is able to store private keys and their certificates.

Listing 1. *Formatting the card*

```
# pkcs15-init --verbose \
  --store-private-key ewak.net_Sub_CA_servers.p12 \
  --format pkcs12 \
  --auth-id 01 \
  --cert-label "ewak.net Sub CA servers"

  Using reader with a card: Feitian SCR310 00 00
  Connecting to card in reader Feitian SCR310 00 00...
  Using card driver entersafe.
  Found ewak.net PKI
  About to store private key.
  Importing 1 certificates:
    0: /C=NL
       /ST=GLD
       /L=T******e
       /O=ewak.net
       /OU=Certificate Services
       /CN=ewak.net Sub CA servers
       /emailAddress=certificate.services@ewak.net
  User PIN [User PIN] required.
  Please enter User PIN [User PIN]:
```

Listing 2. *Contents of the card*

```
# pkcs15-tool --dump

  Using reader with a card: Feitian SCR310 00 00
  PKCS#15 Card [ewak.net PKI]:
     Version         : 0
     Serial number   : 3021303609260511
     Manufacturer ID: EnterSafe
     Last update     : 20110925214004Z
     Flags           : EID compliant

  PIN [User PIN]
     Object Flags    : [0x3], private, modifiable
     ID              : 01
     Flags           : [0x32], local, initialized,
                      needs-padding
     Length          : min_len:4, max_len:16, stored_
                      len:16
     Pad char        : 0x00
     Reference       : 1
     Type            : ascii-numeric
     Path            : 3f005015

  Private RSA Key [Private Key]
     Object Flags    : [0x3], private, modifiable
     Usage           : [0xC], sign, signRecover
     Access Flags    : [0x0]
     ModLength       : 2048
     Key ref         : 1 (0x1)
Native          : yes
     Path            : 3f005015
     Auth ID         : 01
     ID              : 53f70c3ea5be9aef27d959c134d8ebe
                      f77322786
     GUID            : {53f70c3e-a5be-9aef-27d9-
                      59c134d8ebef}

  X.509 Certificate [ewak.net Sub CA servers]
     Object Flags    : [0x2], modifiable
     Authority       : no
     Path            : 3f0050153100
     ID              : 53f70c3ea5be9aef27d959c134d8ebe
                      f77322786
     GUID            : {53f70c3e-a5be-9aef-27d9-
                      59c134d8ebef}
     Encoded serial : 02 01 02
```

Importing Keys

It is possible to have the card generate the private and public keys, so the private key will never be available in plain text. This is the most secure option, but losing the card will also make the keys unrecoverable.

Because I already have a CA in place, I started with importing the existing keys. Generating the keys on the card will be discussed at the end of this article. Importing the key material (in PEM or PKCS12 format) is straightforward (Listing 1).

The user PIN for auth-id 01 is defined during the initialization of the card, `0000` in this article. It can also be

Listing 3. *Finding the card slot*

```
# pkcs11-tool --module /usr/local/lib/opensc-pkcs11.so
                    --list-slots

  Available slots:
  Slot 0 (0xffffffff): Virtual hotplug slot
    (empty)
  Slot 1 (0x1): Feitian SCR310 00 00
    token label:   ewak.net PKI (User PIN)
    token manuf:   EnterSafe
    token model:   PKCS#15
    token flags:   rng, login required, PIN
                   initialized, token initialized
    serial num  :  3021303609260511
```

Listing 4. *Loading the engine*

```
# openssl
OpenSSL> engine dynamic \
  -pre SO_PATH:/usr/local/lib/engines/engine_pkcs11.so
                   \
  -pre ID:pkcs11 \
  -pre LIST_ADD:1 \
  -pre LOAD \
  -pre MODULE_PATH:/usr/local/lib/opensc-pkcs11.so

  (dynamic) Dynamic engine loading support
  [Success]: SO_PATH:/usr/lib/engines/engine_pkcs11.so
  [Success]: ID:pkcs11
  [Success]: LIST_ADD:1
  [Success]: LOAD
  [Success]: MODULE_PATH:/usr/local/lib/opensc-
                   pkcs11.so
  Loaded: (pkcs11) pkcs11 engine
```

supplied to the pkcs15-init tool by adding the `--pin 0000` option. Now let's see what is on the card (Listing 2).

The import was successful. The info for the PIN, the private key and the certificate are displayed. Record the ID of the private key, as we will need it later to tell OpenSSL which signing key to use. A shorter command for retrieving this ID is `pkcs15-tool --list-keys`. More keys can be added to the card. My card has three keys for signing server, client and user certificates.

OpenSSL and PKCS11

OpenSSL can be instructed to use an external crypto provider for generating and storing key material using the PKCS11 API, also known as Cryptoki. When using OpenSSL, we can specify keys on the smart card instead of a keyfile on the disk. To do this, we need the card's slot and the key's ID. The ID we found using the `pkcs15-tool -list-keys` command. The slot can be found with the pkcs11-tool command (Listing 3).

Slot 1 is our card reader with the correct smart card (`ewak.net PKI`) inserted.

Now start an OpenSSL prompt and load the PKCS11 engine: Listing 4.

The engine is loaded and can be used by specifying it in OpenSSL commands with the following parameters:

```
-engine pkcs11 \
-keyform engine \
-key slot_1-key_53f70c3ea5be9aef27d959c134d8ebef77322786
```

For example:

```
OpenSSL> req -new -engine pkcs11 -keyform engine \
  -key slot_1-key_d893afddc82b28fb539e975b2a3e18efc2f3c474 \
  -out cert.csr -text
```

(This will create a certificate signing request from the public key with the id shown.) The use of PKCS11 can be set in the OpenSSL configuration file by adding the following lines to your `/path/to/openssl.cnf` configuration file (Listing 5). All OpenSSL commands can now be run using the PKCS11 engine by specifying:

Listing 5. *openssl.cnf*

```
openssl_conf = openssl_init
[openssl_init]
engines = engine_section
[engine_section]

[pkcs11_section]
engine_id = pkcs11
dynamic_path = /usr/local/lib/engines/engine_pkcs11.so
MODULE_PATH = /usr/local/lib/opensc-pkcs11.so
init = 1
```

Listing 6. *Signing the certificate*

```
# openssl ca -config /etc/ssl/openssl.cnf \
  -engine pkcs11 \
  -keyform engine \
  -keyfile 1:53f70c3ea5be9aef27d959c134d8ebef77322786 \
  -cert ewak.net_Sub_CA_servers.crt \
  -in cert.csr -out cert.pem

  Using configuration from /etc/ssl/openssl.cnf
  engine "pkcs11" set.
  PKCS#11 token PIN:
  Check that the request matches the signature
  Signature ok
  Certificate Details:
  -snip- OpenSSL output omitted -snip-
  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 Updated
```

Listing 7. *Contents of the card*

```
# pkcs15-tool --list-keys
Using reader with a card: Feitian SCR310 00 00
Private RSA Key [Private Key]
     Object Flags   : [0x3], private, modifiable
     Usage          : [0x4], sign
     Access Flags   : [0x1D], sensitive,
                      alwaysSensitive, neverExtract,
                      local
     ModLength      : 1024
     Key ref        : 1 (0x1)
     Native         : yes
     Path           : 3f005015
     Auth ID        : 01
     ID             : d893afddc82b28fb539e975b2a3e18e
                      fc2f3c474
     GUID           : {d893afdd-c82b-28fb-539e-
                      975b2a3e18ef}
```

```
-config /path/to/openssl.cnf -engine pkcs11 \
-keyform engine -key slot_<slot>-id_<id>
```

at the command line.

Signing Keys

The imported keys will be used for signing certificates. The OpenSSL command is not that much different from all available howto's on setting up a self-signed CA (Listing 6).

The signing request *cert.csr* is signed with the specified private key on the smart card. The signed certificate is stored in cert.pem. The certificate of the signing private key is stored in a local file called `ewak.net_Sub_CA_servers.crt`. It is also stored on the smart card, but since it is the public key and meant to be public, storing it on the local file system is more convenient and poses no security risk.

Note the different notation of the parameter for the key selection `-keyfile <slot>:<key id>`. OpenSSL will otherwise look for the keyfile specified in your openssl.cnf and fail to load the key.

Listing 8. *Creating the csr*

```
# openssl req -new \
  -engine pkcs11 \
  -keyform engine \
  -key slot_1-key_d893afddc82b28fb539e975b2a3e18efc2f
                  3c474 \
  -out cert.csr -text

  PKCS#11 token PIN:
  You are about to be asked to enter information that
                  will be
  incorporated into your certificate request.
  -snip- OpenSSL output omitted -snip-
```

Listing 9. *Storing the certificate*

```
# pkcs15-init --store-certificate cert.pem --auth-id
                  01 \
  --id d893afddc82b28fb539e975b2a3e18efc2f3c474

  Using reader with a card: Feitian SCR310 00 00
  User PIN [User PIN] required.
  Please enter User PIN [User PIN]:
```

Generating Keys

If you do not have an existing PKI or are willing to change its private keys, you can have the keys generated by and stored on the smart card. The keys will be generated with the true random number generator on the card and will never have touched you computer's memory or file system. This is done with the pkcs15-init tool.

```
# pkcs15-init --generate-key rsa/1024 --auth-id 01

  Using reader with a card: Feitian SCR310 00 00
  User PIN [User PIN] required.
  Please enter User PIN [User PIN]:
```

Both private and public keys are stored on the smart card. The keys are RSA keys of 1024 bits length. This public key can now be used for generating a certificate signing request. First we need the key ID (Listing 7). Create a signing request from the public key (Listing 8).

The *cert.csr* file can now be signed as shown before. The signed certificate can then be stored on the card using the pkcs15-init tool (Listing 9).

Conclusion

Smart cards can provide a low-cost and relative secure storage for private key material. This article focused on the use of smart cards in a CA environment. It did not focus on storing and using a user certificate for email signing, storing and using a client certificate for OpenVPN or storing and using SSH keys. These are all interesting options I will research further.

ERWIN KOOI

Erwin Kooi is an Information Security Manager for a large grid operator. He started with FreeBSD 4.5 and is an avid fan ever since.