Crypto Escrow User Manual

Version 1.0.0-SNAPSHOT

Frank Cornelis

Published Dec 23, 2014

Abstract

This document is the user manual for the Crypto Escrow tool.

1. Introduction

Software escrow can quickly become expensive when the software is released often. Registering each release at a software escrow office can generate a nice invoice flow. Hence it is cheaper to register once crypto keys at an escrow office and use the crypto keys to protect the software source code. The Crypto Escrow tool aids in the setup of such mechanism. The Crypto Escrow tool supports both software keys and hardware tokens. The usage of hardware tokens is especially interesting as it allows you to physically split the token owner and the token PIN owner. From a security policy point of view this four eyes principle might be mandatory.

The Crypto Escrow tool allows for encryption of multiple files, targeting multiple recipient certificates. The Crypto Escrow tool uses a hybrid crypto system. The files are encrypted using a symmetric crypto system. The symmetric encryption key is encrypted towards each recipient using an asymmetric crypto system. This scheme is very flexible and allows for redundancy towards recipient (hardware token) certificates.

1.1. Requirements

The Crypto Escrow tool requires Java version 1.8.0_25 as it is using JavaFX for its user interface.

Because the Crypto Escrow tool uses AES 256 keys it requires the Unlimited Strength Jurisdiction Policy Files to be installed within the Java 8 runtime.

If you want to use hardware tokens, you need the appropriate PKCS#11 libraries.

2. Getting Started

Start the application under Windows by simply double clicking `crypto-escrow-ui-1.0.0-SNAPSHOT.jar` . If you receive a "Java Exception has occurred" from the "Java Virtual Machine Launcher" this is probably due to the fact that you're not running Java 8. In this case, first install Java 8. If you receive an error message about the "Unlimited Strength Jurisdiction Policy", you need to install these first.

2.1. Create a keystore

First of all we need to create a PKCS#12 keystore file. This keystore file will contain our private encryption key and the corresponding self-signed certificate. Create this keystore via "Keystore" and "New PKCS#12 keystore". Take as "Distinguished name" for example "CN=Test". Keystore files normally use ".p12" or ".pfx" as file extension.

Next we will inspect the PKCS#12 keystore and extract the certificate out of it. You need this certificate later on as encryption recipient (to yourself for testing purposes). Do this via "Keystore" and "Inspect PKCS#12 keystore". Click on "Export" to export your self-signed encryption certificate. Certificate files often use the ".cer" file extension.

2.2. Encrypting and decrypting

Now that we have a PKCS#12 keystore containing our private key, and the certificate stored in a separate file, we can encrypt some files to ourself.

Start via "File" and "Encrypt". Select some files. Next add your own self-signed encryption certificate as recipient certificate. The target container is actually a ZIP file. Thus you can use the ".zip" file extension for this.

Decrypting the ZIP container can be done via "File" and "Decrypt". This time you need to use your PKCS#12 keystore that contains the private encryption key.

2.3. Managing encryption identities

Encrypting and decrypting files to yourself is nice. But most of the time you will want to encrypt files towards somebody else. This requires access to the encryption certificate of the other party. To make this possible the other party needs a secure way to communicate its encryption certificate to you. This is where the concept of encryption identities come into play.

If you send your encryption certificate to somebody, the receiving party needs a way to be able to trust that encryption certificate as coming from you. An encryption identity provides this trust mechanism.

An encryption identity is a proxy certificate based on your encryption certificate and created using your eID authentication certificate. Thus you will create a new encryption certificate that has been issued using your eID authentication certificate.

Create an encryption identity via "Identity" and "Create Identity". This operation requires your eID card to issue the proxy certificate.

If you receive an encryption identity you can import its encryption certificate via "Identity" and "Import Identity Certificate". The proxy certificate issuer will be checked using a PKI validation. This ensures that you will only import encryption certificates from trusted entities.

3. SafeNet eToken

Encryption keys should be kept on hardware tokens. In this manual we demonstrate the usage of SafeNet eToken for management of the encryption keys.

3.1. Token Initialization

Start the SafeNet Authentication Client. Click "Advanced View". Click on the token. Click on "Initialize Token". Set a "Token Name". Set a token password. Uncheck "Token Password must be changed on first logon". Click "Advanced". Ensure "2048-bit RSA key support" is checked. Click "Start". The token now has been initialised.

Check the availability of the token via:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --list-slots --show-info
```

Next we generate an RSA 2048 bit key pair on the eToken itself.

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --keypairgen --key-type rsa:2048
 --login --id 45 --label ESCROW
```

This operation can take a while. Check the available objects on the token via:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --login --list-objects
```

3.2. Self-signed certificate

Generate a self-signed certificate via:

```
openssl req -config openssl-etoken.conf -engine pkcs11 -new -x509 -days 365 -
key 45 -keyform engine -out cert.der -outform DER -sha256
```

With the OpenSSL configuration file `openssl-etoken.conf` containing the following:

```
openssl_conf=openssl_def

[openssl_def]
engines=engine_section

[engine_section]
pkcs11=pkcs11_section

[pkcs11_section]
engine_id=pkcs11
dynamic_path=/usr/lib64/openssl/engines/engine_pkcs11.so
MODULE_PATH=/usr/lib64/libeTPkcs11.so
init=0

[req]
distinguished_name=req_distinguished_name
prompt=no
x509_extensions=req_x509_extensions

[req_distinguished_name]
commonName=Crypto Escrow
OU=Intellectual Property
O=e-Contract.be
C=BE
[req_x509_extensions]
basicConstraints=CA:FALSE
keyUsage=keyEncipherment
subjectKeyIdentifier=hash
```

Check the resulting self-signed certificate via:

```
openssl x509 -noout -text -in cert.der -inform DER
```

Write the certificate to the token via:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --write-object cert.der --type
 cert --login --label ESCROW --id 45 --slot 0
```

Make sure that the label and the identifier corresponds with the one of the private key. Check the available objects on the token via:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --login --list-objects
```

Check whether the Java runtime can load the certificate via:

```
keytool -keystore NONE -storetype PKCS11 -providerClass
 sun.security.pkcs11.SunPKCS11 -providerArg etoken.config -list -v
```

With the `etoken.config` file containing:

```
name=eToken
library=/usr/lib64/libeTPkcs11.so
slotListIndex=0
```

3.3. Delete a certificate

It can happen that you wrote the wrong certificate to the token. In this case, you can delete the certificate from the token via:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --delete-object --type cert --
login --label ESCROW --id 45 --slot 0
```

Check the available objects on the token via:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --login --list-objects
```

To delete the private key from the token, run:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --delete-object --type privkey --
login --label ESCROW --id 45 --slot 0
```

The corresponding public key can be removed from the token via:

```
pkcs11-tool --module /usr/lib64/libeTPkcs11.so --delete-object --type pubkey --
login --label ESCROW --id 45 --slot 0
```

4. Encryption Container Format

The encryption container format is based on the EPUB Open Container Format (OCF) 3.0 specification. The files are encrypted using an AES 256 bit symmetric key. The encryption algorithm is AES-GCM. The AES key is encrypted towards each recipient certificate using SHA 512 RSA-OAEP. This minimum RSA key size is 2048 bit.

A. Crypto Escrow User Manual License

B. Crypto Escrow Project License

The Crypto Escrow Project source code has been released under the GNU LGPL version 3.0.

```
This is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License version 3.0 as published by the Free Software Foundation.

This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this software; if not, see http://www.gnu.org/licenses/.
```