YubiHSM 2 Administration and Usage Guides

May 12, 2022

Yubico
## CONTENTS

1 Introduction .......................................................... 1

2 Resetting Device to Factory Settings ......................... 3
   2.1 Physical Reset ............................................... 3
   2.2 Reset Using YubiHSM Shell ............................... 3

3 Initial Provisioning and Deployment Tasks ................ 5
   3.1 Known Usage Cases ....................................... 5
   3.2 HMAC .................................................. 5
   3.3 PKCS11 / RSA ......................................... 6

4 Practical Guide: Quick Start Tutorial ....................... 9
   4.1 Set Up the Environment .................................. 9
   4.2 Start Up .............................................. 10
   4.3 Set Up YubiHSM 2 Connection ......................... 10
   4.4 Sessions .............................................. 10
   4.5 Adding a New Authentication Key ....................... 11
   4.6 Generate a Key for Signing .............................. 12
   4.7 Prepare to Sign With the New Asymmetric Key ........ 12
   4.8 Export Under Wrap .................................. 13

5 EJBCA Installation .................................................. 15
   5.1 Prerequisites .......................................... 15
   5.2 Configuring a New EJBCA Installation ............... 15
   5.3 Configuring an Existing EJBCA Installation .......... 16

6 OpenSSH Certificates for Host Login ......................... 19
   6.1 Traditional Method ..................................... 19
   6.2 OpenSSH CA ....................................... 19
   6.3 OpenSSH Certificates with YubiHSM .................... 19
   6.4 High-Level Description and Components ............... 20
   6.5 SSH Template ...................................... 20
   6.6 SSH Certificate Request ................................ 21

7 OpenSSL with libp11 for Signing, Verifying and Encrypting, Decrypting ............................... 23
   7.1 Signing and Verifying .................................. 23
   7.2 Encrypting and Decrypting ............................... 24

8 OpenSSL with YubiHSM 2 via engine_pkcs11 and yubihsm_pkcs11 .................. 27
   8.1 Example: Creating an Alias ............................... 27
   8.2 Example: Generating a Key in the Device ............ 27
These Admin and Usage Guides for the YubiHSM 2 incorporate the initial provisioning and deployment guide (previously entitled Admin guide) plus the practical guide that is the YubiHSM quick start tutorial, along with instructions for other basic tasks.
Before deploying the YubiHSM 2 in a production environment, it may be necessary to reset the device to its factory settings, for instance in order to facilitate tests or training.

A reset destroys any objects stored on the device that are not factory-installed.

2.1 Physical Reset

The device can be physically reset to its factory settings. To do this, while inserting the device into a USB port, press the metal rim as you insert the device and continue to press the rim for a minimum of 10 seconds.

2.2 Reset Using YubiHSM Shell

Please refer to documentation of the Reset command.
This topic covers operations pertaining to the initial provisioning and deployment of YubiHSM 2 devices. Familiarity with the device, its features and capabilities is assumed.

**Important:** The YubiHSM 2 ships with a default Authentication Key with a well-known password. It is imperative to remove it prior to production deployment.

### 3.1 Known Usage Cases

When only a single application needs to be provisioned, Yubico recommends that all Authentication Keys and material be provisioned only with Capabilities specific to that use case.

**Note:** This type of deployment requires devices to be physically reset and re-provisioned should a new use case arise.

### 3.2 HMAC

```bash
# Establish a session with the default Authentication Key
yubihs> connect
Session keepalive set up to run every 15 seconds
yubihs> session open 1 password
Created session 0

# Create an Authentication Key for Auditing
yubihs> put authkey 0 0 "Audit auth key" all get-log-entries none
$AUDIT_PASS
Stored Authentication key 0xd054

# Create a Wrap Key for importing application Authentication Keys and secrets
yubihs> get random 0 16
5b61e89468cc8f2a274715c78c3d4753
yubihs> put wrapkey 0 0 "HMAC wrap Key" all import-wrapped
    sign-hmac:verify-hmac 5b61e89468cc8f2a274715c78c3d4753
Stored Wrap key 0xf09a
```

(continues on next page)
# Create an Authentication Key for use with the above Wrap Key
```
yubiHSM> put authkey 0 0 "Provisioning HMAC wrap auth key" all
       import-wrapped none $WRAP_PASS
```
Stored Authentication key 0xf10f

# Delete the default Authentication Key
```
yubiHSM> delete 0 1 authentication-key
```

# Create a wrapped Authentication Key and HMAC Key for the application
```
echo -ne "\x5b\x61\xe8\x94\x68\xcc\x8f\x2a\x27\x47\x15\xc7\x8c\x3d\x47\x53"
   \x47\x53 > wrap.key

echo $HMAC_PASS | yubiHSM-wrap -a aes128-yubico-authentication
   -c sign-hmac,verify-hmac -d 1 -l "HMAC auth key" -k wrap.key --in -
   --out auth.out -e none

echo -ne "\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b"
   \x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b\x0b
   > hmac.key

yubiHSM-wrap -a hmac-sha256 -c sign-hmac,verify-hmac -d 1 -l "HMAC key"
   -k wrap.key --in hmac.key --out hmac.out
```

# Open a Session with the wrap Authentication Key
```
yubiHSM> session open 0xf10f $WRAP_PASS
Created session 1
```

# Import the two wrapped keys in the new Session
```
yubiHSM> put wrapped 1 0xf09a auth.out
Object imported as 0x2a74 of type authentication-key

yubiHSM> put wrapped 1 0xf09a hmac.out
Object imported as 0xd1a2 of type hmac-key
```

# Open a session with the new application Authentication Key
```
yubiHSM> session open 0x2a74 $HMAC_PASS
Created session 2
```

# Run HMAC-SHA256 Test vector #1 and get expected output
```
yubiHSM> hmac 2 0xd1a2 4869205468657265
b0344c61d8db38535ca8afceaf0bf12b883dc200c9833da726e9376c2e32cff7
```

3.3 PKCS11 / RSA

This example assumes that only RSA operations will be performed and that RSA keys will be generated on device over PKCS#11. For using the PKCS#11 module a yubiHSM\_pkcs11.conf file will need to exist and point at the desired connector.

```bash
# Establish a Session with the default Authentication Key
yubiHSM> connect
Session keepalive set up to run every 15 seconds
yubiHSM> session open 1 password
```

(continues on next page)
Created session 0

# Create an Authentication Key for Auditing
yubiHSM> put authkey 0 0 "Audit auth key" all audit none $AUDIT_PASS
Created an Authentication key

# Optionally enable forced audits
yubiHSM> put option 0 force-audit 01

# Create an Authentication Key for usage with the PKCS11 module
yubiHSM> put authkey 0 0 "PKCS11 RSA" 1 delete-asymmetric-key:
    generate-asymmetric-key:sign-pkcs:sign-pss sign-pkcs:sign-pss
$PKCS11_PASS
Created an Authentication key

# Delete the default Authentication Key
yubiHSM> delete 0 1 authentication-key

# Use pkcs11-tool to generate an RSA key
pkcs11-tool --module /path/to/yubiHSM_pkcs11.so -l --pin
    f10f$${PKCS11_PASS} -k --key-type rsa:2048 --usage-sign --label "RSA key"
Using slot 0 with a present token (0x0)
Key pair generated:
Private Key Object; RSA
    label: RSA key
    ID: e77d
    Usage: sign
Public Key Object; RSA 2048 bits
    label: RSA key
    ID: e77d
    Usage: none

3.3. PKCS11 / RSA
The purpose of this tutorial is to demonstrate basic functionalities of different key types: Authentication Key, Asymmetric Key and Wrap Key. We start with a fresh YubiHSM 2 configuration and we will proceed in generating a new Authentication Key. Then we generate an Asymmetric Key for signing purposes. We will sign an arbitrary amount of data and verify that our signature is correct. Part of this documentation is to demonstrate how to back up a key on a second YubiHSM 2. We will do so by wrapping the Asymmetric Key and re-importing it into the same device.

This tutorial will cover:
- Basic YubiHSM 2 setup
- Connecting to YubiHSM 2
- Generating an Authkey on the device
- Generating an Asymmetric Object
- Generating a Wrapkey
- Exporting/Importing an Object under wrap

Before proceeding with this document you should be familiar with concepts such as: Sessions, Domains, Capabilities described in the Concepts section.

Note: The following code samples have arbitrary line-breaks to prevent them from running off the page.

### 4.1 Set Up the Environment

- **Step 1** Get the latest binaries from SDK download YubiHSM2/Releases.
- **Step 2** Install all libraries.
- **Step 3** Make sure your device is accessible by the connector. This is accomplished either by running the connector as a superuser or by using an appropriate udev_rule.
4.2 Start Up

To physically reset the YubiHSM 2 insert the device while holding the touch sensor for 10 seconds.

**Step 1** Start the connector.

```
$ yubihsm-connector -d
```

**Step 2** Check the status of your connector and device by using a browser to visit http://127.0.0.1:12345/connector/status.

4.3 Set Up YubiHSM 2 Connection

**Step 1** Start yubihsm-shell.

```
$ yubihsm-shell
```

**Step 2** Connect to YubiHSM 2.

```
$ yubihsm> connect
```

**Step 3** Enable keepalive to facilitate usability during first time use (keepalive is enabled by default). Remember that this will consume one session.

```
yubihsm> keepalive on
```

**Step 4** Enable debug.

```
yubihsm> debug all
```

4.4 Sessions

Many commands require a Session ID to be specified. To obtain a Session ID use the `session open` command followed by an Authentication Key ID and a derivation password.

By default the YubiHSM 2 comes with a pre-installed Authentication Key with Object ID 1 and derivation password `password`.

4.4.1 Open

To open a Session with this Authentication Key use:

```
yubihsm> session open 1 password
Created session 0
```

The Session ID is the number found in the line directly below a `session open` command. In the example above the Session ID is 0. This value will be used to address the newly created Session.
4.4.2 Close

To close a Session use the command `session close` followed by the Session ID:

```
yubihs> session close 0
```

4.4.3 List

To list the objects in the device use:

```
yubihs> list objects 0
```

**Note:** If you have closed Session 0, the above command will not work. In that situation, open a new Session and use the new Session ID in the command above.

4.5 Adding a New Authentication Key

Before moving on, make sure you are familiar with concepts of Capabilities and Domains.

**Step 1** For our example we are going to generate an Authentication Key with selected Capabilities and Domains. Learn more about existing key Types at YubiHSM2 Concepts.

```
yubihs> put authkey 0 2 yubico 1,2,3 generate-asymmetric-key, export-wrapped, get-pseudo-random, put-wrap-key, import-wrapped, delete-asymmetric-key, sign-ecdsa, sign-ecdsa, exportable-under-wrap, export-wrapped, import-wrapped password
```

**Important:** `export-wrapped` allows the creation of Objects that can perform the Export Wrapped command.

`exportable-under-wrap` allows the creation of Objects that can be exported under wrap.

**Note:** The command above has two distinct sets of Capabilities, separated by a space. This is because Authentication Keys, in addition to having regular Capabilities, also have Delegated Capabilities.

**Step 2** List all Objects to see the newly created Authentication Key.

```
yubihs> list objects 0
```

**Step 3** Next, let’s start using our newly created Authentication Key to establish a Session.

```
yubihs> session open 2 password
Created session 1
```

The new Session has been assigned Session ID 1. We will use this Session ID for most of the commands below. If at any time the Session is closed or expires because of inactivity, open a new one and use the correct Session ID.
4.6 Generate a Key for Signing

We now proceed to generate a new Asymmetric Key. In our example we will use this key to sign some data. We will also export the key under wrap to another YubiHSM, for backup purposes.

Specifically, we will ask the device to generate an Asymmetric Key with ID 100 and a given set of Domains and Capabilities. We will also specify the kind of Asymmetric Key that we would like to generate, an EC key using the NIST P-256 curve in this case.

The command is:

```
yubiHSM> generate asymmetric 1 100 label_ecdsa_sign 1,2,3 exportable-under-wrap,sign-ecdsa ecp256
```

On success, we will see the message:

```
Generated Asymmetric key 0x0064
```

This signifies that an Asymmetric Key with ID 0x0064 (hexadecimal for 100) was generated.

4.7 Prepare to Sign With the New Asymmetric Key

**Step 1** Assuming we have a file called `data.txt` containing the data we would like to sign, we will sign it using ECDSA with the Asymmetric Key we generated in the previous step.

```
yubiHSM> sign ecdsa 1 100 ecdsa-sha256 data.txt
```

By default the output is printed to the standard output and consists of a Base64-encoded signature like the one below.

```
MEUCIQDrBqS04LN5YdyWGiD4iaEjf1ldn+W4c197uM
MXDpoaiQIGEBe/G/Fgp4cummn03K2XToAnPvnuVDOnqHPiuUS0q5g=
```

**Step 2** This behavior can be changed by using the `set outformat` and `set informat` commands, and by specifying an additional output parameter to the `sign` command.

For now we will store the signature as it is in a temporary file so that we will be able to verify it later.

```
$ echo MEUCIQDrBqS04LN5YdyWGiD4iaEjf1ldn+W4c197uM
  MXDpoaiQIGEBe/G/Fgp4cummn03K2XToAnPvnuVDOnqHPiuUS0q5g=
  >signature.b64
```

**Step 3** Next, we will extract the public key from the Asymmetric Key on the device and write it to the file `asymmetric_key.pub`, so that we can use it to verify the signature we just created.

```
yubiHSM> get pubkey 1 100 asymmetric_key.pub
```

**Step 4** We are going to use OpenSSL for the verification process. Since the signature that we created before is in Base64 format, we need to convert it first. Do so with:

```
$ base64 -d signature.b64 >signature.bin
```

**Step 5** It is now possible to verify the signature with OpenSSL.
4.8 Export Under Wrap

Time to export the Asymmetric Key under wrap to a second YubiHSM 2 (in this example, we will export to the same YubiHSM for convenience).

**Step 1** To do that we need a Wrap Key, which fundamentally is an AES key. We will use the random number generator built into the YubiHSM to generate the 16 bytes needed for an AES-128 key.

```bash
yubiHSM> get random 1 16
9207653411df91fd36c12faa6886d5c4
```

**Important:** The result of this command (the bytes) is considered sensitive data and should be stored safely.

**Step 2** We can now store the Wrap Key on the device with ID 200 by doing:

```bash
yubiHSM> put wrapkey 1 200 label_wrapkey 1,2,3 import-wrapped,export-wrapped sign-ecdsa, exportable-under-wrap 9207653411df91fd36c12faa6886d5c4
```

**Note:** For the upcoming export command to be successful, the Delegated Capabilities of the Wrap Key have to include the Capabilities of the Object being exported. Similarly, for the import command to succeed the Delegated Capabilities of the Wrap Key have to include the Capabilities of the Object being imported.

**Step 3** We can now export the Asymmetric Key with ID 100 using the Wrap Key with ID 200 and save it to a file called `wrapped_asymmetric.key`.

```bash
yubiHSM> get wrapped 1 200 asymmetric-key 100 wrapped_asymmetric.key
```

**Step 4** We are going to re-import the Asymmetric Key on the same device so we need to first delete the existing one.

```bash
yubiHSM> delete 1 100 asymmetric-key
```

**Step 5** To import the wrapped EC key back into the YubiHSM use:

```bash
yubiHSM> put wrapped 1 200 wrapped_asymmetric.key
```
EJBCA INSTALLATION

EJBCA and YubiHSM 2 work well together once suitable asymmetric keys have been generated on the YubiHSM 2. Even though the EJBCA Adminweb does provide functionality to generate keys on an HSM, this functionality cannot be used with YubiHSM 2. Instead, keys need to be generated using the YubiHSM 2 Setup tool. Once the keys are generated, they can be used, tested and removed using the functionality provided by EJBCA.

When generating new keys on the YubiHSM 2 for use by an existing installation of EJBCA, the relevant crypto token must be reactivated before the new keys are accessible by EJBCA.

Please note that a *key alias* on EJBCA is equivalent to a *key label* on the YubiHSM 2.

5.1 Prerequisites

Download the installation package suitable for the operation system from the Yubico Developers website. The following packages should be installed:

- YubiHSM Connector
- YubiHSM Shell
- YubiHSM Setup
- YubiHSM PKCS#11 Library

5.2 Configuring a New EJBCA Installation

While following the installation instructions provided by EJBCA, the instructions below need to be executed before deploying EJBCA for the first time:

**Step 1**  Decide how many keys to generate and what aliases they should have. See the documentation in `EJBCA_HOME/conf/catoken.properties.sample` for recommendation on what keys should be generated.

**Step 2**  Use the YubiHSM 2 Setup Tool to generate the keys on the YubiHSM 2, one at a time.

**Step 3**  Set the environment variable `YUBIYHSM_PKCS11_CONF` to the path of the `yubihsn_pkcs11.conf` file. See `PKCS#11 with YubiHSM 2` for the content of that file.

**Step 4**  When configuring EJBCA, make sure to configure the following properties files:

- `EJBCA_HOME/conf/catoken.properties`: 
5.3 Configuring an Existing EJBCA Installation

Step 1  Set the environment variable YUBIHSM_PKCS11_CONF to the path of the yubihsm_pkcs11.conf file. See PKCS#11 with YubiHSM 2 for the content of that file.

Step 2  Configure EJBCA_HOME/conf/web.properties as follows (255 is just an example, read the documentation in the file for more details):

```plaintext
cryptotoken.p11.lib.255.name=<label to identify the YubiHSM 2>
cryptotoken.p11.lib.255.file=/path/to/yubihsm_pkcs11.so
```

Step 3  Re-deploy EJBCA and restart the application server.

Step 4  On EJBCA Adminweb, create a new CryptoToken:
   a. Go to CA Functions > Crypto Tokens.
   b. Click on Create new....
   c. Configure the new CryptoToken as follows:
      • Name: <name for this crypto token>
      • Type: PKCS#11
      • Authentication Code: <password to open a session on the YubiHSM 2. See PKCS#11 with YubiHSM 2 > Logging In.
      • PKCS#11: Library: <from the drop down menu, choose the label you set in step 2.>
      • PKCS#11: Reference Type: Slot ID
      • PKCS#11: Reference: 0
      • PKCS#11: Attribute File: Default
   d. Click Save. If there already are keys on the YubiHSM 2, a list of them will be displayed now (only keys created with the YubiHSM 2 Setup tool will be displayed).
**Step 5** On the command line, use the YubiHSM 2 Setup tool to generate keys on the YubiHSM 2, one at a time.

**Step 6** On EJBCA Adminweb, deactivate and then re-activate the Crypto Token created in step 4. The new keys on the YubiHSM 2 are now ready to be used.

---

**Important:** The slot number of the shared PKCS#11 library must be 0.
OpenSSH supports a proprietary version of certificates that allow simple login to hosts.

6.1 Traditional Method

The usual way to enable a user $U$ to access a specific host $H$ using SSH is to copy the public key of $U$ in a file on $H$ (typically called `authorized_keys`).

This method suffers from a lack of generality. If another user $U'$ were to be given access to $H$, their public key should also be copied in that same file. At the same time, if $U$ were to be given access to a different host $H'$, their public key would have to be added to an equivalent file on that host.

While various automatic provisioning systems have been devised, those still represent a workaround rather than a solution to the problem.

6.2 OpenSSH CA

Since version 5.4 (released 2010-03-08) OpenSSH has had support for so-called OpenSSH Certificates.

By using these, only one OpenSSH CA public key has to be copied onto the target host. At that point any user can be granted access to any such host by giving them a file that contains the following information: their own public key, a validity period, a list of usernames that the user it allowed to login as and a digital signature over the whole content created using the private key of an SSH CA.

This file, the SSH Certificate, is then automatically presented to the SSH server by the SSH client of the user as part of the login process.

6.3 OpenSSH Certificates with YubiHSM

The private key of an SSH CA is a regular private key and can be stored on a YubiHSM. In fact, OpenSSH has builtin support for signing SSH Certificates using CA private keys that reside on a hardware token through the PKCS#11 interface.

The YubiHSM however has specific support for signing SSH Certificates and this guide will describe how to leverage that.
6.4 High-Level Description and Components

A YubiHSM device is able to sign OpenSSH public keys when those are submitted to the device as part of a specific format that we call OpenSSH Certificate Request.

Such a request is granted (i.e., the signature is computed and released), if and only if the following two requirements are fulfilled:

- the user who sends the request to the device has the right privileges to access the OpenSSH CA private key on the device;
- the OpenSSH Certificate Request meets a series of pre-defined constraints.

The first requirement is fulfilled by making sure that the user submitting the request (who may not be the same one who generates the request) can establish a Session with the device through an Authentication Key that has access to the necessary Domains and has the necessary Capability set.

The second requirement is fulfilled by encoding those pre-defined constraints in an object with Type Template and Algorithm SSH Template.

6.5 SSH Template

An SSH Template is a binary object that can be used to restrict how and when an SSH CA private key should be used to sign SSH Certificate Requests.

This is a binary object that encodes a series of constraints. Its format is a collection of Tag-Length-Value tuples whose meaning is described below:

<table>
<thead>
<tr>
<th>Tag Value</th>
<th>Tag Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>Timestamp key algorithm</td>
</tr>
<tr>
<td>0x02</td>
<td>Timestamp public key</td>
</tr>
<tr>
<td>0x03</td>
<td>CA key white-list</td>
</tr>
<tr>
<td>0x04</td>
<td>Not before</td>
</tr>
<tr>
<td>0x05</td>
<td>Not after</td>
</tr>
<tr>
<td>0x06</td>
<td>Principals black-list</td>
</tr>
</tbody>
</table>

The individual tags are further explained below.

6.5.1 Timestamp Key Algorithm

The Algorithm of the public key used to verify timestamp signatures.

6.5.2 Timestamp Public Key

The public key used to verify timestamp signatures.
6.5.3 CA Key White-list

The list of Object IDs describing which Asymmetric Keys can be used with this template.

6.5.4 Not Before

The Not Before time offset to be applied to the current time. If a request contains a time value that is before this computed timestamp, an error will be returned.

6.5.5 Not After

The Not After time offset to be applied to the current time. If a request contains a time value that is after this computed timestamp, an error will be returned.

6.5.6 Principals Black-list

The nul-separated, nul-terminated list of Principals (user names) for which a certificate will not be issued.

6.6 SSH Certificate Request

An SSH certificate format is defined by OpenSSH but it is not too dissimilar from an X.509 certificate. At its core it is a collection of attributes, a time period, a public key and a signature over all the data.

An SSH Certificate Request is the set of information that must be sent to a YubiHSM so that it can generate the aforementioned signature. This consists of all the data present in the certificate (excluding the signature).

6.6.1 Signing an SSH Certificate Request

Once an SSH Template has been stored on the YubiHSM and an SSH Certificate Request has been created, it can be sent to the device for signing.

This is done by issuing the Sign SSH Certificate Command. The parameters required are:

- the Object ID of the SSH CA key which has already been stored on the device
- the Object ID of the SSH Template to use in order to validate the request
- the Algorithm to use to produce the certificate signature
- the timestamp with the definition of Now
- signature S~T~ over the SSH Certificate Request and the timestamp
- the SSH Certificate Request

After the command is issued, the following steps take place in the YubiHSM. First the the signature S~T~ is verified using the public key present within the specified SSH Template. If the verification is successful, the value of Now is recorded. Next the SSH Certificate Request is parsed to extract the Not Before and Not After timestamps together with the list of Principals. The following checks are then performed:

- the ID of the SSH CA key must appear in the SSH CA key white-list present in the SSH Template
- the Not Before timestamp in the SSH Certificate Request must be greater than or equal to Now plus the Not Before offset specified in the SSH Template
• the \texttt{Not After} timestamp in the SSH Certificate Request must be less than or equal to \texttt{Now} plus the \texttt{Not After} offset specified in the SSH Template

• none of the Principals specified in the SSH Certificate Request must appear in the Principals black-list SSH Template

If all the constraints were fulfilled, the YubiHSM will produce a signature using the Algorithm specified in the command. This signature can be appended to the SSH Certificate Request to produce a valid SSH Certificate.
OpenSSL can be used with pkcs11 engine provided by the libp11 library, and complemented by p11-kit that helps multiplexing between various tokens and PKCS#11 modules (for example, the system that the following was tested on supports: YubiHSM 2, YubiKey NEO, YubiKey 4, Generic PIV tokens and SoftHSM 2 software-emulated tokens).

### 7.1 Signing and Verifying

#### 7.1.1 RSA-PKCS#1 v1.5

```bash
$ openssl dgst -engine pkcs11 -keyform engine -sign
   "pkcs11:token=YubiHSM;id=%04%01;type=private" -out t3200.pkcs1.sig
   -sha384 t3200.dat
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:

$ openssl dgst -engine pkcs11 -keyform engine -verify
   "pkcs11:token=YubiHSM;id=%04%01;type=public" -signature t3200.pkcs1.sig
   -sha384 t3200.dat
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
Verified OK
```

#### 7.1.2 RSA-PSS

```bash
$ ~/openssl-1.1/bin/openssl dgst -engine pkcs11 -keyform engine -sign
   "pkcs11:token=YubiHSM;id=%04%01;type=private" -out t6400.txt.sigpss
   -sigopt rsa_padding_mode:pss -sha384 t6400.txt
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:

$ ~/openssl-1.1/bin/openssl dgst -engine pkcs11 -keyform engine -verify
   "pkcs11:token=YubiHSM;id=%04%01;type=public" -signature t6400.txt.sigpss
   -sigopt rsa_padding_mode:pss -sha384 t6400.txt
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
```

(continues on next page)
7.1.3 ECDSA

```
$ openssl dgst -engine pkcs11 -keyform engine -sign
   "pkcs11:token=YubiHSM;id=%02%03;type=private" -sha384 -out
t3200.ecdsa.sig t3200.dat
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:

$ openssl dgst -engine pkcs11 -keyform engine -verify
   "pkcs11:token=YubiHSM;id=%02%03;type=public" -sha384 -signature
t3200.ecdsa.sig t3200.dat
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
Verified OK
```

7.2 Encrypting and Decrypting

7.2.1 RSA-PKCS

```
$ cat t64.txt
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76

$ ~/openssl-1.1/bin/openssl pkeyutl -engine pkcs11 -keyform engine
   -pubin -encrypt -inkey "pkcs11:token=YubiHSM;id=%04%02;type=public"
   -pkeyopt rsa_padding_mode:pkcs1 -in t64.txt -out t64.txt.pkcs1
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:

$ ~/openssl-1.1/bin/openssl pkeyutl -engine pkcs11 -keyform engine
   -decrypt -inkey "pkcs11:token=YubiHSM;id=%04%02;type=private"
   -pkeyopt rsa_padding_mode:pkcs1 -in t64.txt.pkcs1
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76
```

7.2.2 RSA-OAEP

```bash
$ cat t64.txt
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76

$ ~/openssl-1.1/bin/openssl pkeyutl -engine pkcs11 -keyform engine
    -pubin -encrypt -inkey "pkcs11:token=YubiHSM;id=%04%02;type=public"
    -pkeyopt rsa_padding_mode:oaep -pkeyopt rsa_oaep_md:sha384 -pkeyopt
    rsa_mgf1_md:sha384 -in t64.txt -out t64.txt.oaep

engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:

$ ~/openssl-1.1/bin/openssl pkeyutl -engine pkcs11 -keyform engine
    -decrypt -inkey "pkcs11:token=YubiHSM;id=%04%02;type=private"
    -pkeyopt rsa_padding_mode:oaep -pkeyopt rsa_oaep_md:sha384 -pkeyopt
    rsa_mgf1_md:sha384 -in t64.txt.oaep

engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76
```

7.2.3 ECDH

```bash
$ openssl pkeyutl -engine pkcs11 -keyform engine -derive -inkey
    "pkcs11:token=YubiHSM;id=%02%04;type=private" -peerkey peer_key.der

engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
34a03079c38947a679a924f3e20657cd4f69dd36df395b7e759e727524da87dc
```

7.2. Encrypting and Decrypting
Chapter Eight

OpenSSL with YubiHSM 2 via Engine_PKCS11 and YUBIHSM_PKCS11

Install `engine_pkcs11` and `pkcs11-tool` from OpenSC before proceeding. Depending on your operating system and configuration you may have to install [libp11](https://github.com/OpenSC/libp11/blob/master/INSTALL.md) as well. If you are on macOS you will have to [symlink pkg-config](https://gist.github.com/aklap/e885721ef15c8668ed0a1dd64d2ea1a7#gistcomment-2814899) in order to do so.

OpenSSL requires engine settings in the `openssl.cnf` file. Some OpenSSL commands allow specifying `-conf openssl.cnf` and some do not. Setting the environment variable `OPENSSL_CONF` always works, but be aware that sometimes the default `openssl.cnf` contains entries that are needed by commands like `openssl req`.

In other words, you may have to add the engine entries to your default OpenSSL config file (`openssl.cnf` in the directory shown by `openssl version -d`) or add other requirements for your OpenSSL command into the config file.

It is suggested that you create a separate config file for interactions with the HSM in order to prevent conflicts with previous settings or defaults.

### 8.1 Example: Creating an Alias

An alias can be created to easily read from a dedicated config file and ensure compatibility across systems

```
alias yubissl='OPENSSL_CONF=/path/to/yubihsms.conf openssl'
```

### 8.2 Example: Generating a Key in the Device

Here is an example of generating a key in the device, creating a self-signed certificate and then signing a CSR with it:

```
$ pkcs11-tool --module /path/to/yubihsms_pkcs11.so --login --pin
    0001password --keypairgen --key-type rsa:2048 --label "my_key"
    --usage-sign
Using slot 0 with a present token (0x0)
Logging in to "YubiHSM".
Please enter User PIN:
Key pair generated:
Private Key Object; RSA
  label: my_key
  ID: 04ec
  Usage: sign
```

(continues on next page)
YubiHSM 2 Administration and Usage Guides

Public Key Object; RSA 2048 bits
  label:  my_key
  ID:    04ec
  Usage: none

$ openssl req -new -x509 -days 365 -subj '/CN=my key/' -sha256 -config
  engine.conf -engine pkcs11 -keyform engine -key slot_0-label_my_key
  -out cert.pem
engine "pkcs11" set.
PKCS#11 token PIN:

$ OPENSSL_CONF=engine.conf openssl x509 -req -CAkeyform engine -engine
  pkcs11 -in req.csr -CA cert.pem -CAkey slot_0-label_my_key -set_serial
  1 -sha256
engine "pkcs11" set.
Signature ok
subject=/CN=test
Getting CA Private Key
PKCS#11 token PIN:

-----BEGIN CERTIFICATE-----
MIICkzCCAXsCAQEwDQYJKoZIhvcNAQELBQAwETEPMA0GA1UEAwwGbXkga2V5MB4X
DTE3MDQyNDA3Mzc1MFoXDTExMDUyNDA3Mzc1MFowDjEMMAoGA1UEAwwDZm9vMIIB
IjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqBARJLAIjSqK20uRWrs91EC
MYjZhxJ8IAMiDDMw2SuQhB7A2C2W+/d1SG0k5cTEiasDBHbH9Bc2w+xnOL3Dh
8xCafvcFjkcnAbHesrbcwRgItugw7PWBtyopWDTdhVKWS1zkpdD8iKjwiYciweaP
96nEHIQPPRUp7bf3IE7RTXENAqJai6QIYBZ0rzHM9NrIz/6YAr2ua7SY7V/B3xaJ
7KsiQ8oHWuf+RDNkJOhhD+1fgemNNx+w4X4YnCPQjJ/MfjuU2n5EM3Vb/plh9H
uD56oZlIK4FeXgC4pNu8f1v2KE1IXMu1CHRbyb5xk0dkQdvKxtVEfiDcwxwBwID
AQABMA0GSCsqGS1b3DQEBcwUA4IBAQCHyskEUI84T/YGhcjplsmobtyNhWc2ae/z
fmQyP/xGzQkSmU3JA-ZO4jMUbil7UKEOItmqSIU6j0BPw03UjavNHdDPYcUZIS28
fPztIkU2F2EMB/zkPXstBco9+N3414qSdir9hFWM1/CpkfP8PhteUQAgImXjdbDVh
qhrf0g+yK3dAa91kLXuA4YfuC+eEJh0jGUXCiVhGreSLL9jraJHhJ+HStG7HjB
R4U7/hzo3r12X5E5jnnjcaKkAKFXZ932qG/WY1iyLpN5SrN+AmmKrPOHb8My
Zl1aGlfm3K9vLJwrjLQSAIKmpDjUcNE7A+EsETQpy43Vvwi8vL
-----END CERTIFICATE-----

8.3 Example: Certificate Request

For these examples, we assume you have all defaults and the engine config below in engine.conf, and provide an example of how to do the latter in the certificate request example below.

$ cat > engine.conf < <EOF
openssl_conf = openssl_init

[openssl_init]
environments = engine_section

[engine_section]
pkcs11 = pkcs11_section
(continues on next page)
8.4 Example: Retrieve 64 Bytes of Data

Here is an example of using the YubiHSM 2 PRNG via OpenSSL to retrieve 64 bytes of data:

```bash
$ OPENSSL_CONF=engine.conf openssl rand -engine pkcs11 -hex 64
```

```
2aae245fc6d1c0419684ee8968ce26fba2dc3bb48a91bae912c8a82b11db8186493
25800e6e984fedfa1940a24731dc2721431979a287252a214ebb87624dcf1
```

8.5 Example: Adding req entries

The following two examples will fail if you are only using the config above because it doesn’t have the req entries in openssl.cnf. You can integrate the engine.conf entries into the system’s openssl.cnf, or add the following to the end of the above engine.conf:

```ini
[ req ]
distinguished_name = req_dn
string_mask = utf8only
utf8 = yes

[ req_dn ]
commonName = Common Name (eg, your name)
```
8.6 Example: Requesting certificate existing RSA key

Here is an example of requesting a certificate for an existing RSA key with ID 3:

```
$ openssl req -new -subj '/CN=test/' -sha256 -config engine.conf -engine pkcs11 -keyform engine -key 0:0003
PKCS#11 token PIN:
-----BEGIN CERTIFICATE REQUEST-----
MIICVDCCATwcAQAwDzENMAsGA1UEAwEdGVzdDCCASIw0DQUwDAQFMB0GA1UdDgQcMekw
-----END CERTIFICATE REQUEST-----
```

8.7 Example: Self-Signed Certificate Existing RSA Key

Or alternatively a self-signed certificate for the same existing RSA key with ID 3:

```
$ openssl req -new -x509 -days 365 -subj '/CN=test/' -sha256 -config engine.conf -engine pkcs11 -keyform engine -key 0:0003
PKCS#11 token PIN:
-----BEGIN CERTIFICATE-----
MIICmjCCAYICCQDX5mJwg+YmMjANBgkqhkiG9w0BAQUFADAPMQ0wCwYDVQQDBPAR0ZXN0
-----END CERTIFICATE-----
```
8.8 Example: s_server with RSA Key and Certificate

Here is an example of using OpenSSL s_server with an RSA key and cert with ID 3.

By default this command listens on port 4433 for HTTPS connections.

```
$ env OPENSSL_CONF=engine.conf openssl s_server -engine pkcs11 -keyform engine -key 0:0003 -cert rsa.crt -www
```

8.9 Example: s_server with ECDSA Key and Certificate

Here is an example of using OpenSSL s_server with an ECDSA key and cert with ID 2:

```
$ env OPENSSL_CONF=engine.conf openssl s_server -engine pkcs11 -keyform engine -key 0:0002 -cert ecdsa.crt -www
```

8.10 Acknowledgements

We would like to thank Uri Blumenthal (uri@mit.edu) for contributing to this document.
It may be convenient to define a shell-level alias for the `pkcs11-tool --module ...` command. It may also be convenient to add the environment variable to point at the `yubihsmpkcs11.so` library.

To accomplish all of the above for the Bash shell one would add the following lines to the `~/.bash_profile` or `~/.bashrc` file:

```bash
export YUBIHW_PKCS11_CONF=/path/to/user/home/yhsm2-p11.conf
export YUBIHW_PKCS11_MODULE=/usr/local/lib/yubihsmpkcs11.so
alias yhsm2-tool='pkcs11-tool --module ${YUBIHW_PKCS11_MODULE} --login'
```

The `--login` option was added because practically no operation of the HSM device can be performed without logging in to it first.

Assuming that

- RSA signing/verifying keypair has been generated with id \(0x0401\) and capabilities including `asymmetric_sign_pkcs:asymmetric_sign_pss`;
- RSA encrypting/decrypting keypair has been generated with id \(0x0402\) and capabilities including `asymmetric_decrypt_pkcs:asymmetric_decrypt_oaep`;
- ECDSA signing/verifying keypair has been generated with id \(0x0203\) and capabilities including `asymmetric_sign_ecdsa:asymmetric_sign_decdsa`;
- EC keypair for deriving ECDH keys has been generated with id \(0x0204\) and capabilities including `derive-ecdh`;

The following commands illustrate the use of OpenSC `pkcs11-tool` with YubiHSM for cryptographic operations.

**Note:** The `pkcs11-tool` can only perform private key-based cryptographic operations. It can decrypt a ciphertext or create a digital signature, but it cannot encrypt a plaintext or verify a digital signature - OpenSSL is used to accomplish that.

The following files are used as samples:

- `t32.dat` is a binary file containing 32 bytes;
- `t3200.dat` is a binary file containing 3200 bytes;
- `t64.txt` is a text file containing 65 bytes (64 ASCII characters and <CR>);
- `peer_key.der` is a file containing an EC public key in DER format
9.1 Creating Digital Signatures

9.1.1 RSA-PSS

**Step 1** Sign a file using RSA-PSS padding with SHA-384:

```
$ yhsm2-tool --sign -m SHA384-RSA-PKCS-PSS --id 0401 -i t3200.dat -o t3200.dat.sig-pss
Using slot 0 with a present token (0x0)
Logging in to "YubiHSM".
Please enter User PIN:
Using signature algorithm SHA384-RSA-PKCS-PSS
PSS parameters: hashAlg=SHA384, mgf=MGF1-SHA384, salt_len=48
```

**Step 2** Verify the created signature with OpenSSL (with libp11 PKCS#11 engine installed):

```
$ openssl dgst -engine pkcs11 -keyform engine -verify
   "pkcs11:token=YubiHSM;id=%04%01;type=public" -signature
t3200.dat.sig-pss -sigopt rsa_padding_mode:pss -sha384 t3200.dat
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
Verified OK
```

9.1.2 RSA-PKCS#1 v1.5

Sign a file using RSA-PKCS#1 v1.5 padding:

```
$ yhsm2-tool --sign --id 0401 -m SHA384-RSA-PKCS -i t3200.dat -o t3200.pkcs1.sig
Using slot 0 with a present token (0x0)
Logging in to "YubiHSM".
Please enter User PIN:
Using signature algorithm SHA384-RSA-PKCS
```

```
$ openssl dgst -engine pkcs11 -keyform engine -verify
   "pkcs11:token=YubiHSM;id=%04%01;type=public" -signature
t3200.pkcs1.sig -sha384 t3200.dat
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
Verified OK
```

9.1.3 ECDSA

Sign a file using ECDSA with SHA-384 hash:

```
$ yhsm2-tool --sign --id 0203 -m ECDSA-SHA384 -f openssl -i t3200.dat
   -o t3200.ec384.sig2
Using slot 0 with a present token (0x0)
Logging in to "YubiHSM".
Please enter User PIN:
Using signature algorithm ECDSA-SHA384

$ openssl dgst -engine pkcs11 -keyform engine -verify
   "pkcs11:token=YubiHSM;id=%02%03;type=public" -signature
   t3200.ec384.sig2 -sha384 t3200.dat
engine "pkcs11" set.
Enter PKCS#11 token PIN for YubiHSM:
Verified OK
$```

9.2 Performing Decryption

9.2.1 RSA-PKCS#1 v1.5

Decrypt a file using RSA-PKCS#1 v1.5 padding:

```
$ cat t64.txt
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76

$ yhsm2-tool --decrypt --id 0402 -m RSA-PKCS -i t64.txt.pkcs1
Using slot 0 with a present token (0x0)
Logging in to "YubiHSM".
Please enter User PIN:
Using decrypt algorithm RSA-PKCS
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76
$```

9.2.2 RSA-OAEP

Decrypt a file using RSA-OAEP and SHA-384. The file t64.txt was encrypted with RSA-OAEP using SHA-384 for digest and Mask Generation Function (MGF):

```
$ cat t64.txt
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76

$ yhsm2-tool --decrypt --id 0402 -m RSA-PKCS-OAEP --hash-algorithm
   SHA384 --mgf MGF1-SHA384 -i t64.txt.oaep
Using slot 0 with a present token (0x0)
Logging in to "YubiHSM".
Please enter User PIN:
Using decrypt algorithm RSA-PKCS-OAEP
```

(continues on next page)
OAEP parameters: hashAlg=SHA384, mgf=MGF1-SHA384, source_type=0, source_ptr=0x0, source_len=0
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76

$ yhsm2-tool --decrypt --id 0402 -m RSA-PKCS-OAEP --hash-algorithm SHA384 -i t64.txt.oaep
Using slot 0 with a present token (0x0)
Logging in to "YubiHSM".
Please enter User PIN:
Using decrypt algorithm RSA-PKCS-OAEP
OAEP parameters: hashAlg=SHA384, mgf=MGF1-SHA384, source_type=0, source_ptr=0x0, source_len=0
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76
$

9.3 Derive ECDH Key

Derive an ECDH key using a private key on the YubiHSM and a public key read from a file.

$ yhsm2-tool --derive --input-file peer_key.der --id 0204
Logging in to "YubiHSM".
Please enter User PIN:
Using slot 0 with a present token (0x0)
Using derive algorithm 0x00001050 ECDH1-DERIVE
34a03079c38947a679a924f3e20657cd4f69dd36df395b7e759e727524da87dc
$

9.4 Obtaining Random Data

$ yhsm2-tool --pin x0xxxxx --generate-random 64 | xxd -c 64 -p
Using slot 0 with a present token (0x0)
e3384c2a8f7263b46879d27d06879e8f82dabe74bf057637a591a314dea86f12f35
79712950695dcebe54824eebe284430e942e1707991e315148e072d59f7
$

9.5 Acknowledgements

We would like to thank Uri Blumenthal (uri@mit.edu) for contributing to this document.
CHAPTER TEN

YUBIHSM AND OPENSSL ON WINDOWS

This page covers setup, configuration, and usage of the Yubico YubiHSM2 with OpenSSL on Windows 10.

10.1 Overview

The Windows OS does not come with many utilities and support found on Linux. This covers installation and usage on a bare Windows 10 system.

Aside from the bare OS, Visual Studio 2019 (v16.2) was installed. For this example, all of the binaries are 64 bit.

1. Download the YubiHSM2 development kit.
2. Download the libp11 source.
3. Download the OpenSC installer.
4. Download the Shining Light Productions OpenSSL installer.

10.2 Installation

10.2.1 YubiHSM2 Development Kit

Step 1 Unzip the downloaded file to install the development kit. The development kit has utilities and a couple of MSI files.

Step 2 Install the files (connector and CSG provider) to connect to the YubiHSM2. You should now be able to use the yubi-shell.exe to connect to the YubiHSM2.

Step 3 Create the YubiHSM2 connector configuration file, and set the YUBIHAM_PKCS11_CONF environmental variable with its path and name. See below for example.

```Yubihsm_pkcs11.cnf connector = http://127.0.0.1:12345```


10.2.2 OpenSC and OpenSSL Distributions

The Shining Light Productions OpenSSL distribution is not an official distribution, but is provided by volunteers. Throw them a donation!

The OpenSC and OpenSSL distributions will be installed under C:\Program Files.

After OpenSC is installed, you should be able to access the YubiHSM2 using pkcs11-tool.

```bash
C:\PROGRA~1\OpenSC Project\OpenSC\tools>set YUBIHSM_PKCS11_CONF=C:\Users\your_name\yubihsm2-sdk-2019-03-win64-amd64\yubihsm2-sdk\yubihsm_pkcs11.cnf

C:\PROGRA~1\OpenSC Project\OpenSC\tools>pkcs11-tool --module

C:\Users\your_name\yubihsm2-sdk-2019-03-win64-amd64\yubihsm2-sdk\bin\yubihsm_pkcs11.dll --login --pin 0001password -I

Cryptoki version 2.40 Manufacturer Yubico (www.yubico.com)
Library YubiHSM PKCS#11 Library (ver 2.1)
Using slot 0 with a present token (0x0)
```

10.2.3 libp11 Source

Download the libp11 source from GitHub. This will need to be compiled.

**Step 1** Open a Visual Studio x64 Native Tools command prompt.

**Step 2** Go to the source directory.

**Step 3** Type: `nmake -f Makefile.mak OPENSSL_DIR=\progra~1\OPENSS~1 BUILD_FOR=WIN64`

The .dll files will be in the source directory.

10.3 Configuration

**Step 1** Two environmental variables must be set: YUBIHSM_PKCS11_CONF and OPENSSL_CONF. These must be set to the location and file name of the respective configuration files. The OpenSSL configuration file is configured with the engine configuration at the top. The HSM PIN, which is its password, may be set in this file. The password here is the YubiHSM2 default password for the default administrator user.

```bash
yubi_openssl.cnf openssl_conf = openssl_init [ openssl_init ]
engines = engines_section [ engines_section ]
pkcs11 = pkcs11_section [ pkcs11_section ]
engine_id = pkcs11
dynamic_path = C:\\Users\\your_name\\Documents\\sourceproj\\libp11-master\\src
pkcs11.dll MODULE_PATH = C:\\Users\\your_name\\yubihsm2-sdk-2019-03-win64-amd64\\yubihsm2-sdk\\bin
yubihsm_pkcs11.dll PIN = 0001password init = 0
```
Step 2 To run the OpenSSL tool commands, the rest of the file contains the normal configuration sections. OpenSSL v1.1.1c requires more configuration than v1.0.2, which is on Ubuntu. The following sections are for creating a self-signed certificate authority certificate. This is just for demonstration, and not to be placed on the FCT stations.

```
More yubi_openssl.cnf [ req ]
prompt = no
distinguished_name = req_distinguished_name
default_bits = 4096
string_mask = utf8only
default_md = sha256
x509_extensions = v3_ca_ext [ req_distinguished_name ]
countryName = US stateOrProvinceName = Washington
localityName = Seattle
organizationName = Banana Inc.
organizationalUnitName = Fruit Bunch
commonName = Root Test Cert [ v3_ca_ext ]
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always,issuer
basicConstraints = critical, CA:true
keyUsage = critical, digitalSignature, cRLSign, keyCertSign
certificatePolicies = 2.5.29.32, @policysection [ policysection ]
policyIdentifier = 1.3.5.8
userNotice.1 = @notice [ notice ]
explicitText = "Yubi Demo Banana Inc. Development Certificate"
```

Step 3 At this point, you should be able to create a self-signed certificate. In this example, key ID 0:0064 is the identifier for a 4096-bit RSA key.

```
C:\Users\your_name>openssl req -new -x509 -days 365 -sha256
   -engine pkcs11 -keyform engine -key 0:0064 -out cert.pem
   engine "pkcs11" set.
C:\Users\your_name>dir cert.pem
Volume in drive C is OSDisk
Volume Serial Number is AC07-5227
Directory of C:\Users\your_name
08/22/2019 02:20 PM 2,322 cert.pem
1 File(s) 2,322 bytes
0 Dir(s) 179,197,755,392 bytes
free C:\Users\your_name>openssl x509 -noout -text -in cert.pem
Certificate: Data: Version: 3 (0x2)
Signature Algorithm: sha256WithRSAEncryption Issuer: C = US,
ST = Washington, L = Seattle, O = Banana Inc.,
OU = Fruit Bunch,
Not After : Aug 21 21:20:07 2020 GMT Subject: C = US,
ST = Washington, L = Seattle, O = Banana Inc.,
OU = Fruit Bunch,
CN = Root Test Cert Subject Public Key Info:
Public Key Algorithm: rsaEncryption RSA Public-Key: (4096 bit)
Modulus: 00:bd:0c:71:1a:4b:19:86:17:d0:d1:bf:c7:27:83:
```

10.3. Configuration
The purpose of the scripts in this repository is to generate an RSA keypair and enroll for an X.509 certificate to a YubiHSM 2 using YubiHSM-Shell as the primary software tool. In addition to YubiHSM-Shell, Java KeyTool and OpenSSL are used.

Two scripts are published in the folder Scripts: the Windows PowerShell script `YubiHSM_Cert_Enroll.ps1` and the Linux Bash script `YubiHSM_Cert_Enroll.sh`.

When the RSA keypair and certificate have been enrolled to the YubiHSM 2, the YubiHSM 2 PKCS #11 library can then be used with the Sun JCE PKCS #11 Provider.

More specifically, the key/certificate can be used for signing Java code, for example using JarSigner.

The following steps are performed by the scripts:

1. Generate an RSA keypair in the YubiHSM 2.
2. Export the CSR (Certificate Signing Request).
3. Sign the CSR into an X.509 certificate (using OpenSSL CA as an example).
4. Import the signed X.509 certificate into the YubiHSM 2.

The scripts are not officially supported and are provided as-is. The scripts are intended as references, and YubiHSM 2 administrators should ensure to read Yubico’s documentation on managing YubiHSMs before making any deployments in production.

### 11.1 Prerequisites

#### 11.1.1 Operating System and SDKs

Use a computer with Windows 10 or a Linux distribution as the operating system.

Attach the YubiHSM 2 device to one of the available USB ports on the computer.

Install the following software SDKs and tools:

- **YubiHSM SDK** (including YubiHSM-Setup, YubiHSM-Shell and YubiHSM-Connector)
- **OpenSSL**
- **Java JDK** (including KeyTool and JarSigner)
11.1.2 Basic Configuration of YubiHSM 2

Start the YubiHSM-Connector, either as a service or from a command prompt.

Launch the YubiHSM-Shell in a different command prompt, and run the following to make sure that the YubiHSM 2 is accessible:

```
yubiHSM-shell
Using default connector URL: http://127.0.0.1:12345
yubiHSM> connect
Session keepalive set up to run every 15 seconds
yubiHSM> session open 1 password
Created session 0
yubiHSM> list objects 0
Found 1 object(s)
id: 0x0001, type: authentication-key, sequence: 0
```

11.1.3 Configuration File for YubiHSM 2 PKCS #11

Create the configuration file `yubihsm_pkcs11.conf` and store it in the same folder as the `yubihsm_pkcs11` module (which is typically `C:\Program Files\Yubico\YubiHSM Shell\bin\pkcs11\` on Windows and `/usr/lib64/pkcs11/` on Linux).

Configure the `yubihsm_pkcs11.conf` according to the instructions on the YubiHSM 2 PKCS #11 webpage. If the YubiHSM-Connector is running on the same machine, it is sufficient to copy the YubiHSM 2 PKCS #11 configuration sample and paste it into the file `yubihsm_pkcs11.conf`.

11.1.4 Configuration File of Sun JCE PKCS #11 Provider with YubiHSM 2

Next, the YubiHSM 2 PKCS #11 module must be configured for use with the Sun JCE PKCS #11 Provider.

Create the configuration file `sun_yubihsm2_pkcs11.conf` with the following content:

```
name = yubihsm-pkcs11
library = C:\Program Files\Yubico\YubiHSM Shell\bin\pkcs11\yubihsm_pkcs11.dll
attributes(*, CKO_PRIVATE_KEY, CKK_RSA) = {
    CKA_SIGN=true
}
```

11.1.5 Environment Variables

The path to the YubiHSM PKCS #11 configuration file must be set in the environment variables for Windows and Linux:

```
YUBIHSM_PKCS11_CONF = <YubiHSM PKCS11 folder>/yubihsm_pkcs11.conf
```

On Windows it is also recommended to add the following folder paths to the environment variable PATH:

```
'C:\Program Files\Yubico\YubiHSM Shell\bin'
'C:\Program Files\OpenSSL-Win64\bin'
'C:\Program Files\Java\jdk-<version>\bin'
```
11.1.6 Java Keystore

The Java keystore contains a number of pre-configured trusted CA-certificates. The Java signing certificate in the YubiHSM 2 will be validated against the trusted CA-certificates in the Java keystore.

It is therefore recommended to check that the CA-certificate(s) that have been used to issue the Java signing certificates are present in the Java keystore. This can be checked by running the following command:

```
keytool -list -cacerts -storepass <password to Java keystore>
```

If it is not present, add the CA-certificate(s) as trusted certificate(s) to the Java keystore. The Java tool KeyTool can be used for this purpose.

In order to update the Java keystore, start a console in elevated mode (“Run as administrator” on Windows or use “sudo” on Linux), and then run the commands below to import and verify the CA-certificate(s):

```
keytool -import -noprompt -cacerts -storepass <password to Java keystore>
   -alias <alias of the CA-cert> -file <path to the CA-certificate file>

keytool -list -cacerts -storepass <password to Java keystore> -alias <alias of the CA-cert>
```

Below are examples of the commands to import and verify the CA-certificate(s) are:

```
keytool -import -noprompt -cacerts -storepass changeit -alias MyCACert
   -file ./rootCACert.pem

keytool -list -cacerts -storepass changeit -alias MyCACert
```

11.2 Windows PowerShell script for generating keys and certificates

The PowerShell script `YubiHSM_Cert_Enroll.ps1` in the Scripts folder can be executed on Windows to generate an RSA keypair and enroll for an X.509 certificate to a YubiHSM 2.

YubiHSM-Shell is used in command line mode.

OpenSSL is used as a basic CA for test and demo purposes only. For real deployments, however, the OpenSSL CA should be replaced with a proper CA that signs the CSR into an X.509 certificate.

11.2.1 Parameters

The PowerShell script has the following parameters.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>Signature algorithm [Default: RSA2048]</td>
</tr>
<tr>
<td>AuthKeyID</td>
<td>KeyId of the YubiHSM 2 authentication key [Default: 0x0001]</td>
</tr>
<tr>
<td>AuthPW</td>
<td>Password to the YubiHSM 2 authentication key [Default: ]</td>
</tr>
<tr>
<td>CACertificate</td>
<td>CA certificate used by OpenSSL (for test purposes) [Default: TestCACert.pem]</td>
</tr>
<tr>
<td>CAPrivateKey</td>
<td>CA private key used by OpenSSL (for test purposes) [Default: TestCAKey.pem]</td>
</tr>
<tr>
<td>CAPrivateKeyPW</td>
<td>Password of the OpenSSL keystore (for test purposes) [Default: ]</td>
</tr>
<tr>
<td>CreateCSR</td>
<td>Generate keys and export CSR and then exit</td>
</tr>
<tr>
<td>CSRfile</td>
<td>File to save the CSR request to [Default: ./YHSM2-Sig.(date and time).csr]</td>
</tr>
<tr>
<td>Dname</td>
<td>X.500 Distinguished Name to be used as subject fields [Default: ]</td>
</tr>
<tr>
<td>Domain</td>
<td>Domain in the YubiHSM 2 [Default: 1]</td>
</tr>
<tr>
<td>ImportCert</td>
<td>Import signed certificate created with CreateCSR</td>
</tr>
<tr>
<td>KeyID</td>
<td>KeyID where the RSA keypair will be stored [Default: 0x0002]</td>
</tr>
<tr>
<td>KeyName</td>
<td>Label of the key/certificate, same as Java Alias [Default: MyKey1]</td>
</tr>
<tr>
<td>LogFile</td>
<td>Log file path [Default: WorkDirectory/YubiHSM_PKCS11_Enroll.log]</td>
</tr>
<tr>
<td>PKCS11Config</td>
<td>Java JCE PKCS #11 configuration file [Default: ./sun_yubihsm2_pkcs11.conf]</td>
</tr>
<tr>
<td>Quiet</td>
<td>Suppress output</td>
</tr>
<tr>
<td>SignedCert</td>
<td>Signed certificate file. [Default: ]</td>
</tr>
<tr>
<td>WorkDirectory</td>
<td>Working directory where the script is executed [Default: $PSScriptRoot]</td>
</tr>
</tbody>
</table>

All parameters have default settings in the PowerShell script. The parameters can either be modified in the PowerShell script, or be used as input variables when executing the script.

11.2.2 Example of how to execute the PowerShell script:

```
$ .\YubiHSM_PKCS11_Setup.ps1 -KeyID 0x0003
```

11.3 Linux Bash Script for Generating Keys and Certificates

The Bash script YubiHSM_Cert_Enroll.sh in the Scripts folder can be executed on Linux to generate an RSA keypair and enroll for an X.509 certificate to a YubiHSM 2.

YubiHSM-Shell is used in command line mode.

OpenSSL is used as a basic CA for test and demo purposes only. For real deployments, however, the OpenSSL CA should be replaced with a proper CA that signs the CSR into an X.509 certificate.

11.3.1 Parameters

The Bash script has the following parameters.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a, --algorithm</td>
<td>Signature algorithm [Default: RSA2048]</td>
</tr>
<tr>
<td>-c, --cacertificate</td>
<td>CA certificate used by OpenSSL (for test purposes) [Default: ./TestCACert.pem]</td>
</tr>
<tr>
<td>-C, --createcsr</td>
<td>Generate keys and export CSR and then exit</td>
</tr>
<tr>
<td>-d, --domain</td>
<td>Domain in the YubiHSM 2 [Default: 1]</td>
</tr>
<tr>
<td>-f, --pkcs11configfile</td>
<td>Java JCE PKCS #11 configuration file Default: ./sun_yubihsm2_pkcs11.conf</td>
</tr>
<tr>
<td>-F, --csrfile</td>
<td>File to save the CSR request to [Default: ./YHSM2-Sig.(date and time).csr]</td>
</tr>
<tr>
<td>-k, --keyid</td>
<td>KeyID where the RSA keypair will be stored [Default: 0x0002]</td>
</tr>
<tr>
<td>-n, --keyname</td>
<td>Label of the key/certificate, same as Java Alias [Default: MyKey1]</td>
</tr>
<tr>
<td>-o, --dnname</td>
<td>X.500 Distinguished Name to be used as subject fields [Default: ]</td>
</tr>
<tr>
<td>-p, --authpassword</td>
<td>Password to the YubiHSM 2 authentication key [Default: ]</td>
</tr>
<tr>
<td>-q, --quiet</td>
<td>Suppress output</td>
</tr>
<tr>
<td>-r, --caprivatekeypw</td>
<td>Password of the OpenSSL keystore (for test purposes) [Default:]</td>
</tr>
<tr>
<td>-s, --caprivatekey</td>
<td>CA private key used by OpenSSL (for test purposes) [Default: ./TestCAKey.pem]</td>
</tr>
<tr>
<td>-S, --signedcert</td>
<td>Signed certificate file. Mandatory when using --importcert [Default:]</td>
</tr>
<tr>
<td>-t, --logfile</td>
<td>Log file path [Default: ./YubiHSM_PKCS11_Enroll.log]</td>
</tr>
</tbody>
</table>

All parameters have default settings in the Bash script. The parameters can either be modified in the Bash script, or be used as input variables when executing the script.

### 11.3.2 Example of How to Execute the Bash Script

```
$ ./YubiHSM_PKCS11_Setup.sh -k 0x0002 -n MyKey -d 1 -a rsa2048 -i 0x0001
-p password -c ./TestCACert.pem -s ./TestCAKey.pem -f ./sun_yubihsm2_pkcs11.conf
```

### 11.4 List the Objects on YubiHSM 2

The created RSA keypair and X.509 certificate can now be accessed through YubiHSM 2 PKCS11 and be used with Sun JCE PKCS11 Provider.

It is recommended to check that the RSA keypair and the X.509 certificate have been created on the YubiHSM 2. It is possible to use either YubiHSM-Shell or Java KeyTool to list and check those objects on the YubiHSM 2.

### 11.4.1 Example: YubiHSM-Shell Command

```
yubihsn> list objects 0
Found 3 object(s)
id: 0x0001, type: authentication-key, sequence: 0
id: 0x0002, type: opaque, sequence: 1
id: 0x0002, type: asymmetric-key, sequence: 0
yubihsn> get objectinfo 0 0x0002 asymmetric-key
id: 0x0002, type: asymmetric-key, algorithm: rsa2048, label:
".................................", length: 896, domains: 1, sequence: 0, origin: generated, capabilities: exportable-under-wrap:
  sign-attestation-certificate:sign-pkcs:sign-pss
```

### 11.4. List the Objects on YubiHSM 2
11.4.2 Example: Java KeyTool Command

```
keytool -list -keystore NONE -storetype PKCS11 -providerClass
sun.security.pkcs11.SunPKCS11 -providerArg sun_yubihsm2_pkcs11.conf
-storepass 0001password -v
```

Keystore type: PKCS11
Keystore provider: SunPKCS11-yubihsm-pkcs11

Your keystore contains 1 entry

Alias name: MyKey1
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=YubiHSM Attestation id:0xd353
Issuer: EMAILADDRESS=admin@test.se, CN=TestCA, OU=Test, O=Yubico,
L=Stockholm, ST=Stockholm, C=SE
Serial number: 2316118fc1d59fbab75138b562a4b00c8163c3d
CEST 2022
Certificate fingerprints:
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1

11.5 Using YubiHSM 2 with Java Signing Applications

When the YubiHSM 2 has been configured with an RSA keypair and a X.509 certificate, the YubiHSM 2 PKCS11 can now be used with any Java signing application that utilizes the default Sun JCE PKCS11 Provider.

For example, JarSigner can be used to sign a JAR-file with the YubiHSM 2 and validate the signed JAR-file.

11.5.1 Example: Use JarSigner to sign a JAR-file

```
jarsigner -keystore NONE -storetype PKCS11 -providerClass
sun.security.pkcs11.SunPKCS11 -providerArg sun_yubihsm2_pkcs11.conf
lib.jar MyKey1 -storepass 0001password -sigalg SHA256withRSA -tsa
http://timestamp.digicert.com -verbose
...jar signed.
```
11.5.2 Example: Use JarSigner to Validate a Signed JAR-file

```bash
jarsigner -verify lib.jar -verbose -certs
...
jar verified.
```
12.1 Backup and Restore Managed via YubiHSM Key Storage Provider

The YubiHSM 2 supports encrypted export and import of objects using a symmetric AES-CCM based scheme.

ADCS does not set the NCRYPT_ALLOW_EXPORT_FLAG when generating a key through the setup UI or the Install ADCSCertificationAuthority PowerShell module.

When creating an ADCS root CA key via the YubiHSM 2, we add the exportable-under-wrap Capability by default, so that back up and restore functionality is available through the following manual process:

12.1.1 Identify Your Private Key Container Name

- **Step 1** Open an elevated command prompt/shell.
  
  To view the currently installed certificates in the Local Machine “My” store, use the certutil command:

  ```powershell
  PS1> certutil -store My
  ```

- **Step 2** Find the target certificate in the list.

- **Step 3** Find the Key Container property of the target certificate. The Provider property should be equal to YubiHSM Key Storage Provider.

- **Step 4** Record the Cert Hash property to identify the certificate.

12.1.2 Back up the Target Certificate

Using any available means (certmgr.msc, PowerShell, certutil), export the target certificate, but without the private key in DER format. The YubiHSM does not provide a mechanism for returning the raw private key to Windows, so generating a PKCS#12 container is not currently possible. As an example, the following exports the certificate in .crt format to a file named <Cert Hash>.crt.

```powershell
PS1> certutil -split -store My <Cert Hash>
```
12.1.3 Back up the Target Private Key

Using the instructions for exporting a private key under wrap via `yubihsmsHELL` (*Back up Objects Using YubiHSM Shell*), export the target private key with the label property equal to the Key Container property. The Authentication Key that performs this operation must have the export-wrapped capability set.

12.1.4 Restore the Target Private Key

Using the instructions for importing a private key under wrap via `yubihsmsHELL` (*Back up Objects Using YubiHSM Shell*), import the target private key file to your backup YubiHSM. The Authentication Key that performs this operation must have the import-wrapped capability set.

The imported key object should have the same Label property as the original object.

12.1.5 Restore the Target Certificate

**Step 1** Move the target certificate file generated above to the target machine.

**Step 2** Import the certificate to the LocalMachine “My” store using your preferred method.

At this point, the certificate will not have an associated private key.

**Step 3** Use the -repairstore functionality of certutil to re-associate the certificate to the private key.

Make sure that the target private key is visible via the YubiHSM KSP, using

```
PS1> certutil -key -csp "YubiHSM Key Storage Provider"
```

This command lists all private keys (and their corresponding container names — which are equal to the Label property in the YubiHSM visible to the current Authentication Key).

**Step 4** Open an elevated prompt and execute the command:

```
PS1> certutil -repairstore MY <Cert Hash>
```

**Step 5** Repeat the steps under *Identify Your Private Key Container Name* to verify that the certificate has been associated with the YubiHSM Key Storage Provider and has the correct Key Container property value.

12.2 Back up and Restore Keys Using YubiHSM Shell

12.2.1 Back up Objects Using YubiHSM Shell

Make sure you have a Wrap Key with the Capabilities export-wrapped, import-wrapped and applicable Delegated Capabilities set:

```
$ yubihsmsHELL -a get-pseudo-random --count=32 --out=wrap.key
...
yubihsmsHELL -a put-wrap-key -c export-wrapped,import-wrapped
  --delegated=sign-pkcs,decrypt-pkcs,exportable-under-wrap --in=wrap.key
...
Stored Wrap key 0xd581
```
When this Wrap Key is present, any Object in the same Domain and with the Capability exportable-under-wrap and Capabilities matching the Wrap Key’s Delegated Capabilities can be exported:

```
$ yubihsn-shell -a generate-asymmetric-key -A rsa2048
    -c exportable-under-wrap,sign-pkcs,decrypt-pkcs
...
```

```
Generated Asymmetric key 0x6e77
```

```
yubihsn-shell -a get-wrapped --wrap-id=0x6e77 --object-id=0xd581
    --t asymmetric-key --out=key_6e77.yhw
...
```

You now have an encrypted backup of the Asymmetric Key 0x6e77 in the file key_6e77.yhw.

**Important:** The file wrap.key here contains the cleartext version of the Wrap Key loaded into your YubiHSM and should therefore be considered sensitive information that needs to be protected.

### 12.2.2 Back up Objects Using YubiHSM Setup

The tool yubihsn-setup can be used to back up all exportable objects at once:

```
$ yubihsn-setup dump
Enter the wrapping key ID to use for exporting objects: 0xd581
...
```

```
Successfully exported object Asymmetric with ID 0x6e77 to ./0x6e77.yhw
All done
```

### 12.2.3 Restore Objects Using YubiHSM Shell

Assuming a fresh device where you want to restore the previously backed up key 0x6e77:

```
$ yubihsn-shell -a put-wrap-key -A aes256-ccm-wrap
    -c export-wrapped,import-wrapped
    --delegated=sign-pkcs,decrypt-pkcs,exportable-under-wrap
    --in=wrap.key -i 0xd581
...
```

```
Stored Wrap key 0xd581
```

```
yubihsn-shell -a put-wrapped --wrap-id=0xd581 --in=key_6e77.yhw
...
```

```
Object imported as 0x6e77 of type asymmetric-key
```

12.2. Back up and Restore Keys Using YubiHSM Shell

51
CHAPTER THIRTEEN

COPYRIGHT

© 2022 Yubico AB. All rights reserved.

Trademarks
Yubico and YubiKey are registered trademarks of Yubico AB. All other trademarks are the property of their respective owners.

Disclaimer
The contents of this document are subject to revision without notice due to continued progress in methodology, design, and manufacturing. Yubico shall have no liability for any error or damages of any kind resulting from the use of this document.

The Yubico Software referenced in this document is licensed to you under the terms and conditions accompanying the software or as otherwise agreed between you or the company that you are representing.

Contact Information

Yubico Inc.
530 Lytton Street
Suite 301
Palo Alto, CA 94301
USA

Click the links to:
- Submit a support request
- Send a Contact Me request
- See additional contact options for getting touch with us

Document Updated
2022-05-12 07:42:59 UTC