YubiHSM 2 User Guide

Yubico

Mar 14, 2025

CONTENTS

CHAPTER

ONE

INTRODUCTION

The YubiHSM 2 is a USB-based, multi-purpose cryptographic device for servers. Its diminutive physical size is ideal for installation directly into internal or external server ports. It is a Hardware Security Module (HSM) that is cost-effective for all organizations. It provides advanced cryptography including hashing, asymmetric, and symmetric key cryptography to protect the cryptographic keys that secure critical applications, identities, and sensitive data in an enterprise for certificate authorities, databases, code signing and more.

YubiHSM 2 FIPS is FIPS 140-2 Level 3 certified device. Certification by National Institute of Standards and Technology (NIST) can be found at: https://csrc.nist.gov/projects/cryptographic-module-validation-program/certificate/3916

YubiHSM 2 FIPS devices include the text "FIPS" laser-etched onto the surface of the device and allow YubiHSM 2 FIPS to run in FIPS Approved mode.

CHAPTER

TWO

YUBIHSM 2 DEVICE SPECIFICATIONS

2.1 Cryptographic Interfaces

- PKCS#11 API version 2.40
- Yubico Key Storage Provider (KSP) to access Microsoft CNG. The KSP is provided as 64-bit and 32-bit DLLs
- Full access to device capabilities through Yubico's YubiHSM Core Libraries (C, Python)

2.2 Advanced Encryption Standard (AES)

- 128, 192, and 256-bit keys
- Support for Electronic Code Book (ECB), Cipher Block Chaining (CBC) and Counter (CCM) modes

2.3 RSA

- 2048-, 3072-, and 4096-bit keys (with e=65537)
- Signing using PKCS#1v1.5 and PSS
- Decryption using PKCS#1v1.5 and OAEP

2.4 Elliptic Curve Cryptography (ECC)

- Curves: secp224r1, secp256r1, secp256k1, secp384r1, secp521r, bp256r1, bp384r1, bp512r1, Ed25519
- Signing: ECDSA (all except Ed25519), EdDSA (Ed25519 only)
- Derivation: ECDH (all except Ed25519)

2.5 Hashing Functions

SHA-1, SHA-256, SHA-384, SHA-512

2.6 Key Wrap

Import and export using NIST-approved AES-CCM Wrap with 128-, 196-, and 256-bit keys

2.7 Random Numbers

On-chip True Random Number Generator (TRNG) used to seed NIST SP 800-90A Rev.1 AES-256 CTR_DRBG

2.8 Attestation

Asymmetric key pairs generated on-device may be attested using a device-specific Yubico attestation key and certificate, or using your own keys and certificates imported into the HSM.

2.9 Performance

Performance varies depending on usage. The accompanying Software Development Kit includes performance tools that can be used for additional measurements. Example metrics from an otherwise unoccupied YubiHSM 2:

- RSA-2048-PKCS1-SHA256: ~139ms
- RSA-3072-PKCS1-SHA384: ~504ms
- RSA-4096-PKCS1-SHA512: ~852ms
- ECDSA-P224-SHA1: ~64ms
- ECDSA-P256-SHA256: ~73ms
- ECDSA-P384-SHA384: ~120ms
- ECDSA-P521-SHA512: ~210ms
- EdDSA-25519-32Bytes: ~105ms
- EdDSA-25519-64Bytes: ~121ms
- EdDSA-25519-128Bytes: ~137ms
- EdDSA-25519-256Bytes: ~168ms
- EdDSA-25519-512Bytes: ~229ms
- EdDSA-25519-1024Bytes: ~353ms
- AES-(128|192|256)-CCM-Wrap: ~10ms
- HMAC-SHA-(1|256): ~4ms
- HMAC-SHA-(384|512): ~243ms

2.10 Storage Capacity

- All data stored as objects. 256 object slots, 126KB max total
- Stores up to 127 rsa2048 or 93 rsa3072 or 68 rsa4096 or 255 of any elliptic curve type, assuming only one authentication key is present
- *Objects*: Authentication keys (used to establish sessions); Asymmetric private keys; Opaque binary data objects (e.g. x509 certificates); Wrap keys; HMAC keys

2.11 Management

- Mutual authentication and secure channel between applications and the YubiHSM 2
- `M of N` unwrap key restore via YubiHSM Setup Tool

2.12 Physical Characteristics

- Form factor: nano designed for confined spaces such as internal USB ports in servers
- Dimensions: 12mm x 13mm x 3.1mm
- Weight: 1g

2.13 Temperatures

- Operational range: 0°C - 40°C (32°F - 104°F)
- Storage range: -20°C - 85°C (-4°F - 185°F)

2.14 Host Interface

Universal Serial Bus (USB) 1.x Full Speed (12 Mbit/s) Peripheral with bulk interface

CHAPTER

THREE

YUBIHSM 2 SOFTWARE DEVELOPMENT KIT (SDK)

YubiHSM 2 SDK can be downloaded from https://developers.yubico.com/YubiHSM2/Releases/ and contains the following tools and libraries to interface with YubiHSM 2.

Resource	Description
bin/libcrypto-3.dll or lib/libcrypto-3.dylib	Pre-built OpenSSL (Windows and MacOS only)
bin/yubihsm-setup	Deployment tool for YubiHSM 2
bin/yubihsm-wrap	A tool to create wrapped importable objects offline
bin/yubihsm-connector	The Connector, a tool for providing a common interface to the device
bin/yubihsm-shell	The shell, a REPL-style tool for interacting with YubiHSM 2 (and the Connector) See Note (1)
include/pkcs11/pkcs11.h	Common and standard PKCS#11 functions and constants definitions
include/pkcs11/pkcs11y.h	Yubico-specific PKCS#11 functions and constants definitions
include/yubihsm.h	Library functions and constants definitions
lib/libyubihsm.{dylib,so} or bin/libyubihsm.dll	Library binary to interact with YubiHSM 2
lib/yubihsm_pkcs11.{dylib,so} or bin/yubihsm_pkcs11.dll	PKCS#11 module to interact with YubiHSM 2
python-noarch/*	Python implementation of the library
yubihsm-cngprovider-windows-amd64.msi	Installer for CNG/KSP for Windows ADCS (Windows only)
yubihsm-connector-windows-amd64.msi	Installer for the Connector (Windows only)

Details on these tools and libraries can be found in the later sections of this document.

3.1 System Requirements

The YubiHSM 2 SDK is built and provided for the following operating systems.

Operating System	Version	Architecture
Debian	10 Buster	amd64
Debian	11 Bullseye	amd64
Debian	12 Bookworm	amd64
Fedora	39	amd64
Fedora	40	amd64
Ubuntu	14.04 Trusty Tahr	amd64
Ubuntu	16.04 Xenial Xerus	amd64
Ubuntu	18.04 Bionic Beaver	amd64
Ubuntu	20.04 Focal Fossa	amd64
Ubuntu	22.04 Jammy Jellyfish	amd64
Ubuntu	24.04 Noble Numbat	amd64
Windows	Server 2019	x64, x86
Windows	Server 2022	x64, x86
macOS	14 Sonoma	amd64, arm64, universal

CHAPTER

FOUR

QUICK START TUTORIAL

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 proceed in generating a new Authentication Key. Then we generate an Asymmetric Key for signing purposes. We sign an arbitrary amount of data and verify that our signature is correct. Part of this documentation is to demonstrate how to backup a key on a second YubiHSM 2. We do so by wrapping the Asymmetric Key and re-importing it into the same device.

This tutorial covers:

- 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 *Core 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

1. Get the latest binaries from SDK download YubiHSM2/Releases.
2. Install all libraries.
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. The following steps use the yubihsm-connector. Connection can also be made using the direct USB mode which is explained later in this document.

1. Start the connector.

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

   where –

   `-d` runs the connector in debug mode which may slow down the connector. It is not required for normal mode of operations.

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

1. Start yubihsm-shell.

```
$ yubihsm-shell
```

2. Connect to YubiHSM 2.

```
$ yubihsm> connect
```

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`.

4.5 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.

where –

`0` Is the Session ID. This value is used to address the newly created Session.

`1` is the object ID of the pre-installed Authentication Key.

`password` is the password of the pre-installed Authentication Key.

4.6 Close

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

```
yubihsm> session close 0
```

where –

`0` Is the Session ID.

4.7 List

To list the objects in the device use:

```
yubihsm> list objects 0
```

where–

`0` Is the Session ID.

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.

4.8 Adding a New Authentication Key

Before moving on, make sure you are familiar with concepts of *Capability* and *Domain*

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

   ```
   yubihsm> 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
   ```

   where –

   `put authkey` is the command to create a new authentication key.

   `0` is the session ID.

   `2` is the ObjectID of the new authentication key.

   `yubico` is the label of the new authentication key.

   `1,2,3` is the domain where the new authentication key will operate within.

   `generate-asymmetric-key, export-wrapped,get-pseudo-random,put-wrap-key, import-wrapped,delete-asymmetric-key,sign-ecdsa` are the capabilities for the new authentication key.

   `sign-ecdsa,exportable-under-wrap,export-wrapped,import-wrapped` the delegated capabilities for the new authentication key.

`password` is the password used to derive the new authentication key. This is the password you specify when opening a session with the YubiHSM using this authentication key.

Important: `export-wrapped` allows the creation of Objects that can perform the *EXPORT WRAPPED Command* 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 *Capability*.

---

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

```
yubihsm> list objects 0
```

where –

`0` the Session ID used for the open session.

3. Next, let's start using our newly created Authentication Key to establish an encrypted Session.

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

where –

`1` is the Session ID assigned to the new Session. 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.

`2` is the ObjectID of the authentication key used to open the session.

`password` is the password of the authentication key used to open the session.

4.9 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
```

where –

`generate` is YubiHSM shell command.

`asymmetric` is the key type to be generated.

`1` is the session ID.

`100` is the key ID.

`label_ecdsa_sign` is the label for the new key object.

`1,2,3` are the domains where the new key will be accessible.

`exportable-under-wrap` allows this key to be exported under wrap.

`sign-ecdsa` is allows this key to be used to perform ECDSA signature.

`ecp256` specifies NIST P-256 curve for the key.

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.

4.10 Prepare to Sign With the New Asymmetric Key

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
   ```

   where –

   1 is the Session ID.

   `100` is the key ID.

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

   ```
   MEUCIQDrBqS04LN5YdyWGiD4iaEjfl1dn+W4cl97uMMXDpoaiQIgEBe/G/
   ↪FgP4cumnO3K2XWToAnPvnuVDOnqHPiuUS0q5g=
   ```

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 MEUCIQDrBqS04LN5YdyWGiD4iaEjfl1dn+W4cl97uMMXDpoaiQIgEBe/G/
   ↪FgP4cumnO3K2XWToAnPvnuVDOnqHPiuUS0q5g= >signature.b64
   ```

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
   ```

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
   ```

5. It is now possible to verify the signature with OpenSSL.

```
$ openssl dgst -sha256 -signature signature.bin -verify asymmetric_key.pub␣
↪data.txt
Verified OK
```

4.11 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).

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

```
yubihsm> get random 1 16 9207653411df91fd36c12faa6886d5c4
```

Important: The result of this command (the bytes) is considered extremely sensitive data and should be stored safely, and preferably, separate from any production environment.

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

```
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.

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`.

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

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

```
yubihsm> delete 1 100 asymmetric-key
```

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

```
yubihsm> put wrapped 1 200 wrapped_asymmetric.key
```

CHAPTER

FIVE

YUBIHSM 2 SDK TOOLS AND LIBRARIES

5.1 YubiHSM 2 Setup Tool

The SDK ships with a tool called yubihsm-setup that helps with setting up a device for specific use cases. The tool assumes familiarity with the key concepts of YubiHSM such as *Domain*, *Capability* and *Object ID*. It currently supports the following:

- setup for KSP/ADCS and EJBCA;
- restoring a previous configuration
- resetting the device to factory defaults
- exporting all existing objects

The tool is based around the concept of secret-sharing. When setting up Objects, those are exported with a freshly created Wrap Key. The key is never stored on disk, but rather it is printed on the screen as shares. The key concepts here are:

- The number of shares, which is the number of parts the key should be divided into.
- The security threshold, which is the minimum number of shares required to reconstruct the Wrap Key.

Besides splitting the Wrap Key into shares, the tool (by default) also exports under wrap all the newly created objects and saves them in the current directory. This can be used at a later time to "clone" or recover a device. This operation can be performed either with `yubihsm-setup` or manually if the Wrap Key is known.

By default, the Authentication Key used to establish a Session with the device is also normally deleted at the end of the process.

Default behavior can be altered with command line options. For more information, consult the tool's help.

5.1.1 Setup for EJBCA

When setting up the device for use by EJBCA, the setup tool also generates an asymmetric key pair and an X509 certificate suitable for use as a CA key. The setup tool can be re-run as many times as the number of asymmetric keys to be generated since each run will produce only one key pair and one corresponding X509 certificate.

Note: Using the `--no-new-authkey` flag prevents generation of a new Wrap Key and a new Authentication Key.

5.1.2 How It Works

For the JAVA implementation, a key pair can be used to perform PKCS#11 operations only if the key and its corresponding X509 certificate are stored under the same ID on the device (the value of their CKA_ID attributes is the same). To store them under the same ID, run the YubiHSM 2 Setup tool with the `ejbca` subcommand:

1. Generate an Asymmetric Key on the YubiHSM 2.
2. Generate an attestation certificate for the asymmetric key and import it into the YubiHSM 2 under the same ID as the Asymmetric Key.

The attestation certificate stored on the YubiHSM 2 is, in fact, only a placeholder certificate for the public key. It is never used by EJBCA because EJBCA stores the CAs' certificates in a dedicated database.

5.2 YubiHSM Shell

The yubihsm-shell is the administrative and testing tool you can use to interact with and configure the YubiHSM 2 device. All the commands supported by YubiHSM 2 *YubiHSM Command Reference* can be issued to YubiHSM 2 using YubiHSM 2 Shell.

The Shell can be invoked in two different ways: interactively, or as a command line tool useful for scripting.

Additional information on the various commands can be obtained with the `help` command in interactive mode or by referring to the `--help` argument for the command line mode.

Examples of commands can also be found in the *YubiHSM Command Reference* reference.

5.2.1 YubiHSM Shell Command Syntax

Commands and subcommands require specific arguments to work. The Shell will return an error message if the command syntax is incorrect, pointing at the first invalid argument.

Arguments have different types. In interactive mode pre-defined values for command types can be tab-completed (Tab Completion does not work on Windows). Command arguments are explained in the table below.

Arg	Type	Description
A	Algorithm	An algorithm in string form (ex: ecp256)
B	Byte	A generic (hex or dec) 8-bit unsigned number
C	capabilities	A list of Capabilities in either form: hex (ex: 0xffffffffffff) or string (ex: sign-pkcs,sign-pss, get-log-entries)
D	Domains	A list of Domains, either in hex (ex: 0xffff) or string form (ex: 3,5,14)
I	Format	A format specifier in string form (ex: base64)
I	input data	Input data, generally defaults to standard input
U	Number	A generic (hex or dec) unsigned number
O	Option	A device-global option in string form (ex: force-audit)
F	output filename	Output file name, generally defaults to standard output
E	Session	The ID of an already-established Session
S	String	A generic string (use quotes for strings including white spaces)
T	Type	An Object Type in string form (ex: Asymmetric)
W	Word	A generic (hex or dec) 16-bit unsigned number

Different commands have different default formats. These can be listed by invoking `help` on a specific command. For example, the `help sign` will display the following message:

```
pss             Sign data using RSASSA-PSS (default input format: binary)
                e:session,w:key_id,a:algorithm,i:data=-,F:out=-
```

As can be seen, the input format is binary. Additionally, arguments to a command that have `=-` after their type and name (like `i:data` and `F:out` in the example above), use the standard input or standard output by default for reading data.

Different levels of debug output can be enabled by using the `-v` flag in command line mode, or by issuing the `debug LEVEL` command in interactive mode, where LEVEL is one of `all`, `crypto`, `error`, `info`, `intermediate`, `none`, or `raw`.

The following is a list of supported yubihsm-shell commands and their formats.

Blink Device – Blinks the LED of the device to identify it.

Interactive mode

```
$ blink <session> <seconds=10>
```

Command line mode

```
$ yubihsm-shell -a blink-device [--duration <seconds=10>]
```

Change Authentication Key – Replaces the Authentication Key used to establish the current Session.

Interactive mode

```
$ change authkey <session> <key_id> <password=->
```

Close Session – Closes the current session and releases it for re-use.

Interactive mode

```
$ session close <session>
```

Create Otp Aead – Creates a Yubico OTP AEAD using the provided data.

Interactive mode

```
$ otp aead_create <session> <key_id> <key> <private_id> <aead>
```

Decrypt AES CBC – Decrypt data in AES CBC mode.

Interactive mode

```
$ decrypt aescbc  <session> <key_id> <iv> <data=->
```

Command line mode

```
$ yubihsm-shell -a decrypt-aescbc -i <key_id> --iv <iv> --in <data>
```

Decrypt AES ECB – Decrypt data in AES ECB mode.

Interactive mode

```
$ decrypt aesecb  <session> <key_id> <data=->
```

Command line mode

```
$ yubihsm-shell -a decrypt-aesecb -i <key_id> --in <data>
```

Decrypt Oaep – Decrypts data encrypted with RSA-OAEP.

Interactive mode

```
$ decrypt oaep <session> <key_id> <algorithm> <in_data=-> <label=>
```

Command line mode

```
$ yubihsm-shell -a decrypt-oaep -i <key_id> -A <decrypt_algorithm> [--
↪in <in_data> -l <oaep_label>]
```

Decrypt Otp – Decrypts a Yubico OTP with an AEAD and returns counters and timer information (default input format in binary).

Interactive mode

```
$ otp decrypt <session> <key_id> <otp> <aead>
```

Decrypt Pkcs1 – Decrypts data encrypted with RSA-PKCS#1v1.5.

Interactive mode

```
decrypt pkcs1v1_5 <session> <key_id> data=->``
```

Command line mode

```
$ yubihsm-shell -a decrypt-pkcs1v15 -i <key_id> [--in <data>]
```

Delete Object – Deletes an object in the device.

Interactive mode

```
$ delete <session> <object_id> <type>
```

Command line mode

```
$ yubihsm-shell -a delete-object -i <object_id> -t <type>``
```

Derive Ecdh – Performs an ECDH key exchange with the private key in the device.

Interactive mode

```
$ derive ecdh <session> <key_id> <public_key=->
```

Command line mode

```
$ yubihsm-shell -a derive-ecdh -i <key_id> [--in <public_key>]
```

Encrypt AES CBC – Encrypt data in AES CBC mode.

Interactive mode

```
$ encrypt aescbc  <session> <key_id> <iv> <data=->
```

Command line mode

```
$ yubihsm-shell -a encrypt-aescbc -i <key_id> --iv <iv> --in <data>
```

Encrypt AES ECB – Encrypt data in AES ECB mode.

Interactive mode

```
$ encrypt aesecb  <session> <key_id> <data=->
```

Command line mode

```
$ yubihsm-shell -a encrypt-aesecb -i <key_id> --in <data>
```

Export Wrapped – Retrieves an object under wrap from the device. The Object is encrypted using AES-CCM with a 16 bytes MAC and a 13 bytes nonce.

Interactive mode

```
$ get wrapped <session> <wrapkey_id> <type> <object_id> <file=->
```

Command line mode

```
$ yubihsm-shell -a get-wrapped --wrap-id <wrapkey_id> -t <type> -i
↪<object_id> [--out <file>]
```

Generate Asymmetric Key – Generates an Asymmetric Key in the device.

Interactive mode

```
$ generate asymmetric session> <object_id> <label> <domains>
↪<capabilities> <algorithm>
```

Command line mode

```
$ yubihsm-shell -a generate-asymmetric-key -i <object_id> -l <label> -
↪d <domains> -c <capabilities> -A <algorithm>
```

Generate Hmac Key – Generates an HMAC Key in the device.

Interactive mode

```
$ generate hmackey <session> <key_id> <label> <domains> <capabilities>
↪<algorithm>
```

Command line mode

```
$ yubihsm-shell -a generate-hmac-key -i <key_id> -l <label> -d
↪<domains> -c <capabilities> -A <algorithm>
```

Generate Otp Aead Key – Generates an OTP AEAD Key for Yubico OTP decryption.

Interactive mode

```
$ generate otpaeadkey <session> <key_id> <label> <domains>
↪<capabilities> <algorithm> <nonce_id>
```

Command line mode

```
$ yubihsm-shell -a generate-otp-aead-key -i <key_id> -l <label> -d
↪<domains> -c <capabilities> -A <algorithm> --nonce <nonce_id>
```

Generate Symmetric Key – Generates a symmetric key.

Interactive mode

```
$ generate symmetric <session> <key_id> <label> <domains>
↪<capabilities> <algorithm>
```

Command line mode

```
$ yubihsm-shell -a generate-symmetric-key -i <key_id> -l <label> -d
↪<domains> -c <capabilities> -A <algorithm>
```

Generate Wrap Key – Generates a Wrap Key that can be used for export, import, wrap data, and unwrap data.

Interactive mode

```
$ generate wrapkey <session> <key_id> <label> <domains> <capabilities>
↪<delegated_capabilities> <algorithm>
```

Command line mode

```
$ yubihsm-shell -a generate-wrap-key -i <key_id> -l <label> -d
↪<domains> -c <capabilities> --delegated <delegated_capabilities> -A
↪<algorithm>
```

Get Device Info – Gets device version, device serial, supported algorithms and the number of log entries.

Interactive mode

```
$ get deviceinfo
```

Command line mode

```
$ yubihsm-shell -a get-device-info
```

Get Device Public Key – Retrieves the device's public key for the purpose of asymmetric authentication.

Interactive mode

```
$ get devicepubkey
```

Command line mode

```
$ yubihsm-shell -a get-public-key
```

Get Log Entries – Fetches all current entries from the device Log Store.

Interactive mode

```
$ audit get <session>
```

Command line mode

```
$ yubihsm-shell -a get-logs
```

Get Object Info – Fetches all metadata about an object.

Interactive mode

```
$ get objectinfo <session> <object_id> <type>
```

Command line mode

```
$ yubihsm-shell -a get-object-info -i <object_id> -t <type>
```

Get Opaque – Retrieves an Opaque object (like an X.509 certificate) from the device.

Interactive mode

```
$ get opaque <session> <object-id>
```

Command line mode

```
$ yubihsm-shell -a get-opaque -i <object-id>
```

Get Option – Gets device-global options.

Interactive mode

```
$ get option <session> <option>
```

Command line mode

```
$ yubihsm-shell -a get-option --opt-name <option>
```

Get Pseudo Random – Extracts a fixed number of pseudo-random bytes from the device, using the internal PRNG.

Interactive mode

```
$ get random <session> <number_of_bytes> <out=->
```

Command line mode

```
$ yubihsm-shell -a get-pseudo-random [--count <number_of_bytes=256> --
↪out <out>]
```

Get Public Key – Fetches the public key of an Asymmetric Key.

Interactive mode

```
$ get pubkey <session> <key_id>
```

Command line mode

```
$ yubihsm-shell -a get-public-key -i <key_id>
```

Get Storage Info – Reports currently free storage.

Interactive mode

```
$ get storage <session>
```

Get Template – Retrieves a Template object from the device.

Interactive mode

```
$ put template <session> <object_id> <out_data=->
```

Command line mode

```
$ yubihsm-shell -a get-template -i <object_id> [--out <out_data>]
```

Import Wrapped – Imports a wrapped/encrypted object that was previously exported by an YubiHSM 2 device.

Interactive mode

```
$ put wrapped <session> <wrapkey_id> <data=->
```

Command line mode

```
$ yubihsm-shell -a put-wrapped --wrap-id <wrapkey_id> [--in <file>]
```

List Objects – Gets a filtered list of objects from the device.

Interactive mode

```
$ list objects <session> <id=0> <type=any> <domains=0> <capabilities=0>
↪ <algorithm=any> <label=>
```

Command line mode

```
$ yubihsm-shell -a list-objects -t <type> -A <algorithm> [-i <id> -d
↪<domains> -c <capabilities> -l <label>]
```

Put Asymmetric Key – Imports an Asymmetric Key into the device.

Interactive mode

```
$ put asymmetric <session> <object_id> <label> <domains> <capabilities>
↪ <key=->
```

Command line mode

```
$ yubihsm-shell -a put-asymmetric-key -i <object_id> -l <label> -d
↪<domains> -c <capabilities> [--in <key>]
```

Put Authentication Key – Stores an Authentication Key in the device.

Interactive mode

```
$ put authkey <session> <object_id> <label> <domains> <capabilities>
↪<delegated_capabilities> <password=->
```

Command line mode

```
$ yubihsm-shell -a put-authentication-key -i <object-id> -l <label> -d
↪<domains> -c <capabilities> --delegated <delegated_capabilities> [--
↪new-password <password>]
```

Put Hmac Key – Stores an HMAC Key in the device.

Interactive mode

```
$ put hmackey <session> <object_id> <label> <domains> <capabilities>
↪<algorithm> <key>
```

Put Opaque – Stores Opaque data (like an X.509 certificate) in the device.

Interactive mode

```
$ put opaque <session> <object_id> <label> <domains> <capabilities>
↪<algorithm> <data=->
```

Command line mode

```
$ yubihsm-shell -a put-opaque -i <object-id> -l <label> -d <domains> -
↪c <capabilities> -A <algorithm> [--in <data>]
```

Put Otp Aead Key – Imports an OTP AEAD Key used for Yubico OTP Decryption.

Interactive mode

```
$ put otpaeadkey <session> <key_id> <label> <domains> <capabilities>
↪<nonce_id> <key>
```

Put Symmetric Key – Imports a symmetric key.

Interactive mode

```
$ put symmetric <session> <key_id> <label> <domains> <capabilities>
↪<algorithm> <key>
```

Command line mode

```
$ yubihsm-shell -a put-symmetric-key -i <key_id> -l <label> -d
↪<domains> -c <capabilities> -A <algorithm> --in <key>
```

Put Template – Stores a Template in the device (like the template used when signing SSH certificate).

Interactive mode

```
$ put template <session> <object_id> <label> <domains> <capabilities>
↪<algorithm> <in_data=->
```

Command line mode

```
$ yubihsm-shell -a put-template -i <object_id> -l <label> -d <domains>␣
↪-c <capabilities> -A <algorithm> [--in <in_data>]
```

Put Wrap Key – Imports a key for wrapping into the device.

Interactive mode

```
$ put wrapkey <session> <object_id> <label> <domains> <capabilities>
↪<delegated_capabilities> <key>
```

Command line mode

```
$ yubihsm-shell -a put-wrap-key -i <object_id> -l <label> -d <domains>␣
↪-c <capabilities> --delegated <delegated_capabilities> --in <key>
```

Randomize Otp Aead – Creates a new OTP AEAD using random data for key and private ID.

Interactive mode

```
$ opt aead_random <session> <key_id> <aead>
```

Command line mode

```
$ yubihsm-shell -a randomize-otp-aead -i <key_id> --in <aead>
```

Reset Device – Resets and reboots the device, deletes all Objects and restores the default Options and Authentication Key.

Interactive mode

```
$ reset <session>``
```

Command line mode

```
$ yubihsm-shell -a reset
```

Rewrap Otp Aead – Re-encrypts a Yubico OTP AEAD from one OTP AEAD Key to another OTP AEAD Key.

Interactive mode

```
$ otp rewrap <session> <key_id_from> <key_id_to> <aead_in> <aead_out>
```

Session Message – Sends a wrapped command for a previously established session. The command is encrypted and authenticated.

Set Log Index – Informs the device what the last extracted log entry is so logs can be reused.

Interactive mode

```
$ audit set <session> <index>
```

Command line mode

```
$ yubihsm-shell -a set-log-index --log-index <index>
```

Set Option – Sets device-global options that affect general behavior.

Interactive mode

```
$ put option <session> <option> <value>
```

Command line mode

```
$ yubihsm-shell -a put-option --opt-name <option> --opt-value <value>
```

Sign Attestation Certificate – Gets attestation of an Asymmetric Key in the form of an X.509 certificate.

Interactive mode

```
$ attest asymmetric <session> <key_id> <attest_id=0 <file=->
```

Command line mode

```
$ yubihsm-shell -a sign-attestation-certificate -i <key_id> --
↪attestation-id <attest_id> [--out <file>]
```

Sign Ecdsa – Computes a digital signature using ECDSA on the provided data.

Interactive mode

```
$ sign ecdsa <session> <key_id> <signing_algorithm> <in_data=-> <out-
↪data=->
```

Command line mode

```
$ yubihsm-shell -a sign-ecdsa -i <key_id> -A <signing_algorithm> [--in
↪<in_data> --out <out_data>]
```

Sign Eddsa – Computes a digital signature using EdDSA on the provided data.

Interactive mode

```
$ sign eddsa <session> <key_id> <algorithm> <data=-> <out=->
```

Command line mode

```
$ yubihsm-shell -a sign-eddsa -i <key_id> -A <algorithm> [--in <data> -
↪-out <out>]
```

Sign Hmac – Performs an HMAC operation in the device and returns the result.

Interactive mode

```
$ hmac <session> <object_id> <data_to_sign=- in hex> <out=->
```

Command line mode

```
$ yubihsm-shell -a sign-hmac -i <object_id> [--in <data_to_sign_in_hex>
↪ --in <out_data>]
```

Sign Pkcs1 – Computes a digital signature using RSA-PKCS1v1.5 on the provided data.

Interactive mode

```
$ sign pkcs1v1_5 <session> <object_id> <algorithm> <data=-> <out=->
```

Command line mode

```
$ yubihsm-shell -a sign-pkcs1v15 -i <object_id> -A <algorithm> [--in
↪<data> --out <out>]
```

Sign Pss – Computes a digital signature using RSA-PSS on the provided data.

Interactive mode

```
$ sign pss <session> <key_id> <signing_algorithm> <in_data=-> <out_
↪file=->
```

Command line mode

```
$ yubihsm-shell -a sign-pss -i <key_id> -A <signing_algorithm> [--in
↪<in_data> --out <out_file>]
```

Sign Ssh Certificate – Produces an SSH Certificate signature (only works with RSA keys).

Interactive mode

```
$ certify <session> <key_id> <template_id> <algorithm> <in_data_in_
↪binary_format=-> <out_data=->
```

Command line mode

```
$ yubihsm-shell -a sign-ssh-certificate -i <key_id> --template-id
↪<template_id> -A <algorithm> [--in <in_data_in_binary_format> --out
↪<out_data>]
```

Unwrap Data – Decrypts (unwraps) data using a Wrap Key.

Interactive mode

```
$ decrypt aesccm <session> <key_id> <data=->
```

Verify Hmac – Verifies a generated HMAC

Wrap Data – Encrypts (wraps) data using a Wrap Key.

Interactive mode

```
$ encrypt aesccm <session> <key_id> <data=->
```

5.2.2 YubiHSM 2 Shell Algorithm Names

Following table describes algorithm names to be used with YubiHSM Shell for the algorithms supported by YubiHSM 2.

Name	yubihsm-shell name	Comment
AES 128	aes128	
AES 192	aes192	
AES 256	aes256	
AES CBC	aes-cbc	
AES ECB	aes-ecb	
AES128 CCM WRAP	aes128-ccm-wrap	
AES192 CCM WRAP	aes192-ccm-wrap	
AES256 CCM WRAP	aes256-ccm-wrap	
EC BP256	ecbp256	brainpool256r1
EC BP384	ecbp384	brainpool384r1
EC BP512	ecbp512	brainpool512r1
EC ECDH	ecdh	
EC K256	eck256	secp256k1
EC P224	ecp224	secp224r1
EC P256	ecp256	secp256r1
EC P384	ecp384	secp384r1
EC P521	ecp521	secp521r1

continues on next page

Table 1 – continued from previous page

Name	yubihsm-shell name	Comment
ECDSA SHA1	ecdsa-sha1	
ECDSA SHA256	ecdsa-sha256	
ECDSA SHA384	ecdsa-sha384	
ECDSA SHA512	ecdsa-sha512	
ED25519	ed25519	
HMAC SHA1	hmac-sha1	
HMAC SHA256	hmac-sha256	
HMAC SHA384	hmac-sha384	
HMAC SHA512	hmac-sha512	
MGF1 SHA1	mgf1-sha1	
MGF1 SHA256	mgf1-sha256	
MGF1 SHA384	mgf1-sha384	
MGF1 SHA512	mgf1-sha512	
Opaque Data	opaque-data	
Opaque X509 Certificate	opaque-x509-certificate	
RSA 2048	rsa2048	
RSA 3072	rsa3072	
RSA 4096	rsa4096	
RSA OAEP SHA1	rsa-oaep-sha1	
RSA OAEP SHA256	rsa-oaep-sha256	
RSA OAEP SHA384	rsa-oaep-sha384	
RSA OAEP SHA512	rsa-oaep-sha512	
RSA PKCS1 SHA1	rsa-pkcs1-sha1	
RSA PKCS1 SHA256	rsa-pkcs1-sha256	
RSA PKCS1 SHA384	rsa-pkcs1-sha384	
RSA PKCS1 SHA512	rsa-pkcs1-sha512	
RSA PSS SHA1	rsa-pss-sha1	
RSA PSS SHA256	rsa-pss-sha256	
RSA PSS SHA384	rsa-pss-sha384	
RSA PSS SHA512	rsa-pss-sha512	
SSH Template	template-ssh	
Yubico AES Authentication	aes128-yubico-authentication	
Yubico Asymmetric Authentication	ecp256-yubico-authentication	
Yubico OTP AES128	aes128-yubico-otp	
Yubico OTP AES192	aes192-yubico-otp	
Yubico OTP AES256	aes256-yubico-otp	

5.3 YubiHSM 2 Connector

The yubihsm-connector performs the communication between the YubiHSM 2 and the applications that use it.

The Connector must have permissions to access the USB device, and different operating systems behave differently in this regard. The easiest way to get started is to run the Connector with Administrator privileges (e.g. with `sudo`), but the safest way to run the Connector is to use your operating system's configuration to give it only the privileges necessary to access the YubiHSM 2 USB device.

The Connector is not a trusted component. Sessions are established cryptographically between the application and the

YubiHSM 2 using a symmetric mutual authentication scheme that is both encrypted and authenticated.

The Connector is not required to run on the same host as the applications which access it. In that case, configure the Connector to listen on a different address rather than the default `localhost:12345`. Make sure that the client has access. The port number does not need to change, only the address. Also, make sure that OS firewalls are configured properly to allow access to the host machine on the specified port.

To get information regarding the Connector issue a GET request on the `/connector/status` URI.

5.3.1 HTTPS Connections

As mentioned earlier, the Connector is not meant to be a trusted component. For this reason it defaults to HTTP connections. It is possible to use HTTPS, however this requires providing a key and a certificate to the Connector.

Another option is to use a reverse proxy such as nginx before the Connector and have that handle TLS.

5.3.2 Sample Configuration

Sample configuration for the Connector: `yubihsm-connector-config.yaml`

```
# Certificate (X.509)
cert: ""

# Certificate key
key: ""

# Listening address. Defaults to "localhost:12345".
listen: localhost:12345

# Device serial in case of multiple devices
serial: ""

# Log to syslog/eventlog. Defaults to "false".
syslog: false

# Use to enable host header filtering. Default to "false".
# Use this if there is an absolute need to use a web browser on the
# host where the YubiHSM 2 is installed to connect to untrusted web
# sites on the Internet.
enable-host-whitelist: false

# Default list for the host header filter
host-whitelist: localhost,localhost.,127.0.0.1,[::1]
```

Sample udev rule to be placed into `/etc/udev/rules.d/`

```
#This udev file should be used with udev 188 and newer
ACTION!="add|change", GOTO="yubihsm2_connector_end"

# Yubico YubiHSM 2
# The OWNER attribute here has to match the uid of the process
# running the Connector
SUBSYSTEM=="usb", ATTRS{idVendor}=="1050", ATTRS{idProduct}=="0030",
```

(continues on next page)

(continued from previous page)

```
  OWNER="yubihsm-connector"

LABEL="yubihsm2_connector_end"
```

5.4 YubiHSM Wrap

Yubihsm Wrap is a tool that allows the creation of importable objects offline. This is useful when bootstrapping secrets, for example on an air-gapped computer.

The tool requires an unencrypted Wrap Key in binary format and uses that to wrap objects with given Type, *Objects*, *ALGORITHMS*, *Object ID*, *Capability* and, where applicable, Delegated Capabilities.

For the resulting Object to be successfully imported on a YubiHSM 2, the Wrap Key used by `yubihsm-wrap` must already be present on the device.

Currently not all Object Types are supported. Refer to Known Issues and Limitations for more information.

5.5 Libyubihsm

Libyubihsm is the C library used to communicate natively with a YubiHSM 2. It implements and exposes convenience functions for all the commands supported by the device. It also allows the sending of unformatted "raw" messages over an established session or in plain text.

The library is used by:

- **yubihsm-shell**, see YubiHSM Shell Reference
- **PKCS#11 module**, see PKCS#11 with YubiHSM 2 Reference
- **KSP**, see Key Storage Provider Reference

Documentation of the library API can be found as comments within the header file (`yubihsm.h`) in the SDK, or as a pre-built Doxygen bundle.

Libyubihsm includes a connector component to talk to a YubiHSM device. This connector is different from the yubihsm-connector discussed earlier. This component can be one of the following two types.

5.5.1 HTTP Connector

This kind of Connector talks to yubihsm-connector over http(s), allowing remote access to a YubiHSM2, see Connector Reference

In order to select this type of backend the connector URL should use the `http` or `https` scheme; for example, to use a local HTTP Connector use `http://127.0.0.1:12345`.

5.5.2 USB Connector

This kind of Connector is a direct-access USB backend that talks directly with a YubiHSM device. The USB Connector is built into `libyubihsm`. This renders it unnecessary to run an additional component (i.e., the external Connector) at the cost of requiring exclusive access to a YubiHSM device.

To select this type of backend the connector URL should use the `yhusb` scheme. For example, to use a local device with serial number 123456 use `yhusb://serial=123456`.

5.6 Python Library

The Python library allows you to interface with a YubiHSM 2 through both the Connector service and direct USB connection using the Python programming language. It supports Python 3.8 or later.

The recommended way to install the library is by using `pip` inside a `virtualenv`. To create and activate a `virtualenv`, run:

```
$ virtualenv yubihsm
Running virtualenv with interpreter /usr/bin/python3
New python executable in /home/user/yubihsm/bin/python3
Also creating executable in /home/user/yubihsm/bin/python
Installing setuptools, pkg_resources, pip, wheel...done.

$ source yubihsm/bin/activate
(yubihsm) $ pip install yubihsm[http,usb]
Collecting yubihsm-2.0.0
...
Successfully installed asn1crypto-0.22.0 cffi-1.10.0 cryptography-1.8.1
   enum34-1.1.6 idna-2.5 ipaddress-1.0.18 pycparser-2.17 pyusb-1.0.2
   requests-2.13.0 yubihsm-2.0.0
(yubihsm) $
```

Note: The `cryptography` dependency uses C extensions, and therefore has some build dependencies. For detailed instructions, see: https://cryptography.io/en/latest/installation/

```
from yubihsm import YubiHsm
from yubihsm.objects import AsymmetricKey
from yubihsm.defs import ALGORITHM, CAPABILITY

# Connect to the Connector and establish a session using the default
# auth key:
hsm = YubiHsm.connect("http://localhost:12345")
session = hsm.create_session_derived(1, "password")

# Create a new EC key for signing:
key = AsymmetricKey.generate(session, 0, "EC Key", 1, CAPABILITY.SIGN_ECDSA, ALGORITHM.
↪EC_P256)

# Sign a message
data = b'Hello world!'
signature = key.sign_ecdsa(data)
```

(continues on next page)

(continued from previous page)

```
# Delete the key from the YubiHSM 2
key.delete()

# Close session and connection:
session.close()
hsm.close()
```

5.7 Key Storage Provider (KSP) - Windows Only

The Key Storage Provider (KSP) for Windows Cryptography API: Next Generation (CNG) has been thoroughly tested with Active Directory Certificate Services (AD CS) plus 2048-bit, 3072-bit, and 4096-bit keys. It also works with other types of keys, but those have not been tested to the same extent.

The following installs the KSP and the Connector Service, using them for ADCS with the default Authentication Key (1) and password (`password`).

When you run the `Install-AdcsCertificationAuthority` command, you should see the YubiHSM 2 light flash rapidly, because AD CS uses the KSP to generate a 2048-bit key in hardware. For AD CS to work properly, `Restart-Computer` may be needed.

```
PS1> msiexec /i "yubihsm-connector-windows-amd64.msi" /passive ACCEPT=yes
PS1> msiexec /i "yubihsm-cngprovider-windows-amd64.msi" /passive  ACCEPT=yes
PS1> Install-WindowsFeature AD-Certificate -Verbose
PS1> Install-AdcsCertificationAuthority -CAType EnterpriseRootCa    \
    -CryptoProviderName "RSA#YubiHSM Key Storage Provider"          \
    -KeyLength 2048 -HashAlgorithmName SHA256 -ValidityPeriod Years \
    -ValidityPeriodUnits 5
PS1> Install-AdcsOnlineResponder
```

If you are using a different Authentication Key, password, or Connector for the KSP, you can specify them as follows (defaults are shown):

```
PS1> Set-ItemProperty -path HKLM:\SOFTWARE\Yubico\YubiHSM \
    -name ConnectorURL -Type String -Value http://127.0.0.1:12345
PS1> Set-ItemProperty -path HKLM:\SOFTWARE\Yubico\YubiHSM \
    -name AuthKeysetPassword -Type String -Value password
PS1> Set-ItemProperty -path HKLM:\SOFTWARE\Yubico\YubiHSM \
    -name AuthKeysetID -Type DWord -Value 1
```

Warning: Design considerations for Key Storage Providers in Windows prevent the direct USB functionality of libyubihsm (Connector URL `yhusb://`), therefore it is not supported in this version of the YubiHSM KSP.

The default configuration for the connector is: `ProgramData\YubiHSM\yubihsm-connector.yaml` - Administrator rights are required to access the file.

5.7.1 Additional Documentation for YubiHSM Key Storage Provider

- For instructions on how to move a software-based key into the YubiHSM 2 for use with the KSP, see Move Software Keys to Key Storage Provider.
- For an example of how to create an HSM-backed code signing certificate for Windows through the KSP, see *Example: Creating a Code-Signing Certificate using the Key Storage Provider*.
- For more information about status codes, see YubiHSM 2 status codes in Windows.
- For details on how to configure the 32-bit and 64-bit KSP DLLs, please see *YubiHSM 2 with Key Storage Provider for Windows Server*.

5.8 YubiHSM Auth

YubiHSM Auth is a new YubiKey module that serves as a key storage for authenticating against a YubiHSM 2 with a YubiKey instead of just using a session password alone. To leverage this functionality, use the latest release of YubiHSM 2 SDK.

YubiHSM Auth is a YubiKey CCID application that stores the long-lived credentials used to establish secure sessions to a YubiHSM 2. The secure session protocol is based on Secure Channel Protocol 3 (SCP03). YubiHSM Auth is supported by YubiKey v5.4.0 and higher.

YubiHSM Auth uses hardware to protect the long-lived credentials for accessing a YubiHSM 2. This increases the security of the authentication credentials, as compared to the authentication solution for the YubiHSM 2 based on software credentials derived from the Password-Based Key Derivation Function 2 (PBKDF2) algorithm with a password as input.

Note: SCP03 is always used, with yubihsm-auth or not. This means that authentication is always based on a pair of 128 bit AES keys. These keys can be derived from a password on the client side, using authentication in the Yubico command line tools.

5.8.1 Credentials and PIN Codes

Each YubiHSM Auth credential consists of two AES-128 keys which are used to derive the three session-specific AES-128 keys. The YubiHSM Auth application can store up to 32 YubiHSM Auth credentials in the YubiKey.

Each YubiHSM Auth credential is protected by a 16-byte user access code provided to the YubiKey for each YubiHSM Auth operation. The access code is used to access the YubiHSM Auth Credential to derive the session-specific AES-128 keys.

Storing or deleting YubiHSM Auth credentials requires a separate 16-byte admin access code.

Each access code has a limit of eight retries and optionally, verification of user presence (touch).

5.8.2 YubiHSM 2 Secure Channel

YubiKey YubiHSM Auth application can be used to establish an encrypted and authenticated session to a YubiHSM 2. Although the YubiHSM 2 secure channel is based on the protocol Global Platform Secure Channel Protocol **03** (SCP03), there are two important differences:

- The YubiHSM 2 secure channel protocol does not use APDUs, so the commands and possible options are not those of the complete SCP03 specification.
- SCP03 uses key sets with three long-lived AES keys. Two of these long-lived keys are used for authentication and the third is used to encrypt new long-lived keys when they're transferred to the device. Since YubiHSM handles authentication keys like any other keys, the third SCP03 long-lived key is not required therefore YubiHSM 2 secure channel uses key sets with two long-lived AES keys which are required for authentication.

The YubiHSM 2 authentication protocol uses a set of static credentials called a long-lived key set. This consists of two AES-128 keys:

- **ENC**: Used for deriving keys for command and response encryption, as specified in SCP03.
- **MAC**: Used for deriving keys for command and response authentication, as specified in SCP03.

The identical long-lived keyset is protected in the YubiHSM 2 and in the YubiKey YubiHSM Auth application.

Those long-lived key sets are used by the YubiHSM Auth application to derive a set of three session-specific AES-128 keys using the challenge-response protocol as defined in SCP03:

- **Session Secure Channel Encryption Key (S-ENC)**: Used for data confidentiality.
- **Secure Channel Message Authentication Code Key for Command (S-MAC)**: Used for data and protocol integrity.
- **Secure Channel Message Authentication Code Key for Response (S-RMAC)**: Used for data and protocol integrity.

The YubiHSM Auth session-specific keys are output from the YubiKey to the calling library, which uses the session keys to encrypt and authenticate commands and responses during a single session. After the session is over the session keys are discarded. Session keys are only used for a single session and are not sensitive after the session is over.

5.8.3 Architecture Overview

The figure below shows how the YubiHSM Auth application fits in to the YubiHSM 2 architecture.

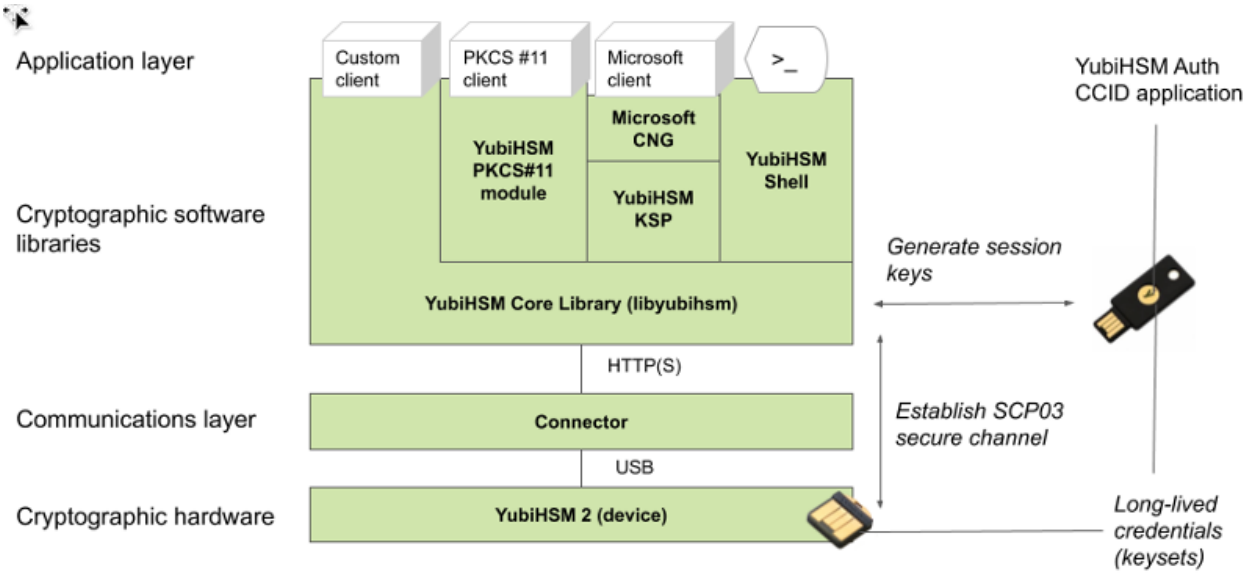


Figure: Architecture Overview

The identical long-lived credentials (key sets) are protected in both the YubiKey YubiHSM Auth application and in the YubiHSM 2. The YubiHSM-Shell software tool can be used for generating the key sets in the YubiHSM 2, and the YubiHSM-Auth software tool can be used for importing the same key sets to the YubiKey YubiHSM Auth application.

At the client, the YubiHSM authentication protocol is implemented in the `libykhsmauth` library, which derives the three session AES-keys by calling the YubiKey YubiHSM Auth CCID application. The session objects that are created can be used by the `libyubihsm` in the communication with YubiHSM.

The YubiHSM session keys are therefore generated on the basis of the long-lived credentials that are protected in the YubiHSM 2 and YubiKey YubiHSM Auth in conjunction with the SCP03 derivation scheme.

5.8.4 YubiHSM Auth Flowchart

The flowchart below illustrates the authentication protocol communication with YubiHSM using the static keys on YubiHSM Auth. It is assumed that the YubiHSM and YubiHSM Auth application share the same static keyset. The steps are explained below.

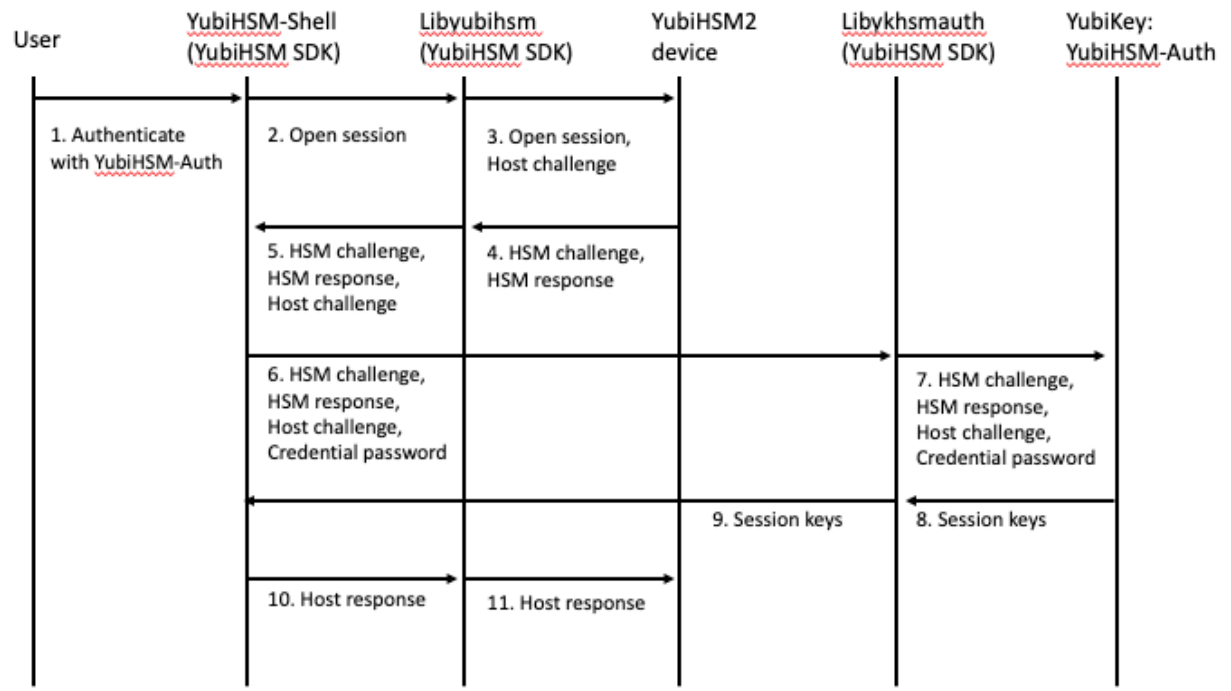


Figure: YubiHSM Auth Flowchart

The following is a description of the steps in the flowchart.

1. The user launches YubiHSM-Shell and enters the commands `connect` and `session open`, with the flag `ykopen` that indicates that the YubiKey with YubiHSM Auth shall be used.
2. The YubiHSM-Shell invokes the `libyubihsm` library, with a request to open a session to the YubiHSM 2.
3. The `libyubihsm` library generates a host challenge and opens a session to the YubiHSM 2 device.
4. The YubiHSM 2 device generates an HSM challenge and generates the session keys based on the HSM challenge, the host challenge, and the static key set in the YubiHSM 2 device. The YubiHSM 2 returns the HSM challenge in an HSM response to the `libyubihsm` library.
5. The `libyubihsm` library propagates the host challenge and HSM challenge to the YubiHSM Shell.
6. The user enters the Credential password for unlocking the static keyset in the YubiHSM Auth application in the YubiKey. The YubiHSM Shell invokes the `libykhsmauth` library, with a request to generate session keys.
7. The `libykhsmauth` library invokes the YubiHSM Auth application in the YubiKey with the Credential password, the HSM challenge and host challenge are used as input parameters.
8. The Credential password unlocks the static keyset in the YubiHSM Auth application, and the YubiHSM Auth application generates the session keys based on the static keys, HSM challenge, and host challenge.
9. The `libykhsmauth` library returns the session keys to YubiHSM Shell.
10. The YubiHSM Shell acknowledges the protocol handshake to `libyubihsm`.
11. The `libyubihsm` sends the host response to the YubiHSM 2 device. The session keys can now be used for secure channel communication between `YubiHSM-Shell/libyubihsm` in the host and the YubiHSM device.

5.8.5 YubiHSM-Auth Software Tool

The YubiHSM-Auth software tool is part of the YubiHSM Shell, which is installed with the YubiHSM SDK. YubiHSM-Auth tool can be used for:

- Storing the YubiHSM Auth credentials on a YubiKey
- Deleting the YubiHSM Auth credentials on a YubiKey
- Listing the YubiHSM Auth credentials on a YubiKey
- Changing the YubiHSM Auth management key on a YubiKey
- Checking the number of retries of the YubiHSM Auth credential password
- Checking the version of the YubiHSM Auth application
- Calculating session keys, mainly for debugging and test purposes
- Resetting the YubiHSM Auth application on a YubiKey

First, the YubiHSM 2 device needs to be configured with an authentication key. The default authentication key password on `KeyID=1` is set to `password`, and this should be changed or replaced with other authentication keys. For the examples in this section, however, it is assumed that the default authentication key is still present on the YubiHSM 2.

In order to generate and store the equivalent YubiHSM Auth credentials on the YubiKey, the `yubihsm-auth` command line tool can be used. To invoke YubiHSM-Auth simply run `yubihsm-auth` with the required commands and parameters.

To get a list of available commands, parameters and their syntax, run:

```
yubihsm-auth --help
```

An example of how to use `yubihsm-auth` for storing YubiHSM Auth credentials on a YubiKey is shown below:

```
$ yubihsm-auth -a put --label="default key" --derivation-password="password" --credpwd=
↪"MyPassword" --touch=on --mgmkey="000000000000000000000000000000000000" --verbose=5
Credential successfully stored
```

where –

`-a put` is the action to insert a YubiHSM Auth credential on the YubiKey

`--label` is the label of the YubiHSM Auth credential on the YubiKey

`--derivation-password` is used as input to the PBKDF2 algorithm, which is used for generating the two AES-128 keys that constitute the YubiHSM Auth credentials to be stored on the YubiKey

`--credpwd` is the password protecting the YubiHSM Auth credentials on the YubiKey

`--touch` is set to 'on', which requires the user to touch the YubiKey when accessing the YubiHSM Auth credential

`--mgmkey` is the management key that is needed for writing the YubiHSM Auth credentials on the YubiKey

`--verbose` is used to print more information as output

Note: We recommend using an offline air-gapped computer when storing the YubiHSM Auth credentials on the YubiKey. Now the YubiKey YubiHSM Auth application can be used with YubiHSM Shell for authentication to the YubiHSM 2.

5.8.6 Using YubiHSM-Auth with YubiHSM Shell

It is now possible to authenticate to the YubiHSM 2 device with static credentials that are protected in the YubiKey application called YubiHSM Auth. For more information on this YubiKey feature and how to configure it, see Using YubiHSM Auth.

The YubiHSM Shell tool supports authentication with YubiHSM Auth credentials in both interactive mode and command line mode.

In order to use yubihsm-shell with the YubiHSM Auth-enabled YubiKey in interactive mode, open a session by executing the following yubihsm-shell command:

`yubihsm> session ykopen <authkey> <label> <password>`

Where, in the context of using YubiHSM-Shell with the YubiHSM Auth application, the following parameters are used:

`authkey` is the identifier of the authentication key in the YubiHSM 2

`label` is the label of the YubiHSM-Auth credetnials stored in the YubiKey

`password` is the password that protects the YubiHSM-Auth credentials stored in the YubiKey.

Below is an example of an interactive command with YubiHSM Shell:

```
yubihsm> session ykopen 1 "default key" "MyPassword"
trying to connect to reader 'Yubico YubiKey OTP+FIDO+CCID 0'
Created session 0
```

To use yubihsm-shell with YubiHSM Auth in command line mode, add the parameter `--ykhsmauth-label` that implicitly invokes the YubiHSM Auth application at the YubiKey. Below is an example of how to use YubiHSM Shell in command line mode:

```
$ yubihsm-shell --ykhsmauth-label "default key" -p "MyPassword"
   -a generate-asymmetric -A rsa2048 -i 11 -c sign-pss -l Signature_Key``
```

If the YubiKey is configured to require touch when accessing the YubiHSM-Auth credentials, the user needs to touch the YubiKey sensor in addition to entering the credential password.

Once the user is authenticated with YubiHSM Auth, all YubiHSM-Shell commands can be used.

CHAPTER

SIX

YUBIHSM 2: BACKUP AND RESTORE

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

The examples below assume the default authentication key (0x0001). If you use some other authentication key make sure that it has the capability `put-wrap-key` and has the correct delegated capabilities, otherwise you will get a `wrong permissions for operation` error.

You can perform these operations using:

- YubiHSM Shell for backing up and restoring
- YubiHSM Setup for backing up and restoring
- YubiHSM Key Storage Provider for backing up and restoring certificate as well as private key.

In all three cases, the process is done by taking the following steps:

1. Create a wrap key, call it *wrapkey*.
2. Import *wrapkey* into the primary YubiHSM2.
3. Export other objects in the primary YubiHSM2 using *wrapkey*.
4. Import *wrapkey* into the backup YubiHSM2.
5. Import the objects exported in step 3 into the backup YubiHSM2.

In order for a full backup to be successful, the following conditions need to be fulfilled (any object that does not fulfill these conditions is not exported):

- *wrapkey* is accessible in all the domains the other objects are available in.
- *wrapkey* has delegated capabilities that include all the capabilities any other object has.
- *wrapkey* has the capabilities `export-wrapped` and `import-wrapped`.
- All other objects have the capability `exportable-under-wrap`.

6.1 Backup and Restore Using YubiHSM Shell

6.1.1 Backup

1. For the purpose of this guide, we will start by generating an asymmetric key that we will then make a backup of.

```
$ yubihsm-shell -a generate-asymmetric-key -A rsa2048 --capabilities exportable-
↪under-wrap, sign-pkcs,decrypt-pkcs
...
```

(continues on next page)

(continued from previous page)

```
Generated Asymmetric key 0x6e77
...
```

OBS: This will generate an asymmetric key accessible in all domains.

2. Start by getting a pseudo random number from the YubiHSM2 and store it in a file. This will be the wrap key.

```
$ yubihsm-shell -a get-pseudo-random --count=32 --out=wrap.key
```

Important: The file `wrap.key` here contains the Wrap Key loaded into your YubiHSM in clear text. It should therefore be considered sensitive.

---

3. Import wrap.key into the primary YubiHSM2.

```
...
yubihsm-shell -a put-wrap-key --capabilities export-wrapped,import-wrapped --
↪delegated=sign-pkcs, decrypt-pkcs,exportable-under-wrap --in=wrap.key
...
Stored Wrap key 0xd581
```

OBS: This will import a wrap key accessible in all domains.

4. Make an encrypted backup of the Asymmetric Key 0x6e77 in the file key_6e77.yhw.

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

6.1.2 Restore

This assumes a fresh device where you want to restore the previously backed up `key 0x6e77`.

1. Import the wrap key into the backup YubiHSM2.

```
$ yubihsm-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
```

2. Import the Asymmetric key 0x6e77 into the backup YubiHEM2.

```
yubihsm-shell -a put-wrapped --wrap-id=0xd581 --in=key_6e77.yhw
...
Object imported as 0x6e77 of type asymmetric-key
```

6.2 Backup and Restore Using YubiHSM Setup

The *YubiHSM 2 Setup Tool* can be used to backup and restore all exportable objects simultaneously.

6.2.1 Backup

OBS: This assumes that a wrap key fulfilling all the conditions mentioned above already exists in the primary YubiHSM2. For the following command line examples, we will assume that such a key has ObjectID 0xd581.

```
$ yubihsm-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
```

Note: When creating a wrap key using `yubihsm-setup` with the subcommand `ksp` or `ejbca`, an option is presented to split the wrap key into shares to be held by different custodians. It would also be possible to set the minimum number of custodians required to reconstruct the wrap key.

Important: Split and reconstruction of the wrap key is done in the software (yubihsm-setup). The YubiHSM2 itself is not aware of such split or any shares.

6.2.2 Restore

Running the store command will import all `\*.yhw` files in the current directory. If some of those files are not encrypted/wrapped with a wrap key that exists in the backup YubiHSM2, they will not be imported.

```
$ yubihsm-setup restore
```

Note: If the wrap key was split, the shares to reconstruct it will need to be provided in this step.

6.3 Backup and Restore Using YubiHSM KSP (Windows Only)

YubiHSM Key Storage Provider (KSP) enables backing up and restoring the keys managed using this tool.

Note: Microsoft Active Directory Certificate Services (ADCS) does not set the `NCRYPT_ALLOW_EXPORT_FLAG` when generating a key, either through the setup UI or the `Install-ADCSCertificationAuthority` PowerShell module.

When creating an ADCS root CA key using the YubiHSM 2, we add the `exportable-under-wrap` Capability by default. Backup and restore functionality is therefore available using the following manual processes.

1. *Identify Your Private Key Container Name*
2. *Backup the Target Certificate*

3. *Backup the Target Private Key*
4. *Restore the Target Private Key*
5. *Restore the Target Certificate*

6.3.1 Identify Your Private Key Container Name

1. To view the currently installed certificates in the Local Machine "My" store, open an elevated command prompt/shell by using the `certutil` command.

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

2. Find the target certificate in the list and then find its `Key Container` property. The Provider property should be the same as `YubiHSM Key Storage Provider`.
3. To identify the certificate, record the `Cert Hash` property.

6.3.2 Backup the Target Certificate

Using any available means (`certmgr.msc`, PowerShell, `certutil`), export the target certificate, but without the private key in DER format.

Note: 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.

For example, to export the certificate in `.crt ``format to a file named ``<Cert Hash>.crt`, use the command.

```
PS1> certutil -split -store My <Cert Hash>.
```

6.3.3 Backup the Target Private Key

Export the target private key with the `label` property equal to the `Key Container` property.

1. Use an Authentication Key with the `export-wrapped` capability set.
2. Use the instructions for exporting a private key under wrap via `yubihsm-shell` (see *Backup and Restore Using YubiHSM Shell*).

6.3.4 Restore the Target Private Key

Import the target private key file to your backup YubiHSM.

1. Use an Authentication Key with the `import-wrapped` capability set.
2. Use the instructions for importing a private key under wrap via `yubihsm-shell` (see *Backup and Restore Using YubiHSM Shell*).

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

6.3.5 Restore the Target Certificate

Before the certificate is imported to the local machine, it does not have an associated private key.

1. Move the target certificate file generated as per *Backup and Restore Using YubiHSM Shell* to the target machine by importing the certificate to the LocalMachine "My" store. Use your preferred method.
2. Re-associate the certificate to the private key by using the `-repairstore` functionality of `certutil`.
3. Verify that the target private key is visible via the YubiHSM KSP: list all private keys (and their corresponding container names - which are equal to the `Label` property in the YubiHSM visible to the current Authentication Key).

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

4. Open an elevated prompt and execute the command:

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

5. To verify that the certificate has been associated with the YubiHSM Key Storage Provider and has the correct `Key Container` property value, repeat the steps under *Identify Your Private Key Container Name*.

CHAPTER

SEVEN

INITIAL PROVISIONING AND DEPLOYMENT GUIDE

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 that it is removed (single use case) or changed prior to production deployment.

7.1 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 (single use case) or changed should a new use case arise.

7.2 HMAC

1. Establish a session with the default Authentication Key.

```
yubihsm> connect
         Session keepalive set up to run every 15 seconds
yubihsm> session open 1 password
         Created session 0
```

2. Create an Authentication Key for Auditing.

```
yubihsm> put authkey 0 0 "Audit auth key" all get-log-entries none
         $AUDIT_PASS
         Stored Authentication key 0xd054
```

3. Create a Wrap Key for importing application Authentication Keys and secrets.

```
yubihsm> get random 0 16
         5b61e89468cc8f2a274715c78c3d4753
yubihsm> put wrapkey 0 0 "HMAC wrap Key" all import-wrapped
```

(continues on next page)

(continued from previous page)

```
        sign-hmac:verify-hmac 5b61e89468cc8f2a274715c78c3d4753
        Stored Wrap key 0xf09a
```

4. 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
```

5. Delete the default Authentication Key.

```
yubihsm> delete 0 1 authentication-key
```

6. 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' > 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' > 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
```

7. Open a Session with the wrap Authentication Key.

```
yubihsm> session open 0xf10f $WRAP_PASS
        Created session 1
```

8. 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
```

9. Open a session with the new application Authentication Key.

```
yubihsm> session open 0x2a74 $HMAC_PASS
        Created session 2
```

10. Run HMAC-SHA256 Test vector #1 and get expected output.

```
yubihsm> hmac 2␣
↪0xd1a2b0344c61d8db38535ca8afceaf0bf12b881dc200c9833da726e9376c2e32cff7
echo -ne '\x48\x69\x20\x54\x68\x65\x72\x65' | openssl dgst -hex -mac hmac -
↪macopt hexkey:0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b -sha256 (stdin)=␣
↪b0344c61d8db38535ca8afceaf0bf12b881dc200c9833da726e9376c2e32cff7
```

7.3 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 with YubiHSM 2* a `yubihsm\_pkcs11.conf` file needs to exist and point at the desired connector.

1. Establish a Session with the default Authentication Key.

```
yubihsm> connect
         Session keepalive set up to run every 15 seconds
yubihsm> session open 1 password
         Created session 0
```

2. Create an Authentication Key for Auditing.

```
yubihsm> put authkey 0 0 "Audit auth key" all audit none $AUDIT_PASS
         Stored Authentication key 0xd054
```

3. Optionally enable forced audits.

```
yubihsm> put option 0 force-audit 01
```

4. 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
         Stored Authentication key 0xf10f
```

5. Delete the default Authentication Key.

```
yubihsm> delete 0 1 authentication-key
```

6. 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
```

CHAPTER

EIGHT

FIPS MODE SUPPORT GUIDE

Note: This guide only applies to YubiHSM 2 FIPS devices.

8.1 Putting YubiHSM 2 into FIPS Mode

To configure the YubiHSM 2 into the FIPS Approved mode of operation:

1. Use the `Set Option` service as follows: 4f000405000101 or

```
put option 0 fips-mode 01
```

2. Import new Authentication Keys to replace the default values.

8.2 Validating the Mode

To check the mode of operation, use the `Get Option` service.

```
get option 0 fips-mode
```

where-

`01` return code indicates the Approved mode.

`00` return code indicates the non-Approved mode.

8.3 Taking it out of FIPS Mode

To configure the YubiHSM 2 into the non-Approved mode of operation.

1. Delete all objects on the YubiHSM 2.
2. Use the `Set Option` service as follows: 4f000405000100 or

```
put option 0 fips-mode 00
```

CHAPTER

NINE

USING KEY STORAGE PROVIDER (KSP) - WINDOWS ONLY

If the target private key is managed by the Microsoft Software Key Storage Provider, another software provider, or any other KSP that allows export via PKCS#12 PFX, it is possible to move your key to the YubiHSM 2, but results may vary.

This process relies on using the `-repairstore` functionality of the `certutil` command, so the private key must only be present via the YubiHSM Key Storage Provider when performing this step. Please refer to the source storage provider documentation for how to cleanly and completely delete a private key.

Because KSP implementations differ, we recommend testing this procedure using your existing provider before affecting a live system.

9.1 Export your Existing Private Key and Certificate

Refer to your current KSP documentation on how to obtain a PKCS#12 PFX export of your certificate and private key.

1. Obtain your PFX file.
2. Split the certificate from the PFX file using `certutil`.

```
   PS1> certutil -split -dump <pfx file>


This creates a file named ``<Cert Hash>.crt``.
```

3. If you are moving the key to the YubiHSM 2 on the same machine, you must delete the original private key in your current provider.

```
PS1> certutil -key
```

4. Locate the key that corresponds with the CA. It may look something like this:

```
Microsoft Software Key Storage Provider:

EXAMPLE-CA abcdef1234fedcba4321abcdef123456_9cfc1053-1b5a-44d7-8a7e-3a8a1c0d0db0␣
↪RSA AT_KEYEXCHANGE
```

5. To delete this example private key.

```
PS1> certutil -delkey -csp "Microsoft Software Key Storage Provider"
↪"abcdef1234fedcba4321abcdef123456_9cfc1053-1b5a-44d7-8a7e-3 a8a1c0d0db0"
```

9.2 Import the Target Private Key

Using the instructions for importing a PFX private key, see *PUT ASYMMETRIC KEY Command* via `yubihsm-shell`, import the target private key file to your YubiHSM 2.

1. Record the `Label` property of your imported key.

   Important: The `certutil` utility does not provide an easy way to split a key exported from the Software KSP into an unencrypted PEM file. It may be necessary to use another tool like OpenSSL to convert the key file to an unencrypted format for import into the HSM.

2. Export the private key.

```
PS1> openssl pkcs12 -in <pfx file> -nocerts -out ca.key -nodes
```

3. To remove the passphrase from the private key.

```
PS1> openssl rsa -in ca.key -out ca.key
```

9.3 Restore the Target Certificate

1. Move the target certificate file (`<Cert Hash>.crt`) to the target machine.
2. Import the certificate to the LocalMachine "My" store via your favorite method.

   At this point, the certificate does not have an associated private key. We use the `-repairstore` functionality of `certutil` to re-associate the certificate to the private key.

3. Make sure that the target private key is visible via the YubiHSM KSP.

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

   This command lists all private keys visible to the current Authentication Key. It also lists the private keys corresponding container names - which are equal to the *Label* property in the YubiHSM 2.

4. Open an elevated prompt and execute the command.

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

5. Verify that the certificate has been associated with the YubiHSM KSP and has the correct `Key Container` property value.

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

6. Inspect the Key Container and Provider properties.

   Warning: If you are moving your CA key to the YubiHSM 2 on the same machine, Windows Certificate Services (CertSvc) on the local machine writes the name of the KSP to its configuration section in the registry. When signing requests, the certificate service will fail if the KSP name does not match the name in the registry.

7. Update the KSP name for the local certificate service.

   - Open an elevated prompt and execute the commands.

```
PS1> certutil -setreg CA\CSP\Provider "YubiHSM Key Storage Provider"
PS1> certutil -setreg CA\EncryptionCSP\Provider "YubiHSM Key Storage
     Provider"
```

- Optionally, if you have multiple CAs on the same machine, or prefer to edit the registry directly. These settings are located at:

  ```
  HKLM\System\CurrentControlSet\Services\CertSVC\Configuration\<CAName>\
  [CSP | EncryptionCSP]
  ```

9.4 Status Codes Reference

The YubiHSM software components have a standard set of status codes to report the status of an HSM operation. To comply with the expectations of specific platforms, these status codes are converted to the appropriate API status code.

Currently, this translation is only performed for the Windows Key Storage Provider. The error codes, their meanings and translated values are as follows.

Libyubihsm Error Code	Description	Windows CNG Translation
YHR_BUFFER_TOO_SMALL	Not enough space to store data	NTE_BUFFER_TOO_SMALL
YHR_CONNECTION_ERROR	Transport Backend error	NTE_DEVICE_NOT_READY
YHR_CONNECTOR_ERROR	Connector operation Failed	NTE_DEVICE_NOT_READY
YHR_CONNECTOR_NOT_FOUNI	Unable to find a suitable connector	NTE_DEVICE_NOT_READY
YHR_CRYPTOGRAM_MISMATCI	Unable to verify cryptogram	NTE_BAD_SIGNATURE
YHR_DEVICE_AUTHENTICATIOI _FAILED	Message encryption / verification failed	NTE_INCORRECT_PASSWORD

continues on next page

Table 1 – continued from previous page

Libyubihsm Error Code	Description	Windows CNG Translation
YHR_DEVICE_COMMAND_UNEXECUTED	The HSM attempted to execute a command, but it did not complete in allotted time. The command has not terminated, and the current state of the session is unavailable	NTE_SYS_ERR
YHR_DEVICE_DEMO_MODE	Demo mode, power cycle device	NTE_DEVICE_NOT_READY
YHR_DEVICE_INSUFFICIENT_PERMISSIONS	Wrong permissions for operation	NTE_PERM
YHR_DEVICE_INVALID_COMMAND	Invalid command	NTE_NOT_SUPPORTED
YHR_DEVICE_INVALID_DATA	Malformed command / invalid data	NTE_INVALID_PARAMETER
YHR_DEVICE_INVALID_ID	Illegal ID used	NTE_INVALID_PARAMETER[]
YHR_DEVICE_INVALID_OTP	Invalid OTP	NTE_INCORRECT_PASSWORD
YHR_DEVICE_INVALID_SESSION	Invalid session	NTE_DEVICE_NOT_READY
YHR_DEVICE_LOG_FULL	Log buffer is full and forced audit is set	NTE_DEVICE_NOT_READY
YHR_DEVICE_OBJECT_EXISTS	An object with the specified ID already exists	NTE_EXISTS

continues on next page

Table 1 – continued from previous page

Libyubihsm Error Code	Description	Windows CNG Translation
YHR_DEVICE_OBJECT_NOT_FOUND	Object not found	NTE_NOT_FOUND
YHR_DEVICE_OK	No error	NTE_OP_OK
YHR_DEVICE_SESSION_FAILED	Session creation failed	NTE_DEVICE_NOT_READY
YHR_DEVICE_SESSIONS_FULL	All sessions are allocated	NTE_DEVICE_NOT_READY
YHR_DEVICE_STORAGE_FAILED	Storage failure	NTE_TOKEN_KEYSET_STORAGE_FULL
YHR_DEVICE_WRONG_LENGTH	Wrong length	NTE_BAD_LEN
YHR_GENERIC_ERROR	Generic error	NTE_FAIL
YHR_INIT_ERROR	Unable to initialize libyubihsm	NTE_PROVIDER_DLL_FAIL
YHR_INVALID_PARAMETERS	Invalid argument to a function	NTE_INVALID_PARAMETER
YHR_MAC_MISMATCH	Unable to verify MAC	NTE_BAD_SIGNATURE
YHR_MEMORY_ERROR	The YubiHSM or software library was not able to allocate memory to perform the requested operation	NTE_NO_MEMORY
YHR_SESSION_AUTHENTICATION_FAILED	Unable to authenticate session	NTE_INCORRECT_PASSWORD
YHR_SUCCESS	The operation completed Successfully	ERROR_SUCCESS

continues on next page

Table 1 – continued from previous page

Libyubihsm Error Code	Description	Windows CNG Translation
YHR_WRONG_LENGTH	This error may occur if there is a mismatch between the YubiHSM firmware version and libyubihsm library version	NTE_BAD_LEN

9.5 Example: Creating a Code-Signing Certificate using the Key Storage Provider

This example will show you how to create a code-signing certificate request using a key generated and stored in the YubiHSM 2 via the Key Storage Provider (KSP). This type of code-signing certificate is appropriate for use with the Microsoft `signtool` utility for digitally signing Windows binaries.

In this example, we use the command line `certreq` utility. All procedures documented here are available in the Certificate Manager (`certmgr.msc`) MMC snap-in if you prefer to use a GUI.

Note: For operations that take input data (from command line or file), releases prior to and including the current yubihsm2-sdk release have a size limit - 4kb in interactive mode, or 8kb in non-interactive mode.

9.5.1 Configure the Key Storage Provider

By default, the KSP will use the factory authentication key in slot 1. If the factory authentication key no longer exists or a different authentication key is desired, the KSP must first be configured with the desired key ID and password.

Note: The configured authentication key must at a minimum have the capabilities `generate-asymmetric-key`, `sign-pkcs`, and delegated capability `sign-pkcs`. If you want the generated key to be exportable, then add the `exportable-under-wrap` delegated capability.

9.5.2 Authentication Key Example

Create a new Authentication Key capable of generating exportable asymmetric keys through KSP.

```
yubihsm> put authkey 0 0 "GenerateKey" 1 generate-asymmetric-key,
         sign-pkcs sign-pkcs,exportable-under-wrap password
         Stored Authentication key 0x0e32
```

9.5.3 Create the Certificate Request Configuration File

To specify your request, the `certreq` utility requires an `.inf` file as input. An example file is supplied here.

Sample sign.inf

```
[Version]
 Signature="$Windows NT$"

[NewRequest]
 Subject = "CN=My Publisher"  ; Entity name (dns name/upn for other cert types)
 HashAlgorithm = sha256       ; Request uses sha256 hash
 KeyAlgorithm = RSA           ; Key pair generated using RSA algorithm
 Exportable = FALSE           ; Private key is not exportable
 ExportableEncrypted = FALSE  ; Private key is not exportable encrypted
 KeyLength = 2048             ; YubiHSM KSP key sizes: 2048, 3072, 4096
 KeySpec = 2                  ; 1 = AT_KEYEXCHANGE, 2 = AT_SIGNATURE
 KeyUsage = 0x80
     ; 80 = Digital Signature, 20 = Key Encipherment (bitmask)
 MachineKeySet = FALSE
     ; True: cert belongs the local computer, False: current user
 KeyUsageProperty = NCRYPT_ALLOW_SIGNING_FLAG
     ; Private key only used for signing, not decryption
 UseExistingKeySet = FALSE    ; Do not use an existing key pair
 ProviderName = "YubiHSM Key Storage Provider"
 ProviderType = 1
 SMIME = FALSE                ; No secure email function
 UseExistingKeySet = FALSE    ; Do not use an existing key pair
 RequestType = PKCS10         ; Can be CMC, PKCS10, PKCS7 or Cert (self-signed)

[Strings]
 szOID_ENHANCED_KEY_USAGE = "2.5.29.37"
 szOID_CODE_SIGN = "1.3.6.1.5.5.7.3.3"
 szOID_BASIC_CONSTRAINTS = "2.5.29.19"

[Extensions]
 %szOID_ENHANCED_KEY_USAGE% = "{text}%szOID_CODE_SIGN%"
 %szOID_BASIC_CONSTRAINTS% = "{text}ca=0&pathlength=0"

; If you are using ADCS with certificate templates, you may add
; a specific template under [RequestAttributes]
; [RequestAttributes]
;  CertificateTemplate= CodeSigning
```

9.5.4 Create the Certificate Request

Once you have created the certificate request configuration file, pass it to `certreq` as the input file argument. For example:

```
certreq -new sign.inf sign.req
```

9.5.5 Sign the Certificate Request

In the above example, the certificate request was written to `sign.req`.

1. Take this file and submit its contents to your CA for signature.
2. Open the resulting file (for example, `sign.crt`) and install the certificate to your personal store.

9.5.6 Sign using Signtool

1. Open a prompt with `signtool` in the path.
2. Sign your binary.

   ```
   > signtool sign <binary name>
   ```

3. Identify your signing certificate by hash, if you have multiple certificates available for code signing.

   `signtool` shows you a list of valid certificates. Re-run sign tool with the `sha1` hash of the certificate:

   ```
   > signtool sign /sha1 <certificate hash> <binary name>
   ```

4. Associate the YubiHSM private key to the certificate.

   When importing the certificate for the first time on a new computer, you need to manually bind the certificate to the private key. This is needed because 1) the key is not stored with the certificate and 2) Windows doesn't automatically create an association between the private key and the certificate.

   After you import the certificate to your personal store, use the `certutil` utility provided by Windows.

   ```
   > certutil -repairstore my <certificate hash>
   ```

9.5.7 Troubleshooting

The error messages returned from `signtool` are often unhelpful in diagnosing why a signing operation failed. In these situations there are a few commands you can use to track down the root cause.

When using `signtool`, use the `/v` and `/debug` flags to get more detailed output.

- The example below shows a response you might receive if the certificate is installed but the YubiHSM is not connected or is misconfigured.

  ```
   > signtool sign /v /debug <binary name>

  After EKU filter, 1 certs were left.
  After expiry filter, 1 certs were left.
  After Hash filter, 1 certs were left.
  ```

(continues on next page)

(continued from previous page)

```
After Private Key filter, 0 certs were left.
SignTool Error: No certificates were found that met all the given criteria.
```

- Use `certutil` to check the validity of the imported certificate.

```
> certutil -verifystore my <certificate hash>

================ Certificate 0 ================
Serial Number: 029fe48291dd587c1e6f42bca341291
...
Certificate is valid
```

- Use `certutil` to check whether the KSP has been installed correctly. You should see `Provider Name: YubiHSM Key Storage Provider` as one of the entries with no errors.

```
> certutil -csplist
...
Provider Name: YubiHSM Key Storage Provider
...
```

- Use `certutil` to check if the key is accessible through the storage provider. You can also add the `-v` flag to get additional details.

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

 YubiHSM Key Storage Provider:
 tq-75c94c4b-5e40-4e44-bcd2-ee3330d4942f
 RSA
   AT_SIGNATURE
```

- Use `certutil` to dump certificate information.

  If the command shows `Cannot find the certificate and private key for decryption.` when using a new computer, it might indicate that `certutil -repairstore` hasn't yet been performed.

```
> certutil -store my <certificate hash>

================ Certificate 0 ================
Serial Number: 029fe48291dd587c1e6f42bca341291
...
Private key is NOT exportable
Signature test passed
```

For a detailed explanation of all options available in the request `.inf` file, see the documentation for the certreq utility.

To generate a similar request using the Certificate Manager:

1. Open the Certificate Manager snap-in.
2. Select the Personal/Certificates store.
3. Right click and select **All Tasks > Advanced Operations > Create Custom Request**.

CHAPTER

TEN

PKCS#11 WITH YUBIHSM 2

10.1 Configuration

The PKCS#11 module requires a configuration file, default location for this file is current directory and default name is `yubihsm_pkcs11.conf` using the environment variable `YUBIHSM_PKCS11_CONF` one can point to a custom location and name.

Configuration options can also be passed as a string in the `pReserved` field of `C_Initialize`, using the OpenSSL PKCS#11 engine this can be set in the `INIT_ARGS` configuration value. This is technically a violation of the PKCS#11 specification (which mandates `pReserved` to be set to `NULL`) and is not supported by all applications.

Accepted configuration options:

- **connector**: URL pointing at the connector to contact, mandatory
- **debug**: Turn on PKCS#11 debugging, default `off`
- **dinout**: Turn on call tracing, default `off`
- **ibdebug**: Turn on debug of libyubihsm, default `off`
- **debug-file**: File to write debug information to, default `stderr`
- **cacert**: File with cacert to verify connector https cert with (not available on Windows)
- **proxy**: Proxy server for reaching the connector (not available on Windows)
- **timeout**: Timeout to use for initial connection to the connector (in seconds), default `5`

A *Configuration File Sample* can be found below.

10.2 Logging In

All interesting operations through the PKCS#11 interface require a logged-in session, and one peculiarity of the PKCS#11 interface is that the user PIN **MUST** be prefixed by the ID (16 bits, in hexadecimal, zero padded if required) of the corresponding Authentication Key.

Assuming the default Authentication Key with ID `1` and password `password`, the user PIN would then be `0001password`. To be compliant with PKCS#11 standards, the Authentication Key password **MUST** be at least `8` characters long.

This is not part of the PKCS#11 requirement, but instead provided through the C_GetTokenInfo function, which means the module decides. Currently the total PIN length must be 12 to to 68 bytes (including the encoded auth key id, so 8 to 64 bytes for the actual PIN). This limit is flexible since the PIN is only used to derive keys.

Note: The concept of a Security Officer (SO) is not supported by the device, and the PIN management functions are not implemented, neither for user nor for SO.

It is recommended that PIN (Authentication Key) management be performed via the `yubihsm-shell` utility or the `libyubihsm` functions.

10.3 PKCS#11 on Windows

After installing yubihsm-shell using the windows installer, in addition to setting `YUBIHSM_PKCS11_CONF` environment variable, the `YubiHSM Shell\bin` directory needs to be added to the system path in order for other applications to be able to load it. This is because the `yubihsm-pkcs11.dll` is dynamically linked to the `libyubihsm\*.dll` and `libcrypto-1_1.dll` libraries and they need to be accessible for the PKCS#11 module to be useful.

On Windows 10, setting the system path is done by following these steps:

1. Go to **Control Panel > System and Security > System > Advanced system setting**.
2. Click **Environment Variables...**.
3. Under System Variables, highlight **Path** and click **Edit...**.
4. Click **New** and add the absolute path to `YubiHSM Shell/bin`.
5. Under System Variables, click **New** and add the environment variable `YUBIHSM_PKCS11_CONF` and set it to the path to the YubiHSM2 PKCS11 configuration file.

If setting the system path is not desirable, the `libyubihsm\*.dll` and `libcrypto-1_1.dll` can be copied into the same directory as the application that needs to access the PKCS#11 module.

10.4 Note for Developers

If `LoadLibrary` is called with an absolute path, it will not look for dependencies of the specified DLL in that directory, but rather in the startup directory of the application that calls `LoadLibrary`. The solution is to either:

- Call `LoadLibraryEx` with the flag `LOAD_WITH_ALTERED_SEARCH_PATH` for absolute paths
- Add the directory where the PKCS#11 module is located to the system `PATH`
- Or copy the dependencies into the application directory.

Note: Calling `LoadLibraryEx` with that flag for a non-absolute path is undefined behavior according to MS docs. For example, the way Pkcs11Interop does it is to set a variable to `LOAD_WITH_ALTERED_SEARCH_PATH` if the path looks absolute, and `0` otherwise; and then always calling `LoadLibraryEx`. If the flags is `0` then `LoadLibraryEx` behaves exactly like `LoadLibrary`.

10.5 PKCS#11 with JAVA

Due to design and implementation choices, there are some peculiarities when generating or importing keys into the YubiHSM 2 using SunPKCS#11 provider and YubiHSM 2 PKCS#11 module. JAVA SunPKCS#11 provider requires the ability to change a key's properties after creation in order for it to be able to use the keys later on. However, YubiHSM 2 does not allow such operation (All key properties have to be set at the time of creation and cannot be changed after the fact). The key information here is that the asymmetric key and its corresponding X509Certificate need to be accessed via the same ID on the device. Later versions of YubiHSM 2 PKCS#11 module provide a way to achieve this via the use of Meta Objects, but it could be worth it to make sure that this requirement is met manually, especially if the number of objects created on the YubiHSM 2 needs to be limited.

10.5.1 Version 2.4.0 or later

In version 2.4.0, the use of Meta Objects is introduced. Meta Objects are opaque objects with algorithm `opaque-data` that store the values of `CKA_ID` and `CKA_LABEL` attributes of another object on the YubiHSM 2, thus working around the hard limit on the length of those values and the inability to change those attributes after the fact. The label of a Meta Object is always `Meta object` for followed by a HEX value representing the ID, type and sequence of the actual object it is tied to (referred to as an Original Object).

Meta Objects are created as needed when the function to create an object is called with `CKA_ID` and/or `CKA_LABEL` values that are longer than 2 and 40 bytes respectively, or when the function to change one of those values is called. Meta Objects store these values as unencrypted raw data. When an Original Object is deleted, its corresponding Meta Object is also deleted automatically.

Meta Objects are only used within PKCS#11 context and their existence and use are invisible to PKCS#11 clients or users. They are, however, visible to `yubihsm-shell` users.

10.5.2 Version 2.3.2 or earlier

When using SunPKCS11 provider, it's important to know that generating asymmetric keys using `C_GenerateKeyPair` does not work. In order for SunPKCS11 to be able to use asymmetric keys on the YubiHSM2 device, both the asymmetric key and its X509Certificate must be stored under the same ObjectID. Once an asymmetric key and its X509Certificate are stored in the YubiHSM 2 under the same ObjectID, there is no problem whatsoever to use and manage the key using PKCS#11, including deleting it.

To generate asymmetric keys on the YubiHSM 2 so that they are accessible by SunPKCS11 provider, either `yubihsm-setup` or `yubihsm-shell` can be used.

yubihsm-setup

Use the subcommand `ejbca` to generate an asymmetric key on the YubiHSM2 and store it and its X509Certificate under the same ObjectID

```
yubihsm-setup -d ejbca
```

yubihsm-shell

Using `yubihsm-shell`, the attestation functionality can be leveraged to produce a self-signed X509Certificate that can then be imported using the same ObjectID as the generated asymmetric key.

```
# Generate asymmetric key and note its ObjectID
yubihsm-shell -a generate-asymmetric-key -i <KEY_OBJECT_ID> -l <OBJECT_LABEL> -d <OBJECT_
↪DOMAINS> -c <KEY_CAPABILITIES> -A <KEY_ALGORITHM>

# Sign an attestation certificate for the generated key using the YubiHSM attestation␣
↪key (with ObjectID=0)
yubihsm-shell -a sign-attestation-certificate -i <KEY_OBJECT_ID> --attestation-id 0 --
↪out cert.pem

# Import the attestation certificate to use it as a template when signing the self-
↪signed certificate. Use the same ObjectID as the generated key
yubihsm-shell -a put-opaque -i <KEY_OBJECT_ID> -l <OBJECT_LABEL> -A opaque-x509-
↪certificate --informat=PEM --in cert.pem

# Sign an attestation certificate for the generated key using the generated key itself
yubihsm-shell -a sign-attestation-certificate -i <KEY_OBJECT_ID> --attestation-id=<KEY_
↪OBJECT_ID> --out selfsigned_cert.pem

# Delete the template certificate to make room for the self-signed certificate to be␣
↪imported
yubihsm-shell -a delete-object -i <KEY_OBJECT_ID> -t opaque

# Import the self-signed certificate using the same ObjectID as the generated key
yubihsm-shell -a put-opaque -i <KEY_OBJECT_ID> -l <OBJECT_LABEL> -A opaque-x509-
↪certificate --informat=PEM --in selfsigned_cert.pem
```

Note that if a YubiHSM 2 device does not come with an attestation key with ObjectID 0, any other asymmetric key can be used instead. Since the whole purpose of signing the first attestation certificate is to produce an X509Certificate to use as a template, any X509Certificate with the desired attributes present can be used as a template instead.

Also note that when using a key for signing an attestation certificate, the signing key's capabilities must include `sign-attestation-certificate`.

10.6 Software Operations

`C_Encrypt` and `C_Verify` for Asymmetric Keys are performed in software, as well as all of the `C_Digest` operations.

10.7 PKCS#11 Attributes

There are a number of attributes defined in PKCS#11 that do not translate to Capabilities of the YubiHSM 2 device and are therefore treated as always having a fixed value.

PKCS#11	YubiHSM 2	Rationale
CKA_PRIVATE	CK_TRUE	Login is always required
CKA_DESTROYABLE	CK_TRUE	Objects can always be deleted from the device
CKA_MODIFIABLE	CK_FALSE	Objects are immutable on the device
CKA_COPYABLE	CK_FALSE	Objects are immutable on the device
CKA_SENSITIVE	CK_TRUE	All objects are sensitive
CKA_ALWAYS_SENSITIVE	CK_TRUE	Objects are immutable on the device

10.8 Capabilities and Domains

Objects created via the PKCS#11 module inherit the Domains of the Authentication Key used to establish the session. The Domains cannot be changed or modified via the module.

Object Capabilities are set on creation, depending on their Type, e.g. an RSA signing key (`CKK_RSA`) created via `C_CreateObject` with the attribute `CKA_SIGN` sets the following Capabilities set `sign-pkcs,sign-pss`.

Similarly for EC (`CKK_EC`), the key has `sign-ecdsa` set.

See the following tables for mappings:

PKCS#11	RSA (CKK_RSA)	EC (CKK_EC)	Wrap (CKK_YUBICO_AES* _CCM _WRAP)	HMAC (CKK_SHA* _HMAC)
CKA_ENCRYPT	N/A	N/A	wrap-data	N/A
CKA_EXTRACTABLE	export-under-wrap	export-under-wrap	export-under-wrap	export-under-wrap
CKA_DECRYPT	decrypt-pkcs, decrypt-oaep	N/A	unwrap-data	N/A
CKA_DERIVE	N/A	derive-ecdh	N/A	N/A
CKA_SIGN	sign-pkcs, sign-pss	sign-ecdsa	N/A	sign-hmac
CKA_VERIFY	N/A	N/A	N/A	verify-hmac
CKA_WRAP	N/A	N/A	export-wrapped	N/A
CKA_UNWRAP	N/A	N/A	import-wrapped	N/A

10.9 PKCS#11 Objects

Not all PKCS#11 Object types are implemented, this is a list of what is implemented and what it maps to.

PKCS#11	Supported CKK	Comment
CKO_CERTIFICATE		Opaque object with algorithm YH_ALGO_OPAQUE_X509_CERTIFICATE
CKO_DATA		Opaque object with algorithm YH_ALGO_OPAQUE_DATA
CKO_PRIVATE_KEY	CKK_RSA, CKK_EC	RSA 2048, 3072 & 4096 with e=0x10001, EC with secp224r1, secp256r1, secp384r1, secp521r1, secp256k1, brainpool256r1, brainpool384r1, brainpool512r1
CKO_PUBLIC_KEY		does not exist in device, only as a property of a private key
CKO_SECRET_KEY	CKK_SHA_1_HMAC, CKK_SHA256_HMAC, CKK_SHA384_HMAC, CKK_SHA512_HMAC, CKK_YUBICO_AES128_CCM_WRAP, CKK_YUBICO_AES192_CCM_WRAP, CKK_YUBICO_AES256_CCM_WRAP	

10.10 PKCS#11 Functions

Not all functions in PKCS#11 are implemented in the module, this is a list of what is implemented.

PKCS#11	Comment
C_CloseSession	
C_CloseAllSessions	
C_CreateObject	Use with CKO_PRIVATE_KEY, CKO_SECRET_KEY, CKO_CERTIFICATE or CKO_DATA
C_Decrypt	
C_DecryptFinal	
C_DecryptInit	Decrypt with Wrap Key or RSA key
C_DecryptUpdate	
C_DeriveKey	Derive key using ECDH as a PKCS#11 session object
C_DestroyObject	
C_Digest	
C_DigestFinal	
C_DigestInit	Do software digest with CKM_SHA_1, CKM_SHA256, CKM_SHA384 or CKM_SHA512
C_DigestUpdate	
C_Encrypt	
C_EncryptFinal	
C_EncryptInit	Encrypt with Wrap Key or do software encryption for RSA key
C_EncryptUpdate	
C_Finalize	
C_FindObjects	
C_FindObjectsFinal	
C_FindObjectsInit	
C_GenerateKey	Generate HMAC Key or Wrap Key
C_GenerateKeyPair	Generate Asymmetric Key
C_GenerateRandom	Generate up to 2021 bytes of random
C_GetAttributeValue	
C_GetFunctionList	
C_GetInfo	
C_GetMechanismList	
C_GetMechanismInfo	
C_GetObjectSize	
C_GetSessionInfo	
C_GetSlotInfo	
C_GetSlotList	
C_GetTokenInfo	
C_Initialize	

continues on next page

Table 1 – continued from previous page

PKCS#11	Comment
C_Login	
C_Logout	
C_OpenSession	
C_Sign	
C_SignFinal	
C_SignInit	Sign with HMAC Key or Asymmetric Key
C_SignUpdate	
C_Verify	
C_VerifyFinal	
C_VerifyInit	Verify HMAC or software verify asymmetric
C_VerifyUpdate	C_UnwrapKey Unwrap an object with Wrap Key
C_WrapKey	Wrap an object with Wrap Key

10.11 PKCS#11 Vendor Definitions

Working with the device Wrap Keys requires using vendor-specific definitions, these are listed in the table below. The Wrap Keys can be used with `C_WrapKey`, `C_Unwrapkey`, `C_Encrypt`, and `C_Decrypt`.

Wrap Type	Wrap Key
CKM_YUBICO_AES_CCM_WRAP	0xd9554204
CKK_YUBICO_AES128_CCM_WRAP	0xd955421d
CKK_YUBICO_AES192_CCM_WRAP	0xd9554229
CKK_YUBICO_AES256_CCM_WRAP	0xd955422a

10.12 Configuration File Sample

Below is a sample of a `yubihsm_pkcs11.conf` configuration file.

```
# This is a sample configuration file for the YubiHSM PKCS#11 module
# Uncomment the various options as needed

# URL of the connector to use. This can be a comma-separated list
connector = http://127.0.0.1:12345

# Enables general debug output in the module
#
# debug

# Enables function tracing (ingress/egress) debug output in the module
#
# dinout

# Enables libyubihsm debug output in the module
#
# libdebug
```

(continues on next page)

(continued from previous page)

```
# Redirects the debug output to a specific file. The file is created
# if it does not exist. The content is appended
#
# debug-file = /tmp/yubihsm_pkcs11_debug

# CA certificate to use for HTTPS validation. Point this variable to
# a file containing one or more certificates to use when verifying
# a peer. Currently not supported on Windows
#
# cacert = /tmp/cacert.pem

# Proxy server to use for the connector
# Currently not supported on Windows
#
# proxy = http://proxyserver.local.com:8080

# Timeout in seconds to use for the initial connection to the connector
# timeout = 5
```

10.13 INIT_ARGS Sample

Below is a sample of using the `INIT_ARGS` configuration with an `openssl.cnf` file.

```
openssl_conf = openssl_init

[openssl_init]
engines = engine_section

[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
dynamic_path = /path/to/engine_pkcs11.so
MODULE_PATH = /path/to/yubihsm_pkcs11.so
INIT_ARGS = connector=http://127.0.0.1:12345 debug
init = 0
```

Note: OpenSSL 1.1 will auto-load modules present in the system engine directory (like `/usr/lib/x86_64-linux-gnu/engines-1.1`) so the `dynamic_path` line has to be dropped there. The error shown will mention "conflicting engine id".

10.14 PKCS#11 Tool Compatibility, Interoperability and Known Restrictions

This topic contains information about the different tools that are either known to work or known not to work with the current version of the YubiHSM 2.

10.14.1 pkcs11-tool

This is the tool produced by OpenSC.

Run with HEAD on master (currently `dfd18389346296f8e4617832e0d5f4171835620d`).

```
pkcs11-tool --module yubihsm_pkcs11.so -l -p 0001password -t
```

All relevant tests are passing with the following notable exceptions:

- `RSA-PKCS-OAEP decryption`: the test appears to be broken. It calls into OpenSSL's `EVP_PKEY_encrypt/EVP_PKEY_encrypt_old` which uses `PKCS1v1.5` padding
- `mechtype-0xD9554204 decryption`: this a Yubico custom mechanism (AES-CCM wrapping) and can't be handled by the tool

10.14.2 pkcs11test

This is a PKCS#11 tester tool by Google. It is built as a test target in the source code. We maintain an internal version to accommodate some differences at https://github.com/Yubico/pkcs11test.

The command used

```
pkcs11test -myubihsm_pkcs11.so -l. -u0001password --gtest_filter= -${SKIPPED_TESTS_STR}
```

where `SKIPPED_TESTS_STR` is the list below.

All relevant tests pass. The following tests have been explicitly skipped:

```
Slot.NoInit
PKCS11Test.EnumerateMechanisms
ReadOnlySessionTest.GenerateRandom
ReadOnlySessionTest.GenerateRandomNone
ReadOnlySessionTest.UserLoginWrongPIN
ReadOnlySessionTest.SOLoginFail
ReadOnlySessionTest.CreateKeyPairObjects
ReadOnlySessionTest.CreateSecretKeyAttributes
ReadOnlySessionTest.SecretKeyTestVectors
ReadOnlySessionTest.SignVerifyRecover
ReadOnlySessionTest.GenerateKeyInvalid
ReadOnlySessionTest.GenerateKeyPairInvalid
ReadOnlySessionTest.WrapUnwrap
ReadOnlySessionTest.WrapInvalid
ReadOnlySessionTest.UnwrapInvalid
ReadWriteSessionTest.CreateCopyDestroyObject
ReadWriteSessionTest.SetLatchingAttribute
ReadWriteSessionTest.FindObjectSubset
```

(continues on next page)

(continued from previous page)

```
ReadWriteSessionTest.ReadOnlySessionSOLoginFail
ReadWriteSessionTest.SOLogin
ReadWriteSessionTest.TookanAttackA1
ReadWriteSessionTest.TookanAttackA3
ReadWriteSessionTest.TookanAttackA4
ReadWriteSessionTest.TookanAttackA5a
ReadWriteSessionTest.TookanAttackA5b
ReadWriteSessionTest.PublicExponent4Bytes
ReadWriteSessionTest.ExtractKeys
ReadWriteSessionTest.AsymmetricTokenKeyPair
RWUserSessionTest.SOLoginFail
DataObjectTest.CopyDestroyObjectInvalid
DataObjectTest.GetMultipleAttributes
DataObjectTest.GetSetAttributeInvalid
RWSOSessionTest.SOSessionFail
RWSOSessionTest.UserLoginFail
RWEitherSessionTest.TookanAttackA2
KeyPairTest.EncryptDecrypt
Ciphers/SecretKeyTest.EncryptDecrypt/0
Ciphers/SecretKeyTest.EncryptDecrypt/1
Ciphers/SecretKeyTest.EncryptDecrypt/2
Ciphers/SecretKeyTest.EncryptDecrypt/3
Ciphers/SecretKeyTest.EncryptDecrypt/4
Ciphers/SecretKeyTest.EncryptDecrypt/5
Ciphers/SecretKeyTest.EncryptFailDecrypt/0
Ciphers/SecretKeyTest.EncryptFailDecrypt/1
Ciphers/SecretKeyTest.EncryptFailDecrypt/2
Ciphers/SecretKeyTest.EncryptFailDecrypt/3
Ciphers/SecretKeyTest.EncryptFailDecrypt/4
Ciphers/SecretKeyTest.EncryptFailDecrypt/5
Ciphers/SecretKeyTest.EncryptDecryptGetSpace/0
Ciphers/SecretKeyTest.EncryptDecryptGetSpace/1
Ciphers/SecretKeyTest.EncryptDecryptGetSpace/2
Ciphers/SecretKeyTest.EncryptDecryptGetSpace/3
Ciphers/SecretKeyTest.EncryptDecryptGetSpace/4
Ciphers/SecretKeyTest.EncryptDecryptGetSpace/5
Ciphers/SecretKeyTest.EncryptDecryptParts/0
Ciphers/SecretKeyTest.EncryptDecryptParts/1
Ciphers/SecretKeyTest.EncryptDecryptParts/2
Ciphers/SecretKeyTest.EncryptDecryptParts/3
Ciphers/SecretKeyTest.EncryptDecryptParts/4
Ciphers/SecretKeyTest.EncryptDecryptParts/5
Ciphers/SecretKeyTest.EncryptDecryptInitInvalid/0
Ciphers/SecretKeyTest.EncryptDecryptInitInvalid/1
Ciphers/SecretKeyTest.EncryptDecryptInitInvalid/2
Ciphers/SecretKeyTest.EncryptDecryptInitInvalid/3
Ciphers/SecretKeyTest.EncryptDecryptInitInvalid/4
Ciphers/SecretKeyTest.EncryptDecryptInitInvalid/5
Ciphers/SecretKeyTest.EncryptErrors/0
Ciphers/SecretKeyTest.EncryptErrors/1
Ciphers/SecretKeyTest.EncryptErrors/2
Ciphers/SecretKeyTest.EncryptErrors/3
```

(continues on next page)

(continued from previous page)

```
Ciphers/SecretKeyTest.EncryptErrors/4
Ciphers/SecretKeyTest.EncryptErrors/5
Ciphers/SecretKeyTest.DecryptErrors/0
Ciphers/SecretKeyTest.DecryptErrors/1
Ciphers/SecretKeyTest.DecryptErrors/2
Ciphers/SecretKeyTest.DecryptErrors/3
Ciphers/SecretKeyTest.DecryptErrors/4
Ciphers/SecretKeyTest.DecryptErrors/5
Ciphers/SecretKeyTest.EncryptUpdateErrors/0
Ciphers/SecretKeyTest.EncryptUpdateErrors/1
Ciphers/SecretKeyTest.EncryptUpdateErrors/2
Ciphers/SecretKeyTest.EncryptUpdateErrors/3
Ciphers/SecretKeyTest.EncryptUpdateErrors/4
Ciphers/SecretKeyTest.EncryptUpdateErrors/5
Ciphers/SecretKeyTest.EncryptModePolicing1/0
Ciphers/SecretKeyTest.EncryptModePolicing1/1
Ciphers/SecretKeyTest.EncryptModePolicing1/2
Ciphers/SecretKeyTest.EncryptModePolicing1/3
Ciphers/SecretKeyTest.EncryptModePolicing1/4
Ciphers/SecretKeyTest.EncryptModePolicing1/5
Ciphers/SecretKeyTest.EncryptModePolicing2/0
Ciphers/SecretKeyTest.EncryptModePolicing2/1
Ciphers/SecretKeyTest.EncryptModePolicing2/2
Ciphers/SecretKeyTest.EncryptModePolicing2/3
Ciphers/SecretKeyTest.EncryptModePolicing2/4
Ciphers/SecretKeyTest.EncryptModePolicing2/5
Ciphers/SecretKeyTest.EncryptInvalidIV/0
Ciphers/SecretKeyTest.EncryptInvalidIV/1
Ciphers/SecretKeyTest.EncryptInvalidIV/2
Ciphers/SecretKeyTest.EncryptInvalidIV/3
Ciphers/SecretKeyTest.EncryptInvalidIV/4
Ciphers/SecretKeyTest.EncryptInvalidIV/5
Ciphers/SecretKeyTest.DecryptInvalidIV/0
Ciphers/SecretKeyTest.DecryptInvalidIV/1
Ciphers/SecretKeyTest.DecryptInvalidIV/2
Ciphers/SecretKeyTest.DecryptInvalidIV/3
Ciphers/SecretKeyTest.DecryptInvalidIV/4
Ciphers/SecretKeyTest.DecryptInvalidIV/3
Ciphers/SecretKeyTest.DecryptInvalidIV/4
Ciphers/SecretKeyTest.DecryptInvalidIV/5
Ciphers/SecretKeyTest.DecryptUpdateErrors/0
Ciphers/SecretKeyTest.DecryptUpdateErrors/1
Ciphers/SecretKeyTest.DecryptUpdateErrors/2
Ciphers/SecretKeyTest.DecryptUpdateErrors/3
Ciphers/SecretKeyTest.DecryptUpdateErrors/4
Ciphers/SecretKeyTest.DecryptUpdateErrors/5
Ciphers/SecretKeyTest.EncryptFinalImmediate/0
Ciphers/SecretKeyTest.EncryptFinalImmediate/1
Ciphers/SecretKeyTest.EncryptFinalImmediate/2
Ciphers/SecretKeyTest.EncryptFinalImmediate/3
Ciphers/SecretKeyTest.EncryptFinalImmediate/4
Ciphers/SecretKeyTest.EncryptFinalImmediate/5
```

(continues on next page)

(continued from previous page)

```
Ciphers/SecretKeyTest.EncryptFinalErrors1/0
Ciphers/SecretKeyTest.EncryptFinalErrors1/1
Ciphers/SecretKeyTest.EncryptFinalErrors1/2
Ciphers/SecretKeyTest.EncryptFinalErrors1/3
Ciphers/SecretKeyTest.EncryptFinalErrors1/4
Ciphers/SecretKeyTest.EncryptFinalErrors1/5
Ciphers/SecretKeyTest.EncryptFinalErrors2/0
Ciphers/SecretKeyTest.EncryptFinalErrors2/1
Ciphers/SecretKeyTest.EncryptFinalErrors2/2
Ciphers/SecretKeyTest.EncryptFinalErrors2/3
Ciphers/SecretKeyTest.EncryptFinalErrors2/4
Ciphers/SecretKeyTest.EncryptFinalErrors2/5
Ciphers/SecretKeyTest.DecryptFinalErrors1/0
Ciphers/SecretKeyTest.DecryptFinalErrors1/1
Ciphers/SecretKeyTest.DecryptFinalErrors1/2
Ciphers/SecretKeyTest.DecryptFinalErrors1/3
Ciphers/SecretKeyTest.DecryptFinalErrors1/4
Ciphers/SecretKeyTest.DecryptFinalErrors1/5
Ciphers/SecretKeyTest.DecryptFinalErrors2/0
Ciphers/SecretKeyTest.DecryptFinalErrors2/1
Ciphers/SecretKeyTest.DecryptFinalErrors2/2
Ciphers/SecretKeyTest.DecryptFinalErrors2/3
Ciphers/SecretKeyTest.DecryptFinalErrors2/4
Ciphers/SecretKeyTest.DecryptFinalErrors2/5
Digests/DigestTest.DigestKey/0
Digests/DigestTest.DigestKey/1
Digests/DigestTest.DigestKey/2
Digests/DigestTest.DigestKey/3
Digests/DigestTest.DigestKey/4
Digests/DigestTest.DigestKeyInvalid/0
Digests/DigestTest.DigestKeyInvalid/1
Digests/DigestTest.DigestKeyInvalid/2
Digests/DigestTest.DigestKeyInvalid/3
Digests/DigestTest.DigestKeyInvalid/4
Signatures/SignTest.SignVerify/0
Signatures/SignTest.SignFailVerifyWrong/0
Signatures/SignTest.SignFailVerifyShort/0
Duals/DualSecretKeyTest.DigestEncrypt/0
Duals/DualSecretKeyTest.DigestEncrypt/1
Duals/DualSecretKeyTest.DigestEncrypt/2
Duals/DualSecretKeyTest.DigestEncrypt/3
Duals/DualSecretKeyTest.DigestEncrypt/4
Duals/DualSecretKeyTest.DigestEncrypt/5
```

10.14.3 python-pkcs11tester

This is a Yubico tool, developed to run additional tests.

```
python setup.py test
```

10.14.4 p11tool

This is a tool shipped with GnuTLS. From version **3.5.2** it can work with the YubiHSM 2. Keys can be generated.

```
p11tool --provider=yubihsm_pkcs11.so "pkcs11:pin-value=0001password" --login --generate-
↪rsa --label="rsa test key" --bits=2048
```

Signatures tested and verified.

```
p11tool --provider=yubihsm_pkcs11.so "pkcs11:pin-value=0001password;object=rsakey" --
↪login --test-sign
```

10.14.5 OpenDNSSEC

OpenDNSSEC contains a `libhsm` and two tools, `ods-hsmutil` and `ods-hsmspeed`, both of these work with the YubiHSM 2 with a small configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>

 <Configuration>
      <RepositoryList>
              <Repository name="default">
                      <Module>yubihsm_pkcs11.so</Module>
                      <TokenLabel>YubiHSM</TokenLabel>
                      <PIN>0001password</PIN>
              </Repository>
      </RepositoryList>
 </Configuration>
```

Using this, it is possible to run through tests.

```
ods-hsmutil -c conf-yubihsm.xml test default
```

This passes all tests using algorithms supported by the YubiHSM 2 (rsa2048, rsa4096, ecp256, ecp384 & randomness).

CHAPTER

ELEVEN

RESETTING DEVICE TO FACTORY SETTINGS

Before deploying the YubiHSM 2 in a production environment, it might be necessary to reset the device to its factory settings, for instance to facilitate tests or training.

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

11.1 Physical Reset

The device can be physically reset to its factory settings. To do this, while inserting the YubiHSM 2 into a USB port, press the metal rim as you insert it and continue to press the rim for a minimum of 10 seconds.

11.2 Reset Using YubiHSM Shell

Please refer to the *RESET DEVICE Command*.

CHAPTER

TWELVE

EJBCA INSTALLATION AND CONFIGURATION GUIDE

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.

Note: A *key alias* on EJBCA is equivalent to a *key label* on the YubiHSM 2.

12.1 Prerequisites

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

- *YubiHSM 2 Connector*
- *YubiHSM Shell*
- *YubiHSM 2 Setup Tool*
- *PKCS#11 with YubiHSM 2*

12.2 Configuring a New EJBCA Installation

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

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.
2. Use the *YubiHSM 2 Setup Tool* to generate the keys on the YubiHSM 2, one at a time.
3. 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.
4. When configuring EJBCA, make sure to configure the following properties files:
    - `EJBCA_HOME/conf/catoken.properties`

```
sharedLibrary=/path/to/yubihsm_pkcs11.so
slotLabelType=SLOT_NUMBER
slotLabelValue=0
#Keys and their aliases as were created in step 2
```

- `EJBCA_HOME/conf/install.properties`

```
ca.tokentype=org.cesecore.keys.token.PKCS11CryptoToken
#ca.tokenpassword=null
ca.tokenproperties=<EJBCA_HOME>/conf/catoken.properties
```

- `EJBCA_HOME/conf/web.properties`

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

Note: The number `255` is just an example. It can be any "available" number. See documentation in `EJBCA_HOME/conf/web.properties`.

12.3 Configuring an Existing EJBCA Installation

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

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

3. Re-deploy EJBCA and restart the application server.
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 is displayed now (only keys created with the YubiHSM 2 Setup tool are displayed).

5. On the command line, use the YubiHSM 2 Setup tool to generate keys on the YubiHSM 2, one at a time.
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.

CHAPTER

THIRTEEN

USING OPENSSH CERTIFICATES FOR HOST LOGIN

OpenSSH supports a proprietary version of certificates that allow simple login to hosts.

13.1 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.

13.2 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 is allowed to login as, and a digital signature over the whole content created using the private key of the 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.

13.3 OpenSSH Certificates with YubiHSM 2

The private key of an SSH CA is a regular private key and can be stored on a YubiHSM 2. OpenSSH has built-in support for signing SSH Certificates using CA private keys that reside on a hardware token through the PKCS#11 interface.

The YubiHSM 2 also has specific support for signing SSH Certificates using an RSA CA key. This guide will also describe how to leverage that.

13.3.1 Example: OpenSSH built-in support for Signing SSH certificates

Signing SSH certificates is performed with OpenSSH's `ssh-keygen` command using the `-s ca_key` option. The `ca_key` specifies the key file containing the signing key. The signing key can be stored in an HSM, in which case the key file only contains the public part of the signing key. The public key is used to locate the corresponding private key on the HSM through the PKCS#11 interface. The PKCS#11 module to use is specified with the `-D` option.

1. Create an SSH CA key on the HSM, export the CA public key, and convert the public key into PKCS8 format for use with OpenSSH.

```
$ yubihsm-shell -a generate-asymmetric-key  --authkey=0x0001 -p password -i 0x000a -
↪l "SSH_CA_Key" -c "sign-pkcs" -A rsa2048

Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 1
Generated Asymmetric key 0x000a

$ yubihsm-shell -p password --authkey=0x0001 -a get-public-key -i 0x000a --out ca_
↪pub.pem

Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 1

$ ssh-keygen -i -f ca_pub.pem -m PKCS8 > ca.pub
```

Note: The CA key needs capability `sign-pkcs` in order to sign SSH pubkeys.

2. Sign the user's pubkey in the file `id_rsa.pub`, using the signing key stored in the HSM.

```
$ ssh-keygen -s ca.pub -D /usr/local/lib/pkcs11/yubihsm_pkcs11.dylib -I key_id id_
↪rsa.pub

Enter PIN for 'YubiHSM':
Signed user key id_rsa-cert.pub: id "key_id" serial 0 valid forever
```

Note: The PIN needs to be prefixed with the ID of the authentication-key in order for `ssh-keygen` to authenticate.

The signed SSH certificate is generated in the file `id_rsa-cert.pub`.

13.3.2 Signing SSH Certificate Requests

Instead of directly signing a user's SSH pubkey directly, the YubiHSM 2 can also be used to sign SSH pubkeys only when a number of conditions are met. This scenario is discussed in the rest of this document.

13.3.3 High-level Description and components

A YubiHSM 2 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.

  This 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 OpenSSH Certificate Request meets a series of pre-defined constraints.

  This is fulfilled by encoding those pre-defined constraints in an object with Type `Template` and Algorithm `SSH Template`.

13.3.4 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:

Tag Value	Tag Description
0x01	Timestamp key algorithm
0x02	Timestamp public key
0x03	CA key white-list
0x04	Not before
0x05	Not after
0x06	Principals black-list

The individual tags are further explained below.

Timestamp Key Algorithm – The *ALGORITHMS* of the public key used to verify timestamp signatures.

Timestamp Public Key – The public key used to verify timestamp signatures.

CA Key White-list – The list of Object IDs describing which Asymmetric Keys can be used with this template.

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.

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.

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

Example template – A hex-dump of an example template file is shown below:

```
01 0001 09
02 0100 cb2702...d71081f1d1
03 0002 000a
04 0004 000012c0
05 0004 00008ca0
06 0005 726f6f7400
```

This template file contains, in order:

- Timestamp Key Algorithm 9 (RSA 2048)
- Timestamp public key (256 bytes)
- CA Key whitelist containing the single Key ID 0x000a
- A Not before offset of 300 seconds (5 minutes)
- A Not before offset of 36000 seconds (10 hours)
- A principal blacklist containing the principal `root`

Although not officially supported, templates can be generated using yubihsm-ssh-tool.

For instance, the above template file and the embedded timestamp key are generated using:

```
$ openssl genrsa -out timestamp.pem
Generating RSA private key, 2048 bit long modulus
.................................+++
.....................................+++
e is 65537 (0x10001)

$ openssl rsa -pubout -in timestamp.pem -out timestamp_pub.pem
writing RSA key

$ pipenv run yubihsm-ssh-tool templ -T timestamp_pub.pem -k 10 -b 300 -a 36000 -p root
```

Here, the file `timestamp_pub.pem` contains the timestamp certificate public key, the CA key ID is 10, certificates should only be issued if their validity is at most 5 minutes in the past (to accommodate for clock skew) and at most 10 hours in the future. Also, certificates for user `root` are not allowed.

13.4 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 2 so that it can generate the aforementioned signature. This consists of all the data present in the certificate (excluding the signature).

For a description, see the `ssh-rsa-cert-v01@openssh.com` key format in the OpenSSH specs.

13.5 Signing an SSH Certificate Request

After an SSH Template has been stored on the YubiHSM 2 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:

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

After the command is issued, the following steps take place in the YubiHSM 2. First the signature `ST` 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:

- ID of the SSH CA key must appear in the SSH CA key white-list present in the SSH Template.
- `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.
- `Not After` timestamp in the SSH Certificate Request must be less than or equal to `Now` plus the `Not After` offset specified in the SSH Template.
- That 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 2 produces 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.

13.5.1 Example request

Although not officially supported, requests can be generated using yubihsm-ssh-tool:

```
$ pipenv run yubihsm-ssh-tool req -s ca_pub.pem -t timestamp.pem -I user-identity -n␣
↪username -V -5h:+5h id_rsa.pub
```

13.5.2 Example: Signing SSH certificates using templates and signing requests

Below is an example of signing SSH certificates using templates and certificate requests.

1. Create an SSH CA key on the HSM, and export the CA public key.

```
$ yubihsm-shell -a generate-asymmetric-key  --authkey=0x0001 -p password -i 10 -l
↪"SSH_CA_Key" -c "sign-ssh-certificate" -A rsa2048

Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 1
Generated Asymmetric key 0x000a
```

(continues on next page)

(continued from previous page)

```
$ yubihsm-shell -p password --authkey=0x0001 -a get-public-key -i 10 --out ca_pub.
↪pem

Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 1
```

Note: This time, the CA key needs capability `sign-ssh-certificate` in order to sign SSH certificate signing requests.

2. Create the template file and store the template in the HSM as an object of type `template-ssh` with object ID 20 and label `SSH_Template`.

```
$ pipenv run yubihsm-ssh-tool templ -T timestamp_pub.pem -k 10 -b 36000 -a 36000 -p␣
↪root

$ yubihsm-shell -a put-template -p password -i 20 -l "SSH_Template" -A template-ssh␣
↪--in templ.dat

Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 1
Stored Template object 0x0014
```

3. Generate a certificate signing request for user `username`.

```
$ pipenv run yubihsm-ssh-tool req -s ca_pub.pem -t timestamp.pem -I user-identity -
↪n username -V -5h:+5h id_rsa.pub

Hash is: b'95dd317189b5e392481de896e7f111228b76d6efe3daa344c2da2819927a05cb'
```

4. Sign the certificate request using the CA key on the HSM.

```
$ yubihsm-shell -a sign-ssh-certificate -p password -i 10 --template-id 20 -A rsa-
↪pkcs1-sha256 --in req.dat --out id_rsa-cert.pub

Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 1
```

The signed SSH certificate is generated in the file `id_rsa-cert.pub`.

13.5.3 Example: constraint violation

To illustrate what happens when the constraints specified in the certificate template are violated, for instance when a certificate is requested for the `root` user.

```
$ pipenv run yubihsm-ssh-tool req -s ca_pub.pem -t timestamp.pem -I user-identity -n␣
↪root -V -5h:+5h id_rsa.pub

Hash is: b'b75d30392a5ea5887affceb593154d2cc860f76f7bbc82757ec3fa20cd2a4d63'

$ yubihsm-shell -a sign-ssh-certificate -p password -i 10 --template-id 20 -A rsa-pkcs1-
↪sha256 --in req.dat --out id_rsa-cert.pub

Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 1
Failed to get certificate signature: SSH CA constraint violation
Unable to get ssh certificate
```

CHAPTER

FOURTEEN

OPENSSL WITH LIBP11 FOR SIGNING, VERIFYING AND ENCRYPTING, DECRYPTING

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).

14.1 Signing and Verifying

Three examples for using openSSL for signing in and verifying access.

14.1.1 RSA-PKCS#1 v1.5

```
$ 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
```

14.1.2 RSA-PSS

```
$ ~/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:
Verified OK
```

14.1.3 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
```

14.2 Encrypting and Decrypting

Three examples for using openSSL for encrypting and decrypting.

14.2.1 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
```

14.2.2 RSA-OAEP

```
$ 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.
```

(continues on next page)

(continued from previous page)

```
Enter PKCS#11 token PIN for YubiHSM:
4aa58c448f3264c777be1b5ad94cf3e0a68911ed3f18db9e568ff2179e263f76
```

14.2.3 ECDH

```
$ 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
```

CHAPTER

FIFTEEN

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 as well. If you are on macOS you will have to symlink pkg-config in order to do so.

OpenSSL requires engine settings in the `openssl.cnf` file. Some OpenSSL commands allow specifying `-conf ossl.conf` 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.

15.1 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/yubihsm.conf openssl'
```

15.2 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/yubihsm_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
Public Key Object; RSA 2048 bits
 label:      my_key
```

(continues on next page)

(continued from previous page)

```
 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
DTE3MDQyNDA3Mzc1MFoXDTE3MDUyNDA3Mzc1MFowDjEMMAoGA1UEAwwDZm9vMIIB
IjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAqBARJLAIjSqKk2OuRWrs91EC
MYjjZhxJE8IAMIiDDM2wSuQhB7A2CVW+/d1SG0k5cTEiasDBHbH9Bc2w+xn0l3Dh
8cXafvcFkjcNabHesrbcwRgItugw7PWBtyopWDtDhVWKS1zkpDO8iKjwiYciweaP
96nEHlQPPRUp7bf3IE7RTXENAqJai6QIYBZOrzHM9NrIz/6YaR2ua7SY7V/B3xaJ
7KsiQ8oHWuf+RDNkJOhbD+1fgeMtN8x+W4XYnCPQPjJ/MfjuHJ2n5EM3Vb/plh9H
uT+D56ozIk41FeXgC4gNu8fIv2KE1XBMuJCGRbyh5xk0dkQdvKxtVEfiDcwxBwID
AQABMA0GCSqGSIb3DQEBCwUAA4IBAQCHyskEU84T/YGhcjlpsdmobtyNhWc2ae/x
fmQpY/XGzQkSmUZJA+Z04JMUbli7UKEOItmqSlU6j0BPy03UjavNHdDPYcUZIS28
fPtzTkU3FdEBM/zkPXStBCo9+N34l4qSdir9hFWM1/CpkfP8PhteUQAqImXjbDVh
qhrfOg+kY3dAz91kLLXuA4YfuC+eEJh0JGuXCivhGre5LL9njrajHnJ+HSt6HHjC
R4U27/hzoK3r12XE5NjznjcaKk1AKFXZE92nqG/WYliyLpNNSrN+AmEKrPOHb8My
ZJlaGAfm3K9vLEjwrLQSAIKpMdpUcNE7Ay+EsEYTQpy43VvwI8vL
-----END CERTIFICATE-----
```

15.3 Example: Certificate Request

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

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

 [openssl_init]
 engines = engine_section

 [engine_section]
 pkcs11 = pkcs11_section

 [pkcs11_section]
 engine_id = pkcs11
# dynamic_path is not required if you have installed
# the appropriate pkcs11 engines to your openssl directory
```

(continues on next page)

(continued from previous page)

```
 dynamic_path = /path/to/engine_pkcs11.{so|dylib}
 MODULE_PATH = /path/to/yubihsm_pkcs11.{so|dylib}
# it is not recommended to use "debug" for production use
 INIT_ARGS = connector=http://127.0.0.1:12345 debug
 init = 0
EOF

$ OPENSSL_CONF=engine.conf openssl engine -t -c pkcs11 (pkcs11) pkcs11 engine [RSA, DSA,␣
↪DH, RAND] [ available ]
```

15.4 Example: Retrieve 64 Bytes of Data

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

```
$ OPENSSL_CONF=engine.conf openssl rand -engine pkcs11 -hex 64
engine "pkcs11" set.
2aae245fc6d1c0419684ee8968ce26fba2dc3bb48a91bae912c8a82b11db8186493
25800e6e984fedfa1940a24731dc2721431979a287252a214ebb87624dcf1
```

15.5 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`:

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

[ req_dn ]
commonName = Common Name (eg, your name)
```

15.6 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
engine "pkcs11" set.
PKCS#11 token PIN:
-----BEGIN CERTIFICATE REQUEST-----
MIICVDCCATwCAQAwDzENMAsGA1UEAwwEdGVzdDCCASIwDQYJKoZIhvcNAQEBBQAD
ggEPADCCAQoCggEBAJoTtK9p5XNDBaqy65IBDSj3mP9cpM0cw/sF/GZai6cx8Skf
DjAhqOkloN+Jdc2OsnaBVSqCbsSjVTXfc83oB2q4M3U/tl/nfzTGHGCA48dbKUiz
M807KoyYzFds9b7ZnGrwCmeXWjt2sAEGiJYEQt9gS9twabnCwxY4KySa9aNSNeHt
```

(continues on next page)

(continued from previous page)

```
AwnfP5V60C73xA7ATOPjuWXq4TWgMWzRD0IwA3h7MIgtevJio2MTPWlspdGbYrxr
KsVfl/AocrSqYb44pMaRbAJAgOpJ8hsPjc9gkJnnrhmbkfV0v0AqjgwqxZa+BCWn
gdGl5HwKVFLu+X3lsBw7xHHJtOYgeFpp8twfvT0CAwEAAaAAMA0GCSqGSIb3DQEB
CwUAA4IBAQAcyImLuv7CrZJ1RPOf5d6u5LfYUadPXSGnozf3Ebgue12B51etKjYK
3cY8m9rRP3jRU5yWk3qoquZ7vCF7RNPf0N+7/blXHfoawx+ffEl/ToUZ5xr7IL0V
Qz9qzEumdNmm6MoQPxPOgrb1oCAz103gkf+S4HZGntO83/D31znsEhCSakoAa44s
3I+7vmzhjwUZsvMUg3sg2NCjRYRX2RPIPmtkDgufqsdAkNyWHlzitjfVMZxf8BcY
9DBrpQe1O6UbE1K9kYj2YBJ9h/FxfNJUk8t+rCcSOcQjmcRtgbHwhk2q77rapmg2
YliaYEU1/e5kl+v+0WEg7rvXgh/VkY2h
-----END CERTIFICATE REQUEST-----
```

15.7 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
engine "pkcs11" set.
PKCS#11 token PIN:
-----BEGIN CERTIFICATE-----
MIICmjCCAYICCQDX5mJwg+YmMjANBgkqhkiG9w0BAQsFADAPMQ0wCwYDVQQDDAR0
ZXN0MB4XDTE3MDMxNTIwMDkzOVoXDTE4MDMxNTIwMDkzOVowDzENMAsGA1UEAwwE
dGVzdDCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAJoTtK9p5XNDBaqy
65IBDSj3mP9cpM0cw/sF/GZai6cx8SkfDjAhqOkloN+Jdc2OsnaBVSqCbsSjVTXf
c83oB2q4M3U/tl/nfzTGHGCA48dbKUizM807KoyYzFds9b7ZnGrwCmeXWjt2sAEG
iJYEQt9gS9twabnCwxY4KySa9aNSNeHtAwnfP5V60C73xA7ATOPjuWXq4TWgMWzR
D0IwA3h7MIgtevJio2MTPWlspdGbYrxrKsVfl/AocrSqYb44pMaRbAJAgOpJ8hsP
jc9gkJnnrhmbkfV0v0AqjgwqxZa+BCWngdGl5HwKVFLu+X3lsBw7xHHJtOYgeFpp
8twfvT0CAwEAATANBgkqhkiG9w0BAQsFAAOCAQEAHeSL6Qwqr8ST4SqnC1T2jjME
cjAT5eK4MqK3ayAy/Y/vYGtzARGIi9tGatyV6AFjs/0Me3/8du4bBVdC2DaP1hTf
m4m1HShHKFdUlwUGcwYoVNquCz8d6hDu3nL0XvtFKX77aHHQZeB3t0uD8evYZdTS
8oAduJpkAdJV7CtCLbGhLlLD3siYkd5fD35lhHlg8T2n5F4srDafQVdrDb/myYmI
2UmrZWvKDWZ3UvzKt1XVS8omIx7aTrUAPqv/SEdpPmJvg0pgWTKvzAtsnsxlRQdd
tdtJ/6nqhwXVSNXlDbyhFVo6J2u8BMEss2iausoSZBzf+YDOw2H+4GH6E11TmA==
-----END CERTIFICATE-----
```

15.8 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
engine "pkcs11" set.
PKCS#11 token PIN:
Using default temp DH parameters
ACCEPT
ACCEPT
```

15.9 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
```

CHAPTER

SIXTEEN

USING OPENSC PKCS11-TOOL

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 `yubihsm_pkcs11.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:

```
export YUBIHSM_PKCS11_CONF=/path/to/user/home/yhsm2-p11.conf
export YUBIHSM_PKCS11_MODULE=/usr/local/lib/yubihsm_pkcs11.so
alias yhsm2-tool='pkcs11-tool --module ${YUBIHSM_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 key pair has been generated with id `0x0401` and capabilities including `asymmetric_sign_pkcs:asymmetric_sign_pss`;
- RSA encrypting/decrypting key pair has been generated with id `0x0402` and capabilities including `asymmetric_decrypt_pkcs:asymmetric_decrypt_oaep`;
- ECDSA signing/verifying key pair has been generated with id `0x0203` and capabilities including `asymmetric_sign_ecdsa:asymmetric_sign_decdsa`;
- EC key pair 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

16.1 Creating Digital Signatures

Examples how to create digital signature.

16.1.1 RSA-PSS

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
```

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
```

16.1.2 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
```

16.1.3 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
```

(continues on next page)

(continued from previous page)

```
$ 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
```

16.2 Performing Decryption

Examples how to run decryption.

16.2.1 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
```

16.2.2 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
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
```

16.2.3 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
```

16.2.4 Obtaining Random Data

```
$ yhsm2-tool --pin xxxxxxxx --generate-random 64 | xxd -c 64 -p
 Using slot 0 with a present token (0x0)

↪e3384c2a8f7263b46879d27d068779ebf82dfabe74bf057637a591a314dea86f12f35a79712950695dcbe54824eebe284430e
```

CHAPTER

SEVENTEEN

YUBIHSM AND OPENSSL ON WINDOWS

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

17.1 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.

17.2 Installation

17.2.1 YubiHSM2 Development Kit

1. Unzip the downloaded file to install the development kit. The development kit has utilities and a couple of MSI files.
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.
3. Create the YubiHSM2 connector configuration file. Then set the `YUBIHSM_PKCS11_CONF` environmental variable with its path and name. See below for example.

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

17.2.2 OpenSC and OpenSSL Distributions

The Shining Light Productions OpenSSL distribution is not an official distribution, it 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 `usingpkcs11-tool`.

```
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)

C:\PROGRA~1\OpenSC Project\OpenSC\tools>
```

17.2.3 libp11 Source

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

1. Open a Visual Studio x64 Native Tools command prompt.
2. Go to the source directory.
3. Type: `nmake -f Makefile.mak OPENSSL_DIR=\progra~1\OPENSS~1 BUILD_FOR=WIN64`

The `.dll` files will be in the source directory.

17.2.4 Configuration

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 `administratoruser`.

   ```
   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
   ```

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"
```

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)
  Serial Number:
      2d:71:6a:fd:8b:ab:5a:b8:3e:5c:cc:c0:bc:b1:a5:11:df:7f:2b:1d
  Signature Algorithm: sha256WithRSAEncryption Issuer: C = US,
      ST = Washington, L = Seattle, O = Banana Inc.,
      OU = Fruit Bunch,
      CN = Root Test Cert Validity Not Before:
      Aug 22 21:20:07 2019 GMT
      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:
```

CHAPTER

EIGHTEEN

CONFIGURING YUBIHSM 2 FOR JAVA CODE SIGNING

The purpose of the scripts in this repository is to generate an RSA key pair 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 key pair 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, see *Introduction* before making any deployments in production.

18.1 Prerequisites

18.1.1 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)

18.2 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
```

18.3 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 *Configuration* webpage. If the YubiHSM-Connector is running on the same machine, it is sufficient to copy the *Configuration File Sample* and paste it into the file `yubihsm_pkcs11.conf`.

18.4 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}
```

18.5 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'
```

18.6 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
```

18.6.1 Signing JAR files

Consider the following minimal Java source file:

```
cat HelloWorld.java
public class HelloWorld {
      public static void main(String [] args) {
              System.out.println("Hello, world");
      }
    }
```

Compile the java source file and create an (unsigned) .jar file:

```
javac HelloWorld.java
jar cfe unsigned.jar HelloWorld HelloWorld.class
```

We can now sign this JAR file with the RSA signing key we have stored in our YubiHSM 2 and create a signed JAR file:

```
jarsigner -tsa http://timestamp.digicert.com -addProvider SunPKCS11 -providerArg ./
↪sunpkcs11.conf -keystore NONE -storetype PKCS11 -storepass 0001password -signedjar␣
↪signed.jar ./unsigned.jar rsaSign

jar signed.

Warning:
The signers certificate is self-signed.
The timestamp will expire on 2031-11-10.
```

In this case, a self-signed certificate was used, but for others to be able to validate the certificate you should use a public CA to sign your Java code.

Note that we are using a timestamp server to record the current time in the signed JAR file. This way we do not need to resign the JAR file when the signing certificate expires.

18.6.2 Verifying signed JAR files

To verify the signature on the signed JAR, we use the public key certificate stored on the YubiHSM 2.

```
jarsigner -verify -addProvider SunPKCS11 -providerArg ./sunpkcs11.conf -keystore NONE -
↪storetype PKCS11 -storepass 0001password ./signed.jar

jar verified.
```

If we trust the signer and the Certificate Authority that issued the signer's certificate, we can decide to run the software in the JAR file:

```
java -jar signed.jar
Hello, world
```

Note that access to the YubiHSM2 is not required when verifying a signature on a signed JAR file as the certificate is included in the JAR file itself. Verification will fail however unless the certificate was signed by a trusted Certification Authority.

18.6.3 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 key pair 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.

18.6.4 Parameters

The PowerShell script has the following parameters.

Parameter	Purpose
Algorithm	Signature algorithm Default: RSA2048
AuthKeyID	KeyId of the YubiHSM 2 authentication key Default: 0x0001
AuthPW	Password to the YubiHSM 2 authentication key Default:
CAcertificate	CA certificate used by OpenSSL (for test purposes) Default: TestCACert.pem
CAPrivateKey	CA private key used by OpenSSL (for test purposes) Default: TestCAKey.pem
CAPrivateKeyPW	Password of the OpenSSL keystore (for test purposes) Default:
CreateCSR	Generate keys and export CSR and then exit
CSRfile	File to save the CSR request to Default: ./YHSM2-Sig.(date and time).csr
Dname	X.500 Distinguished Name to be used as subject fields Default:
Domain	Domain in the YubiHSM 2 Default: 1
ImportCert	Import signed certificate created with CreateCSR
KeyID	KeyID where the RSA key pair is stored Default: 0x0002
KeyName	Label of the key/certificate, same as Java alias Default: MyKey1
LogFile	Log file path Default: WorkDirectory/YubiHSM_PKCS11_Enroll.log
PKCS11Config	Java JCE PKCS #11 configuration file

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.

18.6.5 Example of how to execute the PowerShell script:

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

18.7 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 key pair 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.

18.7.1 Parameters

The Bash script has the following parameters.

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

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.

18.8 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
```

18.9 List the Objects on YubiHSM 2

The created RSA key pair 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 key pair 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.

18.9.1 Example: YubiHSM-Shell Command

```
yubihsm> 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
yubihsm> 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
```

18.9.2 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: 23161118fc1d59fbab75138b562a4b00c8163c3d
```

(continues on next page)

(continued from previous page)

```
Valid from: Wed Apr 14 10:43:28 CEST 2021 until: Sat Aug 27 10:43:28 CEST 2022
Certificate fingerprints:
   SHA1: 38:1E:81:1A:0A:6E:B0:87:E0:B6:5C:8A:B8:C6:EC:91:1D:51:28:1A
   SHA256:␣
↪CC:F7:26:6C:70:12:7E:E3:62:22:71:9B:3C:32:16:C8:C6:34:10:F:49:22:7A:18:70:09:E3:3E:73:42:38:47
Signature algorithm name: SHA256withRSA
Subject Public Key Algorithm: 2048-bit RSA key
Version: 1
```

18.10 Using YubiHSM 2 with Java Signing Applications

When the YubiHSM 2 has been configured with an RSA key pair 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.

18.10.1 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.
```

18.10.2 Example: Use JarSigner to Validate a Signed JAR-file

```
jarsigner -verify lib.jar -verbose -certs
...
jar verified.
```

18.11 Signing XML files using YubiHSM 2

Many applications make use of XML to structure data stored in files, databases, or elsewhere. To establish trust in such data, these documents can be signed using XML Signatures.

In order to sign XML documents you can use a tool called xmlsectool. As `xmlsectool` is implemented as a Java application using the JCA en JCE standards, we can use a YubiHSM 2 to store the signing keys we use for generating XML signatures.

18.11.1 A simple example

As an example, generate an RSA key pair and a self-signed certificate stored on the YubiHSM 2:

```
$ keytool -keystore NONE -storetype PKCS11 -storepass 0001password -addProvider␣
↪SunPKCS11 -providerArg ./sunpkcs11.conf -genkey -alias rsaSign -keyalg RSA -dname␣
↪CN=rsaSign
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a␣
↪validity of 90 days
for: CN=rsaSign
```

As before, we are using the SunPKCS11 provider to interface with the YubiHSM2, similar to other examples in this chapter.

18.11.2 Signing XML files

Let's generate a very simple XML file:

```
$ echo '<x></x>' > unsigned.xml
```

Sign the XML file using `xmlsectool`:

```
$ xmlsectool --sign --pkcs11Config ./sunpkcs11.conf --inFile unsigned.xml --keyAlias␣
↪rsaSign --keyPassword 0001password --outFile signed.xml

INFO  XMLSecTool - Reading XML document from file 'unsigned.xml'
INFO  XMLSecTool - XML document parsed and is well-formed.
INFO  XMLSecTool - XML document successfully signed
INFO  XMLSecTool - XML document written to file /home/user/signed.xml
```

The signed XML document nog contains a `Signature` element containing the a `SignatureValue` and a `KeyInfo` element containing a copy of the X.509 certificate on the YubiHSM 2:

```
<x>
  <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
  <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
  <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
  <ds:Reference URI="">
    <ds:Transforms>
      <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
      <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    </ds:Transforms>
  <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
    <ds:DigestValue>9hy1oK7rXCJu4rTqLZ7cGUH3rPyGm4QllC8VRv6mX60=</ds:DigestValue>
  </ds:Reference>
    </ds:SignedInfo>
  <ds:SignatureValue>ce5SooQsD...aiUDiOkaBiWI8A4olAuRcIgme0PqeLg==</ds:SignatureValue>
  <ds:KeyInfo>
    <ds:KeyValue>
      <ds:RSAKeyValue>
        <ds:Modulus>uSsZh/aAk...MK4yY1LTUqF2HzSO9d4vGdWzwm4Z63ot6w==</ds:Modulus>
        <ds:Exponent>AQAB</ds:Exponent>
```

(continues on next page)

(continued from previous page)

```
      </ds:RSAKeyValue>
    </ds:KeyValue>
    <ds:X509Data>
      <ds:X509Certificate>MIICxzCCAa+g.../BUkO7i8reQw+6qA==</ds:X509Certificate>
      </ds:X509Data>
    </ds:KeyInfo>
  </ds:Signature>
</x>
```

In the above document, we have shortened the Base64 encoded text elements for brevity.

18.11.3 Verifying XML digital signatures

To verify the signed XML file, we can also use `xmlsectool`:

```
$ xmlsectool --verifySignature --inFile signed.xml --pkcs11Config ./sunpkcs11.conf --
↪keyAlias rsaSign --keyPassword 0001password
INFO  XMLSecTool - Reading XML document from file 'signed.xml'
INFO  XMLSecTool - XML document parsed and is well-formed.
INFO  XMLSecTool - XML document signature verified.
```

Here, we are referring to the signing certificate stored on the YubiHSM 2 to be able to verify signatures when direct access to the YubiHSM 2 is not available, we need to export the signing certificate and distribute it to whoever needs to be able to verify such signatures.

To export the signing certificate stored on a YubiHSM 2 using `keytool`:

```
$ keytool -keystore NONE -storetype PKCS11 -storepass 0001password -addProvider␣
↪SunPKCS11 -providerArg ./sunpkcs11.conf -exportcert -alias rsaSign -rfc > signing-crt.
↪pem
```

We can now use `xmlsectool` to verify an XML digital signature using the public key in the signing certificate:

```
xmlsectool  --verifySignature --inFile signed.xml --certificate signing-crt.pem
INFO  XMLSecTool - Reading XML document from file 'signed.xml'
INFO  XMLSecTool - XML document parsed and is well-formed.
INFO  XMLSecTool - XML document signature verified.
```

In case the signature does not verify, `xmlsectool` will complain:

```
$ xmlsectool  --verifySignature --inFile signed.xml --certificate signing-crt.pem
INFO  XMLSecTool - Reading XML document from file 'signed.xml'
INFO  XMLSecTool - XML document parsed and is well-formed.
WARN  XMLSignature - Signature verification failed.
ERROR XMLSecTool - XML document signature verification failed
make: *** [verify] Error 7
```

In this case, either the XML document was changed after its signature was generated, or the public key in the certificate does not match the private key used for signing. Either way, the XML signature cannot be used to establish trust in the XML document's authenticity.

For more information, see Using PKCS11 Credentials from the `xmlsectool` documentation.

18.11.4 A real-world example: SAML metadata signing

One example application of using XML signatures is in identity federation, where users can logon to a web application after authenticating somewhere else. A well-known protocol used for identity federation is SAML 2.0, and this protocol is based on XML.

The parties where users may want to logon (called Service Providers) need to exchange information with the parties where users authenticate (called Identity Providers), and this SAML 2.0 Metadata is typically signed using XML Signatures so it can be automatically verified by SAML peers.

Consider the following SAML 2.0 metadata document for a fictitious Service Provider which specifies its identifier (entity ID), its SAML signing certificate and the URL endpoint an Identity Provider can direct users to in order to process a SAML authentication response:

```
<md:EntityDescriptor  ID="XYZ123456"
    xmlns:md="urn:oasis:names:tc:SAML:2.0:metadata" entityID="https://example.com/saml/
↪sp.xml">
<md:SPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol" >
<md:KeyDescriptor>
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:X509Data>
      <ds:X509Certificate>
          MIHnMIGiAgEBMA0GCSqGSIb3DQEBBAUAMA8xDTALBgNVBAMMBHNpZ24wHhcNMjMwM
          TA1MTI0ODExWhcNMjgwNjI3MTI0ODExWjAPMQ0wCwYDVQQDDARzaWduMEwwDQYJKo
          ZIhvcNAQEBBQADOwAwOAIxAKrBRhYU03MSaU8jBPNUx9wcc6bWhMpinZmINR0JNdh
          3SkPddh7zskcLGonFsmasQIDAQABMA0GCSqGSIb3DQEBBAUAAzEADng7opb78PNoL
          ZH1QzYqmxV0ZSc3rE0OlTW00W/Xq7+77OhU5vVAVYnXpQLlv6sB
  </ds:X509Certificate>
    </ds:X509Data>
  </ds:KeyInfo>
</md:KeyDescriptor>
<md:AssertionConsumerService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"␣
↪index="0" Location="https://example.com/saml/acs"/>
</md:SPSSODescriptor>
</md:EntityDescriptor>
```

Note that the certificate in the Metadata is intended for validating SAML protocol messages and is typically different from the certificate used for validating SAML 2.0 metadata. Either or both certificates can have their private keys stored on the YubiHSM 2, but be aware that SAML protocol messages are signed much more frequently than SAML metadata documents, so the former may require multiple YubiHSM 2 deployments in order to scale with the load on your SAML IdP or SP.

To sign this SAML metadata document, we again use `xmlsectool` with the signing key stored in a YubiHSM 2. We also specify `ID` as the name of the XML attribute to use in the XML signature.

```
xmlsectool --sign --pkcs11Config ./sunpkcs11.conf  --inFile unsigned.xml --keyAlias␣
↪rsaSign --keyPassword 0001password --outFile signed.xml --referenceIdAttributeName ID
```

As before, we will need to export the SAML signing certificate to distribute among our SAML peers so they can validate our signed metadata.

18.12 Example Java code using YubiHSM 2

To interface to cryptographic keys stored on a YubiHSM 2 from Java code, we can use the SunPKCS11 provider.

This has the added benefit that we can write code that is independent of the specific HSM used, as long as the HSM has a PKCS#11 module available.

Apart from writing code, we need to configure all components correctly in order for the code to work correctly. This includes the configuration of the YubiHSM 2 connector, Java `keytool`, and the SunPKCS11 provider.

To illustrate, we will code a simple RSA signing example below.

18.12.1 Setup

Let's assume we have a single YubiHSM 2 connected locally via USB. Store the connector configuration in a file named `yubihsm.conf` and point to it via the `YUBIHSM_PKCS11_CONF` environment variable so that the YubiHSM 2 PKCS#11 module will be able to find it:

```
$ echo "connector=yhusb://" > yubihsm.conf export YUBIHSM_PKCS11_CONF=yubihsm.conf
```

We will be using Java's `keytool` to manage keys and certificates on the YubiHSM 2. For convenience, store the PKCS#11 configuration options in a file named `keytool.config`:

```
$ cat keytool.config
keytool.all = -keystore NONE -storetype PKCS11 -storepass 0001password -addProvider␣
↪SunPKCS11 -providerArg ./sunpkcs11.conf
```

The file `sunpkcs11.conf` is used to configure the PKCS#11 module we want to use, and the PKCS#11 attributes we want to define for objects created or imported via the SunPKCS11 provider:

```
$ cat sunpkcs11.conf
name = YubiHSM2
library = /usr/local/lib/pkcs11/yubihsm_pkcs11.dylib
attributes(*, CKO_PRIVATE_KEY, CKK_RSA) = {
   CKA_SIGN=true
}
```

Finally, we can create an RSA key pair and a self-signed certificate using Java's `keytool`.

```
 $ keytool -conf keytool.config -genkey -alias rsaSign -keyalg RSA -dname CN=rsaSign
Generating 2,048 bit RSA key pair and self-signed certificate (SHA256withRSA) with a␣
↪validity of 90 days for: CN=rsaSign
```

Note that when using `keytool`, the keys are generated in software and subsequently imported into the YubiHSM 2. To generate keys on the YubiHSM 2 itself, use the `yubihsm-shell` tool.

The generated certificate should now be visible from `keytool`. For example:

```
$ keytool -conf keytool.config -list
Keystore type: PKCS11
Keystore provider: SunPKCS11-YubiHSM2

Your keystore contains 1 entry

rsaSign, PrivateKeyEntry,
```

(continues on next page)

(continued from previous page)

```
Certificate fingerprint (SHA-256):
   02:50:E4:1B:D8:FB:1B:07:AB:8C:05:85:37:BD:FB:89:6F:57:1F:CC:86:EC:
   E5:F2:BE:61:76:68:38:58:F0:39
```

18.12.2 Code

Now, let's turn to coding. In this example, we will be signing files using the RSA private key stored on our YubiHSM 2.

The data to be signed is a simple text file, for instance:

```
$ date > datatosign
```

The code needs to do some file I/O, and use the JCA and JCE standards to generate and verify signatures. Let's start with the necessary imports:

```
import java.nio.file.Path;
import java.nio.file.Files;
import java.io.FileOutputStream;
import java.security.cert.X509Certificate;
import java.security.*;
```

To keep things simple, we do not handle Exceptions here and define some hardcoded parameters:

```
public class RsaSignP11 {
   public static void main(String... argv) throws Exception {

     String pkcs11Conf = "sunpkcs11.conf";
     String userPin = "0001password";
     String keyAlias = "rsaSign";
     String infile = "datatosign";
     String outfile = "signature.bin";

   // see below ...
  }
}
```

Using hard-coded parameters is only to keep the example concise. Normally these would be command-line parameters or read from a configuration file. Passwords should never be hard-coded and are typically read from a terminal on demand.

To continue with our example code, first load and configure the SunPKCS11 provider:

```
Provider provider = Security.getProvider("SunPKCS11"); provider = provider.
↪configure(pkcs11Conf); Security.addProvider(provider);
```

Load the PKCS11 KeyStore, authenticating with the User PIN:

```
KeyStore ks = KeyStore.getInstance("PKCS11", provider); ks.load(null, userPin.
↪toCharArray());
```

Retrieve the private key, and sign the data read from the `datatosign` file using the `SHA256withRSA` algorithm:

```
PrivateKey privateKey = (PrivateKey) (ks.getKey(keyAlias, null));
Signature rsaSig = Signature.getInstance("SHA256withRSA"); rsaSig.initSign(privateKey);␣
↪byte[] datatosign = Files.readAllBytes(Path.of(infile)); rsaSig.update(datatosign);␣
↪byte[] sigBytes = rsaSig.sign();
```

Optionally, the signature can be stored in a signature file for others to verify:

```
new FileOutputStream(outfile).write(sigBytes);
```

While we are at it, let's also verify if the signature generated in `sigBytes` can be verified using the corresponding public Key.

First we need to extract the public key from the certificate pointed to by our `rsaSign` alias:

```
X509Certificate cert = (X509Certificate) ks.getCertificate(keyAlias); PublicKey␣
↪publicKey = cert.getPublicKey();
```

Again using the `SHA256withRSA` algorithm, verify that the signature in `sigBytes` matches the data in `datatosign` using our `publicKey`:

```
rsaSig = Signature.getInstance("SHA256withRSA");
rsaSig.initVerify(publicKey);
rsaSig.update(datatosign);
assert rsaSig.verify(sigBytes) == true : "verify failed";
```

To test, compile the source:

```
$ javac RsaSignP11.java
```

Run the program:

```
$ java RsaSignP11
```

There is no output, meaning the `assert` was passed without issues and the signatures verifies.

18.12.3 Troubleshooting

Debugging issues with HSMs can be difficult. It may help to enable logging using the following JVM system properties:

For PKCS#11 keystore specific debugging info:

```
-Djava.security.debug=pkcs11keystore
```

For general SunPKCS11 provider debugging info:

```
-Djava.security.debug=sunpkcs11
```

Also, refer to the documentation on *PKCS#11 with YubiHSM 2* for generating debug logs from the PKCS#11 module itself.

CHAPTER

NINETEEN

DEPLOYING YUBIHSM 2 WITH ACTIVE DIRECTORY CERTIFICATE SERVICES

This document is intended to enable systems administrators to deploy YubiHSM 2 with YubiHSM Key Storage Provider so that the Active Directory Certificate Services Certificate Authority (ADCS CA) root key is created securely on the YubiHSM 2 and so that a hardware-based backup copy of key materials has been produced.

As a guide for deployment, it covers basic topics. Instructions should be modified as required for your specific environment. It is assumed that installation is performed on a single server destined to become a production or lab Certificate Authority root. It is also assumed that you are familiar with the concepts and processes of working with Microsoft ADCS.

Plan a public key infrastructure (PKI) that is appropriate for your organization. For guidance on setting up a PKI, see Microsoft's TechNet article on Public Key Infrastructure Design Guidance

We recommend that you install and test the installation and setup of the YubiHSM 2 in a test or lab environment before deploying to production.

Scenario: In a Windows PKI environment, protect the CA root key in hardware.

Benefits: YubiHSM 2 guards the CA root key and protects all signing and verification services using the root key.

Note: Although the screenshots in this guide are specific to Windows Server 2016, Server 2019 is also supported.

19.1 Prerequisites and Preparations

The audience of this document is expected to be an experienced systems administrator with a good understanding of Windows Server management. In addition, it helps to be familiar with the terminology, software and tools specific to YubiHSM 2. As a primer for these, refer to the :: *Glossary* in this guide.

In order to follow the steps provided in this guide, be sure you meet the following prerequisites, having:

- Access to Microsoft Windows Server 2012, R2/2016, 2019 with Active Directory in an offline, air-gapped environment, such as a secure computer network that is physically isolated from unsecured networks such as the internet. You must also have elevated system privileges.
- YubiHSM 2 software and tools for Windows downloaded from the YubiHSM 2 Release page and available on the system to be used.
- Two (2) factory-reset YubiHSM 2 devices, one for deployment and one for backup in hardware.
- Key custodians identified as per local requirements and available to participate. For more information about key custodians and the associated `M of N` key shares, see the next chapter in this guide.

19.2 Key Splitting and Key Custodians

The preferred method for backing up the YubiHSM 2 keys calls for key splitting and restoring or regenerating, often referred to as setting up an `M of n` scheme (Shamir's Secret Sharing (SSS)). This process ensures no individual can export key material from the YubiHSM 2 and provides a way to control the import of key material that has been exported under wrap from one device into other devices. For example, you would export and import objects for backup purposes, as described in *Backup and Restore Using YubiHSM KSP (Windows Only)*.

The key that is split among a predetermined number (`n`) of **key custodians** (also known as key shareholders) is known as the wrap key. Each custodian receives their own unique share. To use the key, a minimum number of shares (`m`) must be present so that the key can be regenerated (sometimes called "rejoined"). This minimum number of custodians is called the **privacy threshold**. If this threshold is not attained, the wrap key cannot be regenerated. This minimum number, `n`, should be larger than one.

The exact number of key shares and the privacy threshold are determined by the requirements of your organization. If your organization has policies in place that define how this procedure should be performed, be sure you know these policies before proceeding. You should also have a predetermined practice in place specifying both:

- How the key shares must be recorded (written on paper, photographed, locally printed, or some other means) and
- How they must be stored between uses (for example, offsite archive, safety deposit box, sealed envelope).

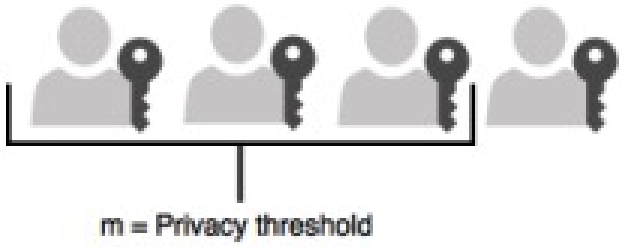


Figure: Privacy Threshold

The YubiHSM Setup Tool enables you to perform the key splitting and assigning of shares to key custodians. To carry out the setup process, you need to know who the wrap key custodians will be. During setup, all key custodians must be physically present to record their shares. Exact instructions for key splitting and assigning of shares are given in *Configuring the Primary YubiHSM 2 Device*.

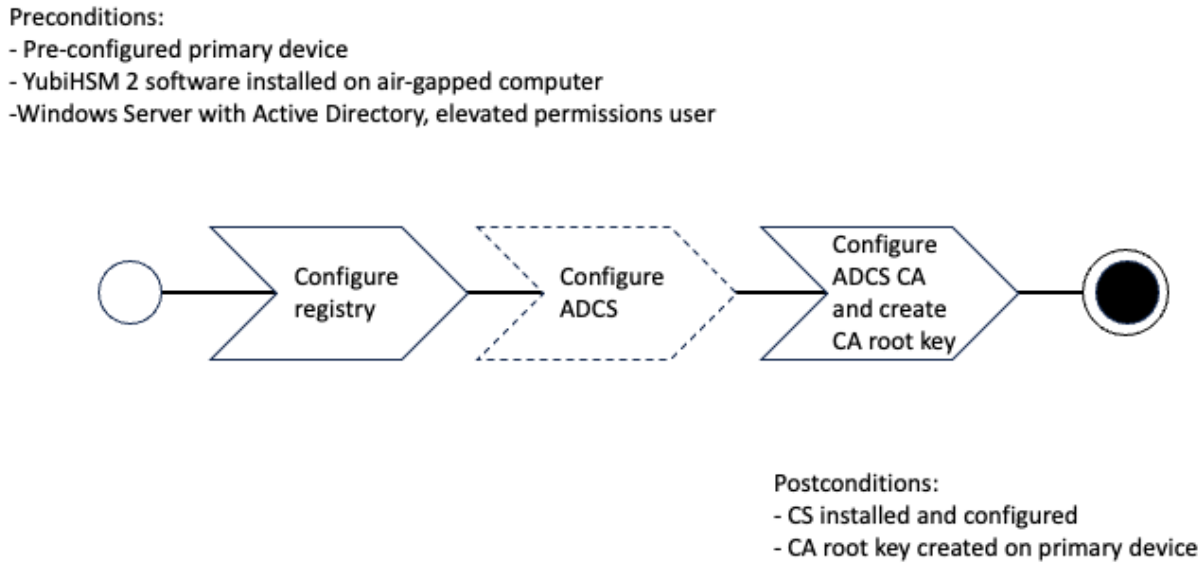
19.3 Deploying YubiHSM2 with ADCS Overview

With a YubiHSM 2 device now configured for use with YubiHSM Key Storage Provider and Microsoft Active Directory Certificate Services, the next set of steps covers the deployment in the ADCS environment. Note that YubiHSM Key Storage Provider software must be installed on the system before proceeding.

Deploying YubiHSM consists of three steps as follows. These steps are described in detail in the following procedure.

1. Configuring the Windows Registry for the YubiHSM Key Storage Provider for the primary YubiHSM 2 device that was configured earlier
2. Configuring ADCS (if not already present)
3. Configuring a new ADCS CA with a root CA key being generated on the device

Figure: Pre- and Post-Conditions



The host that these steps are performed on is assumed to be a member server in the Active Directory domain (domain-joined, not a Domain Controller).

These instructions include steps for a basic configuration and should be performed by an experienced system administrator.

19.4 Configuring the Windows Registry

For ADCS to use the YubiHSM 2, the following registry entries need to be changed from their default values. The `HKEY_LOCAL_MACHINE\SOFTWARE\Yubico\YubiHSM` subkey was created during installation. Be sure to make a backup of your Registry before you make any changes. To configure the Windows Registry:

1. Click **Start > Run**, type `regedit` in the Run dialog box, and click **OK**.
2. Locate and then click the registry subkey for YubiHSM (`HKEY_LOCAL_MACHINE\SOFTWARE\Yubico\YubiHSM`).
3. To change the URI where the connector is listening, change the following entry: `"ConnectorURL"=http://127.0.0.1:12345`
4. To change the ID of the application authentication key (object ID `3` was used as an example in this guide; if you used another object ID be sure to enter the correct information). For our example, because the hexadecimal value of `0x00000003` resolves to `3` in the Windows Registry, change the entry as follows: `"AuthKeysetID"=3`
5. To change the password for the application authentication key that is stored in the registry change the entry for: `"AuthKeysetPassword"={password}`. The password is stored here for the Key Storage Provider to use when authenticating to the device.
6. To save your changes, exit the Windows Registry.

   The YubiHSM Connector service reads the configuration file, `yubihsm-connector-config.yaml`.

   Depending on your local setup, for instance if you are running multiple instances of the software on the same host, you may need to edit this configuration file to make sure that parameters are consistent between the configuration file and the Windows Registry. On Windows, the `yubihsmconnector.config.yaml` file is available at `C:\programdata\yubiHSM\yubihsmconnector.yaml` - you will need administrator rights to modify the file.

19.5 Setting Up Your Enterprise Certificate Authority

19.5.1 To Configure ADCS

If you already have Certification Services installed, you can skip these steps.

1. On a Windows Server host, joined to an existing Active Directory domain, log on into the server as a domain administrator.
2. Click **Start > Administrative Tools**, then click **Server Manager**.
3. Under Roles Summary, click **Add roles and features**.
4. Use the Add Roles and Features Wizard to add the Active Directory Certificate Services role, and click **Next**.
5. In the Select role services wizard page, select the option for **Certification Authority**, then click **Next**.
6. Complete the wizard and reboot the host if prompted.

19.5.2 To Configure the ADCS CA and Create the Root Key

After you have completed the feature installation, you need to create the Enterprise CA instance.

1. If you haven't already, do the following:
    a. On a Windows Server host, joined to an existing Active Directory domain, log into the server as a `domain administrator`.
    b. Click **Start > Administrative Tools**, then click **Server Manager**.
2. In Server Manager, start the **Add Roles and Features Wizard** and select **Role-based** or **feature based** installation. Click **Next**.
3. In the Credentials page, confirm that you are logged in as a `domain administrator`. If you are not, you will not be able to create an Enterprise CA in the subsequent steps. Click **Next**.
4. In the Role Services page, select the option for **Certification Authority**, and then click **Next**.
5. In the Setup Type page, select the option for **Enterprise CA**, and then click **Next**.
6. In the CA Type page, select the option for **Root CA**, and then click **Next**.
7. In the Private Key page, select the option for **Create a new private key**, and then click **Next**.
8. In the Cryptography for CA page, do the following:
    a. Click **Select a cryptographic provider** and select **RSA#YubiHSM Key Storage Provider** from the list displayed. This indicates that the root key should be generated on the device.
    b. Click **Key Length** and select the key size you want from the list displayed. Options for key size `2048-bit` or `4096-bit`. The default setting is `2048`.
    c. For Select the hash algorithm for signing certificates issued by this CA, select a desired hash algorithm, such as `SHA256`. The default setting is `SHA256`.
    d. Select the option to **Allow administrator interaction when the private key is accessed by the CA**. This allows the private key to be exported for backup purposes (so it can be restored to another server). Click **Next**.
9. In the CA Name page, accept the defaults. Click **Next**.
10. In the Validity Period page, accept the default or set another validity period appropriate for your purposes. Click **Next**.

11. In the CA Database page, accept the default location for logs. Click **Next**.
12. In the Confirmation page, the important detail is that the `YubiHSM Key Storage Provider` is being used to store the CA private key. Click **Configure**.

    The Progress page appears, briefly, as the local CA database is created, and changes are written to Active Directory.
13. Finally, confirm the presence of the `Configuration succeeded` message in the Results page. Click **Close**.

CHAPTER

TWENTY

INSTALLING THE YUBIHSM 2 TOOLS AND SOFTWARE

To complete the procedures in this guide, install the YubiHSM 2 tools and software that will be needed for this.

Tip: A generic prompt, `$`, is used in command line examples in this document. Depending on your command line application, your prompt may be different.

20.1 About the YubiHSM Software

The following YubiHSM items of software are used in this guide. They are included as part of the archive file you downloaded.

20.2 Installation

1. Unzip the downloaded archives of the SDK containing the YubiHSM libraries and tools and move the contents to an appropriate location.
2. Complete the step for your operating system.
   - On your **Windows** system, run both installers:
     - `yubihsm-cngprovider-windows-amd64.msi` (YubiHSM Key Storage Provider)
     - `yubihsm-connector-windows-amd64.msi` (YubiHSM Connector for Windows)
   - On a **Debian**-based system, run the following command:

     ```
     $ dpkg -i ./libykhsmauth1_*.deb ./libyubihsm-usb1_*.deb ./
     libyubihsm-http1_*.deb ./libyubihsm1_*.deb ./yubihsm-shell_*.deb
     ```

   - On a **Redhat**-based system, run the following command:

     ```
     $ yum install ./yubihsm-shell-*.rpm
     ```

3. (**Windows** system) Set the ADCS service dependency for the YubiHSM Connector service via an elevated/admin Windows Command Prompt. This prevents an error which occurs if the ADCS services start before the YubiHSM connector is running.
   a. List the current dependencies with `sc qc "certsvc"`

```
> sc qc “certsvc”
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: certsvc
TYPE               : 110  WIN32_OWN_PROCESS (interactive)
START_TYPE         : 2   AUTO_START
ERROR_CONTROL      : 1   NORMAL
BINARY_PATH_NAME   : C:\Windows\system32\certsrv.exe
LOAD_ORDER_GROUP   :
TAG                : 0
DISPLAY_NAME       : Active Directory Certificate Services
DEPENDENCIES       :
SERVICE_START_NAME : localSystem
```

b. Add the YubiHSM Connector dependency to ADCS with the command: `sc config "certsvc" depend="yhconsrv"`

```
> sc config "certsvc" depend="yhconsrv"
[SC] ChangeServiceConfig SUCCESS
```

After the command is entered, the dependency can be verified with `sc qc “certsvc”`

```
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: certsvc
TYPE               : 110  WIN32_OWN_PROCESS (interactive)
START_TYPE         : 2   AUTO_START
ERROR_CONTROL      : 1   NORMAL
BINARY_PATH_NAME   : C:\Windows\system32\certsrv.exe
LOAD_ORDER_GROUP   :
TAG                : 0
DISPLAY_NAME       : Active Directory Certificate Services
DEPENDENCIES       : yhconsrv
SERVICE_START_NAME : localSystem
```

To remove dependencies for ACDS, use the same command for adding dependencies with a blank depend field: `sc config "certsvc" depend=""`

CHAPTER

TWENTYONE

VERIFYING THE DEFAULT CONFIGURATION OF THE YUBIHSM 2

Verify the results of the YubiHSM Setup program using the YubiHSM Shell program. Log in using the application authentication key.

The YubiHSM 2 device comes with a single factory-installed authentication key whose default password is `password`. As part of the configuration in this guide, this default authentication key will be destroyed. If the YubiHSM 2 is reset to its default configuration, any non factory-installed objects stored on it are also destroyed. Reset instructions can be found in *Resetting Device to Factory Settings*.

We reiterate that you will need two YubiHSM 2 devices to complete all steps of this guide, because you will be deploying the first device and creating a backup of all key material on the second device.

These steps also verify that neither of the YubiHSM 2 devices have been tampered with.

To verify that YubiHSM 2 devices still have the default configuration by following the steps below:

1. Verify the YubiHSM 2 setup, in your Command Prompt, run the following command:

   ```
   $ yubihsm-shell
   ```

   Do one of the following:

   - If the application that calls the YubiHSM Connector is **running on a local host**, start the Connector with the command `yubihsm-connector` without additional parameters. In Windows Server 2012 SP2 or higher, `yubihsm-connector.exe` is located in `C:\Program Files\YubiHSM Connector\`.
   - If the application is **running on a VM or a different server**, start the YubiHSM Connector on the host operating system in networking mode. For example, if the host machine's IP address is `192.168.100.252`, launch the Connector on the host OS with the command `yubihsm-connector -l 192.168.100.252:12345`

   Tip: For testing or debugging the YubiHSM Connector, the flag `-d` can be set.

2. To gain shell access to the YubiHSM 2, launch the YubiHSM Shell program:

   a. Open a Command Prompt.

   b. Run the command `yubihsm-shell`.

   c. If a networked Connector is used, set the parameter `--connect <connector URL>`.

      If the YubiHSM Connector is running on a host machine to which the YubiHSM 2 is physically connected, start the YubiHSM Shell program in networked mode.

      ```
      $ yubihsm-shell --connector http://192.168.100.252:12345
      ```

where –

The host server's IP address is 192.168.100.252

Tip: For testing or debugging the YubiHSM Shell, the flag `-d` can be set.

3. To connect to the YubiHSM 2, at the `yubihsm` command line, type `connect`. A message saying that you have a successful connection is displayed.
4. To open a session with the YubiHSM 2, type `session open 1` (where 1 is the ID of the default authentication key pre-installed on the device).
5. Type in the default password: `password`. A message confirming that the session has been set up successfully is displayed.
6. You now have an administrative connection to the YubiHSM 2 and you can list the objects available by typing `list objects 0` and pressing **Enter**. Your results should be similar to the following:

```
Found 3 object(s)
id: 0x0002, type: wrap-key, sequence: 0
id: 0x0003, type: authentication-key, sequence: 0
id: 0x0004, type: authentication-key, sequence: 0
```

As you can see by looking at their IDs, these objects correspond to the wrap key, the application authentication key and the audit key that were just created.

7. To obtain more information about any of the objects and its capabilities — for example, the application authentication key (object ID 3) — run the `objectinfo` command with the appropriate ID format, for example:

```
yubihsm> get objectinfo 0 3 authentication-key
```

The response you receive should look similar to the following:

```
id: 0x0003, type: authentication-key, algorithm:
aes128-yubico-authentication, label: "Application auth key", length:
40, domains: 1, sequence: 0, origin: imported, capabilities:
exportable-under-wrap:generate-asymmetric-key:
sign-attestation-certificate:sign-pkcs:sign-pss:sign-ecdsa,
delegated_capabilities:exportable-under-wrap:
generate-asymmetric-key:sign-attestation-certificate:sign-pkcs:
sign-pss:sign-ecdsa
```

8. Review the responses to confirm that YubiHSM 2 has now been configured to:
   - Generate asymmetric objects
   - Compute signatures using RSA-PKCS1v1.5
   - Compute signatures using RSA-PSS
   - Export other objects under wrap
   - Import wrapped objects
   - Mark an object as exportable under wrap

   In addition, this object (the application authentication key, object ID 3) also has delegated capabilities that can be bestowed on other objects that it creates. For more information on delegated capabilities, see *Capability*.
9. To exit, type `quit`.

CHAPTER

TWENTYTWO

CONFIGURING THE PRIMARY YUBIHSM 2 DEVICE

The YubiHSM Setup program, which is part of the YubiHSM 2 tool set, is used to perform the initial configuration of the primary YubiHSM 2 device. This program configures the device with the requirements needed for deploying YubiHSM 2 to safely store the ADCS root CA key. Specifically, during the setup process the YubiHSM is configured so that the necessary key material is generated on the device:

- One wrap key – The wrap key is split among a determined number of key custodians, and each share is recorded by each custodian. See *Key Splitting and Key Custodians*.
- One **application authentication key (auth key)** – The auth key for authenticating to the YubiHSM 2 through the KSP. This allows the KSP to perform operations in the YubiHSM 2.

  Note: This initial configuration replaces the default auth key with a new one, which will only be operable in the same domain as the asymmetric key. The *Domain* that is used to compartmentalize the YubiHSM 2 determines this behavior.

  Tip: For test purposes you can set the `yubihsm-setup -d` flag to keep the default auth key with the administrative privileges; this will allow you to delete keys on the YubiHSM 2 for test purposes only. For production purposes, however, the `yubihsm-setup` command must be executed without the `-d` flag to ensure that the factory preset auth key is properly deleted from the YubiHSM 2 device.

- One **audit key** – The audit key is used for accessing the internal audit log of the device and resetting the audit log. The audit log retains information about the last 62 operations. It is also used to purge the log if needed. Depending on your local requirements, you may not need to create an audit key. If you are unsure of your requirements, we suggest you create an audit key.

The auth key and the audit key are exported under wrap to a file in the current working directory on the machine where the YubiHSM Setup program is installed.

Tip: The YubiHSM Setup tool has a help argument that you can call to learn more about its usage.

Note: To safeguard the integrity of the device, configuration must be performed in an air-gapped environment.

22.1 Summary of Configuration Steps

After you have inserted the primary device into the air-gapped system, the configuration steps are diagrammed in the following image, and listed below. They are described in detail in the next section, *Configure Primary YubiSHM 2 Procedure*.

22.2 Configuration Steps

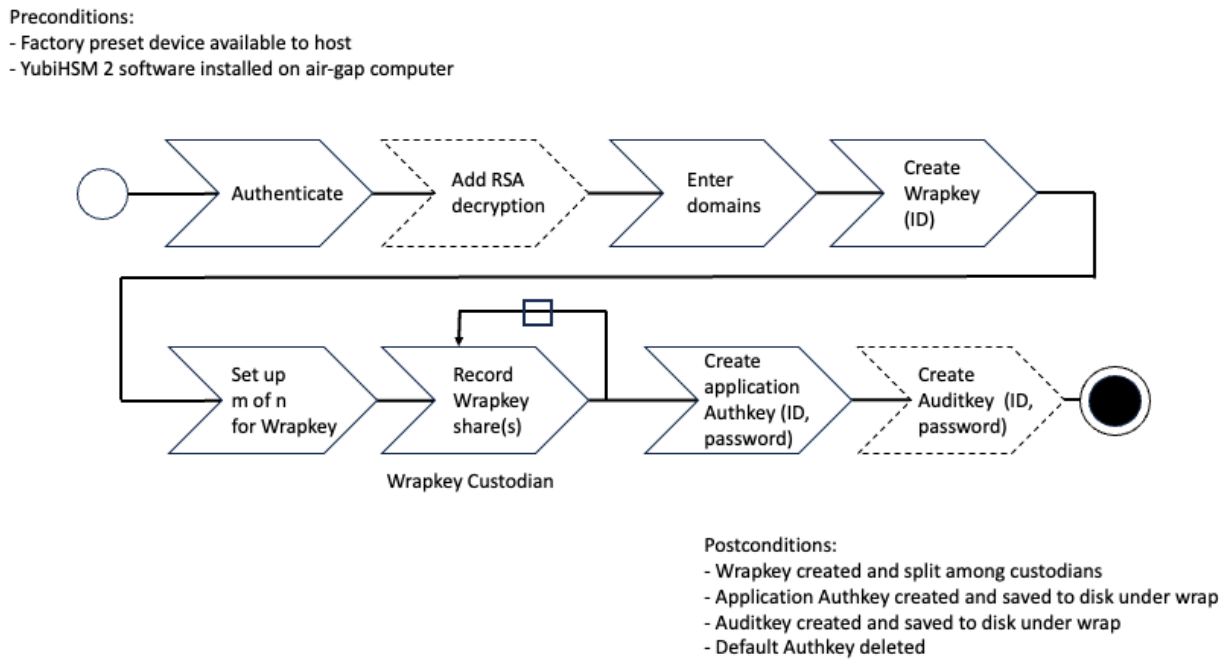


Figure: Pre- and Post-Conditions

1. Authenticate:
   a. Set up communication between the YubiHSM 2 tools and the device.
   b. Start the configuration process. Run the YubiHSM Setup with the argument `ksp`, specifying the Connector URL if necessary.
   c. Start the YubiHSM Setup process and authenticate to the YubiHSM device.
2. Add RSA decryption and capabilities if required. For example:
   - Active Directory Certificate Services (ADCS), does not require RSA decryption.
   - Microsoft SQL Server Always Encrypted, needs RSA decryption capabilities.
3. Enter Domains. Enter the names of the domains in which you need the auth key and audit key to be available.
4. Create the **wrap key** and its ID.
5. Setup `m of n` for Wrap key. Split the wrap key into shares and specify the privacy threshold.
6. Record Wrap key shares. Have the wrap key custodians record the number of shares required to rejoin the wrap key.

7. Create the **application authentication key (auth key)**. Includes creating the ID and password that are used to authenticate to the device by the KSP in Windows so the KSP can perform operations in YubiHSM 2.
8. Create the **audit application key (audit key)** (optional), include ID and password.

The original default auth key is deleted and setup process finishes.

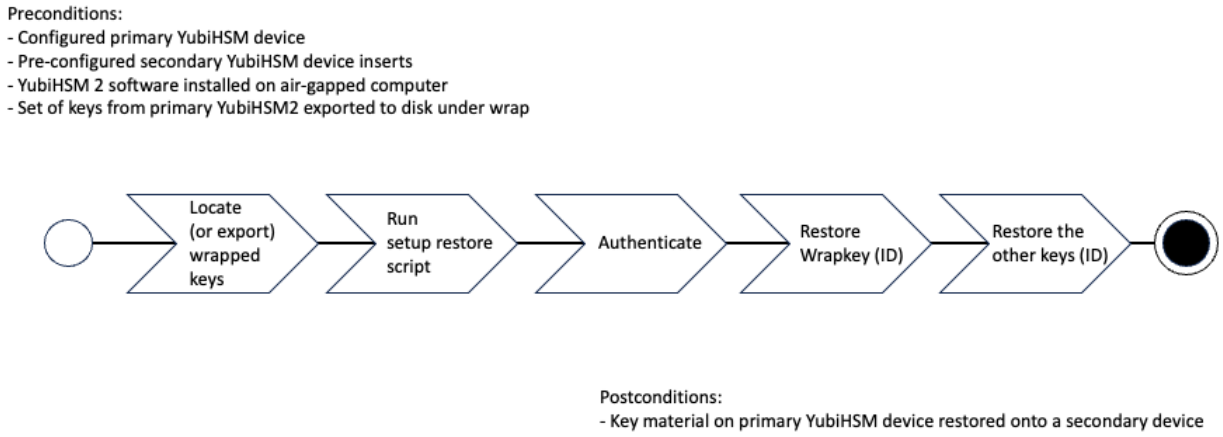


Figure: Flowchart illustrating the YubiHSM 2 setup for Windows

22.3 Configure Primary YubiSHM 2 Procedure

1. Authenticate
   a. Enable communication with the YubiHSM 2 device by ensuring that the YubiHSM Connector service (`yhconsrv` in Windows) is running the YubiHSM Connector on the system where the device is inserted.

      If the YubiHSM Connector is running on a host machine to which the YubiHSM 2 is physically connected, the Connector should be started in networked mode. For example, if the host IP address is 192.168.100.252, the Connector should be started on the host machine with the following command:

      ```
      yubihsm-connector -l 192.168.100.252:12345
      ```

      You can validate that the connector is running properly by typing the following URI into your browser: http://192.168.100.252:12345/connector/status. The output in the web browser should be similar to:

      ```
      status=OK
      serial=*
      version=1.0.0
      pid=*
      address=192.168.100.252
      port=12345
      ```

   b. Run YubiHSM Setup with the argument `ksp`. To do this:
      i. Launch your command line application.
      ii. Navigate to the directory for which you have write access and that contains the YubiHSM Setup program.
      iii. Run the YubiHSM Setup with the argument `ksp`.

```
yubihsm-setup ksp
```

If the application calling the Setup is installed on a machine other than the YubiHSM Connector, use the connector flag to specify the Connector URL, for example:

```
yubihsm-setup --connector http://192.168.100.252:12345 ksp
```

Tip: The setup tool also has a help argument that you can call to learn more about its usage.

c. Start the YubiHSM Setup process. Type the default auth key password: `password` and press **Enter**.

A message confirms that the default auth key was used and that you have successfully authenticated to the device: `Using authentication key 0x0001.`

Object IDs are displayed in the YubiHSM Setup Tool using hexadecimal numbers, in this case the default auth key has ID 1, or `0x0001` in hexadecimal.

2. You are prompted to add RSA decryption capabilities. Do one of the following:

- If you plan to use your YubiHSM 2 exclusively with an application that only needs signing capabilities, RSA decryption is not required. Active Directory Certificate Services (ADCS), for example, does not require RSA decryption.

  Type `n`.

- If you are planning on using the same YubiHSM 2 device for something that does require the capability to decrypt RSA, then you do need RSA decryption. The Microsoft SQL Server Always Encrypted, for example, needs RSA decryption capabilities.

  Type `y`.

Tip: If you are unsure what selection to make, type `n`.

3. At the prompt, enter the domain(s) you need the auth key and audit key to be available in.

The auth and audit keys are generated after you create the wrap key. You will only need one domain for the purposes of completing this guide. Do the following:

Unless you have a requirement to assign more than one domain, type a single number between 1 to 16 and press **Enter**.

In this guide, we assume that domain 1 was entered. Confirmation will look like the following:

```
got domains [
One
]
```

or

```
Using domains:
One
Enter wrap keyID (0 to choose automatically):
```

4. Generate a **wrap key** and enter its ID.

The wrap key is very important as it allows you to export and import objects from and to the device. For example, you would export and import objects for backup purposes, as described in *Backup and Restore Key Material*. Do one of the following:

- To manually assign a wrap key ID, type the number and press **Enter**. As object ID `1` is already in use by the default auth key, we recommend you assign id `2` to the wrap key. Type `2` and press **Enter**.
- To allow the system to assign a wrap key ID automatically, type `0` and press **Enter**.

In both cases, a confirmation message similar to the following is displayed:

```
Stored wrap key with ID 0x0002 on the device
```

5. Specify the number of shares to split the wrap key to distribute it equally among a number of key custodians. Also, specify the privacy threshold, which is the number of shares that must be present for the wrap key to be regenerated.

   For this example, we assume that the wrap key is split into three shares, out of which at least two shares must be present in order to use the key. If there are not two key custodians present, the wrap key cannot be rejoined. At the prompt, do the following:

   a. Enter the number of shares. In this example, enter `3`.

   b. Enter the privacy threshold. In this example, enter `2`.

   When defined, the three wrap key custodians each take their turn in front of the screen to record their respective share. A warning notice appears advising you that the shares are not stored anywhere.

6. Have each custodian record their key shares. Each custodian completes these steps.

   a. At the prompt, record their key share.

   b. Confirm their key share by typing `y` and press **Enter**.

   c. Turn it over to the next custodian. The screen buffer is cleared before each share is presented.

   The following is an example of a share presented on the screen:

```
2-1-WWmTQj5PHGJQ4H9Y2ouURm8m75QkDOeYzFzOX1VyMpAOeF3YKYZyAVdM0WY4GErclVuAC
Have you recorded the key share? (y/n)
```

   A notice is displayed, warning if the shares are not stored anywhere.

Note: Be sure to record the shares and store them safely if you want to re-use the wrap key for this device in the future.

Important: Each custodian must record the whole string presented, including the prefix (in this case) `2-1-` which indicates the number of shares determined to be required to rejoin (or the privacy threshold) and the number of the share itself out of the total number of shares being created.

Tip: For non-production and test purposes, such as in a lab scenario, it is not necessary to specify that the wrap key should be shared between key custodians but instead just use one solitary key. To do this, when configuring the device using YubiHSM Setup, indicate the number of shares to be `1` and the privacy threshold to also be `1`.

When this step is completed, the wrap key generated is saved to the HSM 2 device.

7. Create an **auth key**.

   The auth key is used to authenticate to the device by the Key Storage Provider (KSP) in Windows, allowing the KSP to perform operations in YubiHSM 2.

   a. Since object ID `1` and `2` are already in use by the default auth key and the wrap key respectively, the example in this guide assumes that the auth key to be created next gets ID `3`. Do one of the following:

      - To manually assign an auth key ID, type `3` and press **Enter**.
      - To instead allow the system to assign a wrap key ID automatically, type `0` and press **Enter**.

   b. Create and enter a password of at least eight (8) characters for the auth key.

      Be sure to store the password of the auth key that you will use in a way so that it cannot be compromised. You need this password later to configure the YubiHSM KSP DLL for use later. See *Configure the YubiHSM 2 Software on Windows*.

      Enter the auth key password and press **Enter**. A confirmation message appears.

      ```
      Stored auth key with ID 0x0003 on the device
      Saved wrapped auth key to {path} 0x0003.yhw
      ```

      The wrapped auth key (`0x0003.yhw`) has been saved to the same path as the location of the YubiHSM Setup program. Although encrypted using the wrap key, we recommend that you do not store keys - even under wrap - on a network-accessible or otherwise potentially comparable storage media.

      Leave the `\*.yhw-` file with the wrapped auth key where it was saved for now. It will be used later to create a backup. Delete the auth key **after** you make the backup.

8. Decide whether to create an **audit key**. To log into the YubiHSM 2 with the audit key, both the key ID and the password will be needed.

   The audit key is used to access the internal audit log of the device which holds information about the last 62 operations performed. It is also used to reset the log if needed. Depending on your local requirements, you may not need to create an audit key. If you are unsure of your requirements, we suggest you create an audit key.

   a. At the prompt to create an audit key, type `y`.

   b. Assign a key ID to the audit key.

      Make a note of the ID you enter (for example, key ID `4`).

   c. Enter the audit key password.

      Store this password so that it cannot be compromised.

   The audit key is exported under wrap to the current working directory. Using our example of key ID `4`, the file is named `0x0004.yhw`.

The setup tool finishes by letting you know that the default, factory-installed auth key has been deleted.

```
Previous authentication key 0x0001 deleted
All done
```

The YubiHSM Setup application exits. The YubiHSM 2 device is now equipped with the symmetric keys for wrap, audit, and application authentication.

22.4 Verifying the Setup

You can verify the results of the YubiHSM Setup program by using the YubiHSM Shell program, and logging in using the auth key (we used object ID 3 in this guide). To verify the YubiHSM Setup:

1. In your command line application (where $ is the prompt), run YubiHSM Shell program. To do this, if you haven't already, launch your command line application and navigate to the directory containing the YubiHSM Shell program. Then type the following command and press **Enter**.

```
$ yubihsm-shell
```

2. To connect to the YubiHSM, at the yubihsm prompt, type `connect` and press **Enter**. A message verifying that you have a successful connection is displayed.
3. To open a session with the YubiHSM 2, type `session open 3` and press **Enter**.
4. Type in the password for the auth key. You will receive a confirmation message that the session has been set up successfully.
5. You now have an administrative connection to the YubiHSM 2 and can list the objects available. To list the objects, type `list objects 0` and press **Enter**. Your results should be similar to the following:

```
Found 3 object(s)
id: 0x0002, type: wrapkey, sequence: 0
id: 0x0003, type: authkey, sequence: 0
id: 0x0004, type: authkey, sequence: 0
```

   As you can see by looking at their IDs, these objects correspond to the wrap key, the auth key and the audit key that were just created.

   To obtain more information about any one of the objects, for example, the auth key (object ID 3), including its capabilities, type the following command and press **Enter**:

```
yubihsm> get objectinfo 0 3 authentication-key
```

   The response you receive should look similar to the following:

```
id: 0x0003, type: authkey, algorithm: yubico-aes-auth,
label: "Application auth key", length: 40, domains: 1,
sequence: 0, origin: imported, capabilities:
asymmetric_gen:asymmetric_sign_pkcs:asymmetric_sign_pss:
export_wrapped: import_wrapped:export_under_wrap,
delegated_capabilities:
asymmetric_gen:asymmetric_sign_pkcs:asymmetric_sign_pss:
export_under_wrap
```

   This indicates that YubiHSM 2 as it has now been configured will later on allow the KSP to leverage the device to:

   - Generate asymmetric objects
   - Compute signatures using RSA-PKCS1v1.5
   - Compute signatures using RSA-PSS
   - Export other objects under wrap
   - Import wrapped objects
   - Mark an object as exportable under wrap

In addition, this object (the auth key, object ID 3) also has so-called delegated capabilities. Delegated capabilities define the set of capabilities that can be set or "bestowed" onto other objects that are created by it.

6. To exit, type `quit`.

CHAPTER

TWENTYTHREE

CONFIGURE THE YUBIHSM 2 SOFTWARE ON WINDOWS

Before using the YubiHSM 2 on Windows, there are two YubiHSM 2 software components to be configured:

- The YubiHSM 2 KSP.
- The YubiHSM 2 Connector service.

The configuration steps are described in the sections below.

Important: Make a backup of your Windows Registry before you make any changes.

23.1 Configure the KSP Settings in the Windows Registry

To enable Microsoft Cryptographic API Next Generation (CNG) to access the YubiHSM 2 KSP, the following registry entries must be changed from their default values. The YubiHSM 64-bit KSP subkey and the YubiHSM 32-bit KSP subkey were created during the YubiHSM SDK installation:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Yubico\YubiHSM
```

The edits to be made produce a result like the one illustrated below:

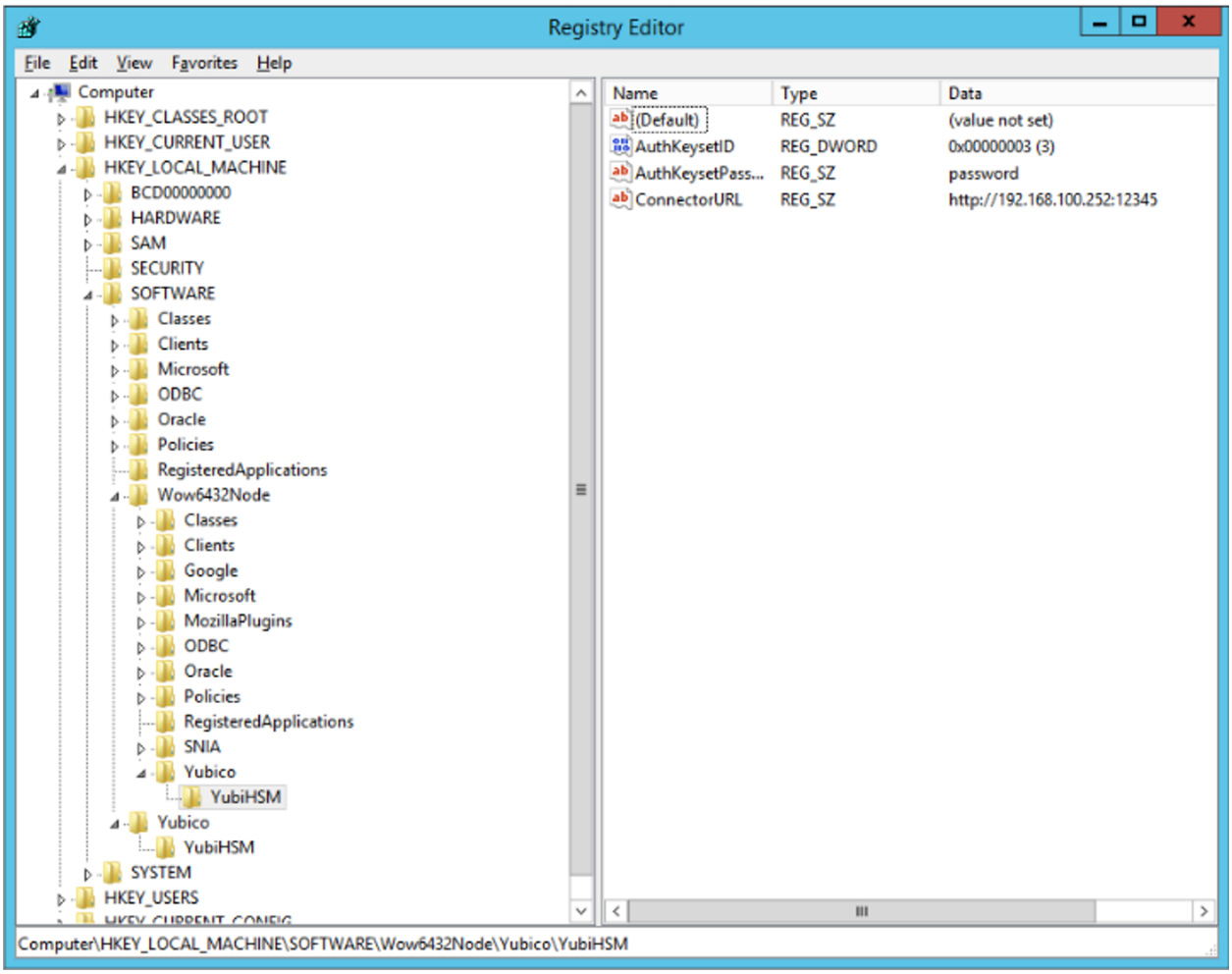


Figure: Registry settings for the YubiHSM 2 KSP

1. Click **Start > Run**, type `regedit` in the Run dialog box, and click **OK**.
2. Select the registry subkey for the **YubiHSM 64-bit KSP**.

   ```
   HKEY_LOCAL_MACHINE\SOFTWARE\Yubico\YubiHSM.
   ```

3. Change the URI to the IP address and port on which the YubiHSM 2 Connector is listening by editing the following registry entry appropriately, for example:

   ```
   "ConnectorURL"=http://127.0.0.1:12345
   ```

   If the Connector is listening on IP address and port `192.168.100.252:12345`, for example, the ConnectorURL value should be changed to:

   ```
   "ConnectorURL"=http://192.168.100.252:12345
   ```

4. Enter the ID of the application authentication key (object ID 3 was used as an example in this guide; if you used another object ID be sure to enter that). For our example, because the hexadecimal value of `0x00000003` resolves to 3 in the Windows Registry, change the entry to:

   ```
   "AuthKeysetID"=3
   ```

5. The application authentication key password is stored in the registry for the KSP to use when authenticating to the device. Enter the new password that you created:

```
"AuthKeysetPassword"={password}
```

6. Select the registry subkey for the `YubiHSM 32-bit KSP`.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Yubico\YubiHSM
```

7. Repeat steps 3-5 above.
8. To save your changes, exit the Windows Registry.

23.2 Configure the YubiHSM 2 Connector Service

The YubiHSM Connector service reads the configuration file `yubihsm-connector-config.yaml`. Depending on your local setup, for instance if you are running multiple instances of the software on the same host, you may need to edit this configuration file to ensure it is consistent with the Windows Registry, i.e., that the parameters and their values are the same in the configuration file and in the Windows Registry.

On Windows, the `yubihsmconnector.config.yaml` file is located at `C:\programdata\yubiHSM\yubihsmconnector.yaml` - you need administrator rights to modify the file.

CHAPTER

TWENTYFOUR

ALTERNATIVE SCENARIOS WITH CA ROOT KEY OR SUBORDINATE CAS

This guide covers only basic setup and use of the YubiHSM 2 with ADCS. Some alternative scenarios include migrating an existing CA root key to YubiHSM 2 or leveraging the YubiHSM 2 and YubiHSM Key Storage Provider in larger PKI installations using multiple hosts to serve the CA including Subordinate CAs. Since conditions can vary a great deal between organizations on these topics, the following contains some references that might be useful when deploying YubiHSM 2 under such circumstances.

24.1 Migrating an Existing CA Root Key to YubiHSM 2

One potential circumstance when deploying YubiHSM 2 to secure ADCS is the fact that a CA root key already exists, either in software or secured by hardware such as another Hardware Security Module. It is normally possible to migrate the CA root key over to the YubiHSM 2, however depending on the pre-existing setup, the steps to take may vary. For more information, see the information on the Yubico developers' website at Move Software Keys to Key Storage Provider.

24.2 Subordinate CAs

To improve security and scalability of your Certification Authority, consider installing the Root CA on a standalone (offline) server, and use a Subordinate CA for all certificate signing. For additional information about implementing advanced configurations, see the relevant Microsoft documentation, such as *AD CS Step by Step Guide: Two Tier PKI Hierarchy Deployment*.

You will need assistance from the wrap key custodians to provide their respective wrap key shares, if applicable. In the example we used in this guide, 2 out of the 3 shares must be available. When you create a backup, you create a duplicate of the objects on your primary YubiHSM 2 onto a secondary device. The actual backup procedure consists of steps as follows. These steps are described in detail in the following procedure.

1. Set up communication between the YubiHSM 2 tools and the device.
2. Start the configuration process and authenticate to the YubiHSM 2 device.
3. Identify the CA root key ID.
4. Export the CA root key.
5. Verify the key material under wrap.
6. Restore the key material onto a secondary (backup) device.
7. Verify the objects on the secondary device are correct.

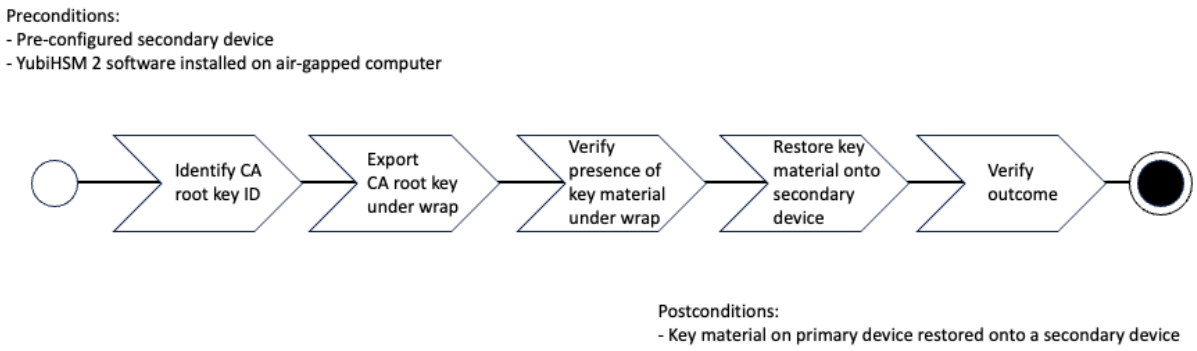


Figure: Pre and Post Conditions

Since the CA root key was created on the device when setting up the CA, it currently only exists on the device. To back it up using the YubiHSM Setup program, it must first be exported from the device using the wrap key that also sits on the device alongside the application authentication key and the audit key. To export the CA root key under wrap using the wrap key on the device:

1. In your command line application, run YubiHSM Shell program. To start the YubiHSM Shell program:
   a. Launch your command line application and navigate to the directory containing the YubiHSM Shell program.
   b. Type the following command and press **Enter**.

   ```
   $ yubihsm-shell
   ```

2. To connect to the YubiHSM, at the `yubihsm` prompt, type `connect` and press **Enter**. A message verifying that you have a successful connection is displayed.
3. To open a session with the YubiHSM 2, type `session open 3` and press **Enter**.
4. Type in the password for the application authentication key.

   You will receive a confirmation message that the session has been set up successfully.
5. If you already know the object ID of the root CA, you can skip this step. If you need to identify the root CA, you can list the objects available.
   a. To list the objects, type `list objects 0` (where `0` is the session number) and press **Enter**.
   b. You will receive a list of the objects on the device that application authentication key with ID `3` has access to, which will include the CA root key. Identify its ID.
6. To export the CA root key under wrap from the primary device to the local file system, in the YubiHSM Shell program, run:

   ```
   $ yubihsm> get wrapped 0 2 asymmetric-key {rootkeyID} rootkey.yhw
   ```

7. Verify that all the keys that were exported under wrap to file reside in the same directory as the YubiHSM Setup program. The tool looks for files with the `.yhw` file extension in the current working directory and attempts to read and import them into the device. The wrap key will be imported when you provide the wrap key shares to the tool. Given the example object IDs in this guide, the following files should be present:
   - `0x0003.yhw` (Application authentication key under wrap)
   - `0x0004.yhw` (Audit key under wrap)
   - `rootkey.yhw` (CA root key under wrap)

8. To begin the process of restoring the data onto the secondary YubiHSM 2, if the primary YubiHSM 2 device is inserted into your computer, remove it and insert the secondary device. Restoring a device must be performed in an air-gapped environment to guarantee integrity.
9. In your command line application (where `$` is the prompt), run YubiHSM Setup with the argument `restore`.
    a. Launch your command line application, navigate to the directory containing the YubiHSM Setup program,
    b. Type the following command, and press **Enter**.

```
$ yubihsm-setup restore
```

10. To start the YubiHSM Setup process, type the default authentication key password: `password` and press **Enter**.

    A confirmation message is displayed that the default authentication key was used and that you successfully have authenticated to the device: `Using authentication key 0x0001`

    You will now start the restore procedure, which involves providing the number of wrap key shares required by the privacy threshold defined when setting up the primary device.
11. When prompted, type the number of shares required by the privacy threshold and press **Enter**.

    In this guide, we have specified that 2 shares are required to be rejoined. These must be present to proceed.
12. When prompted, for share number 1, the wrap key custodian holding the first share inputs this information and presses **Enter**. A message is displayed that the share is received:

```
Received share  2-1␣
↪WWmTQj5PHGJQ4H9Y2ouURm8m75QkDOeYzFzOX1VyMpAOeF3YKYZyAVdM0WY4GErclVuAC
```

13. Continue to have each wrap key custodian enter the share information for each of the wrap key shares required to rejoin the key share. After a sufficient number of wrap key shares have been inserted by the wrap key custodians, a final message is displayed:

```
Stored wrap key with ID 0x0002 on the device
```

Note: The ID of the wrap key on the secondary device is the same as that for the primary device.

After the wrap key has been stored on the secondary device, the YubiHSM Setup program reads the files containing the application authentication key, the CA root key, and, if applicable, the audit key that were saved to file under wrap during the configuration of the primary device.

```
reading ./0x0004.yhw
Successfully imported object Authkey, with ID 0x0004
reading ./0x0003.yhw
Successfully imported object Authkey, with ID 0x0003
reading ./rootkey.yhw
Successfully imported object Asymmetric, with ID {rootkeyID}
```

If there are files containing wrapped objects with the `.yhw` file extension in this directory that were exported with a different wrap key than the one reconstituted by the shares here, the setup tool attempts to also read those but will fail gracefully and only restores the files it can decrypt.

The restore process finishes and the setup tool lets you know that the default, factory-installed authentication key has been deleted.

```
Previous authentication key 0x0001 deleted
All done
Finally, the YubiHSM Setup application exits.
```

24.2.1 Confirming the Duplicated YubiHSM 2

You now have a duplicate of the device configured with the three key objects you created on the primary device earlier. These are identical to the primary device that was configured earlier.

To confirm the duplicated YubiHSM 2:

1. In your command line application, run YubiHSM Shell program.
   a. Launch your command line application and navigate to the directory containing the YubiHSM Shell program.
   b. Enter the following command and press **Enter**.

   ```
   $ yubihsm-shell
   ```

2. To connect to the YubiHSM, at the `yubihsm` prompt, type `connect` and press **Enter**. A message verifying that you have a successful connection is displayed.
3. To open a session with the YubiHSM 2, type `session open 3` (where `3` is the ID for your application authentication key) and press **Enter**.
4. Type in the password for the application authentication key. You will receive a confirmation message that the session has been set up successfully.
5. To list the objects, type `list objects 0` (or instead of `0` some other session number that was given to you in step 4) and press **Enter**. Verify that the secondary device now contains all of the key material that you intended to restore.

   Depending on the order in which the keys under wrap were imported, the order of the enumerated keys on the secondary device may be different than on the primary device when using the list command. This has no practical implementation and the object IDs are identical between the devices.
6. If you have verified that the secondary device now contains all of the key material that you intended to restore, you should now remove the keys under wrap currently on file in the current working directory for the YubiHSM Setup program.

CHAPTER

TWENTYFIVE

BACKUP AND RESTORE KEY MATERIAL

We strongly recommend you make a backup copy of all production objects residing on your primary device, particularly once the CA root key has been generated on the device. If there is an unforeseen hardware failure of the primary device, having a backup ensures that you can resume operations quickly. In addition, this provides a means to backup all objects contained on a device to reside in secure hardware offline.

The backup process will result in two identical YubiHSM 2 devices with the same number of objects, keys, labels, etc.

Note: Specific recommendations for governance of your critical key material is out of scope for this guide. Ensure that you design and document these security procedures to fit the requirements of your organization. In many cases, they are subject to audits.

25.1 Backup the YubiHSM 2 Overview

The backup of the primary YubiHSM 2 is a duplicate of all of the objects stored on the primary device, to be exported under wrap and that are available using the application authentication key used.

The procedure described in this section is appropriate for testing and for smaller installations. For larger and/or more complex installations such as:

- Those whose setup did **not** involve the YubiHSM Setup program
- When moving the YubiHSM 2 device from one server to another

Review the information at *YubiHSM 2: Backup and Restore* to determine whether the procedures set out there are more appropriate for your situation.

This guide gives instructions for duplicating the following on the secondary device:

- Wrap key (previously created with ID 2),
- Application authentication key (ID 3),
- Audit key (ID 4) (if created previously)

The listed objects are exported under wrap.

The factory-installed authentication key (ID 1) on the secondary YubiHSM 2 device will be destroyed, just as it was on the primary YubiHSM 2 device.

If you use actual wrap key custodians (instead of just doing a proof of concept), you will need the custodians to provide their respective wrap key shares. In the example we used in this guide, 2 out of the 3 custodians/shares must be available.

To guarantee integrity, perform these operations in an air-gapped environment.

25.2 Backup and Restore the YubiHSM 2 Procedure Overview

The backup, see *YubiHSM 2: Backup and Restore*, of the primary YubiHSM 2 is a duplicate of all of the objects stored on the primary device. The objects are exported under wrap onto the secondary device. The objects are available using the same application authentication key used.

For instance, when following this guide, the wrap key (created with ID 2 previously), the application authentication key (ID 3), the audit key (ID 4) (if created previously), and the CA root key will be duplicated onto the secondary device. The factory-installed authentication key (ID 1) on the secondary device will be destroyed. You will need assistance from the wrap key custodians to provide their respective wrap key shares, if applicable.

In the example we used in this guide, 2 out of the 3 shares must be available. When you create a backup, you create a duplicate of the objects on your primary YubiHSM 2 onto a secondary device. The actual backup procedure consists of steps as follows. These steps are described in detail in the following procedure.

The backup and restore procedure consists of the steps listed below the following diagram. The steps are explained in detail in the section,:ref:*hsm2-restore-keys-secondary-yubihsm2-device-label*.

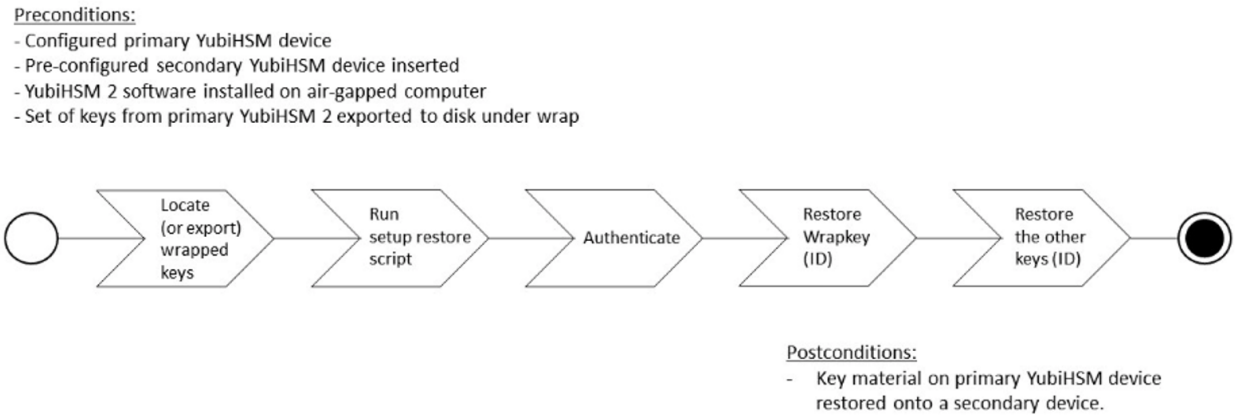


Figure: Flowchart illustrating backup and recovery of YubiHSM 2 keys

1. Locate the wrapped key material that was previously exported by the steps in *Configuring the Primary YubiHSM 2 Device*.
2. Set up communication between the YubiHSM 2 tools and the secondary (backup) YubiHSM 2 device.
3. Start the configuration process and authenticate to the secondary YubiHSM 2 device.
4. Identify the CA root key ID.
5. Export the CA root key.
6. Verify the key material under wrap.
7. Restore the key material onto a secondary (backup) YubiHSM 2 device.
8. Verify the objects on the secondary device are correct.

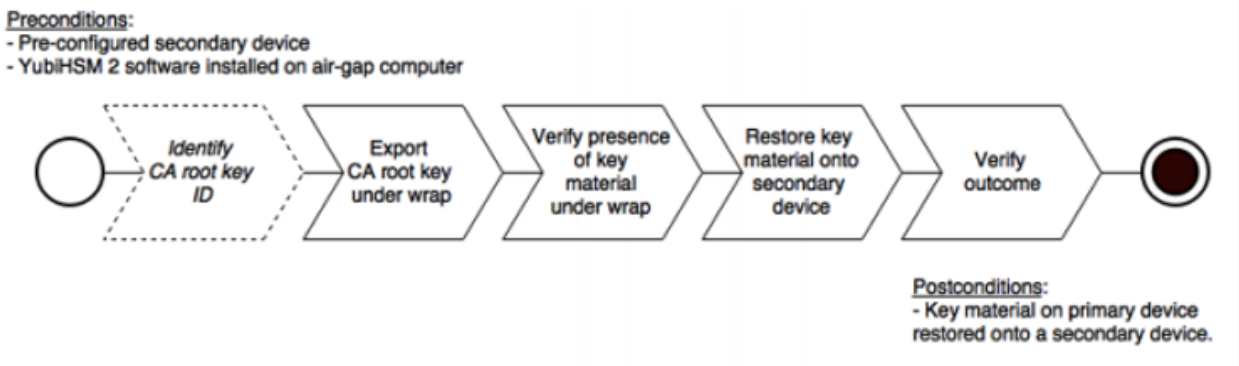


Figure: Pre and Post Conditions

Tip: For **test purposes** you can set the `yubihsm-setup -d` flag to keep the default authentication-key with the administrative privileges. This allows you to delete keys on the YubiHSM 2 for test purposes only. For **production purposes** however, the `yubihsm-setup` command must be executed without the `-d` flag to ensure that the factory preset authentication key is properly deleted on the YubiHSM 2.

25.3 Restore Keys on the Secondary YubiHSM 2 Device

Since the CA root key was created on the device when setting up the CA, it currently only exists on the device. To back it up using the YubiHSM Setup program, it must first be exported from the device using the wrap key that also sits on the device alongside the application authentication key and the audit key. To export the CA root key under wrap using the wrap key on the device:

1. In your command line application, run YubiHSM Shell program. To do this, if you haven't already:
   a. Launch your command line application and navigate to the directory containing the YubiHSM Shell program.
   b. Then run the following command and press **Enter**.

   ```
   $ yubihsm-shell
   ```

2. To connect to the YubiHSM, at the `yubihsm` prompt, type `connect` and press **Enter**. A message verifying that you have a successful connection is displayed.
3. To open a session with the YubiHSM 2, type `session open 3` and press **Enter**.
4. Type in the `password` for the application authentication key.

   You will receive a confirmation message that the session has been set up successfully.
5. If you already know the object ID of the root CA, you can skip this step. If you need to identify the root CA, you can list the objects available.
   a. To list the objects, type `list objects 0` (where `0` is the session number) and press **Enter**.
   b. You will receive a list of the objects on the device that application authentication key with ID `3` has access to, which will include the CA root key. Identify its ID.
6. To export the CA root key under wrap from the primary device to the local file system, in the YubiHSM Shell program, run

```
yubihsm> get wrapped 0 2 seed=0 asymmetric-key {rootkeyID} rootkey.yhw
```

Where `seed=0` does not export a privacy key seed. See *EXPORT WRAPPED Command*.

7. Verify that all the keys, that were previously exported from the primary YubiHSM 2 under wrap, reside in the same directory as the YubiHSM Setup program and that you have read access to that directory.

   If the necessary keys are not yet all available on disk, export the keys under wrap. Run the following command:

   ```
   yubihsm-setup dump
   ```

   The YubiHSM Setup tool looks for files with the `.yhw` file extension in the current working directory and attempts to read and import them into the YubiHSM 2 device. The wrap key itself will be imported when the wrap key shares are provided to the tool. For example, the following files may be present:

   - `0x0003-AuthenticationKey.yhw` (Application authentication key under wrap)
   - `0x0004-AuthenticationKey.yhw` (Audit key under wrap)
   - `rootkey.yhw` (CA root key under wrap)
   - `x427a-Opaque.yhw` (Certificate under wrap - not referenced by this guide in the configuration of the primary HSM 2)
   - `x427a-AsymmetricKey.yhw` (Private asymmetric key under wrap - not referenced by this guide in the configuration of the primary HSM 2)

   If the initial authentication key (by default available as ID 0x0001) has been deleted, the new authentication application key is identified with the flag `yubihsm-setup --authkey`. For example:

   ```
   $ yubihsm-setup --authkey 0x0003 dump
   ```

8. To begin the process of restoring the data onto the secondary YubiHSM 2, if the primary YubiHSM 2 device is inserted into your computer, remove it and insert the secondary device.

   Important: Restoring a device must be performed in an air-gapped environment in order to guarantee integrity.

9. In your command line application (where `$` is the prompt), run YubiHSM Setup with the argument `restore`.

   a. Change the directory containing the `*.yhw` files,

   b. Run `yubihsm-setup` with the `restore` argument:

   ```
   $ yubihsm-setup restore
   ```

10. To start the YubiHSM Setup process. Type the default authentication key password: `password` and press **Enter**.

    A confirmation message confirms that the default authentication key was used and that you successfully have authenticated to the YubiHSM 2 device:

    ```
    Using authentication key 0x0001
    ```

11. When prompted, type the minimum number of wrap key shares required by the privacy threshold and press **Enter**.

    The require number of wrap key shares were defined when you set up the primary YubiHSM 2 device. In this guide, we have specified that 2 shares are required to regenerate the key. These must be present in order to proceed.

12. When prompted for share number 1: Have the wrap key custodian holding the first share input this information and press **Enter**. A message confirms that the share is received:

```
Received share 2-
↪1WWmTQj5PHGJQ4H9Y2ouURm8m75QkDOeYzFzOX1VyMpAOeF3YKYZyAVdM0WY4GErclVuAC
```

13. Continue to have each wrap key custodian enter the share information for each of the wrap key shares required to regenerate the wrap key. When the sufficient number of wrap key shares have been entered by the wrap key custodians, a final message is displayed indicating that the wrap key from the primary YubiHSM 2 is now on the secondary YubiHSM 2 as well:

```
Stored wrap key with ID 0x0002 on the device
```

Note: The ID of the wrap key on the secondary device is the same as the ID of the wrap key on the primary device.

---

14. Review the output to verify Certificate Authority (CA) root key was also generated and exported along with a private asymmetric key, both under wrap.

    After the wrap key has been stored on the secondary YubiHSM 2 device, the YubiHSM Setup program reads the files containing the application authentication key, the CA root key, and, if applicable, the audit key that were saved to file under wrap during the configuration of the primary device.

    The output below shows that in this case, the Certificate Authority (CA) root key was also generated and exported along with a private asymmetric key, both under wrap.

```
reading ./0x0004.yhw
Successfully imported object Authkey, with ID 0x0004
reading ./0x0003.yhw
Successfully imported object Authkey, with ID 0x0003
reading ./0x427a-AsymmetricKey.yhw
Successfully imported object Asymmetric, with ID 0x427a
reading ./0x427a-Opaque.yhw
Successfully imported object Opaque, with ID 0x427a
reading ./rootkey.yhw
Successfully imported object Asymmetric, with ID {rootkeyID}
```

15. Review the output to note if there are files containing wrapped objects with the `.yhw` file extension in this directory that were exported with a wrap key **other than** the one reconstituted by the shares here. The Setup tool attempts to read those too, but fails gracefully. The Setup tool only restores the files it can decrypt.

16. Wait for the restore process to finish and the setup tool informs you that the default, factory-installed authentication key has been deleted.

```
Previous authentication key 0x0001 deleted
All done
```

    The YubiHSM Setup application exits.

25.4 Verify the Duplicated YubiHSM 2

With the steps in the previous sections completed, you now have a secondary (duplicate) of the YubiHSM 2 device configured with the three key objects you created on the primary YubiHSM 2 device earlier.

Confirm that the key objects are identical on both the secondary (configured in previous section) and the primary device (configured earlier).

1. At your command prompt, run the YubiHSM Shell program. To do this, if you haven't already:
   a. Launch your command line application and navigate to the directory containing the YubiHSM Shell program.
   b. Then run the following command and press **Enter**.

   ```
   $ yubihsm-shell
   ```

2. Connect to the YubiHSM 2, at the `yubihsm` prompt, type `connect` and press **Enter**. A message confirms that you have a successful connection.
3. Open a session with the YubiHSM 2, type `session open 3` and press **Enter**.

   where - `3` is the ID for your application authentication key.
4. Type the *password* for the application authentication key. A message confirms that the session has been set up successfully.
5. List the objects, type `list objects 0` and press **Enter**.

   where - `0` is session number that was given to you in step 4. Replace `0` with your session number, if it is different.
6. Review the output and verify that the secondary device now contains all of the key material that you intended to restore.

   Depending on the order in which the keys under wrap were imported, the order of the enumerated keys on the secondary device may be different than on the primary device when using the list command. This has no practical implication and the object IDs are identical between the devices.
7. After you verify that the secondary device contains all of the key material that you intended to restore, remove the keys under wrap currently on file in the current working directory for the YubiHSM Setup program. The computer's hard drive can be erased.

CHAPTER

TWENTYSIX

DEPLOYING YUBIHSM 2 FOR MICROSOFT HOST GUARDIAN SERVICE (HGS) GUIDE

In a Microsoft Host Guardian Service (HGS) environment, the signing key and the encryption key must be protected in hardware. The YubiHSM 2 protects these keys in hardware and thereby guards the HGS.

This guide is intended to help systems administrators deploy YubiHSM 2 for use with HGS in a Windows server environment. The expected outcome is that the signing key and the encryption key are generated and stored securely on a YubiHSM 2 and that a hardware-based backup copy of key materials has been produced.

These guidelines for deployment cover basic topics, so the instructions should be modified as required for your particular environment. It is assumed that you are familiar with the concepts and processes for working with HSG. It is also assumed that the installation is performed on a single HSG, but the concept can be extended to multiple servers.

Important: We recommend that you install and test the HGS installation and setup of the YubiHSM 2 in a test or lab environment before deploying to production.

For guidance on enabling the HGS in a production environment, see Microsoft's documentation on how to deploy a guarded fabric and shielded virtual machines (VMs).

26.1 The Host Guardian Service – Guarded Fabric Concept

In order to raise the security level for virtualization, Microsoft Windows Server 2016 introduced the concept of Guarded Fabric to increase the security of Hyper-V Virtual Machines (VMs). A guarded fabric is used to protect hosts from a VM running malicious software and to protect VMs from a compromised host.

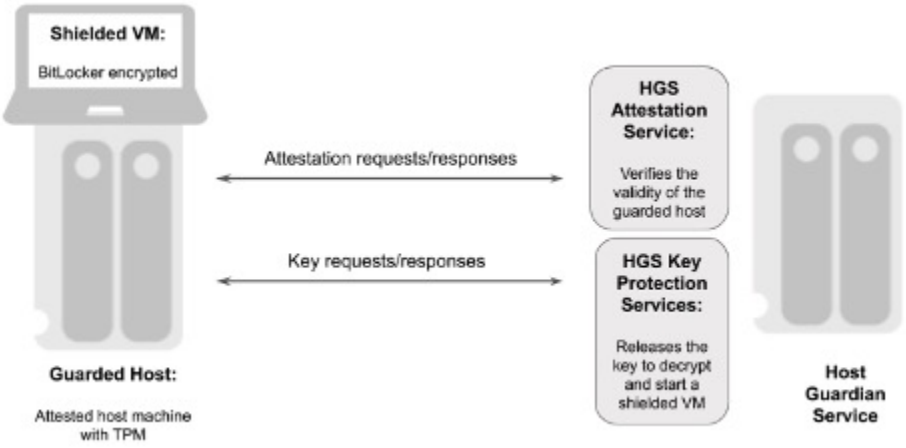


Figure: Overview of a Guarded Fabric and Main Components

A guarded fabric is comprised of the following main components:

- Host Guardian Service (HGS) This is a Windows Server role that is typically installed on a cluster of physical servers. The HGS in turn is composed of the Attestation Service and the Key Protection Service. The Attestation Service verifies the Trusted Computing Group (TCG) logs of a guarded host and issues a health certificate if the Guarded Host is attested by HGS. The HGS Key Protection Service is described in "HGS Key Protection Service" below.
- Guarded Host This is an attested host machine, equipped with a Trusted Platform Module (TPM) that can run shielded Hyper-V VMs. The guarded Hyper-V host must be attested by the HGS Attestation Service in order to power on or migrate shielded VMs.
- Shielded VM This is a Hyper-V VM equipped with a virtual TPM, that is encrypted using BitLocker and can run only on attested guarded hosts in a guarded fabric.

The guarded fabric components are described in Microsoft's overview of guarded fabric and shielded VMs.

26.2 HGS Key Protection Service

The HGS Key Protection Service (KPS) is configured with at least two certificates (and corresponding private keys), which are used for signing and encrypting the keys used to start up shielded VMs. The two mandatory certificates are:

- **Encryption certificate**: This certificate is used to encrypt and decrypt the key protector, which itself contains the symmetric key that encrypts the virtual TPM of a shielded VM at rest. When a shielded VM is booting up on an attested guarded host, the HGS KPS decrypts and releases its symmetric key, which is used by the guarded host to decrypt the virtual TPM and the hard drive of a shielded VM.
- **Signing certificate**: This certificate is used to digitally sign the key protector to ensure its authenticity.

In addition to these mandatory certificates, the HGS KPS can also be configured with four optional certificates:

- Communications certificate
- Attestation signer certificate
- HTTPS (SSL/TLS) certificate
- Dump encryption certificate.

If those certificates are not configured, the Encryption certificate and Signing certificate will provide the necessary operations.

The Encryption certificate and Signing certificate can either be self-signed or issued by a Certification Authority (CA).

The private keys corresponding to the certificates can be stored in an HSM or in software in PKCS #12 format. The recommended option is to protect the keys in hardware in an HSM.

For more information on these topics, see Frequently Asked Questions About HGS Certificates in the Microsoft Tech Community (requires community login).

26.3 Scope of this Guide

The scope of this guide is to describe how to use the HGS KPS to generate the Encryption and Signing certificates/keys using the YubiHSM. In this document, the Encryption and Signing certificates will be self-signed and created with PowerShell scripts.

How to use CA to issue the certificates is out of scope for this guide.

How to deploy and configure the HGS Attestation Service, guarded hosts, shielded VMs, and additional features of a guarded fabric are also out of scope for this guide.

For information on how to install and configure a complete guarded fabric, see Microsoft's documentation on guarded fabric deployment.

26.4 Prerequisites and Preparations

The audience of this document is an experienced systems administrator with a good understanding of Microsoft Hyper-V virtualization management. In addition, it is helpful to be familiar with the terminology, software, and tools specific to YubiHSM 2. As a primer for these terms, see the *Glossary*.

To complete the steps provided in this guide, complete the following prerequisites:

- Microsoft Windows Server 2016 or higher. The operating system should be installed in a secure computer network. The system administrator must also have elevated system privileges.
- YubiHSM 2 software and tools for Windows downloaded from the Yubico YubiHSM 2 Release page and available on the system to be used.
- Two (2) YubiHSM 2 devices, one for deployment and one for backup in hardware.
- Your organization's policies may require key custodians to be available for the YubiHSM 2 deployment. For more information about key custodians and the associated `M of N` key shares, see *YubiHSM 2 SDK Tools And Libraries* in the YubiHSM 2 Windows Deployment Guide.

Configuration for this Integration For the integration described in this guide, the following hardware and software configuration was used:

- Microsoft Windows Server 2016.
- Yubico YubiHSM v 2.1.2.
- Yubico YubiHSM v 2.1.2 software tools.

26.5 Basic Setup of YubiHSM 2 and Host Guardian Service

26.5.1 Install and Configuring YubiHSM 2

Install and configure the YubiHSM 2 and software using the instructions in the following sections in the *YubiHSM 2 with Key Storage Provider for Windows Server*.

1. *Installing the YubiHSM 2 Tools and Software*
2. *Configuring the Primary YubiHSM 2 Device*
3. *Configure the YubiHSM 2 Software on Windows*

Once these instructions have been followed, the YubiHSM 2 should be configured with the example we are using, one domain with a wrap key (`id 0x0002`), an application authentication key (`id 0x0003`), and an audit key (`id 0x0004`). The configuration of the YubiHSM 2 can be inspected by using the YubiHSM-Shell in a command prompt as shown in the screenshot below.

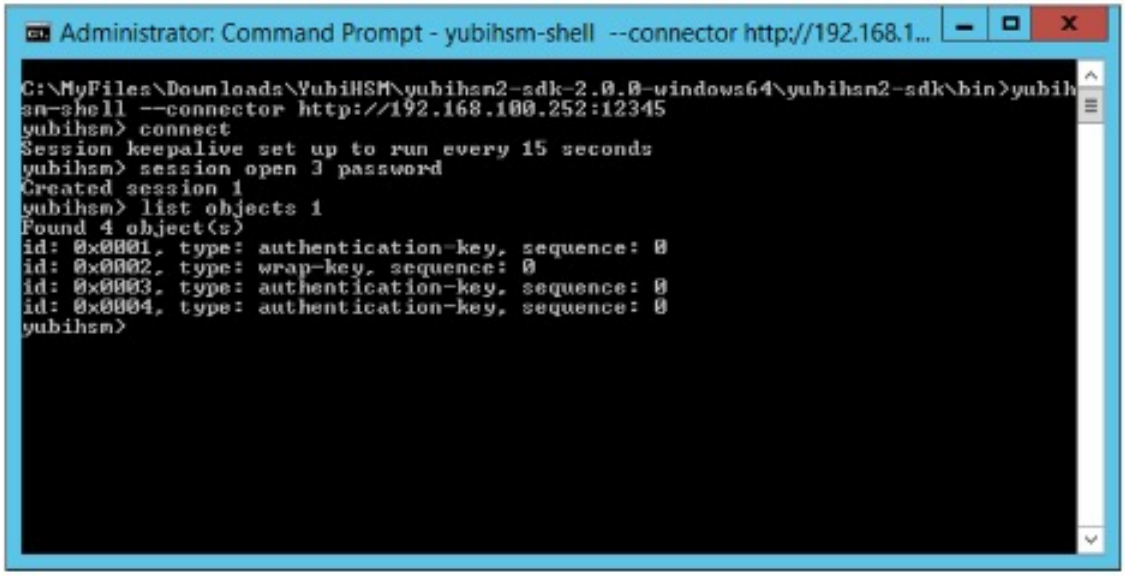


Figure: Example of the YubiHSM 2 Basic Configuration

26.5.2 Basic Deployment of HGS

To test the encryption and signing certificate/key generation for HGS Key Protection Services, configure a basic HGS environment on a single server. For more information on how to install and configure a complete guarded fabric, see Microsoft's documentation on guarded fabric deployment.

To use shielded VMs, begin by adding the HGS role and configuring the HGS domain. In the following, we are showing the PowerShell prompt as `PS C:\users\your-username\`.

1. Add HGS Role.

   To add the HGS role to a Windows Server, open a PowerShell console and enter the following command:

   ```
   PS C:\users\your-username\ Install-WindowsFeature -Name HostGuardianServiceRole -
   ↪IncludeManagementTools -Restart
   ```

   For more information on this PowerShell command, see Microsoft's documentation on how to Install HGS.

2. Install Host Guardian Server on Bastion Host.

   To configure the Active Directory (AD) forest for HGS, configure the HGS service, and lock down the Windows Server to a bastion host, open a PowerShell console and enter the following command:

   ```
   PS C:\users\your-username\ $adminPassword = ConvertTo-SecureString -AsPlainText '
   ↪<password>' -Force

   PS C:\users\your-username\ Install-HgsServer -HgsDomainName 'bastion.local' -
   ↪SafeModeAdministratorPassword $adminPassword -Restart
   ```

   For more information on this PowerShell command, see Microsoft's documentation on how to Install HGS.

26.6 Create Signing and Encryption Keys for HGS

26.6.1 Generate Signing and Encryption Keys and Certificates

Generate the signing and encryption keys and certificates for HGS by using the PowerShell cmdlet `New-SelfSignedCertificate`. In this guide, self-signed certificates will be used for HGS.

The HGS signing and encryption certificates must adhere to the following specifications:

- Crypto provider: YubiHSM Key Storage Provider.
- Key algorithm: RSA
- Minimum key size: 2048 bits
- Signature algorithm: SHA256
- Key usage: Digital signature and data encipherment
- Enhanced key usage: Server authentication
- Subject name: Recommended: your company's name or web address

Do the following to create the self-signed HGS certificates:

1. Create the Self-signed HGS Signing Certificate and Key.

   Start a command prompt with administrator rights and type the command `PowerShell`. In the PowerShell command prompt, run the following cmdlet:

```
PS New-SelfSignedCertificate -Provider "YubiHSM Key Storage Provider" -Subject
↪"CN=HGS Signing Certificate" -KeyExportPolicy NonExportable -KeyUsage␣
↪DigitalSignature,DataEncipherment -TextExtension @("2.5.29.37={text}1.3.6.1.5.5.7.
↪3.1") -KeyAlgorithm RSA -KeyLength 2048 -CertStoreLocation "Cert:\LocalMachine\My
↪" -Verbose
```

2. Create the Self-signed HGS Encryption Certificate and Key.

   In the PowerShell command prompt, run the following cmdlet:

```
PS C:\users\your-username\ New-SelfSignedCertificate -Provider "YubiHSM Key Storage␣
↪Provider" -Subject "CN=HGS Encryption Certificate" -KeyExportPolicy NonExportable␣
↪-KeyUsage DigitalSignature,DataEncipherment -TextExtension @("2.5.29.37={text}1.3.
↪6.1.5.5.7.3.1") -KeyAlgorithm RSA -KeyLength 2048 "Cert:\LocalMachine\My" -Verbose
```

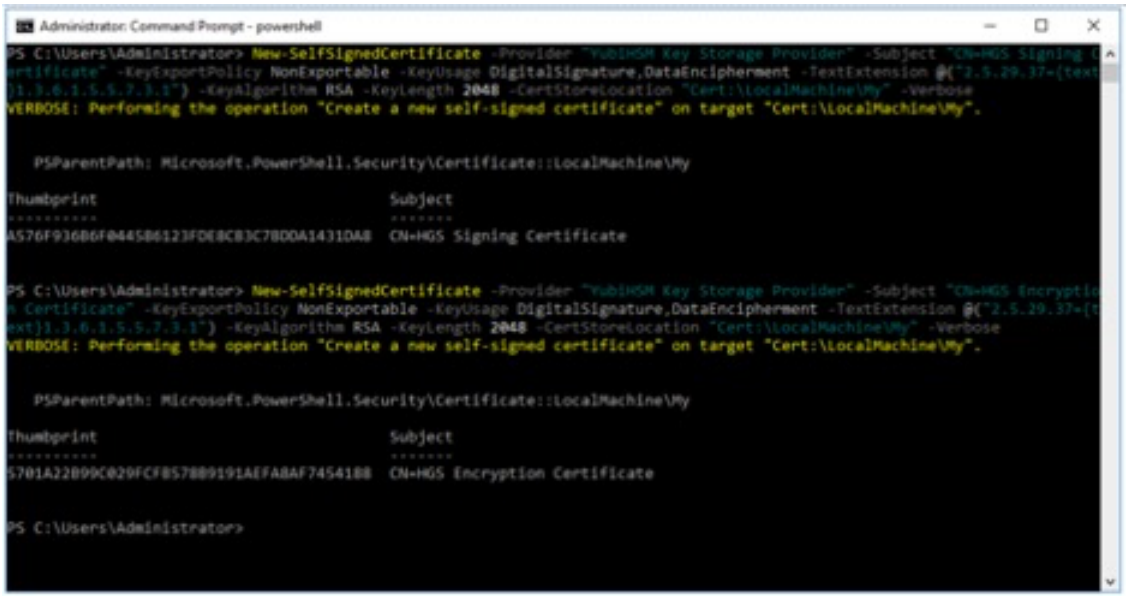


Figure: Example of PowerShell cmdlet to Create Self-Signed Certificates

Make a note of the thumbprints of the self-signed certificates. In this example, the signing certificate thumbprint is `A576F936B6F044586123FDE8CB3C7BDDA1431DA8` and the encryption certificate thumbprint is `5701A22B99C029FCFB578B9191AEFA8AF7454188`.

3. Verify Generation and Storage of HGS Key-pairs in YubiHSM 2.

   Verify that the HGS key-pairs have been properly generated and stored in YubiHSM 2 by starting a command prompt and using `YubiHSM-Shell` to list the objects, as shown in the figure below.

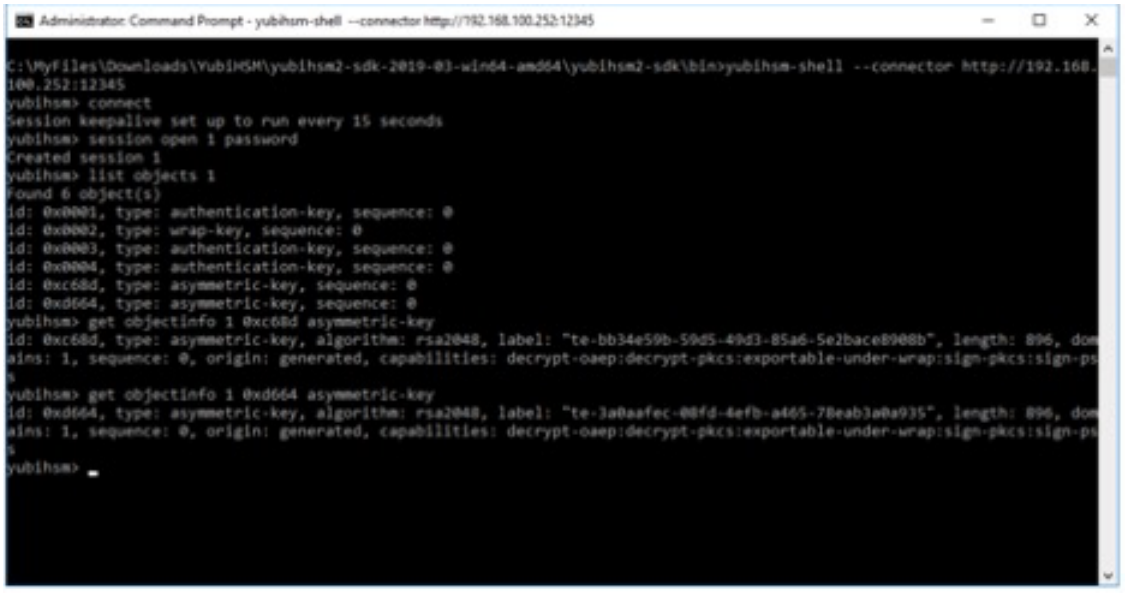


Figure: Example of HGS Keys in YubiHSM-Shell

4. Verify Storage of HGS Certificates in Microsoft Certificate Store.

   Verify that corresponding HGS certificates have been stored in Microsoft certificate store. Launch the Microsoft Management Console (MMC) by going to the command line and typing `MMC.exe`.

   a. In MMC, select **File > Add/remove Snap-in**.

b. In the Add or Remove Snap-ins window, select the option **Certificates > Computer Account > Local Computer**.

c. In the Certificates (Local Computer) console, expand the folders **Personal > Certificates**, and verify that the self-signed HGS signing and encryption certificates appear.

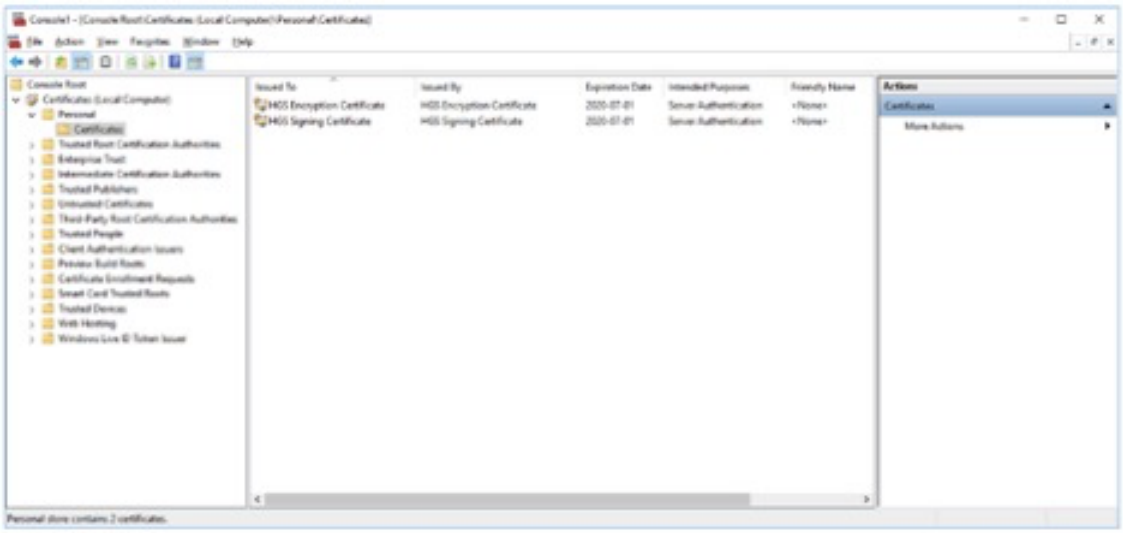


Figure: Example of HGS Certificates in Microsoft Certificate Store

For more information on how to generate HGS signing and encryption keys and certificates, see Microsoft's documentation on HGS certificate management.

26.6.2 Initialize HGS with Signing and Encryption Keys and Certificates

Once the HGS signing and encryption keys and certificates have been generated, use them to initialize HGS.

Create the self-signed HGS certificates by starting a command prompt with administrator rights and typing the command `PowerShell`. In the PowerShell command prompt, run the following cmdlet to initialize HGS with the signing and encryption certificates.

Note: The parameters `SigningCertificateThumbprint` and `EncryptionCertificateThumbprint` should be set to the output values from the PowerShell cmdlet `New-SelfSignedCertificate` as described in the previous section.

```
PS C:\users\your-username\ Initialize-HgsServer -HgsServiceName 'MyHgsService' -
↪SigningCertificateThumbprint '<SigningCertificateThumbprint>' -
↪EncryptionCertificateThumbprint '<EncryptionCertificateThumbprint>'
```

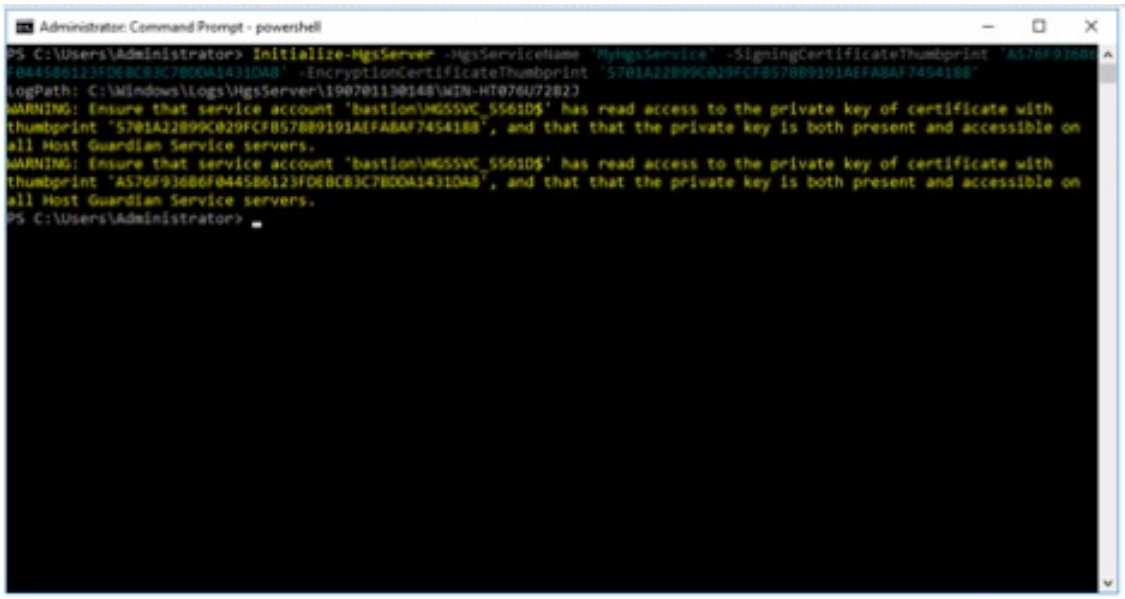


Figure: Example of PowerShell cmdlet to Initialize HGS with the Certificates

For more information on how to initialize HGS with the signing and encryption certificates, see Microsoft's documentation on HGS initialization.

CHAPTER

TWENTYSEVEN

YUBIHSM 2 FOR MICROSOFT SQL SERVER DEPLOYMENT GUIDE

27.1 YubiHSM 2 for Microsoft SQL Server Guide

In a Microsoft SQL Server environment, the Column Master Key (CMK) must be protected in hardware. The YubiHSM 2 protects the CMK in hardware and guards the Microsoft SQL Server database encryption services.

This guide is intended to help systems administrators deploy YubiHSM 2 for use with Microsoft SQL Server in a Windows server environment. The expected outcome is that the Column Master Key (CMK) is created securely on a YubiHSM 2 and that a hardware-based backup copy of key materials has been produced.

These guidelines for deployment cover basic topics, so the instructions should be modified as required for your specific environment. It is assumed that you are familiar with the concepts and processes for working with Microsoft SQL Server. It is also assumed that the installation is performed on a single Microsoft SQL Server database, but the concept can be extended to more servers and databases.

Important: Before deploying to production, we recommend that you install and test the Microsoft SQL Server installation and setup of the YubiHSM 2 in a test or lab environment.

For guidance on enabling the Microsoft SQL Server feature Always Encrypted in a production environment, see the Microsoft SQL Docs for Always Encrypted.

27.2 Introduction to Always Encrypted

Introduced in 2016, Microsoft SQL Server's Always Encrypted feature enables the encryption of selected columns in a database.

Note: The YubiHSM 2 requires Microsoft SQL Server 2017 and Microsoft SQL Server Management Studio (SSMS) 2018.

The Always Encrypted encryption mechanisms rely upon two cryptographic keys, described in detail in the Microsoft SQL Docs, Overview of Key Management for Always Encrypted. In brief:

- The **Column Encryption Key (CEK)** is a symmetric key used for encrypting the contents of a selected database column.
- The **Column Master Key (CMK)** is an asymmetric key that is used for protecting the encryption key. The CMK for Always Encrypted can be protected in a local key store, which is in the scope of this document, or in a centralized key store, which is not in scope.

A CMK can be stored in a local key store that supports Microsoft's Cryptography Next Generation (CNG) API. To protect the CMK in hardware, a hardware security module (HSM) should be used. In this scenario, Always Encrypted accesses the HSM through the CNG API by using a key storage provider (KSP).

To protect the CMK in hardware, the YubiHSM 2 can be deployed as the local key store. Microsoft's Always Encrypted accesses the YubiHSM 2 through the KSP that is provided with the YubiHSM software tools. With this setup, the Microsoft SQL Server Management Studio (SSMS) can be used to manage the CMK in the YubiHSM 2 device. This deployment guide describes two ways to generate the CMK and CEK in YubiHSM 2:

- By using the SSMS wizard, as described in *Use SSMS to Generate the CMK and CEK*.
- By running a PowerShell script, as described in *Use PowerShell Script to Generate the CMK and CEK*.

27.3 Prerequisites and Preparations

The audience of this document is an experienced system administrator with a good understanding of Microsoft SQL Server management. In addition, it is helpful to be familiar with the terminology, software, and tools specific to YubiHSM 2. As a primer for these terms, see the *Glossary*.

To follow the steps provided in this guide, the complete the following prerequisites:

- Microsoft Windows Server 2022 or higher, with Microsoft .NET Framework 4.8 or higher. The operating system should be installed in a secure computer network. The system administrator must also have elevated system privileges.
- Access to Microsoft SQL Server 2019 with SQL Server Management Studio (SSMS) 2018 or higher.
- YubiHSM 2 software and tools for Windows downloaded from the Yubico YubiHSM 2 Release page and available on the system to be used.

  Note: The 32-bit version of the YubiHSM KSP DLL is needed for use with SSMS.

- Two (2) YubiHSM 2 devices, one for deployment and one for backup in hardware.
- Your organization policies may require key custodians to be available for the YubiHSM 2 deployment. For more information about key custodians and the associated `M of N` key shares, see *Key Splitting and Key Custodians*.

27.3.1 Configuration for this Integration

For the integration described in this guide, the following hardware and software configuration was used:

- Microsoft Windows Server 2022.
- Microsoft .NET Framework 4.8.
- Microsoft SQL Server 2019.
- Microsoft SQL Server Management Studio (SSMS) 2018.
- Yubico YubiHSM v 2.1.2.
- Yubico YubiHSM software tools v 2021.12c. In particular, the 32-bit YubiHSM KSP DLL is needed for use with SSMS (which is a 32-bit application).

27.4 Basic Setup of YubiHSM 2 and SQL Server

27.4.1 Installing and Configuring YubiHSM 2

Install and configure the YubiHSM 2 device and software using the instructions in the following sections in the YubiHSM 2 with Key Storage Provider for Windows Server—Configure YubiHSM 2 Key Storage Provider for Microsoft Windows Server, see *Key Splitting and Key Custodians*.

- *Installing the YubiHSM 2 Tools and Software*
- *Configuring the Primary YubiHSM 2 Device*
- *Configure the YubiHSM 2 Software on Windows*

When these instructions have been completed, the YubiHSM 2 should be configured with — for example — one domain with a **wrap key** (id `0x0002`), an application **auth key** (id `0x0003`), and an **audit key** (id `0x0004`). The configuration of the YubiHSM 2 can be inspected by using the YubiHSM-Shell in a command prompt as shown in the screenshot below.

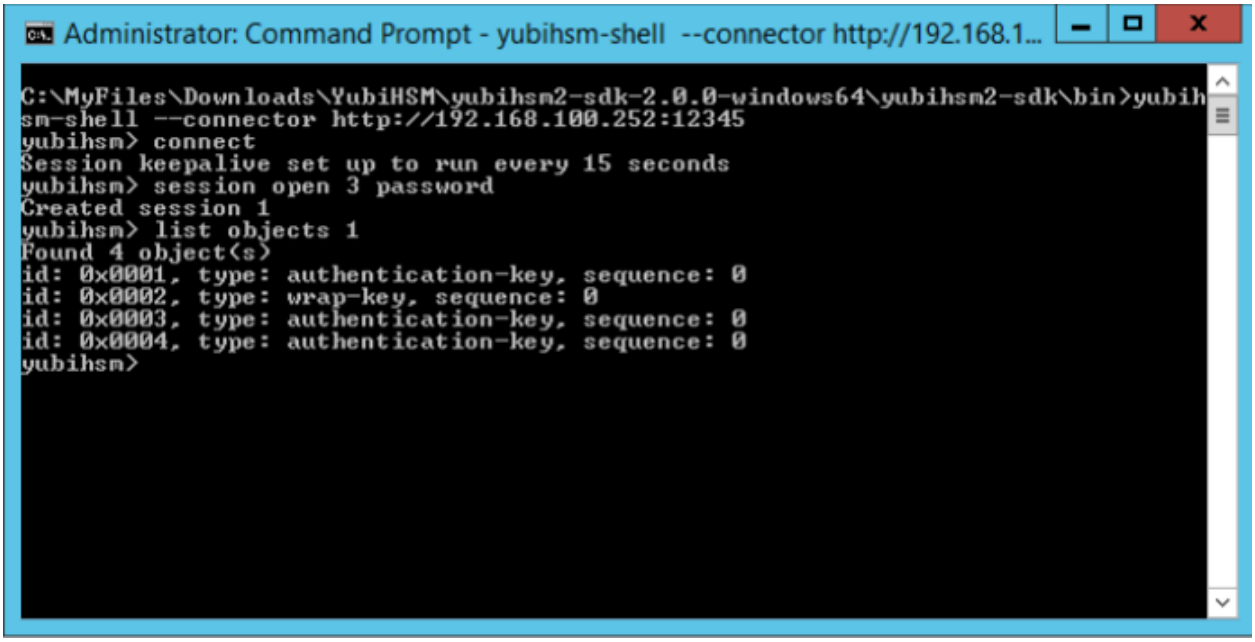


Figure: Example of the YubiHSM 2 basic configuration

27.4.2 Creating a Test Database

Create a test database that will be used for the Always Encrypted deployment with YubiHSM 2. A test database can be downloaded from Microsoft's official repository at Wide World Importers sample database v1.0 If you already have a Microsoft SQL Server database installed, you can skip ahead to *Configure SSMS for Database Encryption*.

Note: At least one row with values needs to be inserted into the database table before the columns are encrypted (see the example of a test database below).

---

1. Create a test database.
    a. Launch Microsoft SQL Server Management Studio (SSMS) 2018.

b. Right-click on the **Databases** icon.

c. Select **New Database…**.

d. Enter an appropriate name for the database.

In this guide, a test database named “Sales3” is used for the tests with Always Encrypted in conjunction with YubiHSM 2.

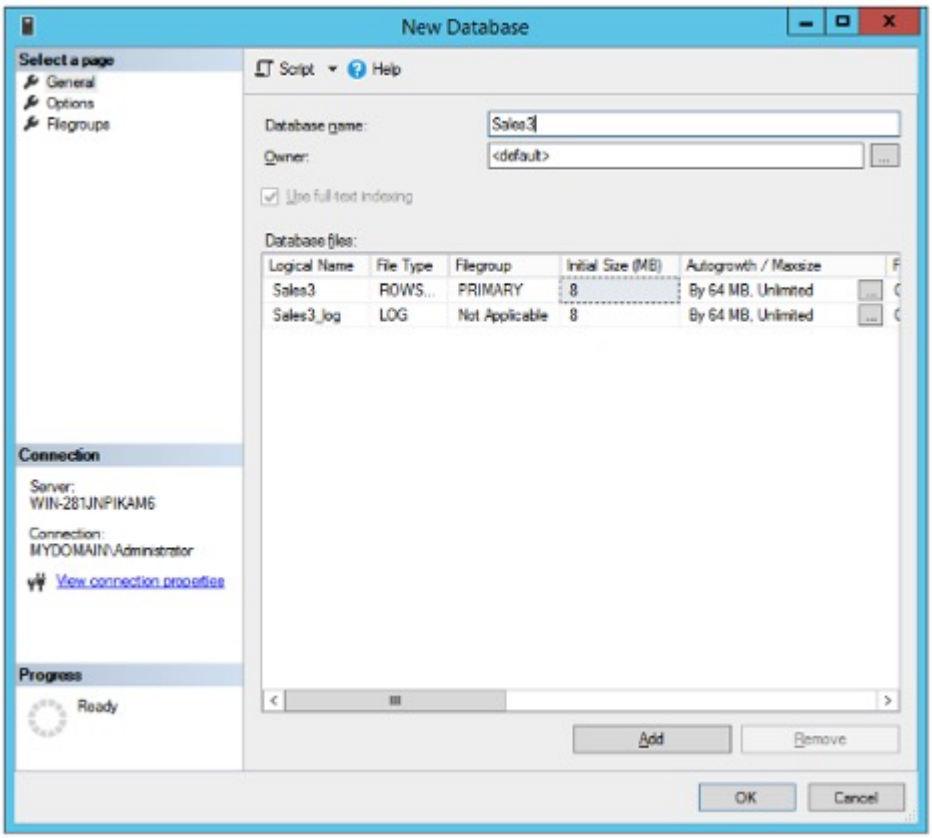


Figure: Example of test database

2. Create table:

a. Expand **Databases > Sales3 > Tables**.

b. Right-click on **Tables** and select **Create new table**.

c. Add some columns, for example “Name”, “Address”, “ZipCode”, “City”, “Country”.

d. Save the table and give it a name - “Table_Customers” for example.

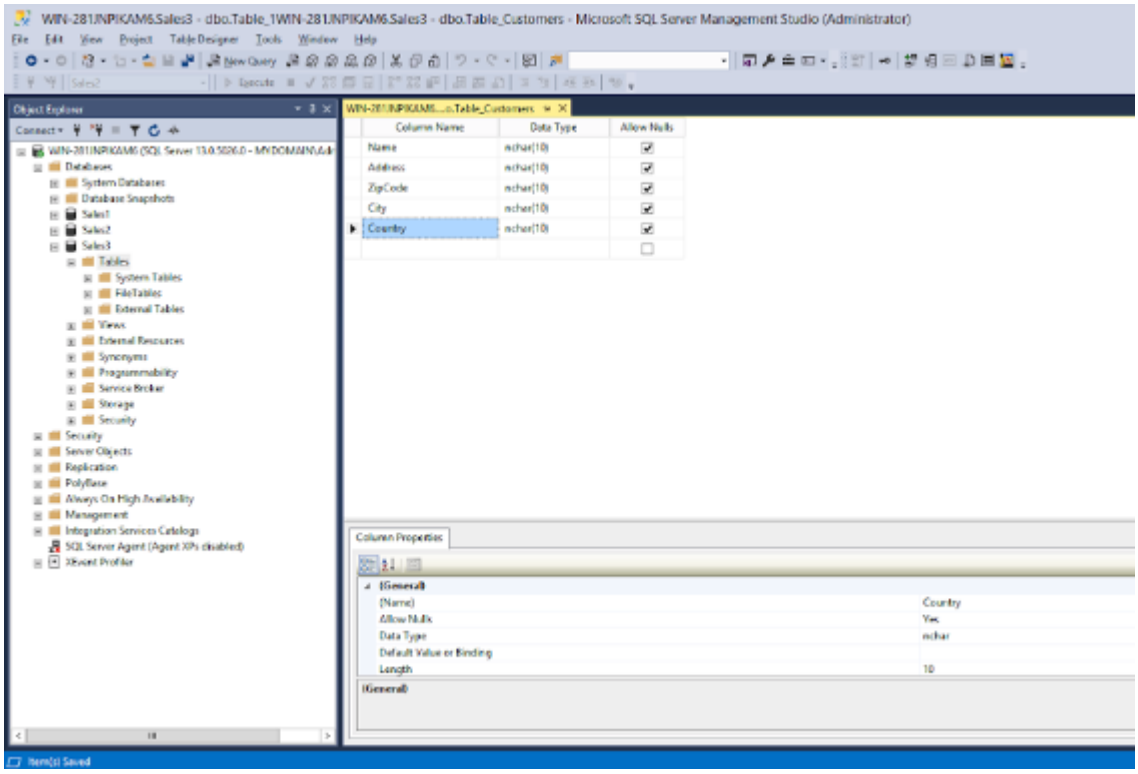


Figure: Example of test table

3. Add one or more rows to the table:

   a. Expand **Databases > Sales3**.

   b. Right-click on **Table** and selecting **New > Query...**.

   c. Use the SQL Query window to insert rows into the database table, for example, with the SQL query shown below.

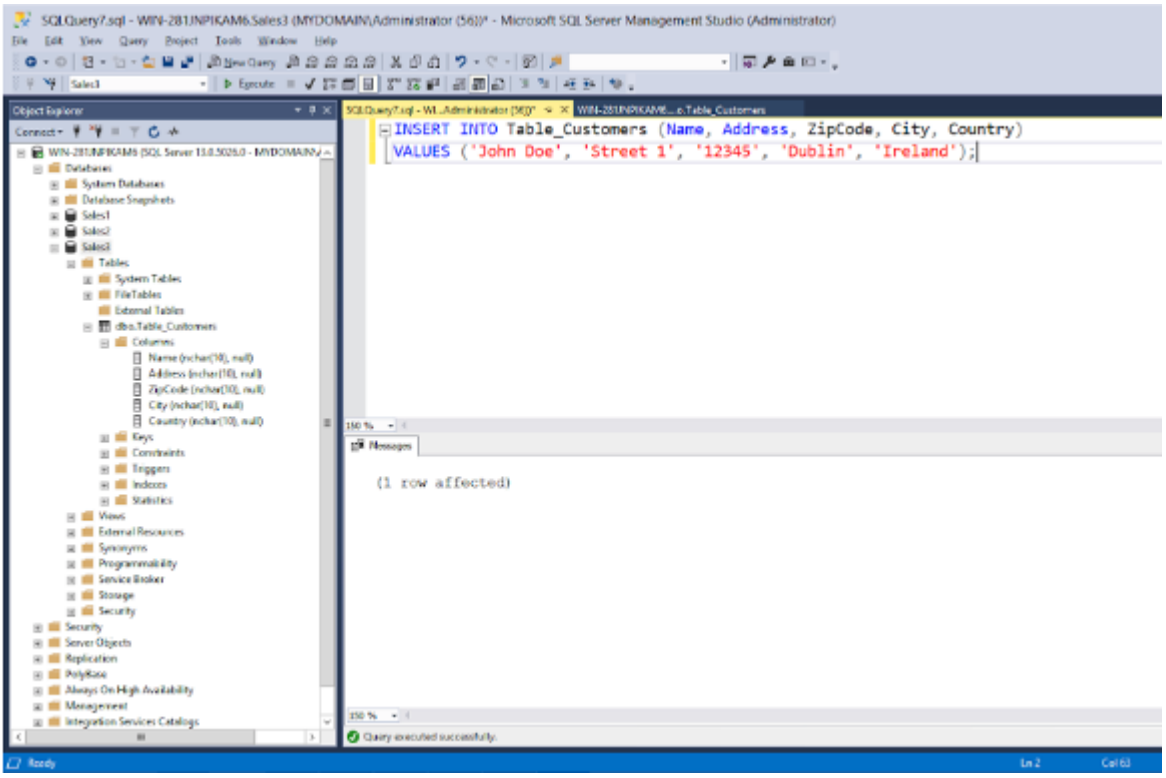


Figure: Example of SQL query to insert values into the table

27.5 Use SSMS to Generate the CMK and CEK

The Microsoft SQL Server Management Studio (SSMS) provides tools for manual creation of the CMK and CEK. However, using a PowerShell script (see *Use PowerShell Script to Generate the CMK and CEK*) results in a uniform configuration and ensure no options are missed. Note that all the examples and screenshots in this document are based on different key names being used for the SSMS wizard and the PowerShell script.

27.5.1 Generate the CMK

1. To generate the CMK for a database, create and save the following PowerShell script to generate Always Encrypted Key. Save this script as `AlwaysEncryptedKey-PS.ps1`.

```
$cngProviderName = "YubiHSM Key Storage Provider"
$cngAlgorithmName = "RSA"
$cngKeySize = 2048 # Recommended key size for column master keys
$cngKeyName = "AlwaysEncryptedKey-PS" # Name identifying your key in the KSP
$cngProvider = New-Object System.Security.Cryptography.CngProvider(
↪$cngProviderName)
$cngKeyParams = New-Object System.Security.Cryptography.
↪CngKeyCreationParameters
$cngKeyParams.provider = $cngProvider
$cngKeyParams.KeyCreationOptions = [System.Security.Cryptography.
↪CngKeyCreationOptions]::OverwriteExistingKey
$keySizeProperty = New-Object System.Security.Cryptography.CngProperty(
```

(continues on next page)

(continued from previous page)

```
↪"Length",[System.BitConverter]::GetBytes($cngKeySize),[System.Security.
↪Cryptography.CngPropertyOptions]::None);
$cngKeyParams.Parameters.Add($keySizeProperty)
$cngAlgorithm = New-Object System.Security.Cryptography.CngAlgorithm (
↪$cngAlgorithmName)
$cngKey = [System.Security.Cryptography.CngKey]::Create($cngAlgorithm,
↪$cngKeyName, $cngKeyParams)
```

2. Run the `AlwaysEncryptedKey-PS.ps` script from a PowerShell Window with elevated/administrator permissions.
3. Once completed, verify the Network Location is set to Private or Domain. To do so:
   - View the current Profile assigned to the Network Connection by using the command.

     ```
     Get-NetConnectionProfile.
     ```

   - If the Profile is set to `Public` change it to `Private` or `Domain` so that SQL can communicate properly with the YubiHSM.
   - To change it, use the command.

     ```
     Set-NetConnectionProfile -InterfaceAlias Ethernet1 -NetworkCategory "Private"
     ```

27.6 Validate Generation of the CMK

The presence of the asymmetric CMK in the YubiHSM 2 can also be validated by running the following sequence of YubiHSM-Shell commands in a command prompt.

```
$yubihsm> connect
$yubihsm> session open <slot-ID> <password>
$yubihsm> list objects <session-ID>
$yubihsm> get objectinfo <session-ID> <key-ID> asymmetric-key
```

Example output from the YubiHSM-Shell commands is shown in the screenshot below.

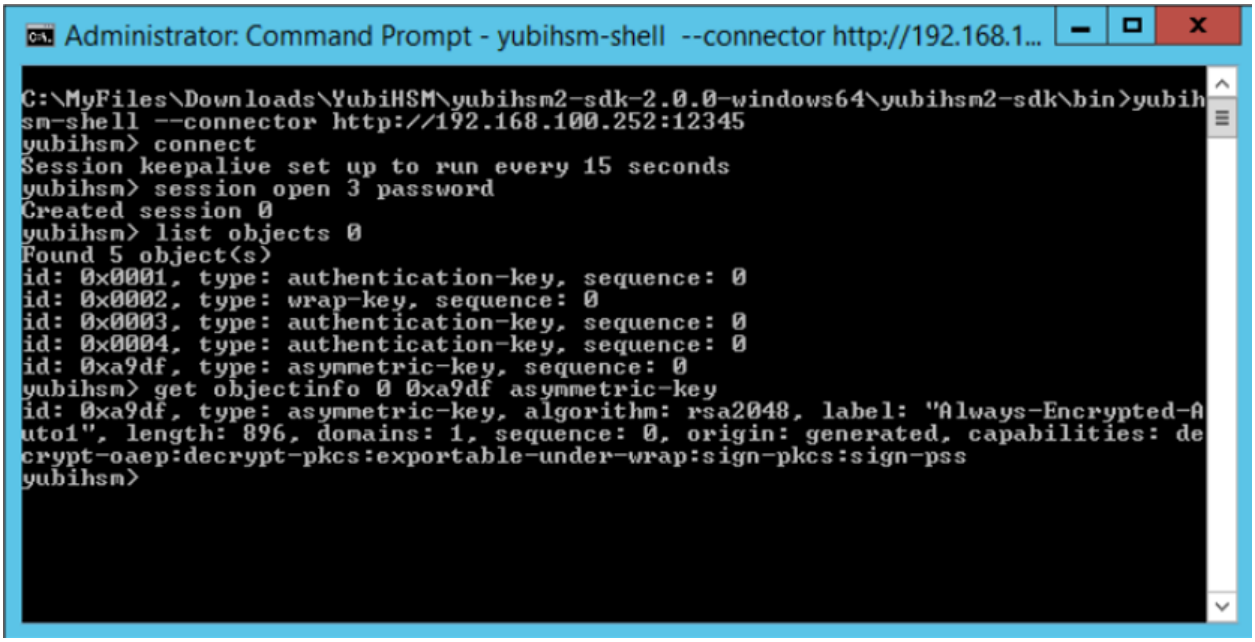


Figure: New Column Master Key listed in YubiHSM

27.6.1 Assign the CMK to a Database

1. To assign the CMK for a database.
   a. Launch SSMS.
   b. Expand **Databases > Database-Name > Security > Always Encrypted Keys > Column Master Key**.

   We use the example shown below, expanding the tree **Databases > Sales3 > Security > Always Encrypted Keys > Column Master Key**.

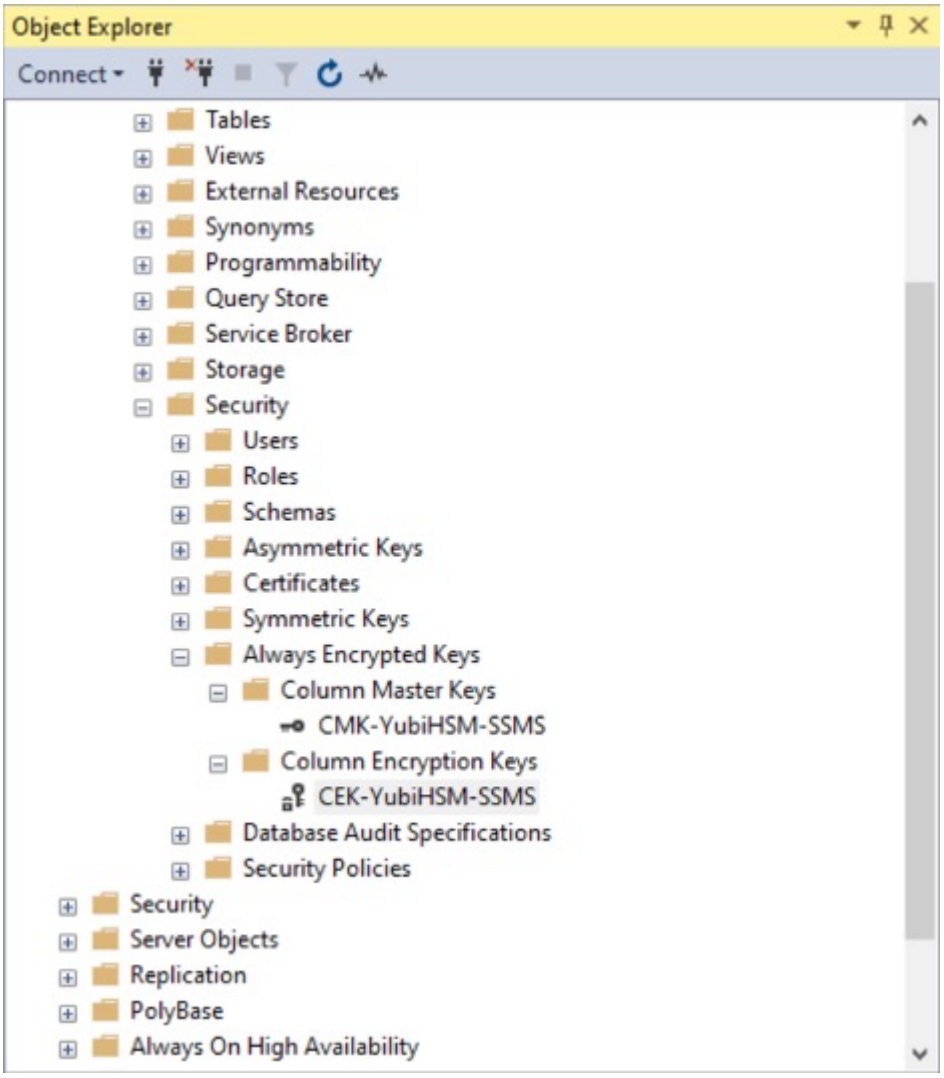


Figure: Assigning the CMK

2. Right-click on **Column Master Keys**, and select **New Column Master Key…** in the **New Column Master Key** window, enter the following values:
   - In the **Name** text field, enter an appropriate name for the CMK, for example, "CMK-YubiHSM-SSMS".
   - In the **Key Store** drop-down list, select **Key Storage Provider (CNG)**.
   - In the **Select a provider** drop-down list, select **YubiHSM Key Storage Provider**.
   - In the bottom field, select **AlwaysEncryptedKey-PS**.

27.6.2 Generate the CEK

The next task is to generate the CEK for a database.

1. Generate the CEK.
   a. Launch SSMS.
   b. Expand **Databases > Database-Name > Security > Always Encrypted Keys > Column Encryption Key**.

   In our example, expand the tree **Databases > Sales3 > Security >Always Encrypted Keys > Column Encryption Key**, which is illustrated in the screenshot below.

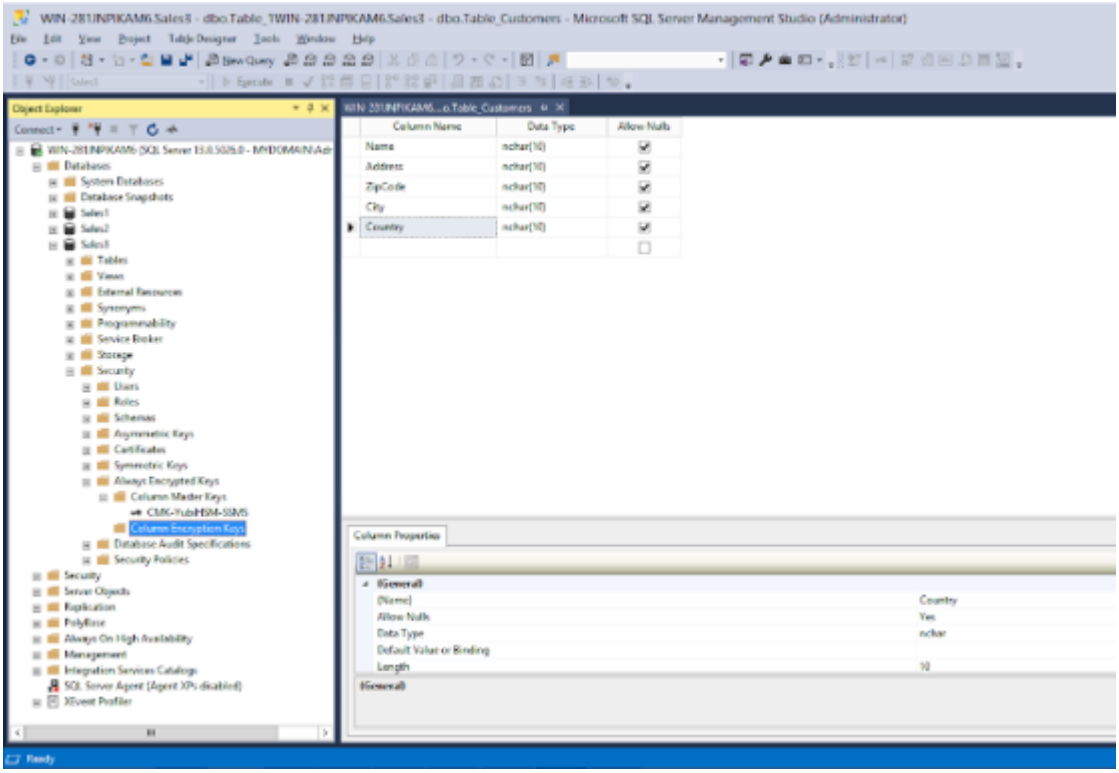


Figure: Column Encryption Keys in SSMS

2. Right-click **Column Encryption Keys** and select **New Column Encryption Key...**.
3. In the **New Encryption Master Key** window, enter the following values:
   a. In the **Name** text field, enter an appropriate name for the CEK, for example `CEK-YubiHSM-SSMS`.
   b. In the **Column master key** drop-down list, select the CMK that was generated on the YubiHSM, for example `CMK-YubiHSM-SSMS`.

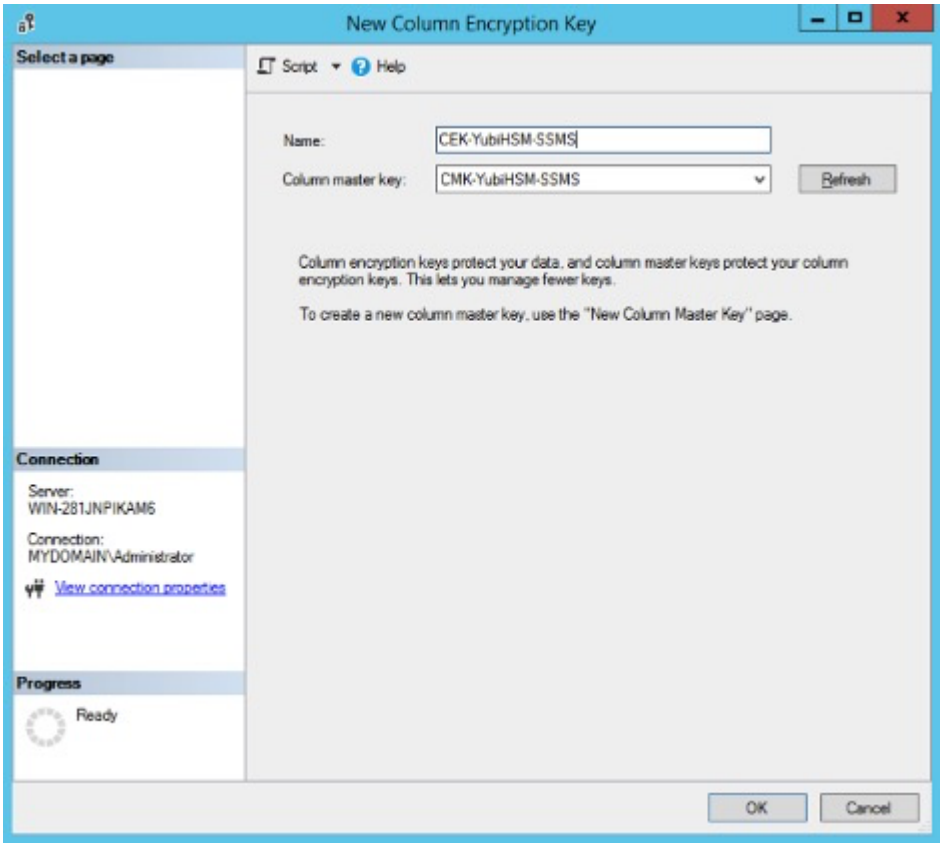


Figure: Create new Column Encryption Key with SSMS

4. Generate and verify the CEK.
    a. Press **OK**. To verify the success of the operation.
    b. Check to see whether the CEK is listed under **Always Encrypted Keys in SSMS**.

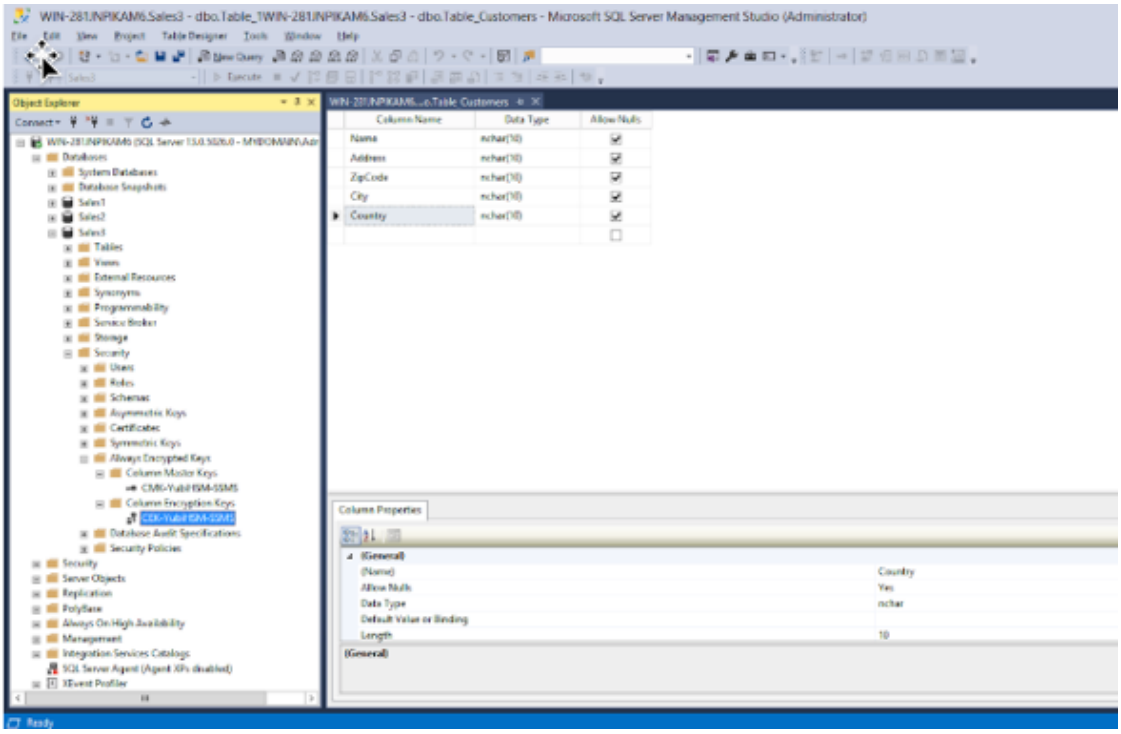


Figure: Column Master Key and Column Encryption Key in SSMS

27.7 Use PowerShell Script to Generate the CMK and CEK

Instead of using SSMS to generate the CMK and CEK (as described in the foregoing section, *Use SSMS to Generate the CMK and CEK*), another option is to use PowerShell to generate the CMK and CEK. Microsoft has published a PowerShell script that can be used to generate the CMK in an HSM. The following instructions list the activities in the script, then describe how to modify that PowerShell script to generate the CMK in the YubiHSM 2 by calling its KSP.

27.7.1 Create a CMK in the YubiHSM 2 with CNG Provider (KSP)

```
$cngProviderName = "YubiHSM Key Storage Provider"
$cngAlgorithmName = "RSA"
$cngKeySize = 2048 # Recommended key size for column master keys
$cngKeyName = "AlwaysEncryptedKey-PS" # Name identifying your key in the KSP
$cngProvider = New-Object System.Security.Cryptography.CngProvider($cngProviderName)
$cngKeyParams = New-Object System.Security.Cryptography.CngKeyCreationParameters
$cngKeyParams.provider = $cngProvider
$cngKeyParams.KeyCreationOptions = System.Security.Cryptography.CngKeyCreationOptions]::␣
↪OverwriteExistingKey
$keySizeProperty = New-Object System.Security.Cryptography.CngProperty("Length",[System.
↪BitConverter]::GetBytes($cngKeySize),[System.Security.Cryptography.
↪CngPropertyOptions]::None);
$cngKeyParams.Parameters.Add($keySizeProperty)
$cngAlgorithm = New-Object System.Security.Cryptography.CngAlgorithm($cngAlgorithmName)
$cngKey = [System.Security.Cryptography.CngKey]::Create($cngAlgorithm,$cngKeyName,
↪$cngKeyParams)
```

27.7.2 Import SQL Server Module

```
Import-Module "SqlServer"
```

27.7.3 Connect to your Database

```
$serverName = "<server name>"
$databaseName = "<database name>"
$connStr = "Server = " + $serverName + "; Database = " + $databaseName + ";Integrated␣
↪Security = True"
$connection = New-Object Microsoft.SqlServer.Management.Common.ServerConnection
$connection.ConnectionString = $connStr
$connection.Connect()
$server = New-Object Microsoft.SqlServer.Management.Smo.Server($connection)
$database = $server.Databases[$databaseName]
```

27.7.4 Create SQL CMK Settings Object for your CMK

```
$cmkSettings = New-SqlCngColumnMasterKeySettings -CngProviderName
$cngProviderName -KeyName $cngKeyName
```

27.7.5 Create CMK Metadata in Database

```
$cmkName = "CMK-YubiHSM-PS"
New-SqlColumnMasterKey -Name $cmkName -InputObject $database -ColumnMasterKeySettings
↪$cmkSettings -Verbose
```

27.7.6 Generate CEK, Encrypt with CMK, and Create CEK Metadata in Database

```
$cekName = "CEK-YubiHSM-PS"
New-SqlColumnEncryptionKey -Name $cekName -InputObject $database -ColumnMasterKeyName
↪$cmkName -Verbose
```

27.7.7 Customize the Script

1. To customize this script, change the placeholders `server name` and `database name` to the actual values of the Microsoft SQL Server name and the database used.

   For the test database used in this example, the `database name` is set to `Sales3`, while the `server name` should be set to the name of your Windows server.

2. Save the PowerShell script file in a folder with an appropriate name, for example `CreateColumnMasterAndEncryptionKeys-YubiHSM.ps1`.

3. Execute the script.

   a. Launch a command prompt with administrator privileges

b. Enter the PowerShell mode by typing `PowerShell`.

c. Navigate to the directory where the PowerShell script is located.

d. Execute the script:

```
PS> .\CreateColumnMasterAndEncryptionKeys-YubiHSM.ps1
```

The PowerShell script generates the CMK and the CEK and displays the output from these operations. Output from the script given in *Create a CMK in the YubiHSM 2 with CNG Provider (KSP)* is shown in the screenshot below.

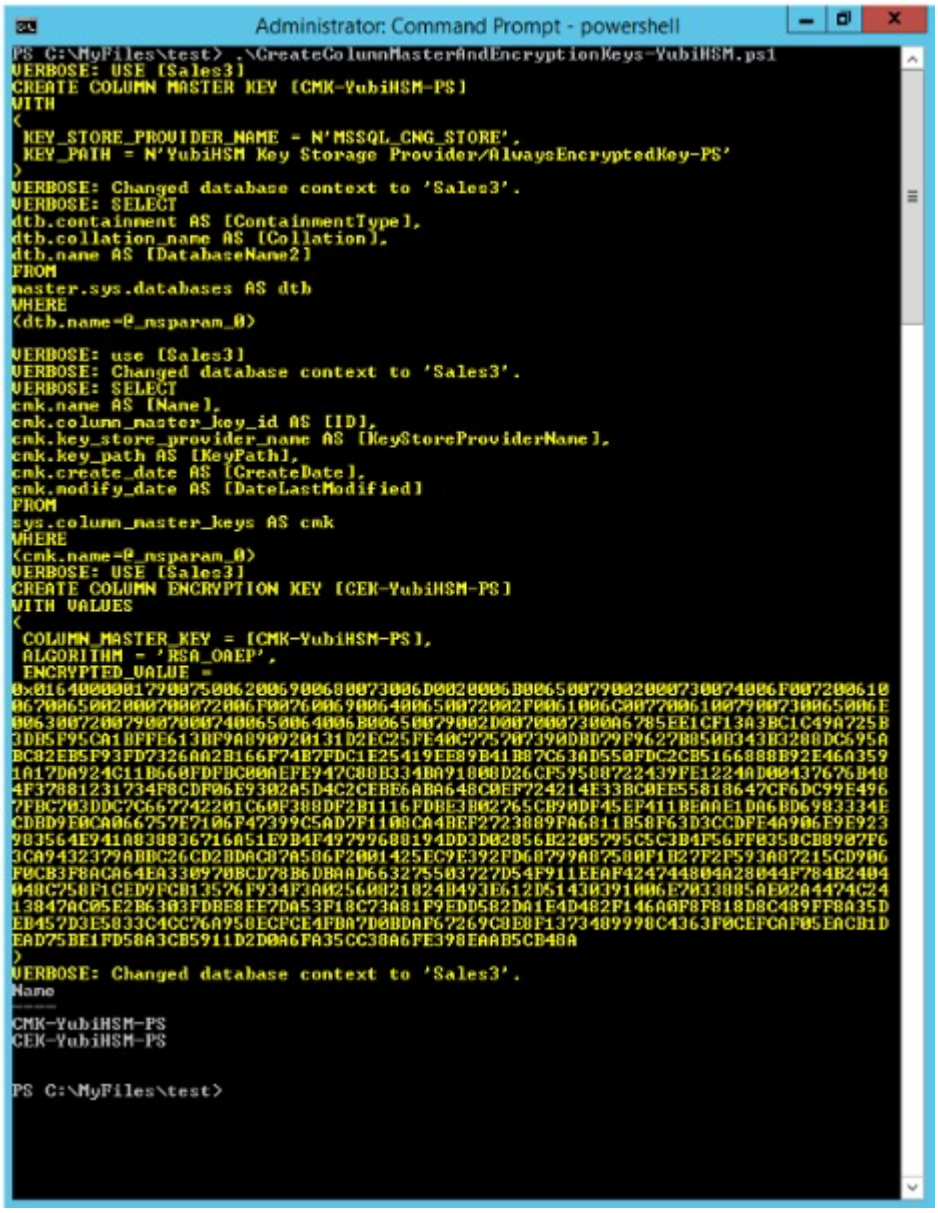


Figure: PowerShell script to create Column Master Key and Column Encryption Key

27.7.8 Validate Generation of the CMK and the CEK

1. After executing the PowerShell script.
   a. Switch back to SSMS.
   b. Expand the objects **Databases > Database-Name > Security > Column Master Key and Databases > Database-Name > Security > Column Encryption Key**.
   c. Right-click each object and select the **Refresh** option.

   The CMK and CEK that were generated by the PowerShell script appear in SSMS as `CMK-YubiHSM-PS` and `CEK-YubiHSM-PS` respectively.

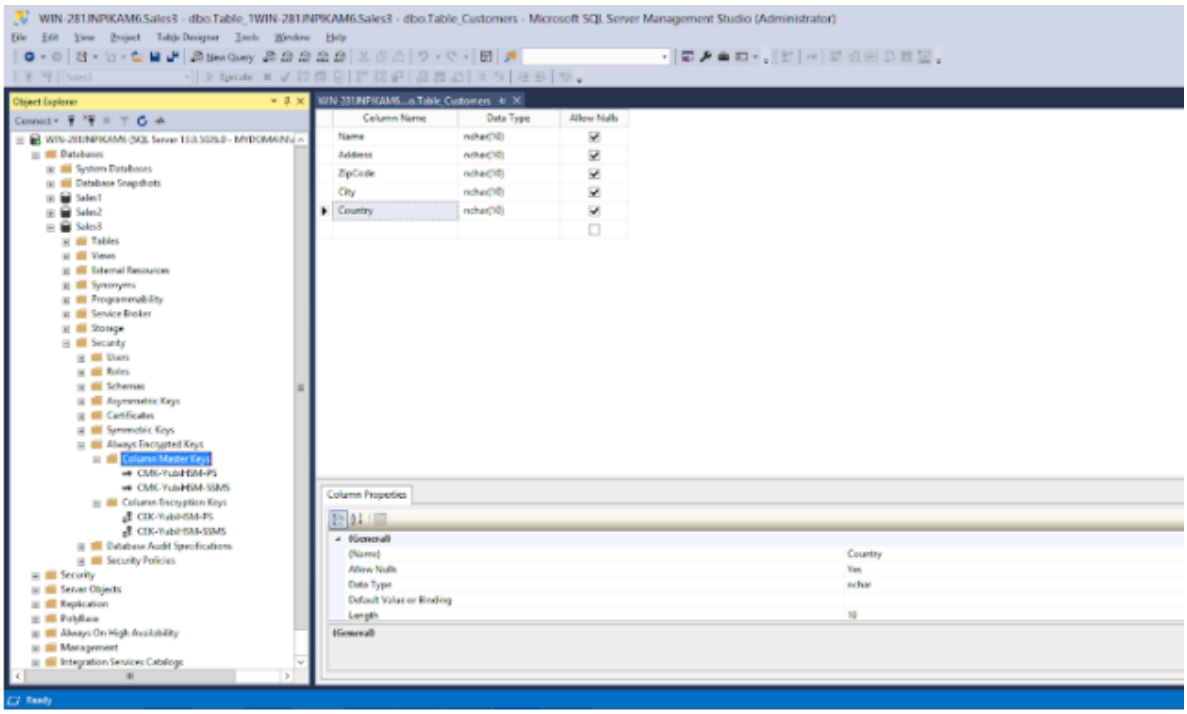


Figure: Refreshed Column Master Keys and Column Encryption Keys in SSMS

2. Verify the presence of the asymmetric CMK in the YubiHSM 2 by running the following sequence of YubiHSM-Shell commands in a command prompt.

```
$yubihsm> connect
$yubihsm> session open <slot-ID> <password>
$yubihsm> list objects <session-ID>
$yubihsm> get objectinfo <session-ID> <key-ID> asymmetric-key
```

   Example output for the YubiHSM-Shell commands is shown in the screenshot below.

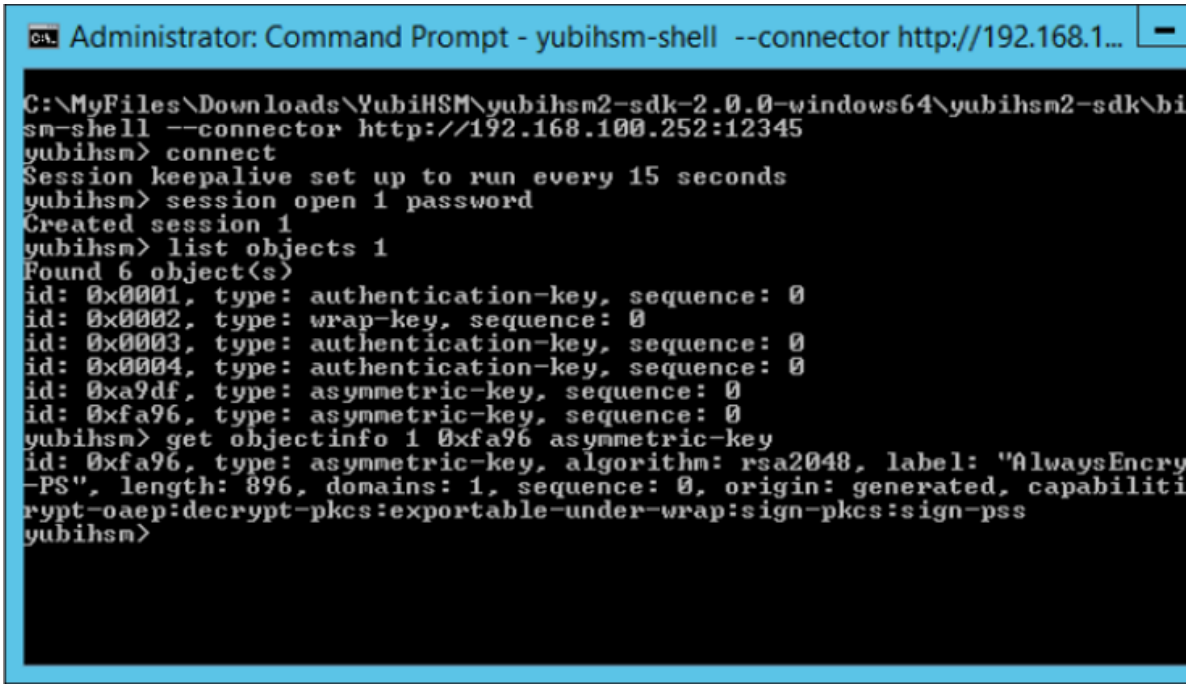


Figure: Column Master Keys in YubiHSM 2

27.8 Encrypt Database Columns

Database columns can be encrypted with PowerShell- or SSMS-generated keys.

27.8.1 Encrypt Database Column with PowerShell-Generated Keys

1. To encrypt a database column, expand the database's columns: **Databases > Database-Name > Tables > Table-Name > Columns**.

   Our example expands the tree **Databases > Sales3 > Tables > dbo.Table_Customers > Columns**, as shown in the screenshot below.

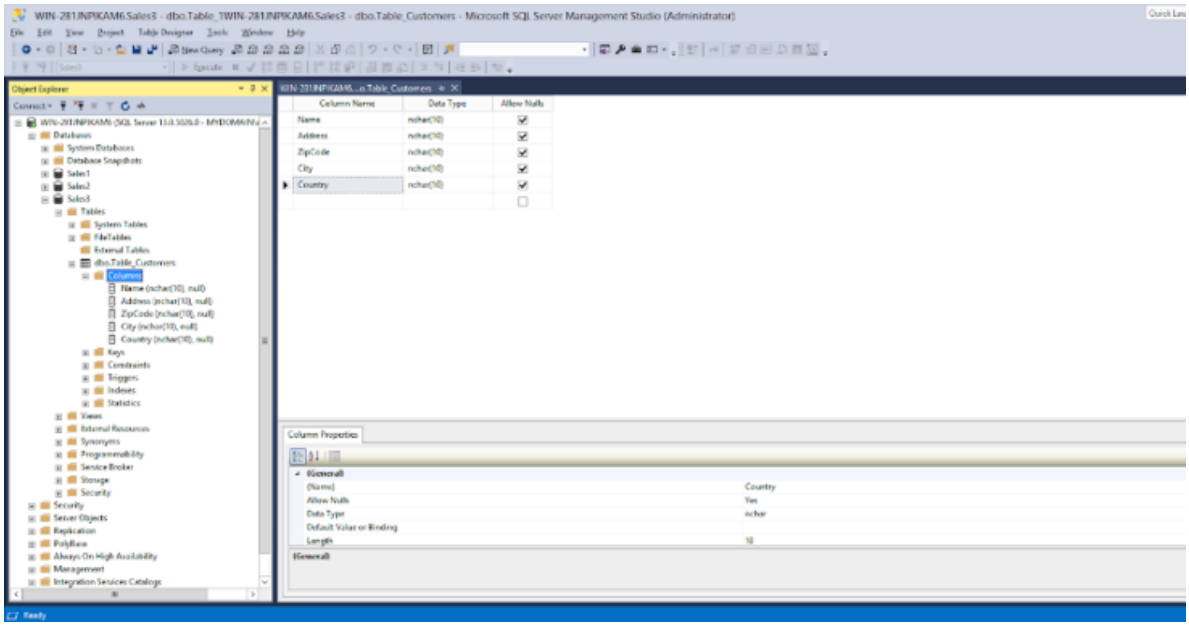


Figure: Expanded columns to be encrypted

2. Right-click the column to be encrypted and select **Encrypt Column…**.

   In our example, right-click the table **Name** and select **Encrypt Column…**. The **Introduction** window in the SSMS Always Encrypted wizard appears:

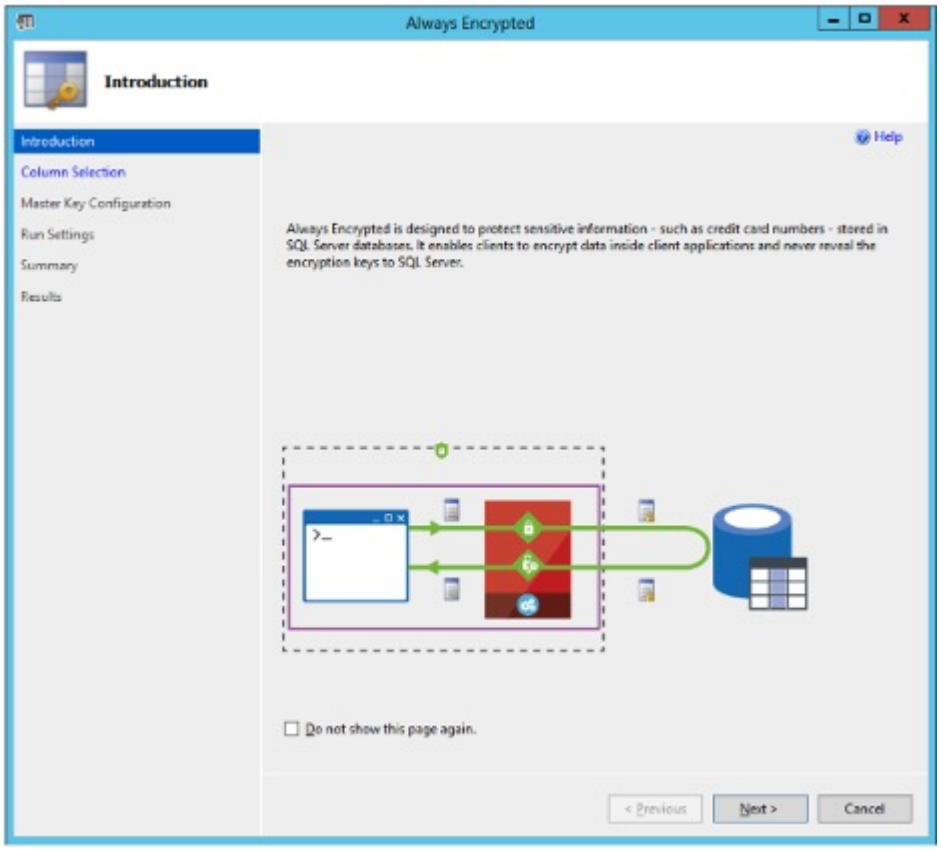


Figure: Always Encrypted wizard: Introduction

3. Click **Next**. The Column Selection window of the Always Encrypted wizard appears:

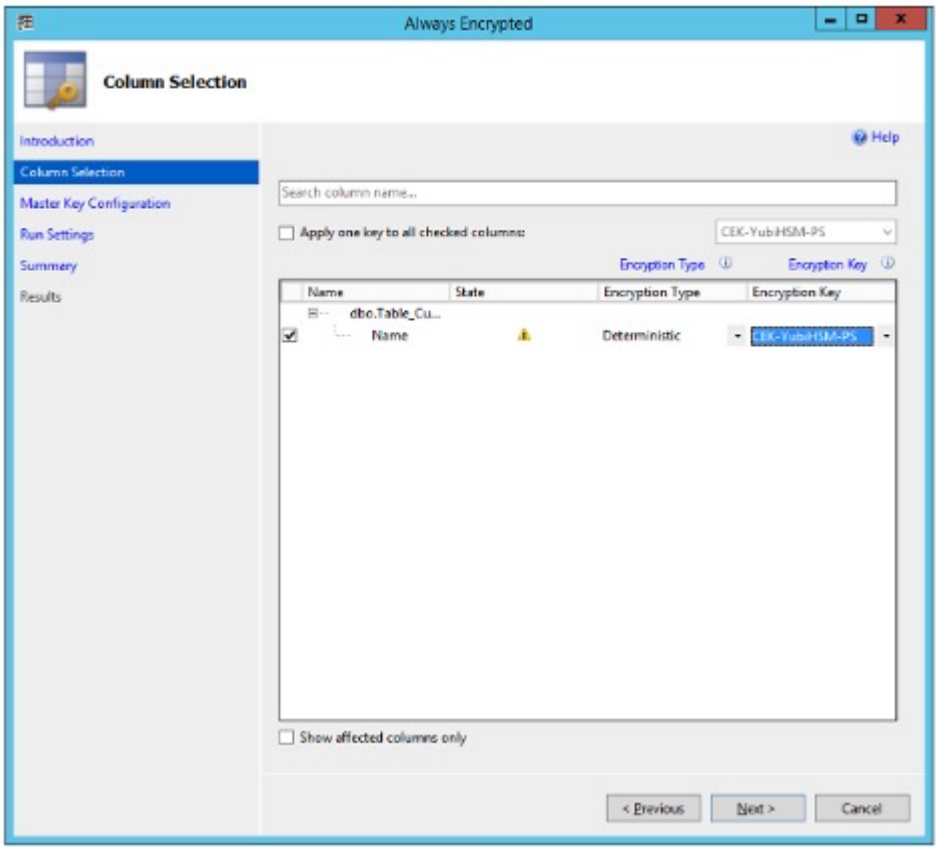


Figure: Always Encrypted wizard: Column Selection

4. In this example, the CEK that was generated with the PowerShell script is used for encrypting the database column.

   a. In the **Column Selection** window, select the Encryption Key named **CEK-YubiHSM-PS**.

      The **Encryption Type** can be set to either **Deterministic** or **Randomized**. In this example Deterministic is selected.

   b. Click **Next**, and the **Master Key Configuration** window in the Always Encrypted wizard appears.

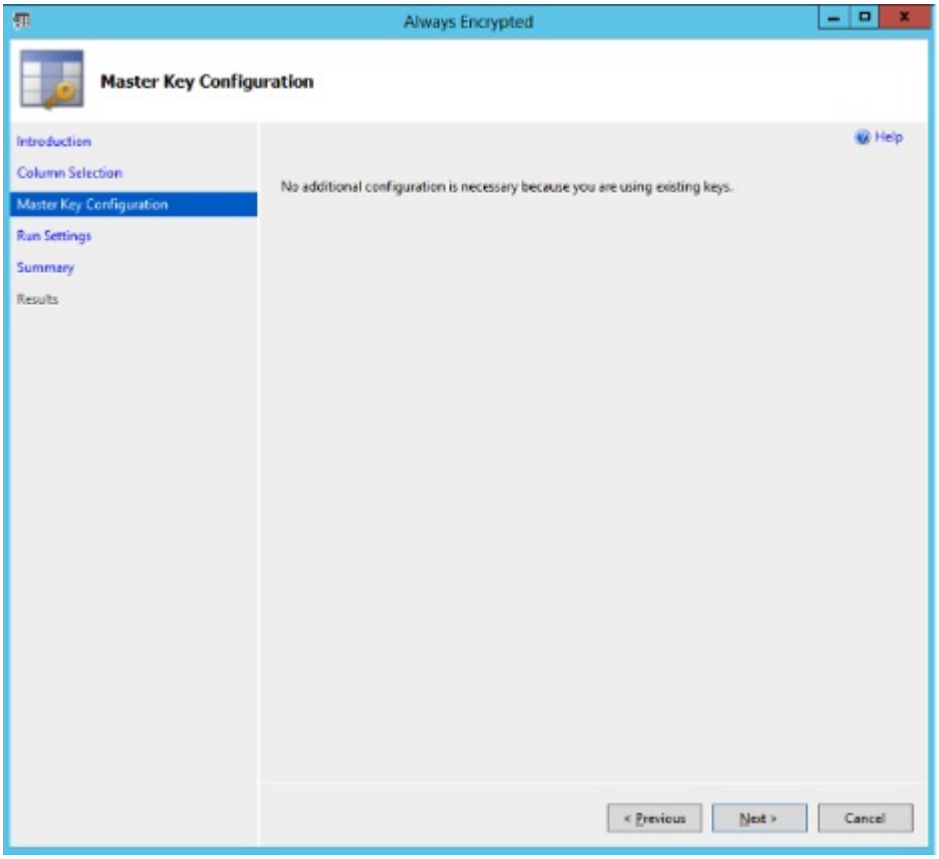


Figure: Always Encrypted wizard: Master Key Configuration

5. In the **Master Key Configuration** window, click **Next**, since the master column key in the YubiHSM 2 will be used. The **Run Settings** window in the Always Encrypted wizard appears.

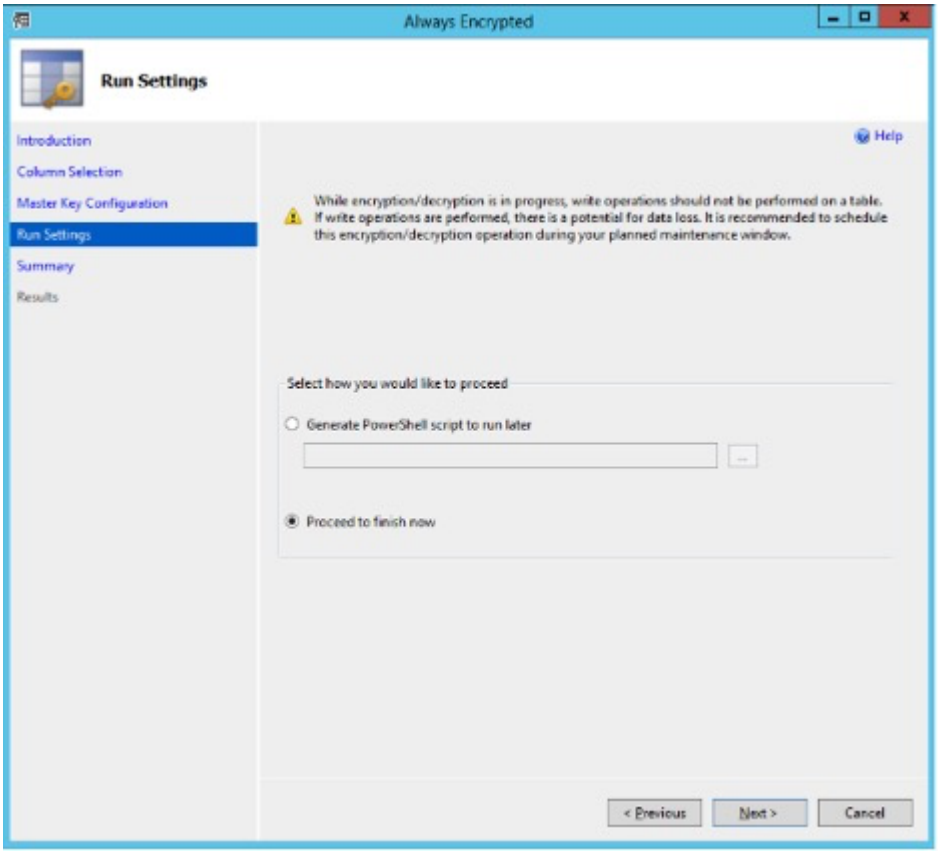


Figure: Always Encrypted wizard: Run Settings

6. In the **Run Settings** window, select **Proceed to finish now** (unless you want to generate a PowerShell script to run later) and click **Next**. The **Summary** window in the Always Encrypted wizard appears.

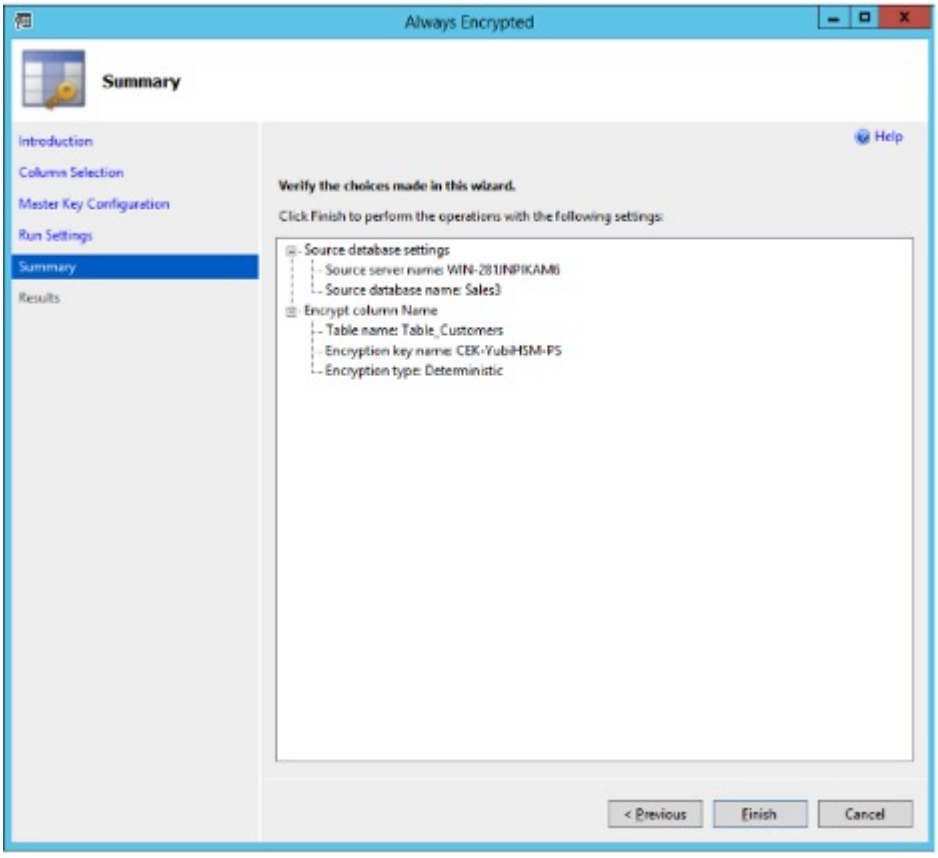


Figure: Always encrypted wizard: Summary

7. Review the settings in the **Summary** window and click **Finish**. The **Results** window appears:

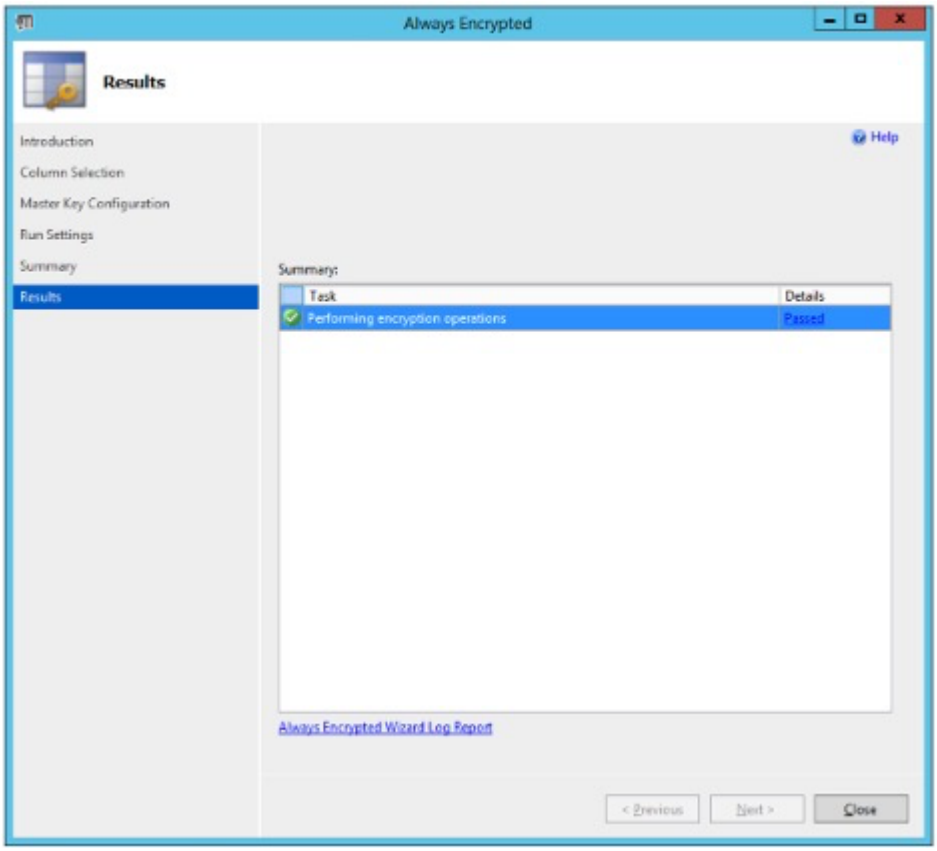


Figure: Always encrypted wizard: Results

When the column encryption operation succeeds, the word "Passed" is displayed in the **Details** column of the relevant row in the **Results** window.

27.8.2 Encrypt Database Column with SSMS-generated Keys

To use the CMK and CEK that were generated in *Use SSMS to Generate the CMK and CEK* follow the instructions above for encrypting a database column with PowerShell-generated keys (Encrypt Database Column with PowerShell-generated Keys), but select a different column (for example, **Address**) and use the column encryption key **CEK-YubiHSM-SSMS** and the related column master key **MK-YubiHSM-SSMS**.

27.8.3 Verify Encrypted Database Column

To check that the columns have been encrypted.

1. Expand the object **Database > Database-Name**. In our example the database name is `Sales3`.
2. Select **New Query** in the top menu.
3. Type the SQL query in the example below and click **Execute**.

```
SELECT * FROM Table_Customers;
```

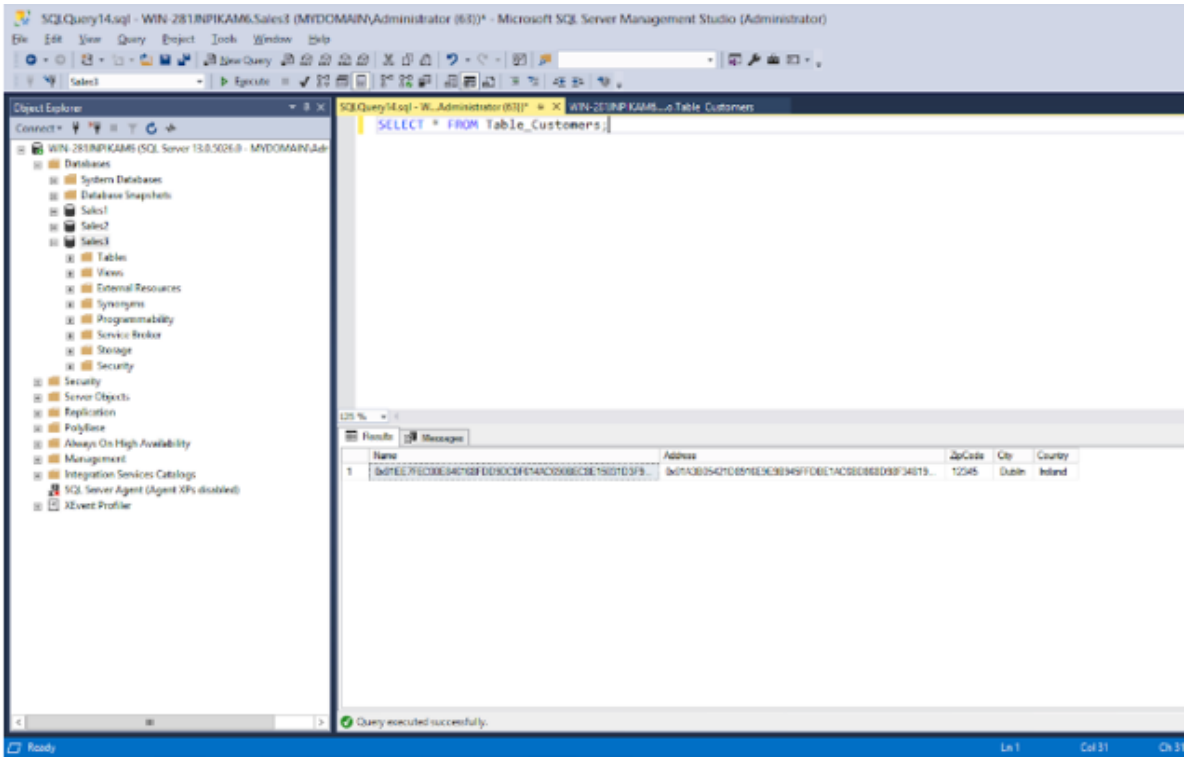


Figure: Checking the encrypted columns

27.9 Configure SSMS for Database Encryption

To configure Microsoft SQL Server and SSMS with the basic database settings needed for testing Always Encrypted in conjunction with YubiHSM 2, set SSMS to display the encrypted columns in clear text.

1. Select **Connect Object Explorer** settings.
    a. Click the **Connect Object Explorer** icon. The Connect to Server window appears.
    b. Click **Options**.
    c. Select the **Always Encrypted** tab and select **Enable Always Encrypted** (column encryption).
    d. To make the changes take effect, click the **Disconnect** icon and then the **Connect** icon.

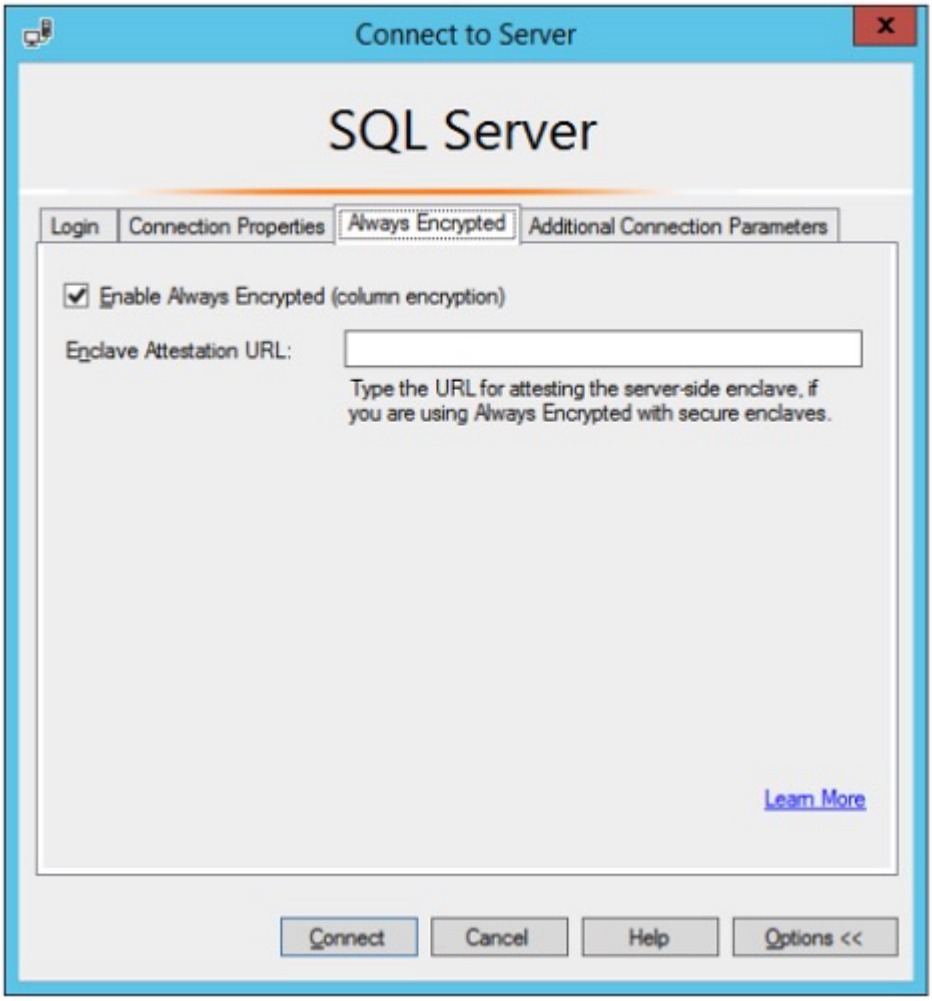


Figure: Enable Always Encrypted in SSMS

2. Select Enable Parameterization.
   a. In the main menu, click **Query** and from the drop-down list, select **Query options…**.

      The **Query Options** window appears.

   b. Select **Execution > Advanced**.

   c. Select the checkbox for **Enable Parameterization for Always Encrypted**.

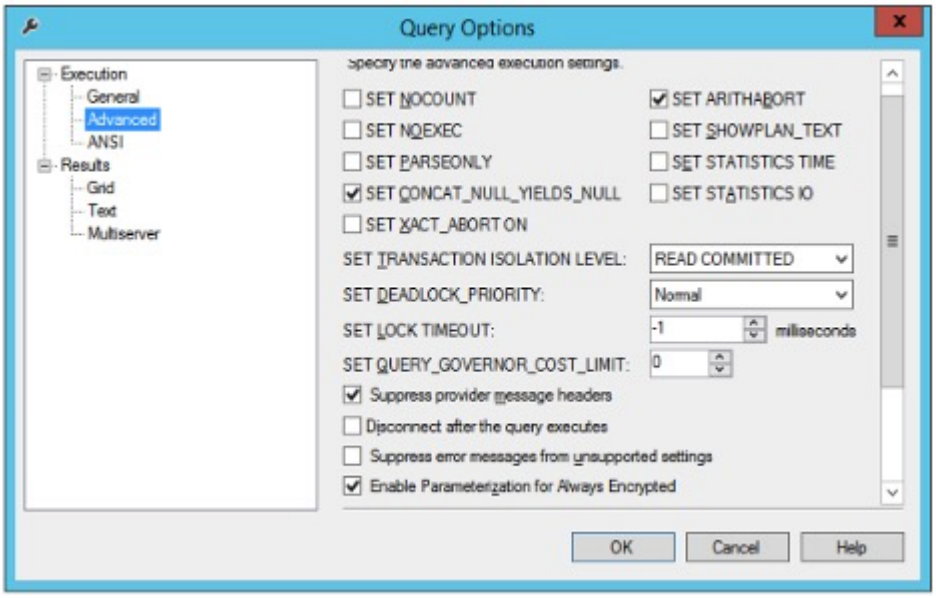


Figure: Enable Parameterization for Always Encrypted queries in SSMS

These are the basic database settings in Microsoft SQL Server and SSMS for testing Always Encrypted in conjunction with YubiHSM.

3. To verify the settings.

   a. Expand the object **Database > Database-Name**. In our example the database name is `Sales3`.

   b. Select **New Query** in the top menu again.

   c. Re-enter the SQL query in the example below and click **Execute**.

```
SELECT * FROM Table_Customers;
```

When the SSMS settings take effect, the encrypted database columns are decrypted, and the values displayed in clear text as shown in the screenshot below.

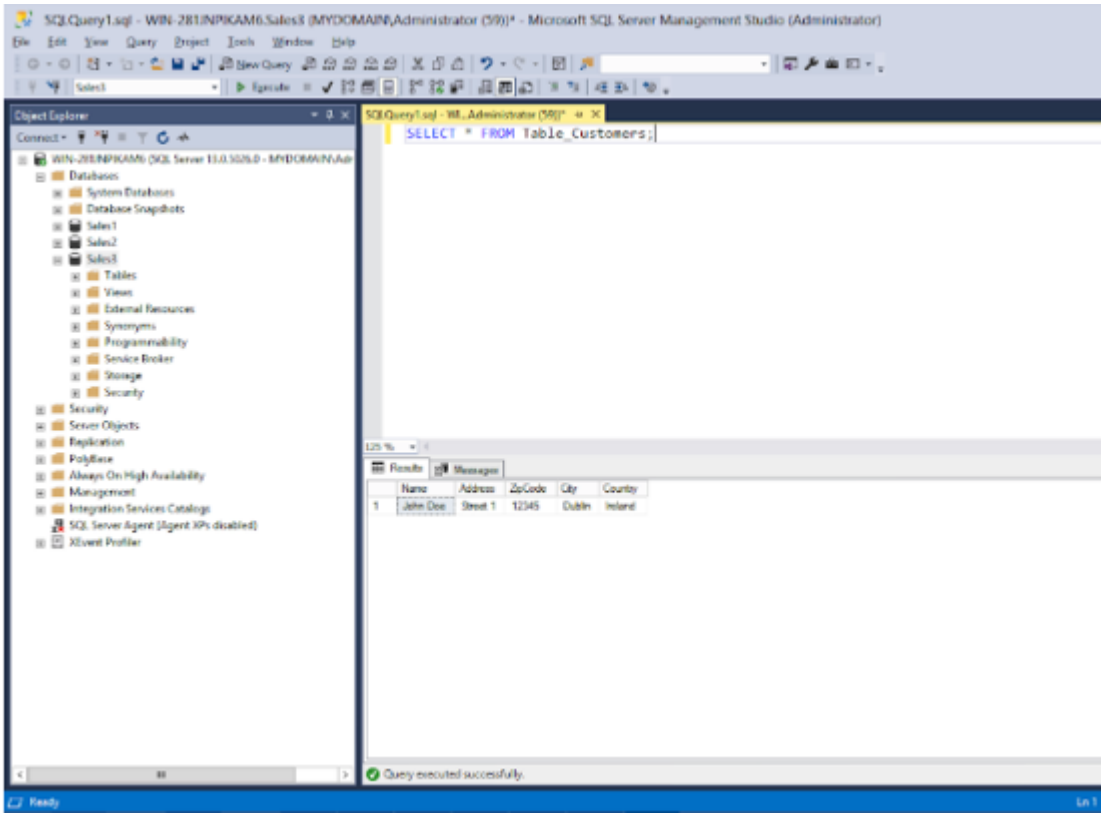


Figure: Decrypted values in the database columns

CHAPTER

TWENTYEIGHT

YUBIHSM 2 WITH KEY STORAGE PROVIDER FOR WINDOWS SERVER

28.1 Configure YubiHSM 2 Key Storage Provider (KSP) for Microsoft Windows Server

This guide is intended to help systems administrators deploy YubiHSM 2 for use in a Windows server environment. The expected outcome is that the YubiHSM 2 is installed and configured with authentication keys, audit keys, and wrap keys. This guide also explains how to make backups and restore keys on a YubiHSM 2.

These guidelines for deployment cover basic topics, so the instructions should be modified as required for your specific environment. It is assumed that you are familiar with the concepts and processes for working with Microsoft Windows Server. It is also assumed that the installation is performed on a single Microsoft Windows Server, but the concept can be extended to more servers.

Important: Before deploying to production, we recommend that you use this guide for installing and testing the setup of the YubiHSM 2 with the Microsoft Windows Server installation in a test or lab environment.

28.2 About the YubiHSM Software

The following YubiHSM 2 software is used in this guide. These items are included as part of the archive file you download from the *YubiHSM 2 SDK Tools And Libraries*.

YubiHSM Connector - Enables communication between the YubiHSM 2 and applications that use it. We recommend that the YubiHSM Connector run on the host operating system if the calling application is deployed to a VM. The Connector must always be running.

YubiHSM Shell - The administrative command line tool used to interact with and configure the YubiHSM 2 device. If the YubiHSM Shell is installed on a VM, it will connect to the Connector over a networked connection.

YubiHSM Setup - Helps with setting up a device for specific use cases. Currently supports setting up for use with Microsoft Windows KSP.

YubiHSM Key Storage Provider (KSP) - Acts like a driver for the YubiHSM 2 device on Windows and enables it to work with applications that leverage Microsoft's Cryptographic API Next Generation (CNG). Examples of calling applications are Microsoft Certificate Services or Microsoft SQL Server Always Encrypted.

28.3 Prerequisites and Preparations

The audience of this guide is an experienced systems administrator with a good understanding of Microsoft Windows Server management. In addition, it is helpful to be familiar with the terminology, software, and tools specific to YubiHSM 2. As a primer for these, refer to Glossary.

In order to follow the steps provided in this guide, the following prerequisites must be met:

- Access to Microsoft Windows Server 2012 SP2 or higher, installed in a secure computer network. The system administrator must have elevated system privileges.
- The YubiHSM 2 SDK downloaded from the Yubico YubiHSM 2 Release page and available on the system to be used. Installation instructions are given in the following.
- Two (2) YubiHSM 2 devices, one for deployment and one for backup in hardware.
- Key custodians, if your organization policies require them for the YubiHSM 2 deployment. For more information about key custodians and the associated `M of N` key shares, see *Key Splitting and Key Custodians*.

Important: Although it is possible to configure the YubiHSM 2 on a networked machine, to safeguard its integrity, it is recommended that its configuration be performed on a fresh system in an air-gapped environment, i.e., the steps in this guide should be performed on a stand-alone computer with both Windows Server 2012 SP2 or higher and the YubiHSM 2 software installed. And we recommend that you do not store keys - even under wrap - on network-accessible or otherwise compromise-able storage media.

CHAPTER

TWENTYNINE

KEY SPLITTING AND KEY CUSTODIANS

The preferred method for backing up the YubiHSM 2 keys calls for key splitting and restoring or regenerating, often referred to as setting up an `M of n` scheme (Shamir's Secret Sharing (SSS). This process ensures no individual can export key material from the YubiHSM 2 and provides a way to control the import of key material that has been exported under wrap from one device into other devices. For example, you would export and import objects for backup purposes, as described in *Backup and Restore Key Material*.

The key that is split among a predetermined number (`n`) of **key custodians** (also known as key shareholders) is known as the wrap key. Each custodian receives their own unique share. To use the key, a minimum number of shares (`m`) must be present so that the key can be regenerated (sometimes called "rejoined"). This minimum number of custodians is called the **privacy threshold**. If this threshold is not attained, the wrap key cannot be regenerated. This minimum number, `n`, should be larger than one.

The exact number of key shares and the privacy threshold are determined by the requirements of your organization. If your organization has policies in place that define how this procedure should be performed, be sure you know these policies before proceeding. You should also have a predetermined practice in place specifying both:

- How the key shares must be recorded (written on paper, photographed, locally printed, or some other means) and
- How they must be stored between uses (for example, offsite archive, safety deposit box, sealed envelope).

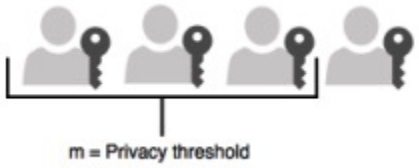


Figure: Privacy threshold

The YubiHSM Setup Tool enables you to perform the key splitting and assigning of shares to key custodians. To carry out the setup process, you need to know who the wrap key custodians will be. During setup, all key custodians must be physically present to record their shares. Exact instructions for key splitting and assigning of shares are given in *Configuring the Primary YubiHSM 2 Device*.

CHAPTER

THIRTY

CORE CONCEPTS

30.1 Objects

The first concept that we will present is the Object. Any persistently stored and self-contained piece of information present in a YubiHSM 2 is an Object. This is intentionally a very generic and broad definition which can be easily rephrased as *everything is an Object*. Objects have associated properties that characterize them and give them different meanings. Regardless of the kind and the specific properties, any YubiHSM 2 device can store up to 256 Objects. Their combined size cannot exceed 126 KB.

30.1.1 Object Type

To identify what an Object can and cannot do, we define an attribute called Object Type, or simply Type. A Type is not enough to *uniquely* identify an Object, but it defines the set of operations that can be performed with or on it. The following types are defined:

Name	Value	yubihsm-shell name
Opaque	0x01	opaque
Authentication Key	0x02	authentication-key
Asymmetric Key	0x03	asymmetric-key
Wrap Key	0x04	wrap-key
HMAC Key	0x05	hmac-key
Template	0x06	template
OTP AEAD Key	0x07	otp-aead-key
Symmetric Key	0x08	symmetric-key
Public Wrap Key	0x09	public-wrap-key

30.1.2 Authentication Key

An Authentication Key is one of the most fundamental Objects there are. Authentication Keys can be used to establish a Session with a device. See *Create and Authenticate a Session*. An Authentication Key is basically two long-lived AES keys: an encryption key and a MAC key. When establishing a Session, the long-lived keys are used to generate three session keys:

- An encryption key used to encrypt the messages exchanged with the device
- A MAC key used to create an authentication tag for each message sent to the device
- A response MAC key used to create an authentication tag for each response message sent by the device

The session keys are temporary and are destroyed when the Session is no longer in use.

30.1.3 Asymmetric Key

An Asymmetric Key Object is what the YubiHSM 2 uses to represent an asymmetric key-pair where only the private key can be used to perform cryptographic operations.

30.1.4 HMAC Key

An HMAC Key is a secret key used when computing and verifying HMAC signatures.

30.1.5 Opaque

An Opaque Object is an unchecked kind of Object, normally used to store raw data in the device. No specific restrictions (besides size limitations) are imposed to this type of Object.

30.1.6 OTP AEAD Key

An OTP AEAD Key Object is a secret key used to decrypt Yubico OTP values for further verification by a validation process.

30.1.7 Symmetric Key

Available with firmware version 2.3.1 or later.

A Symmetric Key Object is a secret key used when encrypting and decrypting AES.

Object Types are encoded as an 8-bit value.

30.1.8 Template

A Template Object is a binary template used for example to validate SSH certificate requests.

30.1.9 Wrap Key

A Wrap Key Object is a secret key used to wrap and unwrap Objects during the export and import process.

Object Types are encoded as an 8-bit value.

30.1.10 Public Wrap Key

A Public Wrap Key Object is an RSA public key used to wrap Objects and (a)symmetric keys during the export process.

30.2 ALGORITHMS

Name	Value	yubihsm-shell name	EC Curve	Usage
RSA PKCS1 SHA1	1	rsa-pkcs1-sha1		RSA sign with PKCS1.5
RSA PKCS1 SHA256	2	rsa-pkcs1-sha256		RSA sign with PKCS1.5
RSA PKCS1 SHA384	3	rsa-pkcs1-sha384		RSA sign with PKCS1.5
RSA PKCS1 SHA512	4	rsa-pkcs1-sha512		RSA sign with PKCS1.5
RSA PSS SHA1	5	rsa-pss-sha1		RSA sign with PSS
RSA PSS SHA256	6	rsa-pss-sha256		RSA sign with PSS
RSA PSS SHA384	7	rsa-pss-sha384		RSA sign with PSS
RSA PSS SHA512	8	rsa-pss-sha512		RSA sign with PSS
RSA 2048	9	rsa2048		Generate RSA key
RSA 3072	10	rsa3072		Generate RSA key
RSA 4096	11	rsa4096		Generate RSA key
EC P256	12	ecp256	secp256r1	Generate EC key
EC P384	13	ecp384	secp384r1	Generate EC key
EC P521	14	ecp521	secp521r1	Generate EC key
EC K256	15	eck256	secp256k1	Generate EC key
EC BP256	16	ecbp256	brainpool256r1	Generate EC key
EC BP384	17	ecbp384	brainpool384r1	Generate EC key
EC BP512	18	ecbp512	brainpool512r1	Generate EC key
HMAC SHA1	19	hmac-sha1		Generate HMAC key
HMAC SHA256	20	hmac-sha256		Generate HMAC key
HMAC SHA384	21	hmac-sha384		Generate HMAC key
HMAC SHA512	22	hmac-sha512		Generate HMAC key
ECDSA SHA1	23	ecdsa-sha1		ECDSA sign
EC ECDH	24	ecdh		
RSA OAEP SHA1	25	rsa-oaep-sha1		RSA decrypt with OAEP

continues on next page

Table 1 – continued from previous page

Name	Value	yubihsm-shell name	EC Curve	Usage
RSA OAEP SHA256	26	rsa-oaep-sha256		RSA decrypt with OAEP
RSA OAEP SHA384	27	rsa-oaep-sha384		RSA decrypt with OAEP
RSA OAEP SHA512	28	rsa-oaep-sha512		RSA decrypt with OAEP
AES128 CCM WRAP	29	aes128-ccm-wrap		Generate Wrap key
Opaque Data	30	opaque-data		Store raw data as an opaque object
Opaque X509 Certificate	31	opaque-x509-certificate		Store X509Certificate as an opaque object
MGF1 SHA1	32	mgf1-sha1		RSA sign with PSS and RSA decrypt with OAEP
MGF1 SHA256	33	mgf1-sha256		RSA sign with PSS and RSA decrypt with OAEP
MGF1 SHA384	34	mgf1-sha384		RSA sign with PSS and RSA decrypt with OAEP

continues on next page

Table 1 – continued from previous page

Name	Value	yubihsm-shell name	EC Curve	Usage
MGF1 SHA512	35	mgf1-sha512		RSA sign with PSS and RSA decrypt with OAEP
SSH Template	36	template-ssh		Store an SSH template (a binary object used to restrict how and when an SSH CA private key should be used)
Yubico OTP AES128	37	aes128-yubico-otp		Generate OTP AEAD key
Yubico AES Authentication	38	aes128-yubico-authentication		Store authentication key
Yubico OTP AES192	39	aes192-yubico-otp		Generate OTP AEAD key
Yubico OTP AES256	40	aes256-yubico-otp		Generate OTP AEAD key
AES192 CCM WRAP	41	aes192-ccm-wrap		Generate and store wrap key
AES256 CCM WRAP	42	aes256-ccm-wrap		Generate and store wrap key
ECDSA SHA256	43	ecdsa-sha256		ECDSA sign
ECDSA SHA384	44	ecdsa-sha384		ECDSA sign

continues on next page

Table 1 – continued from previous page

Name	Value	yubihsm-shell name	EC Curve	Usage
ECDSA SHA512	45	ecdsa-sha512		ECDSA sign
ED25519	46	ed25519		Generate ED key
EC P224	47	ecp224	secp224r1	Generate EC key
AES KWP	55	aes-kwp		Internal use only

30.3 Attestation

Asymmetric keys in the YubiHSM can be attested by another Asymmetric key. The attestation process creates a new x509 certificate for the attested key.

The device comes pre-loaded with an attestation key and certificate referenced by ID `0`. It is possible to use your own key and certificate for attestation, these then must have the same ID and the key has to have the `sign-attestation-certificate` Capability set.

30.3.1 Details

- Serial is a random 16 byte integer
- Issuer is the subject of the attesting certificate
- Dates is copied from the attesting certificate
- Subject is the string `YubiHSM Attestation id 0x` with the attested ID appended
- If the attesting key is RSA the signature is SHA256-PKCS#1v1.5
- If the attesting key is EC the signature is ECDSA-SHA256

30.3.2 Certificate Extensions

Some certificate extensions are added in the generated certificate and/or the pre-loaded certificate:

OID	Description	Data Type	Generated/Pre-loaded
1.3.6.1.4.1.41482.4.1	Firmware version	Octet String	Both
1.3.6.1.4.1.41482.4.2	Serial number	Integer	Both
1.3.6.1.4.1.41482.4.3	*Origin*	Bit String	Generated
1.3.6.1.4.1.41482.4.4	*Domain*	Bit String	Generated
1.3.6.1.4.1.41482.4.5	*Capability*	Bit String	Generated
1.3.6.1.4.1.41482.4.6	*Object ID*	Integer	Generated
1.3.6.1.4.1.41482.4.9	*Label*	Utf8String	Generated
1.3.6.1.4.1.41482.4.10	*FIPS*	Integer	Pre-loaded
1.3.6.1.4.1.41482.4.12	*FIPS*	Boolean	Generated

30.3.3 Pre-Loaded Certificates

The pre-loaded certificate can be fetched as an opaque object with ID 0. This will in turn be signed by an intermediate CA which is signed by a Yubico root CA.

30.3.4 Intermediates:

E45DA5F361B091B30D8F2C6FA040DB6FEF57918E.pem

30.4 Capability

A Capability is an attribute that can be given to an *Objects* allowing specific operations to be performed on or with it. Commands like digital signature generation and data decryption require (and check) for a predetermined set of Capabilities to be present on an Object. Further below is the list of existing Capabilities.

It is important to know that there are no restrictions on which Capabilities can be set on an Object. Specifically, this means that it is possible to assign meaningless Capabilities to Objects that will never be able to use them, for example it is possible to have an Asymmetric Object with the Capability `verify-hmac`. Such a Capability only makes sense for HMAC Key objects, but the device allows defining a superset. Lack of Capabilities required for a specific operation causes a command requiring that Capability to fail.

30.4.1 Delegated Capabilities

Every Object stored on the device has an associated set of Capabilities. There is a second set of so-called Delegated Capabilities that only Authentication Keys and Wrap Keys have. This is used to capture the indirection that Authentication Keys and Wrap Keys can be used as a means of storing more Objects on a device. In both cases Delegated Capabilities are used as a filter.

For Authentication Keys, Delegated Capabilities define the set of Capabilities that can be set or "bestowed" onto an Object created by the Authentication Key. Any operation attempting to create Objects with a Capability outside of this set fails.

For Wrap Keys, Delegated Capabilities define the set of Capabilities that an Object can have when imported or exported using the Wrap Key. A larger set of Capabilities causes the import operation to fail.

30.4.2 Protocol Details

A Set of Capabilities is an 8-byte value. Each Capability is identified by a specific bit, as shown in the `Hex Mask` column below.

Name	Hex Mask	Applicable Objects	Description
Asymmetric Keys			

continues on next page

Table 2 – continued from previous page

Name	Hex Mask	Applicable Objects	Description
	0x0000020000000000		
delete-asymmetric -key		authentication -key	Delete Asymmetric Key Objects
	0x0000000000000010		
generate-asymmetric -key		authentication -key	Generate Asymmetric Key Objects
	0x0000000000000008		
put-asymmetric-key		authentication -key	Write Asymmetric Key Objects
——————————**Authentication Keys**——————————-			
	0x0000010000000000		
delete-authen- tication-key		authentication -key	Delete Authentication Key Objects
	0x0000000000000004		
put-authentication -key		authentication -key	Write Authentication Key Objects
	0x0000400000000000		
change- authentication-key		authentication -key	Replace Authentication Key Objects
——————————**Certificate**——————————-			
	0x0000000400000000		
sign-attestation- certificate		authentication -key, asymmetric-key	Attest properties of Asymmetric Key Objects
sign-ssh-certificate	0x0000000002000000	authentication -key, asymmetric-key	Sign SSH certificates
——————————**Data**——————————			

continues on next page

Table 2 – continued from previous page

Name	Hex Mask	Applicable Objects	Description
decrypt-cbc	0x0010000000000000	authentication-key, symmetric-key	Decrypt data using AES CBC mode. Available with firmware version 2.3.1 or later.
decrypt-ecb	0x0004000000000000	authentication-key, symmetric-key	Decrypt data using AES ECB mode. Available with firmware version 2.3.1 or later.
decrypt-oaep	0x0000000000000400	authentication-key, asymmetric-key	Decrypt data using RSA-OAEP
decrypt-pkcs	0x0000000000000200	authentication-key, asymmetric-key	Decrypt data using RSA-PKCS1v1.5
encrypt-cbc	0x0020000000000000	authentication-key, symmetric-key	Encrypt data using AES CBC mode. Available with firmware version 2.3.1 or later.
encrypt-ecb	0x0008000000000000	authentication-key, symmetric-key	Encrypt data using AES ECB mode. Available with firmware version 2.3.1 or later.
——————————**ECDH**——————————			

continues on next page

Table 2 – continued from previous page

Name	Hex Mask	Applicable Objects	Description
derive-ecdh	0x0000000000000800	authentication-key, asymmetric-key	Perform ECDH
Global			
get-option	0x0000000000040000	authentication-key	Read device-global options
set-option	0x0000000000020000	authentication-key	Write device-global options
HMAC			
delete-hmac-key	0x0000080000000000	authentication-key	Delete HMAC Key Objects
generate-hmac-key	0x0000000000200000	authentication-key	Generate HMAC Key Objects
put-mac-key	0x0000000000100000	authentication-key	Write HMAC Key Objects
sign-hmac	0x0000000000400000	authentication-key, hmac-key	Compute HMAC of data
verify-hmac	0x0000000000800000	authentication-key, hmac-key	Verify HMAC of data
Log			
get-log-entries	0x0000000001000000	authentication-key	Read the Log Store
Opaque			
delete-opaque	0x0000008000000000	authentication-key	Delete Opaque Objects

continues on next page

Table 2 – continued from previous page

Name	Hex Mask	Applicable Objects	Description
get-opaque	0x0000000000000001	authentication-key	Read Opaque Objects
put-opaque	0x0000000000000002	authentication-key	Write Opaque Objects
——————————OTP——————————			
create-otp-aead	0x0000000040000000	authentication-key, otp-aead-key	Create OTP AEAD
decrypt-otp	0x0000000020000000	authentication-key, otp-aead-key	Decrypt OTP
delete-otp-aead-key	0x0000200000000000	authentication-key	Delete OTP AEAD Key Objects
generate-otp-aead-key	0x0000001000000000	authentication-key	Generate OTP AEAD Key Objects
put-otp-aead-key	0x0000000800000000	authentication-key	Write OTP AEAD Key Objects
randomize-otp-aead	0x0000000080000000	authentication-key, otp-aead-key	Create OTP AEAD from random data
rewrap-from-otp-aead-key	0x0000000100000000	authentication-key, otp-aead-key	Rewrap AEADs from one OTP AEAD Key Object to another

continues on next page

Table 2 – continued from previous page

Name	Hex Mask	Applicable Objects	Description
	0x0000000200000000		
rewrap-to-otp-aead-key		authentication-key, otp-aead-key	Rewrap AEADs to one OTP AEAD Key Object from another
——————————**Random**——————————			
get-pseudo-random	0x0000000000080000	authentication-key	Extract random bytes
——————————**Reset**———————————-			
reset-device	0x0000000010000000	authentication-key	Perform a factory reset on the device
——————————**Signatures**———————————			
sign-ecdsa	0x0000000000000080	authentication-key, asymmetric-key	Compute digital signatures using ECDSA
sign-eddsa	0x0000000000000100	authentication-key, asymmetric-key	Compute digital signatures using EDDSA
sign-pkcs	0x0000000000000020	authentication-key, asymmetric-key	Compute signatures using RSA-PKCS1v1.5
sign-pss	0x0000000000000040	authentication-key, asymmetric-key	Compute digital signatures using using RSA-PSS

continues on next page

Table 2 – continued from previous page

Name	Hex Mask	Applicable Objects	Description
——————————**Template**——————————-			
delete-template	0x0000100000000000	authentication -key	Delete Template Objects
get-template	0x0000000004000000	authentication -key	Read Template Objects
put-template	0x0000000008000000	authentication -key	Write Template Objects
——————————**Wrap** ——————————-			
delete-wrap-key	0x0000040000000000	authentication -key	Delete Wrap Key Objects
export-wrapped	0x0000000000001000	authentication -key, wrap-key	Export other Objects under wrap
exportable-under -wrap	0x0000000000010000	all	Mark an Object as exportable under wrap
generate-wrap-key	0x0000000000008000	authentication -key	Generate Wrap Key Objects
import-wrapped	0x0000000000002000	authentication -key, wrap-key	Import wrapped Objects
put-wrap-key	0x0000000000004000	authentication -key	Write Wrap Key Objects
unwrap-data	0x0000004000000000	authentication -key, wrap-key	Unwrap user-provided data

continues on next page

Table 2 – continued from previous page

Name	Hex Mask	Applicable Objects	Description
wrap-data	0x0000002000000000	authentication -key, wrap-key	Wrap user-provided data
Public Key Wrap			
put-public-wrap -key	0x0040000000000000	authentication -key, wrap-key	Write RSA Public Wrap Key
delete-public-wrap -key	0x0080000000000000	authentication -key, wrap-key	Delete RSA Public Wrap Key
Symmetric Keys			
generate-symmetric -key	0x0001000000000000	authentication -key	Generate AES key. Available with firmware version 2.3.1 or later.
put-symmetric-key	0x0000800000000000	authentication -key	Import AES key. Available with firmware version 2.3.1 or later.
delete-symmetric-key	0x0002000000000000	authentication -key	Delete AES key. Available with firmware version 2.3.1 or later.

30.5 Domain

A Domain is a logical partition that can be conceptually mapped to a container. In a YubiHSM 2 there are 16 independent Domains; an Object can belong to one or more Domains.

Note: Authentication Keys are Objects and thus can belong to multiple Domains.

Domains serve as a means to secure Objects so that they cannot be addressed by independent applications running on the same device. This is achieved by specifying the Object's Domain. Only users or applications that belong to the same Domain as an Object can access it or use it.

The details involved in accessing an Object are explained in the *Effective Capabilities (Tying It All Together)* page.

30.5.1 Protocol Details

Domains are encoded as 16-bit values, where each Domain is represented by a bit

Domain Number	Hex Mask
1	0x0001
2	0x0002
3	0x0004
4	0x0008
5	0x0010
6	0x0020
7	0x0040
8	0x0080
9	0x0100
10	0x0200
11	0x0400
12	0x0800
13	0x1000
14	0x2000
15	0x4000
16	0x8000

30.6 Effective Capabilities (Tying It All Together)

This document describes how Object-related concepts interact with each another.

Let us assume that we are establishing a Session with Authentication Key `0xabcd` so that the Session can use the Asymmetric Key `0x1234` to sign some data. We are assuming that Asymmetric Key `0x1234` is an RSA 2048-bit key and that we would like to generate a signature using RSASSA-PSS.

30.6.1 Create and Authenticate a Session

Creating and authenticating a Session requires knowledge of what the long-lived keys are (or what the associated derivation password is).

When a valid Session is established, certain properties of the Authentication Key used to create the Session are inherited by the Session itself. These are:

- The Domain(s) to which the Authentication Key belongs (for more information, see *Domain*),
- The Capabilities of the Authentication Key (see *Capability*) and
- The Delegated Capabilities (see *Capability*) associated with Authentication Key `0xabcd` .

The Session's inherited properties serve to ensure that the only Objects stored in the HSM 2 that we can see and access are those that belong to the same Domain(s) as Authentication Key `0xabcd`.

30.6.2 Generate a Signature

The required capability must be set on both the Authentication Key used to establish the Session (Authentication Key `0xabcd`) and the target Object used to perform the operation (Asymmetric Key `0x1234`).

Assuming that Asymmetric Key `0x1234` is in one such Domain, we can now continue and ask the HSM 2 to generate a signature. To do so we will send the `Sign Data` command over the Session. It will not execute successfully unless the arguments of the command are valid, i.e., no malformed data can be sent to the device or an error will occur.

Both Authentication Key `0xabcd` and Asymmetric Key `0x1234` must have the Capability `sign-pss` set.

30.6.3 Effective Capabilities and Role Definition

The overlap between

- The Capabilities of the Authentication Key used to establish the Session and
- The Capabilities of the target Object involved in the operation

defines the **Effective Capabilities**. An operation on a given target Object over a given Session can succeed only if the Capabilities required by the operation are included in the Effective Capabilities.

The interaction between Domains and Effective Capabilities enables flexible setup and role definition. For example,

- It is possible to assign a set of Capabilities to an Object, and then distribute those Capabilities across different Authentication Keys so that each key is enabled to perform only a single operation on the target Object, and no key performs the same operation as any other key.
- Similarly, it is possible to disable specified operations by not assigning the requisite Capabilities to an Authentication Key. For example, an "Administrator" Authentication Key could be enabled only to create keys while a "User" Authentication Key could be enabled only to use those same keys.

30.6.4 Workflow

1. Determine which Objects will have operations performed on them
2. Determine which Authentication Keys you will use
3. Determine which operations will be performed
4. Use a spreadsheet (if necessary) to map out the interaction between the first three items
5. With the aid of the spreadsheet, create domains to enable the interaction.

   Note: Authentication Keys are Objects and thus can belong to multiple Domains.

6. You could construct your domains:
   - per operation - put an Object and an Authentication Key into each domain, or
   - per Object - put the Authentication Key(s) for all the operations to be performed on each Object into a single domain
   - per Authentication Key - put the requisite Object(s) into each Domain.

   For example, if you wanted Jan to do the signing and Ola to do the importing, you could adopt any of the above options, but the Effective Capabilities enable you to assign far more complex webs of responsibilities.
7. Use the spreadsheet to set the Capabilities and Delegated Capabilities appropriately, "appropriateness" being determined by the Objects and operations to be performed on them.

30.7 Errors

Below are error codes returned by a YubiHSM device.

Name	Value	Description
OK	0x00	Success
INVALID COMMAND	0x01	Unknown command
INVALID DATA	0x02	Malformed data for the command
INVALID SESSION	0x03	The session has expired or does not exist
AUTHENTICATION FAILED	0x04	Wrong Authentication Key
SESSIONS FULL	0x05	No more available sessions
SESSION FAILED	0x06	Session setup failed
STORAGE FAILED	0x07	Storage full
WRONG LENGTH	0x08	Wrong data length for the command
INSUFFICIENT PERMISSIONS	0x09	Insufficient permissions for the command
LOG FULL	0x0a	The log is full and force audit is enabled
OBJECT NOT FOUND	0x0b	No object found matching given ID and Type
INVALID ID	0x0c	Invalid ID
SSH CA CONSTRAINT VIOLATION	0x0e	Constraints in SSH Template not met
INVALID OTP	0x0f	OTP decryption failed
DEMO MODE	0x10	Demo device must be power-cycled
OBJECT EXISTS	0x11	Unable to overwrite object

30.8 FIPS

Note: This section applies to YubiHSM 2 FIPS devices only.

The YubiHSM 2 is available in a FIPS-capable version called YubiHSM 2 FIPS.

The YubiHSM 2 FIPS is certified at FIPS 140-2 Level 3, which means it can be used in solutions that are meant to comply with FIPS 140-2 requirements.

The YubiHSM 2 FIPS can be configured in an approved mode and a non-approved mode of operation. In the approved mode, only FIPS-approved algorithms are supported. In the non-approved mode, additional non-approved algorithms such as `rsa-pkcs1-sha1` are supported.

FIPS-approved mode can be configured only after a device reset by enabling the `fips-mode` option and immediately changing the default Authentication key.

For instructions on configuring the YubiHSM 2 FIPS in FIPS-approved mode, see FIPS Mode Support Guide.

A key attestation generated on a YubiHSM 2 FIPS device with firmware version 2.4.1 or newer will have an X.509 extension present with OID `1.3.6.1.4.1.41482.4.12`. If the key attestation was generated in FIPS-approved mode, this extension will have the BOOLEAN value `TRUE`. Otherwise, it will have the BOOLEAN value `FALSE`.

The pre-loaded certificate of a YubiHSM 2 FIPS device will have an X.509 extension present with OID `1.3.6.1.4.1.41482.4.10`. This extension will have an INTEGER value encoding its FIPS certificate. Currently, the value `6` refers to the YubiHSM 2 FIPS certificate for firmware version 2.2.

30.9 Label

A Label is a sequence of bytes that can be used to add a mnemonic reference to Objects.

30.9.1 Protocol Details

Labels are `40` bytes long. As far as the YubiHSM is concerned, the label is only a string of raw bytes and is not restricted to printable characters or valid UTF-8 glyphs.

30.10 Logs

A YubiHSM 2 device maintains a list of recently executed commands in a portion of non-volatile memory known as the Log Store. This allows logging commands across different power cycles. Specific commands are used to extract logs from the device. Since the Log Store uses non-volatile memory, it can only store up to `62` different entries. When the Log Store is full, it is used as a circular buffer, meaning that the least recently used entry is overwritten.

It is possible to set the device in Force Audit mode. When this is done entries from the Log Store must be retrieved or commands that cannot be logged will fail. Together with individual commands, power-on and reboot events are also logged.

The establishment of a session is logged like any other operation; however those commands are always allowed, independent of the current status of the Log Store. This is so that it is always possible to retrieve logs and free up the Log Store, even when the device is in Force Audit mode and the Log Store is full. However, the number of unlogged authentication and power-up events is stored in a counter that is retrieved as part of the log retrieval.

Entries in the Log Store are organized to form a chain of hashes. This enables auditors to verify that a given set of entries has not been tampered with after extraction, and that all entries are present. More details on the format of log entries can be found in the protocol description document for *GET LOG ENTRIES Command*.

30.11 Object ID

The ID property is used to identify an Object of a given Type. This means that to **uniquely** identify an Object stored on a YubiHSM 2, the couple `(Type, ID)` is required. There can be more than one Object with a given ID and more than one Object with a given Type, but only one Object with a specific ID and Type. This is so that logical connections between Objects can be established by giving a set of connected Objects of different Types the same ID.

An Object ID can have values in the range `[0-65535]` or `[0x0000-0xffff]` in hexadecimal. Note that this range is larger than the maximum number of Objects that can be stored in the device (256). Regardless of the type, ID `0x0000` and `0xffff` are reserved for internal Objects.

30.11.1 Protocol Details

Object IDs are encoded as 16-bit values.

30.12 Options

Options are device-global settings. The following Options are defined:

Option Name	Hex Value
force-audit	0x01
command-audit	0x03

The data payload is Option-specific.

30.12.1 Force Audit

This Option is used to enable Force Audit mode which prevents the device from performing additional operations when the *Logs* is full.

The Option accepts three different values:

- 0x00: Option disabled
- 0x01: Option enabled
- 0x02: Option permanently enabled (only possible to turn off through factory reset)

30.12.2 Command Audit

This Option is used to enable or disable logging of specific commands. Logging commands impacts performance. By default logging is enabled for all operations.

The Option accepts three different values:

- 0x00: Option disabled
- 0x01: Option enabled
- 0x02: Option permanently enabled (only possible to turn off through factory reset)

Multiple commands can be specified at once with the syntax `C1 V1, C2 V2, ..., Cn Vn` where `Ci` is the Command Code and `Vi` is the Option Value. An example of this syntax can be found at the *SET OPTION Command* description.

30.13 Origin

The Origin is a one-byte value that is part of the metadata associated with an asymmetric key object. The origin indicates whether the asymmetric key was generated on a YubiHSM 2 device or generated externally and subsequently imported. If a key was imported, the origin also indicates whether the key was imported in plaintext or using a wrap key.

Origins are also used when generating a key attestation. The attestation certificate will contain the key's origin as an X.509 extension. See *Attestation*.

30.13.1 Protocol Details

Origins are encoded as 8-bit values, where each defined origin is represented by a bit according to the following table:

Name	Hex Mask
generated	0x0001
imported	0x0002
imported_wrapped	0x0010

Note that not all combinations of these bits are valid. In practice, only the combinations `0x0001`, `0x0002`, and `0x0011`, `0x0012` can occur.

30.14 Sequence

The Sequence is a one-byte value that is part of the metadata associated with an Object. The Sequence describes how many times an Object with a given ID and Type has been written. This is mostly useful for caching to determine if new data needs to be fetched from the device.

30.14.1 Protocol Details

Sequence is 8 bits long and will wrap.

30.15 Session

A Session is not a property of a specific Object, but rather it is used to describe a logical connection between an application and a device. Sessions are end-to-end encrypted and authenticated using Session Keys. These keys are derived from long-lived, pre-shared Authentication Key Objects as part of the sessions authentication process. The Session creation and authentication protocol is based on Global Platform SCP03.

On a single YubiHSM 2 it is possible to establish up to `16` independent and concurrent Sessions. Note that while multiple concurrent Sessions can be active at a given time, the device still serves as a rendezvous point. This means that time-consuming operations such as generating a long RSA key will block commands in other Sessions. Sessions are addressed with a number in the range `[0-15]`.

Sessions have an expiration period of 30 seconds of inactivity in order to prevent resource starvation. After 30 seconds, the device will consider a Session inactive and will move it to the pool of re-usable Sessions. Whenever a command is executed on a given Session, the inactivity timer is reset, meaning that if a Session is being constantly used, it will not expire.

Some of the operations that can be performed on a YubiHSM 2 do **not** require a Session. The implication is that the command and its response will travel unencrypted to and from the device. These commands are only generic status commands, making Sessions required for any meaningful operation.

The long-lived keys required to derive Sessions can be explicitly used in the relevant commands. However, there are built-in functionalities to derive those keys from a password using `10,000` iterations of `PBKDF2` with the salt `Yubico`, making the process more human-friendly. **Every new or factory-reset YubiHSM 2 has a default Authentication Key with ID 1 and all Capabilities and all Domains set.** This is equivalent to a superuser or an administrator. The long-lived keys for this Object are derived using the process previously described with the password password.

Warning: It is crucial to delete this well-known Authentication Key before deployment.

CHAPTER

THIRTYONE

YUBIHSM COMMAND REFERENCE

This section contains a list of the commands supported by the YubiHSM 2.

Important: The YubiHSM 2 is certified at FIPS 140-2 Level 3 .

The low-level format for each command message and the relative response is provided, together with an example of how that command can be used within the yubihsm-shell.

31.1 OPEN SESSION Command

This command is the combination of sending two commands in sequence to the YubiHSM:

- The command to create a session
- The command to authenticate the session

The user of yubihsm-shell does not need to run these commands separately as that is taken care of by the session open command that uses those two commands behind the scenes.

Opens an authenticated session to the device. Subsequent commands can be communicated to the device over this authenticated session.

31.1.1 Interactive Mode

```
yubihsm> session open w:authkey, i:password=-
```

Parameters

- **authkey** Required.

  Authentication key object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **password**

  The password used to authenticate the session. The password is prompted for if not specified.

Example

Create a new session with Authentication Key 1 using the password `password`. This does both the session creation and authentication steps.

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

31.1.2 Command Line Mode

A session is automatically created when executing yubihsm-shell commands on the command line.

31.2 AUTHENTICATE SESSION Command

Complete the mutual authentication process started with *CREATE SESSION Command*.

Finish the Session negotiation and authenticate the Session to the device. After this command completes successfully the Session is authenticated and can be used.

31.2.1 Shell Example

Create a new Session with Authentication Key 1 using the password `password`, this performs both the creation and authentication steps.

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

31.2.2 Protocol Details

Command

```
Tc = 0x04
Lc = 17
Vc = S || B || M
```

where –

`S` = Session ID (1 byte)

`B` = Host Cryptogram (8 bytes)

`M` = CMAC(S-MAC, 016 || T || Lc + 8 || S || B) (8 bytes)

This is the first authenticated message in the chain.

The device verifies M and B, both using S-MAC.

Response

```
Tr = 0x84
Lr = 0
Vr = Ø
```

31.3 OPEN SESSION ASYMMETRIC Command

Available with firmware version 2.3.1 or later.

Opens an authenticated session to the device using an asymmetric key. The YubiHSM2 and a client should have exchanged public keys earlier. The asymmetric keys are created from the curve EC-P256.

A session opened with an asymmetric authentication key does not need to be authenticated separately. The command is immediately usable if the CREATE SESSION command is successful.

Subsequent commands can be communicated to the device over this authenticated session.

31.3.1 Interactive Mode

```
yubihsm> open_asym w:authkey,i:privkey=-
```

Parameters

- **authkey** Required.

  ObjectID of the asymmetric authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal

- **privkey** Required.

  The private key to open the session with

  Possible Values: Password, path to file or - for stdin

  Default format: PEM

Example

Create a new session with Authentication Key 100 using a private key stored in `priv.key`. This does both the session creation and authentication steps.

```
yubihsm> session open_asym 100 priv.key
Created session 0
```

31.3.2 Command Line Mode

Asymmetric authentication keys cannot be used in command line mode.

31.3.3 Protocol Details

Command

```
Tc = 0x03
Lc = 67
Vc = I || K
```

where –

I = Key ID of an asymmetric authentication key (2 bytes)

K = Ephemeral client public key (65 bytes)

On success the device generates a Session ID S (1 byte) and sets the message counter for the current Session to 1.

The error ERROR_INV_DATA if K is not a valid EC-P256 key.

Response

```
Tr = 0x83
Lr = 82
Vr = S || Kd || R
```

where –

S = Session ID (1 bytes)

Kd = Ephemeral device public key (65 bytes)

R = Recipient (16 bytes)

31.4 BLINK DEVICE Command

Blink the LED of the device to identify it.

This device must be sent over an authenticated session.

31.4.1 Shell Example

Blink the device for 15 seconds.

```
yubihsm> blink 0 15
```

31.4.2 Interactive Mode

```
yubihsm> blink e:session, b:seconds=10
```

Parameters

- **seconds**

  Number of seconds to blink.

  Default Value: 10

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Blink the device for 15 seconds.

```
yubihsm> blink 0 15
```

31.4.3 Command Line Mode

```
$ yubihsm-shell -a blink-device [ --authkey <authKeyID> -p <password> --duration
↪<duration> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `--duration=INT`

  Number of seconds to blink.

  Default Value: 10

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

Example

Blink the device for 15 seconds.

```
$ yubihsm-shell -a blink-device --duration 15
```

31.4.4 Protocol Details

Command

```
Tc = 0x6b
Lc = 1
Vc = S
```

where –

S = Seconds to blink for (1 byte)

Response

```
Tr = 0xeb
Lr = 0
Vr = Ø
```

31.5 CHANGE ASYMMETRIC AUTHENTICATION KEY Command

Available with firmware version 2.3.1 or later.

Replace the Asymmetric Authentication Key used to establish the current Session. It is not possible to modify any of the metadata connected to the Object such as Domains or Capabilities. Only the public key will be modified.

This command must be sent over an authenticated session.

31.5.1 Interactive Mode

```
yubihsm> change authkey_asym e:session,w:key_id,i:pubkey=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  ObjectID of the authentication key used to open the current session and whose public key will be changed. Object ID is a 2 bytes integer. Can be specified in hex or decimal

- **pubkey**

The new public key.

- When using stdin, click CTRL-D to mark end of input.
- Input format for a password string is `password`.
- If `password` format is used, the tool will derive an ec-p256 private key from the input string and calculate the public key from that. The private key is not used for anything else.

Possible Values: File containing the client's public key as an uncompressed ec-p256 public key, `password` or - for stdin

Default Value: stdin

Possible Format for public key file: PEM, HEX, binary

Default format: PEM

Example

Change the current Asymmetric Authentication Key to `newkey.pub`:

```
yubihsm> change authkey_asym 0 100 newkey.pub
Changed Authentication key 0x0064
```

31.5.2 Command Line Mode

This command is not available in command line mode.

31.5.3 Protocol Details

Command

```
Tc = 0x6c
Lc = 2 + 1 + 16 + 16
Vc = I || A || Key
```

Replace the currently used Authentication Key with a new set of keys.

where –

`I` = *Object ID* of the Authentication Key (2 bytes)

`A` = *ALGORITHMS* (1 byte) (ec-p256-yubico-authentication = 0x31)

`Key` = Uncompressed EC-P256 public key (64 bytes)

Response

```
Tr = 0xec
Lr = 2
Vr = I
```

where –

I = *Object ID* of the changed Object (2 bytes)

Note: This command returns ERROR_INV_DATA if Key is not a valid EC-P256 key.

31.6 CHANGE AUTHENTICATION KEY Command

Available with firmware version 2.2.0 or later.

Replace the Authentication Key used to establish the current Session. It is not possible to modify any of the metadata connected to the Object such as Domains or Capabilities. Only the payload data of the Object (for example, the long-lived symmetric keys) will be modified.

The same PBKDF2 derivation scheme described in *Session* is available.

This device must be sent over an authenticated session.

31.6.1 Shell Example

Change the current Authentication Key deriving it from the password newpassword.

```
yubihsm> change authkey 0 1 newpassword
Changed Authentication key 0x0001
```

31.6.2 Interactive Mode

```
yubihsm> change authkey e:session, w:key_id, i:password=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  The ObjectID of the authentication key used to open the current session and whose password will be changed. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **password**

  The new password for key_id. The password is prompted for if not specified.

Example

Change the current Authentication Key deriving it from the password `newpassword`.

```
yubihsm> change authkey 0 1 newpassword
Changed Authentication key 0x0001
```

31.6.3 Command Line Mode

This command is not available in command line mode.

31.6.4 Protocol Details

Command

```
Tc = 0x6c
Lc = 2 + 1 + 16 + 16
Vc = I || A || Ke || Km
```

Replace the currently used Authentication Key with a new set of keys.

where –

`I` = *Object ID* of the Authentication Key (2 bytes)

`A` = *ALGORITHMS* (1 byte)

`Ke` = Encryption Key (16 bytes)

`Km` = Mac Key (16 bytes)

Response

```
Tr = 0xec
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of the changed Object (2 bytes)

31.7 CLOSE SESSION Command

Close the current *Session* and release it for re-use. This device must be sent over an authenticated session.

31.7.1 Shell Example

Close Session 0.

```
yubihsm> session close 0
```

31.7.2 Interactive Mode

```
yubihsm> session close e:session
```

Parameters

session Required.

The ID of the session to close.

Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Close Session `0`.

```
yubihsm> session close 0
```

31.7.3 Command Line Mode

This command does not need to be run separately on the command line. The session will automatically close after the command has been executed.

31.7.4 Protocol Details

Command

```
Tc = 0x40
Lc = 0
Vc = Ø
```

Response

```
Tr = 0xc0
Lr = 0
Vr = Ø
```

31.8 CREATE OTP AEAD Command

Create a Yubico OTP AEAD using the provided data. This device must be sent over an authenticated session.

31.8.1 Shell Example

Create a new AEAD using Otp-aead Key `0x027c` with the key `000102030405060708090a0b0c0d0e0f` and private ID `010203040506`. Store the result in the file `aead`.

```
yubihsm> otp aead_create 0 0x027c 000102030405060708090a0b0c0d0e0f 010203040506 aead
```

31.8.2 Interactive Mode

```
yubihsm> otp aead_create e:session, w:key_id, i:key, i:private_id, F:aead
```

Parameters

- **aead** Required.

  The file to store the Yubico OTP AEAD.

  Default input format: hex

- **key** Required.

  The key used to create the Yubico OTP AEAD.

- **key_id** Required.

  OTP AEAD key object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **private_id** Required.

  The private ID used to create the Yubico OTP AEAD.

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Create a new AEAD using Otp-aead Key `0x027c` with the key `000102030405060708090a0b0c0d0e0f` and private ID `010203040506`. Store the result in the file `aead`.

```
yubihsm> otp aead_create 0 0x027c 000102030405060708090a0b0c0d0e0f 010203040506 aead
```

31.8.3 Command Line Mode

This command is not available in command line mode.

31.8.4 Protocol Details

Command

```
Tc = 0x61
Lc = 24
Vc = I || K || P
```

where –

`I` = *Object ID* of the OTP AEAD Key (2 bytes)

`K` = OTP Key (16 bytes)

`P` = OTP Private ID (6 bytes)

Response

```
Tr = 0xe1
Lr = LA
Vr = A
```

where –

`A` = Nonce concatenated with AEAD (36 bytes)

31.9 CREATE SESSION Command

Begin the mutual authentication process for establishing a *Session*.

Start negotiating a Session with the device. This command tells the device which Authentication Key to use and sends the host challenge part. The response contains the device challenge and device authentication part. To establish the session continue with *AUTHENTICATE SESSION Command*.

31.9.1 Shell Example

Create a new session with Authentication Key 1 using the password `password`. This does both the session creation and authentication steps.

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

31.9.2 Protocol Details

Command

```
Tc = 0x03
Lc = 10
Vc = I || H
```

where –

I = Key set ID (2 bytes)

H = Host Challenge (8 bytes)

The device generates a random Card Challenge C (8 bytes).

The device derives three Session Keys (S-ENC, S-MAC and S-RMAC) starting from the set of two static keys identified by I (K-ENC and K-MAC) and the two challenges H and C, using the same procedure described in SCP03.

The device uses S-MAC together with H and C to compute the Card Cryptogram A. The host will compute the Host Cryptogram B after having received C and derived S-MAC.

On success the device generates a Session ID S (1 byte) and sets the message counter for the current Session to 1.

Response

```
Tr = 0x83
Lr = 17
Vr = S || C || A
```

31.10 DECRYPT CBC Command

Available with firmware version 2.3.1 or later.

Decrypt data in CBC mode.

31.10.1 Interactive Mode

```
yubihsm> decrypt aescbc e:session,w:key_id,s:iv,i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the symmetric key to decrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **iv** Required.

Encryption initialization vector. 16 bytes in HEX format.

- **data**

  Data to decrypt. When using stdin, the end of input is marked with CTRL-D.

  Possible Values: Data or - for stdin

  Default Value: stdin

  Input format: PEM

  Output format: HEX

Example

Decrypt data using key `0x0064`:

```
yubihsm> decrypt aescbc 0 0x0064 00000000000000000000000000000000␣
↪SG0OU4CT2pH2dnd967KyTU5OgdJ8edxJjOf3Yt52gGQ=
c5cffa1c2333fd824a86951cf602bca1
```

31.10.2 Command Line Mode

```
$ yubihsm-shell -a decrypt-aescbc -i <key_id> --iv <iv> [--in <data> --out <out> --
↪authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password will be prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the symmetric key to decrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--iv=STRING` Required.

  Encryption initialization vector. 16 bytes in HEX format.

- `--in=STRING`

  Data to decrypt. Multiple of 16 bytes. When using stdin, the end of input is marked with CTRL-D.

  Possible Values: data or stdin

  Default Value: stdin

  Input format: Binary

- `--out=STRING`

Decrypted data.

Possible Valued: Path to file or stdout

Default Value: stdout

Output format: HEX

Example

Decrypt data using key 0x0064:

```
$ yubihsm-shell -a decrypt-aescbc -i 0x0064 --iv 00000000000000000000000000000000 --in␣
↪data.enc
c5cffa1c2333fd824a86951cf602bca1
```

31.10.3 Protocol Details

Command

```
Tc = 0x71
Lc = 2 + 16 + LE
Vc = I || V || E
```

where –

`I` = *Object ID* of the symmetric key (2 bytes)

`V` = Encryption initialization vector (IV) in HEX (16 bytes)

`E` = Data to decrypt

Response

```
Tr = 0xf1
Lr = LD
Vr = D
```

where –

`D` = Decrypted data

31.11 DECRYPT ECB Command

Available with firmware version 2.3.1 or later.

Decrypt data in ECB mode.

31.11.1 Interactive Mode

```
yubihsm> decrypt aesecb e:session,w:key_id,i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the symmetric key to decrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **data**

  Data to decrypt. When using stdin, the end of input is marked with CTRL-D.

  Possible Values: Data to sign or - for stdin

  Default Value: stdin

  Input format: PEM

  Output format: HEX

Example

Decrypt data using key `0x0064`:

```
yubihsm> decrypt aesecb 0 0x0064 SG0OU4CT2pH2dnd967KyTQSIdJILAhWsmhdFIkHAZMQ=
c5cffa1c2333fd824a86951cf602bca1
```

31.11.2 Command Line Mode

```
$ yubihsm-shell -a decrypt-aesecb -i <key_id> [--in <data> --out <out> --authkey
↪<authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password will be prompted for if not specified.

- `-i, --object-id=SHORT` Required.

Object ID of the symmetric key to decrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal

- `--in=STRING`

  Data to decrypt. Multiple of 16 bytes. When using stdin, the end of input is marked with CTRL-D.

  Possible Values: data or stdin

  Default Value: stdin

  Input format: Binary

- `--out=STRING`

  Decrypted dat.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Output format: HEX

Example

Decrypt data using key `0x0064`:

```
$ yubihsm-shell -a decrypt-aesecb -i 0x0064 --in data.enc
c5cffa1c2333fd824a86951cf602bca1
```

31.11.3 Protocol Details

Command

```
Tc = 0x6f
Lc = 2 + LE
Vc = I || E
```

where –

`I` = *Object ID* of the symmetric key (2 bytes)

`D` = Data to decrypt

Response

```
Tr = 0xef
Lr = LD
Vr = D
```

where –

`D` = Decrypted data

31.12 DECRYPT OAEP Command

Decrypt data encrypted with RSA-OAEP.

31.12.1 Example

Decrypt data stored in file `enc` using key `0x79c3`:

```
yubihsm> decrypt oaep 0 0x79c3 rsa-oaep-sha1 enc
xlwIc7yQf/KkV5v4Y87Q9ZSqLReoNAxlCmmMPA4W08U=
```

31.12.2 Protocol Details

Command

```
T~c~ = 0x59
L~c~ = 2 + 1 + L~D~ + L~H~
V~c~ = I \|\| M \|\| D \|\| H~l~
```

where –

`I` = link:../Concepts/Object_ID.adoc[Object ID] of the Asymmetric Key (2 bytes)

`M` = Hash link:../Concepts/Algorithms.adoc[Algorithm] to use for MGF1 (1 byte)

`D` = Decryption data (256, 384 or 512 bytes)

`H~l~` = Hash of OAEP Label (20, 32, 48 or 64 bytes)

Response

```
T~r~ = 0xc9
L~r~ = L~R~
V~r~ = R
```

where –

`R` = Decrypted data with OAEP padding removed

31.13 DECRYPT OTP Command

Decrypt a Yubico OTP and return counters and timer information.

31.13.1 Shell Example

Decrypt a (hex encoded) Yubico OTP using key ID `0x027c`.

```
yubihsm> otp decrypt 0 0x027c 2f5d71a4915dec304aa13ccf97bb0dbb aead
OTP decoded, useCtr:1, sessionCtr:1, tstph:1, tstpl:1
```

31.13.2 Interactive Mode

```
yubihsm> otp decrypt e:session, w:key_id, s:otp, i:aead
```

Parameters

- **aead** Required.

  Nonce concatenated with AEAD (36 bytes).

  Possible Values: Path to file containing the AEAD

  Default format: binary

- **key_id** Required.

  OTP AEAD key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **otp** Required.

  OTP to decrypt.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384, rsa-oaep-sha512

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Decrypt a (hex encoded) Yubico OTP using key ID `0x027c`.

```
yubihsm> otp decrypt 0 0x027c 2f5d71a4915dec304aa13ccf97bb0dbb aead
OTP decoded, useCtr:1, sessionCtr:1, tstph:1, tstpl:1
```

31.13.3 Command Line Mode

This command is not available in command line mode.

31.13.4 Protocol Details

Command

```
Tc = 0x60
Lc = 2 + 36 + 16
Vc = K || A || O
```

where –

`I` = *Object ID* of the OTP AEAD Key (2 bytes)

`A` = Nonce concatenated with AEAD (36 bytes)

`O` = OTP (16 bytes)

Response

```
Tr = 0xe0
Lr = 6
Vr = S || U || Th || Tl
```

where –

`S` = Session counter (2 bytes)

`U` = Usage counter (1 byte)

`Th` = Timestamp high (1 byte)

`Tl` = Timestamp low (2 bytes)

31.14 DECRYPT PKCS1 Command

Decrypt data encrypted with RSA-PKCS#1v1.5.

31.14.1 Shell Example

Decrypt the file `enc` using key `0xa930`.

```
yubihsm> decrypt pkcs1v1_5 0 0xa930 enc xlwIc7yQf/KkV5v4Y87Q9ZSqLReoNAxlCmmMPA4W08U=
```

31.14.2 Interactive Mode

```
yubihsm> decrypt pkcs1v1_5 e:session, w:key_id, i:data=-
```

Parameters

- **data**

  Input data to decrypt.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Default data format: binary

- **key_id** Required.

  RSA key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Decrypt the file `enc` using key `0xa930`.

```
yubihsm> decrypt pkcs1v1_5 0 0xa930 enc xlwIc7yQf/KkV5v4Y87Q9ZSqLReoNAxlCmmMPA4W08U=
```

31.14.3 Command Line Mode

```
$ yubihsm-shell -a decrypt-pkcs1v15 -i <key_id> [ --authkey <authKeyID> -p <password> --
↪in <data> --out <out_data> --informat <data_format> --outformat <outdata_format> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-i, --object-id=SHORT` Required.

  Object ID or an RSA key. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--in=STRING`

  Data to decrypt.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default data format: binary

- `--informat=ENUM`

  Input data format.

  Possible Values: base64, binary, PEM, hex

- `--out=STRING`

  Decrypted data.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Default data format: binary

- `--outformat=ENUM`

  Output data format.

  Possible Values: base64, binary, PEM, hex

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

Example

Decrypt data stored in file `enc` using key `0x79c3`.

```
$ yubihsm-shell -a decrypt-pkcs1v15 -i 0x79c3 --in enc xlwIc7yQf/
↪KkV5v4Y87Q9ZSqLReoNAxlCmmMPA4W08U=
```

31.14.4 Protocol Details

Command

```
Tc = 0x49
Lc = 2 + LD
Vc = I || D
```

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`D` = Decryption data (256, 384 or 512 bytes)

The data is padded using the PKCS#1v1.5 scheme with `Block Type 2`. The data is decrypted and conformance to the padding scheme must be checked. Padding is then removed and the contained message is returned.

Response

```
Tr = 0xc9
Lr = LR
Vr = R
```

where –

`R` = Decrypted data with padding removed.

31.15 DELETE OBJECT Command

Delete an Object in the device.

31.15.1 Shell Example

Delete Asymmetric Key `0x52b6`.

```
yubihsm> delete 0 0x52b6 asymmetric-key
```

31.15.2 Interactive Mode

```
yubihsm> delete e:session, w:id, t:type
```

Parameters

- **id** Required.

  Object ID of the object to delete. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **type** Required.

  Type of the object to delete.

  Possible Values: asymmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key

Example

Delete Asymmetric Key `0x52b6`.

```
yubihsm> delete 0 0x52b6 asymmetric-key
```

31.15.3 Command Line Mode

```
$ yubihsm-shell -a delete-object -i <id> -t <type> [ --authkey <authKeyID> -p <password>␣
↪]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-i, --object-id=SHORT` Required.

  Object ID of the object to delete. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-t, --object-type=STRING` Required.

  Type of the object to delete.

  Possible Values: asymmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key

Example

Delete Asymmetric Key `0x52b6`.

```
$ yubihsm-shell -a delete-object -i 0x52b6 -t asymmetric-key
```

31.15.4 Protocol Details

Command

```
Tc = 0x58
Lc = 2 + 1
Vc = I || T
```

where –

`I` = *Object ID* (2 bytes)

`T` = Type, *Objects* (1 byte)

Response

```
Tr = 0xd8
Lr = 0
Vr = Ø
```

31.16 DERIVE ECDH Command

Perform an ECDH key exchange with the private key in the device.

31.16.1 Shell Example

Perform an ECDH operation with key `0x52b6` and a public key in the file `pubkey.pem`.

```
yubihsm> derive ecdh 0 0x52b6 pubkey.pem
5898516bcb0cb3db89d53471137c2d1c741b8ba6ebf2bb01f4a62d97342e97b2
```

31.16.2 Interactive Mode

```
yubihsm> derive ecdh e:session, w:key_id, i:pubkey=-
```

Parameters

- **key_id** Required.

  Object ID of an EC key. Object ID is a 2 bytes integer. Can be specified in hex or decimal

- **pubkey**

  The public key of another EC key.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Default data format: PEM

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Perform an ECDH operation with key `0x52b6` and a public key in the file `pubkey.pem`.

```
yubihsm> derive ecdh 0 0x52b6 pubkey.pem
5898516bcb0cb3db89d53471137c2d1c741b8ba6ebf2bb01f4a62d97342e97b2
```

31.16.3 Command Line Mode

```
$ yubihsm-shell -a derive-ecdh -i <key_id> [ --authkey <authKeyID> -p <password> --in
↪<pubkey> --out <ecdh> --informat <pubkey_format> --outformat <ecdh_format> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-i, --object-id=SHORT` Required.

  EC key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--in=STRING`

  The public key of another EC key.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Data Format: PEM

- `--informat=ENUM`

  Format of public key.

  Possible Values: base64, binary, PEM, hex

  Default Value: PEM

- `--out=STRING`

  ECDH key.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Default Data Format: PEM

- `--outformat=ENUM`

  Format of ECDH key.

  Possible Values: base64, binary, PEM, hex

  Default Value: PEM

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

Example

Perform an ECDH operation with key `0x52b6` and a public key in the file `pubkey.pem`.

```
$ yubihsm-shell -a derive-ecdh -i 0x52b6 --in pubkey.pem
5898516bcb0cb3db89d53471137c2d1c741b8ba6ebf2bb01f4a62d97342e97b2
```

31.16.4 Protocol Details

Command

```
Tc = 0x57
Lc = 2 + LD
Vc = K || D
```

where –

`I` = Object ID of the Asymmetric Key (2 bytes)

`D` = Uncompressed public key to perform the exchange with (57, 65, 97, 129 or 133 bytes)

Response

```
Tc = 0xd7
Lc = LX
Vc = X
```

where –

`X` = X coordinate of the completed key exchange

31.17 DEVICE INFO Command

Gets device version, device serial, supported *ALGORITHMS* and available log entries.

31.17.1 Shell Example

Fetch device info for currently connected device. In YubiHSM with firmware version 2.4 and above, the device info also return `Part number` which is required by FIPS.

```
yubihsm> get deviceinfo
Version number:         2.0.0
Serial number:          2000000
Log used:               2/62
Supported algorithms:   rsa-pkcs1-sha1, rsa-pkcs1-sha256, rsa-pkcs1-sha384,
                        rsa-pkcs1-sha512, rsa-pss-sha1, rsa-pss-sha256,
                        rsa-pss-sha384, rsa-pss-sha512, rsa2048,
                        rsa3072, rsa4096, ecp256, ecp384, ecp521, eck256,
                        ecbp256, ecbp384, ecbp512, hmac-sha1, hmac-sha256,
                        hmac-sha384, hmac-sha512, ecdsa-sha1, ecdh,
```

(continues on next page)

(continued from previous page)

```
                        rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384,
                        rsa-oaep-sha512, aes128-ccm-wrap, opaque-data,
                        opaque-x509-certificate, mgf1-sha1, mgf1-sha256,
                        mgf1-sha384, mgf1-sha512, template-ssh,
                        aes128-yubico-otp, aes128-yubico-authentication,
                        aes192-yubico-otp, aes256-yubico-otp,
                        aes192-ccm-wrap, aes256-ccm-wrap,
                        ecdsa-sha256, ecdsa-sha384, ecdsa-sha512,
                        ed25519, ecp224,
Part number:            78CLUFX5000P
```

31.17.2 Interactive Mode

```
yubihsm> get deviceinfo
```

Example

Fetch device info for currently connected device.

```
yubihsm> get deviceinfo
Version number:             2.0.0
Serial number:              2000000
Log used:                   2/62
Supported algorithms:  rsa-pkcs1-sha1, rsa-pkcs1-sha256, rsa-pkcs1-sha384,
                       rsa-pkcs1-sha512, rsa-pss-sha1, rsa-pss-sha256,
                       rsa-pss-sha384, rsa-pss-sha512, rsa2048, rsa3072,
                       rsa4096, ecp256, ecp384, ecp521, eck256, ecbp256,
                       ecbp384, ecbp512, hmac-sha1, hmac-sha256, hmac-sha384,
                       hmac-sha512, ecdsa-sha1, ecdh, rsa-oaep-sha1,
                       rsa-oaep-sha256, rsa-oaep-sha384, rsa-oaep-sha512,
                       aes128-ccm-wrap, opaque-data, opaque-x509-certificate,
                       mgf1-sha1, mgf1-sha256, mgf1-sha384, mgf1-sha512,
                       template-ssh, aes128-yubico-otp,
                       aes128-yubico-authentication, aes192-yubico-otp,
                       aes256-yubico-otp, aes192-ccm-wrap, aes256-ccm-wrap,
                       ecdsa-sha256, ecdsa-sha384, ecdsa-sha512, ed25519,
                       ecp224
```

31.17.3 Command Line Mode

```
$ yubihsm-shell -a get-device-info
```

Example

Fetch device info for currently connected device.

```
$ yubihsm-shell -a get-device-info
Version number:             2.0.0
Serial number:              2000000
Log used:                   2/62
Supported algorithms: rsa-pkcs1-sha1, rsa-pkcs1-sha256, rsa-pkcs1-sha384,
                      rsa-pkcs1-sha512, rsa-pss-sha1, rsa-pss-sha256,
                      rsa-pss-sha384, rsa-pss-sha512, rsa2048, rsa3072,
                      rsa4096, ecp256, ecp384, ecp521, eck256, ecbp256,
                      ecbp384, ecbp512, hmac-sha1, hmac-sha256, hmac-sha384,
                      hmac-sha512, ecdsa-sha1, ecdh, rsa-oaep-sha1,
                      rsa-oaep-sha256, rsa-oaep-sha384, rsa-oaep-sha512,
                      aes128-ccm-wrap, opaque-data, opaque-x509-certificate,
                      mgf1-sha1, mgf1-sha256, mgf1-sha384, mgf1-sha512,
                      template-ssh, aes128-yubico-otp,
                      aes128-yubico-authentication, aes192-yubico-otp,
                      aes256-yubico-otp, aes192-ccm-wrap, aes256-ccm-wrap,
                      ecdsa-sha256, ecdsa-sha384, ecdsa-sha512, ed25519,
                      ecp224
```

31.17.4 Protocol Details

Command

```
Tc = 0x06
Lc = /{ LI /}
Vc = /{ I /}
```

where –

`I` = Page index, can only be `0` = general device info or `1` = part number. Optional and is only available with firmware 2.4 or higher (1 byte)

Response

Unspecified or page index `0`:

```
Tr = 0x86
Lr = 9 + algorithms
Vr = VMajor || VMinor || VBuild || S || Ltotal || Lused || A
```

where –

`VMajor` = Major version number (1 byte)

`VMinor` = Minor version number (1 byte)

`VBuild` = Build version number (1 byte)

`S` = Serial number (4 bytes)

`Ltotal` = Log Store size expressed in number of log entries (1 byte)

`Lused` = Log lines used (1 byte)

`A` = List of supported *ALGORITHMS*

Page index 1:

```
Tr = 0x86
Lr = 13
Vr = P
```

where –

`P` = Part number (13 byte)

31.18 ECHO Command

Return the byte sequence present within the data field, without any modification. Can be sent over an encrypted Session or as a bare command.

31.18.1 Shell Example

Plain echo

```
yubihsm> plain echo 0x3c 10
Response (10 bytes):
3c3c3c3c3c3c3c3c 3c3c
```

Echo over session 0

```
yubihsm> echo 0 0x3c 10
Response (10 bytes):
3c3c3c3c3c3c3c3c 3c3c
```

31.18.2 Interactive Mode

Over Encrypted Session

```
yubihsm> echo e:session, b:byte, w:count
```

Bare Command

```
yubihsm> plain echo b:byte, w:count
```

Parameters

- **byte** Required.

  The byte to be echoed.

- **count** Required.

  How many times the byte will be echoed.

- **session**

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Echo over session 0

```
yubihsm> echo 0 0x3c 10
Response (10 bytes):
33c3c3c3c3c3c3c3c 3c3c
```

Plain echo

```
yubihsm> plain echo 0x3c 10
Response (10 bytes):
3c3c3c3c3c3c3c3c 3c3c
```

31.18.3 Command Line Mode

This command is not available in command line mode.

31.18.4 Protocol Details

Command

```
Tc = 0x01
Lc = LE
Vc = E
```

where –

E = Data to echo (1-2021 bytes)

Response

```
Tr = 0x81
Lr = LE
Vr = E
```

where –

E = Data to echo (1-2021 bytes)

31.19 ENCRYPT CBC Command

Available with firmware version 2.3.1 or later.

Encrypt data in CBC mode.

31.19.1 Interactive Mode

```
yubihsm> encrypt aescbc e:session,w:key_id,s:iv,i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the symmetric key to encrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **iv** Required.

  Encryption initialization vector. 16 bytes in HEX format.

- **data**

  Data to encrypt. Multiple of 16 bytes. When using stdin, the end of input is marked with CTRL-D.

  Possible Values: Data or - for stdin

  Default Value: stdin

  Input format: HEX

  Output format: PEM

Example

Encrypt data using key 0x008c:

```
yubihsm> encrypt aescbc 0 0x008c 00000000000000000000000000000000␣
↪c5cffa1c2333fd824a86951cf602bca1
SG0OU4CT2pH2dnd967KyTU5OgdJ8edxJjOf3Yt52gGQ=
```

31.19.2 Command Line Mode

```
$ yubihsm-shell -a encrypt-aescbc -i <key_id> --iv <iv> [--in <data> --out <out> --
↪authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password will be prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the symmetric key to encrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--iv=STRING` Required.

  Encryption initialization vector. 16 bytes in HEX format

- `--in=STRING`

  Data to encrypt. Multiple of 16 bytes. When using stdin, the end of input is marked with CTRL-D.

  Possible Values: data or stdin

  Default Value: stdin

  Input format: HEX

- `--out=STRING`

  Encrypted data.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Output format: Binary

Example

Encrypt data using key `0x008c`:

```
$ yubihsm-shell -a encrypt-aescbc -i 0x008c --in c5cffa1c2333fd824a86951cf602bca1 --out␣
↪data.enc
```

31.19.3 Protocol Details

Command

```
Tc = 0x72
Lc = 2 + 16 + LD
Vc = I || IV || D
```

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`IV` = Encryption initialization vector (IV) in HEX (16 bytes)

`D` = Data to encrypt (multiple of 16 bytes)

Response

```
Tr = 0xf2
Lr = LE
Vr = E
```

where —

`E` = Encrypted data

31.20 ENCRYPT ECB Command

Available with firmware version 2.3.1 or later.

Encrypt data in ECB mode.

31.20.1 Interactive Mode

```
yubihsm> encrypt aesecb e:session,w:key_id,i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the symmetric key to encrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **data**

  Data to encrypt. Multiple of 16 bytes. When using stdin, the end of input is marked with CTRL-D.

  Possible Values: Data to sign or - for stdin

  Default Value: stdin

  Input format: HEX

  Output format: PEM

Example

Encrypt data using key `0x0064`:

```
yubihsm> encrypt aesecb 0 0x0064 c5cffa1c2333fd824a86951cf602bca1␣
↪SG0OU4CT2pH2dnd967KyTQSIdJILAhWsmhdFIkHAZMQ=
```

31.20.2 Command Line Mode

```
$ yubihsm-shell -a encrypt-aesecb -i <key_id> [--in <data> --out <out> --authkey
↪<authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password will be prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the symmetric key to encrypt with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--in=STRING`

Data to encrypt. Multiple of 16 bytes. When using stdin, the end of input is marked with CTRL-D.

Possible Values: data or stdin

Default Value: stdin

Input format: HEX

- `--out=STRING`

  Encrypted data.

  Possible Value: Path to file or stdout

  Default Value: stdout

  Output format: Binary

Example

Encrypt data using key `0x0064`:

```
$ yubihsm-shell -a encrypt-aesecb -i 0x0064 --in c5cffa1c2333fd824a86951cf602bca1 --out␣
↪data.enc
```

31.20.3 Protocol Details

Command

```
Tc = 0x70
Lc = 2 + LD
Vc = I || D
```

where –

`I` = *Object ID* of the symmetric Key (2 bytes)

`D` = Data to encrypt (multiple of 16 bytes)

Response

```
Tr = 0xf0
Lr = LE
Vr = E
```

where –

`E` = Encrypted data

31.21 EXPORT WRAPPED Command

Retrieves an Object under wrap from the device. The Object is encrypted using AES-CCM with a 16 bytes MAC and a 13 bytes nonce.

Both the Wrap Key and the Authentication Key must have the capability `export-wrapped` and the wrapped object must have the capability `exportable-under-wrap`.

For YubiHSM devices with firmware version 2.4 or later, this command has been extended to enable exporting ed25519 keys with the seed for a private key. To ensure backward compatibility with older versions of the HSM, the default is to not export the seed. Importing such a legacy format results in an all-zero seed if such a key is exported in the future.

31.21.1 Shell Example

Fetch the Asymmetric Key `0x997e` encrypted with Wrap Key `0xcf94` and store the result in the file `key.enc`.

```
yubihsm> get wrapped 0 0xcf94 asymmetric-key 0x997e 0 key.enc
```

31.21.2 Interactive Mode

```
yubihsm> get wrapped e:session, w:wrapkey_id, t:type, w:id, b:include_seed=0, F:file=-
```

Parameters

- **file**

  Encrypted/wrapped object.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

- **id** Required.

  Object ID of the object to be wrapped. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **include_seed**

  Export ED25519 key with its seed. Default is `0`, which does not export the seed for a privacy key.

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **type** Required.

  Type of the object to be wrapped.

  Possible Values: asymmetric-key, symmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key, public-wrap-key

- **wrapkey_id** Required.

  Wrap key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal

Example

Fetch the Asymmetric Key `0x997e` encrypted with Wrap Key `0xcf94` and store the result in the file `key.enc`.

```
yubihsm> get wrapped 0 0xcf94 asymmetric-key 0x997e 0 key.enc
```

31.21.3 Command Line Mode

```
$ yubihsm-shell -a get-wrapped --wrap-id <wrapkey_id> -t <type> -i <object_id> [ --
↪include-seed --authkey <authKeyID> -p <password> --out <out_data> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-i, --object-id=SHORT` Required.

  Object ID of the object to be wrapped. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--include_seed`

  Export ED25519 key with its seed. Default is not to.

- `--out=STRING`

  Encrypted/wrapped object.

  Possible Values: Path to file or stdout

  Default Value: stdout

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified

- `-t, --object-type=STRING` Required.

  Type of the object to be wrapped.

  Possible Values: symmetric-key, symmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key, public-wrap-key

- `--wrap-id=INT` Required.

  Wrap key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

Example

Fetch the Asymmetric Key `0x997e` encrypted with Wrap Key `0xcf94` and store the result in the file `key.enc`.

```
$ yubihsm-shell -a get-wrapped --wrap-id 0xcf94 -t symmetric-key -i 0x997e --out key.enc
```

31.21.4 Protocol Details

Command

```
Tc = 0x4a
Lc = 2 + 1 + 2 /{ + 1 /}
Vc = Iw || T || Io /{ || S /}
```

where –

`Iw` = *Object ID* of Wrap Key to use (2 bytes)

`T` = Type, *Objects* of Object to wrap (1 byte)

`Io` = *Object ID* of Object to wrap (2 bytes)

`S` = 1 to include seed when exporting ED25519 key, `0` otherwise. Optional with firmware 2.4 or higher (1 byte)

Response

```
Tr = 0xca
Lr = 13 + LR
Vr = N || R
```

where –

`N` = Nonce used for this wrap (13 bytes)

`R` = Wrapped data (Length dependent on object)

31.22 EXPORT RSA WRAPPED Command

Available on YubiHSM devices with firmware version 2.4 or higher.

Both the Public Wrap Key and the Authentication Key must have the capability `export-wrapped` and the wrapped object must have the capability `exportable-under-wrap`.

Retrieves an Object under RSA wrap from the device. The wrapped object is serialized using the YubiHSM-internal format.

31.22.1 Interactive Mode

```
yubihsm> get rsa_wrapped e:session, w:wrapkey_id, t:type, w:id, a:aes=aes256, a:hash=rsa-
↪oaep-sha256, a:mgf1=mgf1-sha256, F:file=-
```

Parameters

- **aes**

  Algorithm of the ephemeral AES key

  Possible Values: aes128, aes192 or aes256

  Default Value: aes256

- **file**

  Encrypted/wrapped object.

  Possible Values: Path to file or - for stdin

  Default Value: stdout

  Format: Binary

- **hash**

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- **id** Required.

  Object ID of the object to be wrapped. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **mgf1**

  Hash algorithm to use for MGF1.

  Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

  Default Value: mgf1-sha256

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **type** Required.

  Type of the object to be wrapped.

  Possible Values: asymmetric-key, symmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key, public-wrap-key

- **wrapkey_id** Required.

  Public Wrap Key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal

Example

Fetch the Asymmetric Key `0x997e` encrypted with RSA Public Wrap Key `0xcf94` and store the result in the file `object.enc`.

```
yubihsm> get rsa_wrapped 0 0xcf94 asymmetric-key 0x997e aes192 rsa-oaep-sha384 mgf1-sha1␣
↪object.enc
```

31.22.2 Command Line Mode

```
$ yubihsm-shell -a get-rsa-wrapped --wrap-id <wrapkey_id> -t <type> -i <object_id> [ -A
↪<aes> --oaep <oaep> --mgf1 <mgf1> --authkey <authKeyID> -p <password> --out <out_data>␣
↪]
```

Parameters

- `-A, --algorithm=STRING`

  Algorithm of the ephemeral AES key

  Possible Values: aes128, aes192 or aes256

  Default Value: aes256

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-i, --object-id=SHORT` Required.

  Object ID of the object to be wrapped. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--mgf1=STRING`

  Hash algorithm to use for MGF1.

  Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

  Default Value: mgf1-sha256

- `--oaep=STRING`

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- `--out=STRING`

  Encrypted/wrapped object.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Format: Binary

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified

- `-t, --object-type=STRING` Required.

  Type of the object to be wrapped.

  Possible Values: symmetric-key, symmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key, public-wrap-key

- `--wrap-id=INT` Required.

  Wrap key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

Example

Fetch the Asymmetric Key `0x997e` encrypted with Wrap Key `0xcf94` and store the result in the file `object.enc`.

```
$ yubihsm-shell -a get-rsa-wrapped --wrap-id 0xcf94 -t symmetric-key -i 0x997e -A aes192␣
↪--oaep rsa-oaep-sha384 --mgf1 mgf1-sha1 --out object.enc
```

31.22.3 Protocol Details

Command

```
Tc = 0x76
Lc = 2 + 1 + 2 + 1 + 1 + 1 + LHL
Vc = Iw || To || Ti || Ae || H || M || LH
```

where –

`Iw` = *Object ID* of Wrap Key to use (2 bytes)

`To` = Type, *Objects* of Object to wrap (1 byte)

`Ti` = *Object ID* of Object to wrap (2 bytes)

`Ae` = *ALGORITHMS* of the ephemeral AES key (1 byte)

`H` = *ALGORITHMS* to use for OAEP label (1 byte)

`M` = *ALGORITHMS* to use for MGF1 (1 byte)

`LH` = The label digest (Length dependent on OAEP algorithm)

Response

```
Tr = 0xf6
Lr = LR
Vr = R
```

where –

`R` = RSA wrapped data (Length dependent on object)

31.23 EXPORT RSA WRAPPED KEY Command

Available on YubiHSM devices with firmware version 2.4 or higher.

Both the Public Wrap Key and the Authentication Key must have the capability `export-wrapped` and the wrapped object must have the capability `exportable-under-wrap`.

Wrap an (a)symmetric key. Only asymmetric and symmetric key objects are valid targets. Asymmetric keys are serialized as PKCS#8

31.23.1 Interactive Mode

```
yubihsm> get rsa_wrapped_key e:session, w:wrapkey_id, t:type, w:id, a:aes=aes256,␣
↪a:hash=rsa-oaep-sha256, a:mgf1=mgf1-sha256, F:file=-
```

Parameters

- **aes**

  Algorithm of the ephemeral AES key

  Possible Values: aes128, aes192 or aes256

  Default Value: aes256

- **file**

  Encrypted/wrapped object.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Format: Binary

- **hash**

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- **id** Required.

  Object ID of the object to be wrapped. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **mgf1**

  Hash algorithm to use for MGF1.

  Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

  Default Value: mgf1-sha256

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **type** Required.

Type of the object to be wrapped.

Possible Values: asymmetric-key, symmetric-key

- **wrapkey_id** Required.

  Wrap key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal

Example

Fetch the Asymmetric Key `0x997e` encrypted with RSA Public Wrap Key `0xcf94` and store the result in the file `key.enc`.

```
yubihsm> get rsa_wrapped_key 0 0xcf94 asymmetric-key 0x997e aes192 rsa-oaep-sha384 mgf1-
↪sha1 key.enc
```

31.23.2 Command Line Mode

```
$ yubihsm-shell -a get-rsa-wrapped-key --wrap-id <wrapkey_id> -t <type> -i <object_id> [␣
↪-A <aes> --oaep <oaep> --mgf1 <mgf1> --authkey <authKeyID> -p <password> --out <out_
↪data> ]
```

Parameters

- `-A, --algorithm=STRING`

  Algorithm of the ephemeral AES key

  Possible Values: aes128, aes192 or aes256

  Default Value: aes256

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-i, --object-id=SHORT` Required.

  Object ID of the object to be wrapped. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--mgf1=STRING`

  Hash algorithm to use for MGF1.

  Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

  Default Value: mgf1-sha256

- `--oaep=STRING`

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- `--out=STRING`

Encrypted/wrapped object.

Possible Values: Path to file or stdout

Default Value: stdout

Format: Binary

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified

- `-t, --object-type=STRING` Required.

  Type of the object to be wrapped.

  Possible Values: symmetric-key, symmetric-key

- `--wrap-id=INT` Required.

  Wrap key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

Example

Fetch the Asymmetric Key `0x997e` encrypted with Wrap Key `0xcf94` and store the result in the file `key.enc`.

```
$ yubihsm-shell -a get-rsa-wrapped-key --wrap-id 0xcf94 -t symmetric-key -i 0x997e -A␣
↪aes192 --oaep rsa-oaep-sha384 --mgf1 mgf1-sha1 --out key.enc
```

31.23.3 Protocol Details

Command

```
Tc = 0x74
Lc = 2 + 1 + 2 + 1 + 1 + 1 + LHL
Vc = Iw || To || Ti || Ae || H || M || LH
```

where –

`Iw` = *Object ID* of Wrap Key to use (2 bytes)

`To` = Type, *Objects* of Object to wrap (1 byte)

`Ti` = *Object ID* of Object to wrap (2 bytes)

`Ae` = *ALGORITHMS* of the ephemeral AES key (1 byte)

`H` = *ALGORITHMS* to use for OAEP label (1 byte)

`M` = *ALGORITHMS* to use for MGF1 (1 byte)

`LH` = The label digest (Length dependent on OAEP algorithm)

Response

```
Tr = 0xf4
Lr = LR
Vr = R
```

where –

R = RSA wrapped key (Length dependent on key)

31.24 GENERATE ASYMMETRIC KEY Command

Generate an Asymmetric Key in the device.

31.24.1 Shell Example

Generate a new key using `secp256r1` in the device.

```
yubihsm> generate asymmetric 0 0 eckey 1 sign-ecdsa ecp256
Generated Asymmetric key 0x2846
```

31.24.2 Interactive Mode

```
yubihsm> generate asymmetric e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪a:algorithm
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Value: Maximum of 40 characters string.

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

Possible Values: none, sign-pkcs, sign-pss, sign-ecdsa, sign-eddsa, decrypt-pkcs, decrypt-oaep, derive-ecdh, exportable-under-wrap, sign-ssh-certificate, sign-attestation-certificate

- **algorithm** Required.

  Key algorithm.

  Possible Values: rsa2048, rsa3072, rsa4096, ecp256, ecp384, ecp521, eck256, ecbp256, ecbp384, ecbp512, ed25519, ecp224

Example

Generate a new key using `secp256r1` in the device.

```
yubihsm> generate asymmetric 0 0 eckey 1 sign-ecdsa ecp256
Generated Asymmetric key 0x2846
```

31.24.3 Command Line Mode

```
$ yubihsm-shell -a generate-asymmetric-key -i <key_id> -l <label> -d <domains> -c
↪<capabilities> -A <algorithm> [--authkey <authKeyID> -p <password>]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING`

  Required. The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT`

  Required. Object ID of the asymmetric key. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Maximum of 40 characters string. Can be empty.

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

Possible Values: none, sign-pkcs, sign-pss, sign-ecdsa, sign-eddsa, decrypt-pkcs, decrypt-oaep, derive-ecdh, exportable-under-wrap, sign-ssh-certificate, sign-attestation-certificate

- `-A, --algorithm=STRING` Required.

  Key algorithm.

  Possible Values: rsa2048, rsa3072, rsa4096, ecp256, ecp384, ecp521, eck256, ecbp256, ecbp384, ecbp512, ed25519, ecp224

Example

Generate a new key using `secp256r1` in the device.

```
$ yubihsm-shell -a generate-asymmetric-key -i 0 -l eckey -d 1 -c sign-ecdsa -A ecp256
Generated Asymmetric key 0x2846
```

31.24.4 Protocol Details

Command

```
Tc = 0x46
Lc = 2 + 40 + 2 + 8 + 1
Vc = I || L || D || C || A
```

Generate an Asymmetric key-pair with a given ID. Each parameter has a fixed length and the order is compulsory.

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Effective Capabilities (Tying It All Together)* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

Response

```
Tr = 0xc6
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of the created Asymmetric Key (2 bytes)

31.25 GENERATE HMAC KEY Command

Generate an HMAC Key in the device.

31.25.1 Shell Example

Generate an HMAC-SHA512 key.

```
yubihsm> generate hmackey 0 0 hmackey 1 sign-hmac:verify-hmac hmac-sha512
Generated HMAC key 0xa9bf
```

31.25.2 Interactive Mode

```
yubihsm> generate hmackey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪a:algorithm
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over. 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string.

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key.Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, sign-hmac, verify-hmac, exportable-under-wrap

- **Algorithm** Required.

  Key algorithm.

  Possible Values: hmac-sha1, hmac-sha256, hmac-sha384, hmac-sha512

Example

Generate an HMAC-SHA512 key.

```
yubihsm> generate hmackey 0 0 hmackey 1 sign-hmac:verify-hmac hmac-sha512
Generated HMAC key 0xa9bf
```

31.25.3 Command Line Mode

```
$ yubihsm-shell -a generate-hmac-key -i <key_id> -l <label> -d <domains> -c
↪<capabilities> -A <algorithm> [--authkey <authKeyID> -p <password>]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, sign-hmac, verify-hmac, exportable-under-wrap

- `-A, --algorithm=STRING` Required.

  Key algorithm.

  Possible Values: hmac-sha1, hmac-sha256, hmac-sha384, hmac-sha512

Example

Generate a new key using `secp256r1` in the device:

```
$ yubihsm-shell -a generate-hmac-key -i 0 -l hmackey -d 1 -c sign-hmac,verify-hmac -A␣
↪hmac-sha512
Generated HMAC key 0xa9bf
```

31.25.4 Protocol Details

Command

```
Tc = 0x5a
Lc = 2 + 40 + 2 + 8 + 1
Vr = I || L || D || C || A
```

where –

`I` = *Object ID* of the HMAC Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

Response

```
Tr = 0xda
Lr = 2
Vr = I
```

where –

`I` = *Object ID*

31.26 GENERATE OTP AEAD KEY Command

Generate an OTP AEAD Key for Yubico OTP decryption.

31.26.1 Shell Example

Generate a new AES-256 OTP AEAD Key that can decrypt Yubico OTPs and create new AEADs.

```
yubihsm> generate otpaeadkey 0 0 otpaeadkey 1 decrypt-otp,
 create-otp-aead aes256-yubico-otp 0x01020304
Generated OTP AEAD key 0x027c
```

31.26.2 Interactive Mode

```
yubihsm> generate hmackey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪a:algorithm, u:nonce_id
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Posible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Maximum of 40 characters string. Can be empty

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16.

- **capabilities** Required.

  Capabilities of the key. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, create-otp-aead, decrypt-otp, randomize-otp-aead, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, exportable-under-wrap Use `none` to include no capability.

- **algorithm** Required.

  Key algorithm. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possble Values: aes128-yubico-otp, aes192-yubico-otp, aes256-yubico-otp

- **nonce_id** Required.

  OTP nonce. 4 bytes

Example

Generate a new AES-256 OTP AEAD Key that can decrypt Yubico OTPs and create new AEADs.

```
yubihsm> generate otpaeadkey 0 0 otpaeadkey 1 decrypt-otp,create-otp-aead aes256-yubico-
↪otp 0x01020304
Generated OTP AEAD key 0x027c
```

31.26.3 Command Line Mode

```
$ yubihsm-shell -a generate-otp-aead-key -i <key_id> -l <label> -d <domains> -c
↪<capabilities> -A <algorithm> --nonce <nonce_id> [--authkey <authKeyID> -p <password>]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the asymmetric key. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Can be empty

  Possible Values: Maximum of 40 characters string.

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

  Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `none` to include no capability.Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, create-otp-aead, decrypt-otp, randomize-otp-aead, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, exportable-under-wrap

- `-A, --algorithm=STRING` Required.

  Key lgorithm

  Possible Values: aes128-yubico-otp, aes192-yubico-otp, aes256-yubico-otp

- `--nonce=INT` Required.

  OTP nonce

Example

Generate a new AES-256 OTP AEAD Key that can decrypt Yubico OTPs and create new AEADs.

```
$ yubihsm-shell -a generate-otp-aead-key -i 0 -l otpaeadkey -d 1 -c decrypt-otp,create-
↪otp-aead -A aes256-yubico-otp --nonce 0x01020304
Generated OTP AEAD key 0x027c
```

31.26.4 Protocol Details

Command

```
Tc = 0x66
Lc = 2 + 40 + 2 + 8 + 1 + 4
Vc = I || L || D || C || A || N
```

where –

`I` = *Object ID* of the OTP AEAD Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`N` = Nonce ID (4 bytes)

Response

```
Tr = 0xe6
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of the created OTP AEAD Key (2 bytes)

31.27 GENERATE SYMMETRIC KEY Command

Available with firmware version 2.3.1 or later.

Generate a symmetric Key in the device.

31.27.1 Interactive Mode

```
yubihsm> generate symmetric e:session,w:key_id,s:label,d:domains,c:capabilities,
↪a:algorithm
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by `,` or `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between

  Possible Values: none, decrypt-ecb, encrypt-ecb, decrypt-cbc, encrypt-cbc, exportable-under-wrap

- **algorithm** Required.

  Key algorithm

  Possible Values: aes128, aes192, aes256

Example

Generate a new key using aes256 in the device:

```
yubihsm> generate symmetric 0 0 aeskey 1 encrypt-ecb,decrypt-ecb aes256
Generated symmetric key 0xc040
```

31.27.2 Command Line Mode

```
$ yubihsm-shell -a generate-symmetric-key -i <key_id> -l <label> -d <domains> -c
↪<capabilities> -A <algorithm> [--authkey <authKeyID> -p <password>]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password will be prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the asymmetric key. Use '0' to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by `,` or `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between

  Possible Values: none, encrypt-ecb, decrypt-ecb, encrypt-cbc, decrypt-cbc, exportable-under-wrap

- `-A, --algorithm=STRING` Required.

  Key algorithm

  Possible Values: aes128, aes192, aes256

Example

Generate a new key using secp256r1 in the device:

```
$ yubihsm-shell -a generate-symmetric-key -l aeskey -d 1 -c encrypt-ecb,decrypt-ecb -A␣
↪aes256
Generated symmetric key 0xc040
```

31.27.3 Protocol Details

Command

```
Tc = 0x6e
Lc = 2 + 40 + 2 + 8 + 1
Vc = I || L || D || C || A
```

Generate a symmetric key with a given ID. Each parameter has a fixed length and the order is compulsory.

where –

`I` = *Object ID* of the symmetric Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

Response

```
Tr = 0xee
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of the created symmetric Key (2 bytes)

31.28 GENERATE WRAP KEY Command

Generate a Wrap Key that can be used for export, import, wrap data and unwrap data.

31.28.1 Shell Example

Generate a new Wrap Key that can be used for `wrap` and `unwrap`.

```
yubihsm> generate wrapkey 0 0 wrapkey 1 wrap-data:unwrap-data none
 aes256-ccm-wrap
Generated Wrap key 0x5b3a
```

31.28.2 Interactive Mode

```
yubihsm> generate hmackey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪c:delegated_capabilities, a:algorithm
```

Parameters

- **Session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **Label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string.

- **Domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, wrap-data, unwrap-data, export-wrapped, import-wrapped, exportable-under-wrap

- **delegated_capabilities** Required.

  Delegated capabilities. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- **Algorithm** Required.

  Key algorithm.

  Possible Values: aes128-ccm-wrap, aes192-ccm-wrap, aes256-ccm-wrap, rsa2048, rsa3072, rsa4096

Example

Generate a new Wrap Key that can be used for wrap and unwrap.

```
yubihsm> generate wrapkey 0 0 wrapkey 1 wrap-data:unwrap-data none aes256-ccm-wrap
Generated Wrap key 0x5b3a
```

31.28.3 Command Line Mode

```
$ yubihsm-shell -a generate-wrap-key -i <key_id> -l <label> -d <domains> -c
↪<capabilities> --delegated <delegated_capabilities> -A <algorithm> [--authkey
↪<authKeyID> -p <password>]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified

- `-i, --object-id=SHORT` Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, wrap-data, unwrap-data, export-wrapped, import-wrapped, exportable-under-wrap

- `--delegated=STRING`

  Delegated capabilities of kry. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque,

delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

Default Value: none

- `-A, --algorithm=STRING` Required.

  Key algorithm.

  Possible Values: aes128-ccm-wrap, aes192-ccm-wrap, aes256-ccm-wrap

Example

Generate a new Wrap Key that can be used for wrap and unwrap.

```
$ yubihsm-shell -a generate-wrap-key -i 0 -l wrapkey -d 1 -c wrap-data:unwrap-data -A␣
↪aes256-ccm-wrap
Generated Wrap key 0x5b3a
```

31.28.4 Protocol Details

Command

```
Tc = 0x5b
Lc = 2 + 40 + 2 + 8 + 1 + 8
Vc = I || L || D || C || A || DC
```

where –

`I` = *Object ID* of the Wrap Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`DC` = Delegated *Capability* (8 bytes)

Response

```
Tr = 0xdb
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of created Wrap Key (2 bytes)

31.29 GET DEVICE PUBLIC KEY Command

Available with firmware version 2.3.1 or later.

Fetch the device public key to use with asymmetric authentication to the device. This is end as a bare command and not over an encrypted session.

31.29.1 Example

Get device public key:

```
yubihsm> get devicepubkey
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEfSE6zN59ONnsOf
9C8VGNym+oBgnWO5mjJZJ5Z9kkbpMIhLwkjsqKOhgKI+Slfv3o
XmrcwVzUstLAkQe1HdC/uA==
-----END PUBLIC KEY-----
```

31.29.2 Protocol Details

Command

```
T~c~ = 0x0a
L~c~ = 0
V~c~ = Ø
```

Response

```
T~r~ = 0x8a
L~r~ = 1 + 64
V~r~ = A | | K
```

where –

`A` = *ALGORITHMS* (1 byte)

`K` = Uncompressed EC-P256 public key (64 bytes)

The algorithm will currently always be `ec-p256-yubico-authentication`.

The uncompressed EC key marker is omitted (hence the 64 bytes), similarly to how other EC keys are handled.

31.30 GET LOG ENTRIES Command

Fetch device audit log. Fetch all current entries from the device Log Store.

31.30.1 Shell Example

```
yubihsm> audit get 0
0 unlogged boots found
0 unlogged authentications found
Found 6 items
item:    46 -- cmd: 0x4b -- length:  234 -- session key: 0x0001
 -- target
key: 0xcf94 -- second key: 0x997e -- result: 0xcb -- tick: 335725
 -- hash: 415f51f1f035a1b713e730e4464e4033
item:    47 -- cmd: 0x4c -- length:   77 -- session key: 0x0001
 -- target
key: 0xaff7 -- second key: 0xffff -- result: 0xcc -- tick: 351714
 -- hash: 5496a60d478c2b9c801d8d32ca66b554
item:    48 -- cmd: 0x00 -- length:    0 -- session key: 0xffff
 -- target
key: 0x0000 -- second key: 0x0000 -- result: 0x00 -- tick: 0 -- hash:
  14ac7747ba9bbb243cfc70befeb5349b
item:    49 -- cmd: 0x03 -- length:   10 -- session key: 0xffff
 -- target
key: 0x0001 -- second key: 0xffff -- result: 0x83 -- tick: 139 -- hash:
  b20a8f25c025e693a8e869b433294a20
item:    50 -- cmd: 0x04 -- length:   17 -- session key: 0xffff
 -- target
key: 0x0001 -- second key: 0xffff -- result: 0x84 -- tick: 139 -- hash:
  ebfae425c319ac7a0afbb8b92597de7c
item:    51 -- cmd: 0x67 -- length:    2 -- session key: 0x0001
 -- target
key: 0xffff -- second key: 0xffff -- result: 0xe7 -- tick: 697 -- hash:
  2e395d1b706668737e1d2215813db47e
```

31.30.2 Interactive Mode

```
yubihsm> audit get e:session, F:file=-
```

Parameters

- **Session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **File**

  Log entries. Default output format: ASCII.

  Possible Values: Path to file or - for stdout

Default Value: stdout

Example

```
yubihsm> audit get 0
0 unlogged boots found
0 unlogged authentications found
Found 6 items
item:       46 -- cmd: 0x4b -- length:  234 -- session key: 0x0001 -- target key: 0xcf94␣
↪-- second key: 0x997e -- result: 0xcb -- tick: 335725 -- hash:␣
↪415f51f1f035a1b713e730e4464e4033
item:       47 -- cmd: 0x4c -- length:   77 -- session key: 0x0001 -- target key: 0xaff7␣
↪-- second key: 0xffff -- result: 0xcc -- tick: 351714 -- hash:␣
↪5496a60d478c2b9c801d8d32ca66b554
item:       48 -- cmd: 0x00 -- length:      0 -- session key: 0xffff -- target key:␣
↪0x0000 -- second key: 0x0000 -- result: 0x00 -- tick: 0 -- hash:␣
↪14ac7747ba9bbb243cfc70befeb5349b
item:       49 -- cmd: 0x03 -- length:   10 -- session key: 0xffff -- target key: 0x0001␣
↪-- second key: 0xffff -- result: 0x83 -- tick: 139 -- hash:␣
↪b20a8f25c025e693a8e869b433294a20
item:       50 -- cmd: 0x04 -- length:   17 -- session key: 0xffff -- target key: 0x0001␣
↪-- second key: 0xffff -- result: 0x84 -- tick: 139 -- hash:␣
↪ebfae425c319ac7a0afbb8b92597de7c
item:       51 -- cmd: 0x67 -- length:      2 -- session key: 0x0001 -- target key:␣
↪0xffff -- second key: 0xffff -- result: 0xe7 -- tick: 697 -- hash:␣
↪2e395d1b706668737e1d2215813db47e
```

31.30.3 Command Line Mode

```
$ yubihsm-shell -a get-logs --out <file> [--authkey <authKeyID> -p <password>]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--out=STRING`

  Log entries.

  Possible Values: Path to file or stdout

  Default Value: stdout

- `--outformat=ENUM`

Output data format.

Possible Values: default, base64, binary, PEM, hex, ASCII

Default Format: ASCII A

Example

```
$ yubihsm-shell -a get-logs
0 unlogged boots found
0 unlogged authentications found
Found 6 items
item:       46 -- cmd: 0x4b -- length:  234 -- session key: 0x0001 -- target key: 0xcf94␣
↪-- second key: 0x997e -- result: 0xcb -- tick: 335725 -- hash:␣
↪415f51f1f035a1b713e730e4464e4033
item:       47 -- cmd: 0x4c -- length:   77 -- session key: 0x0001 -- target key: 0xaff7␣
↪-- second key: 0xffff -- result: 0xcc -- tick: 351714 -- hash:␣
↪5496a60d478c2b9c801d8d32ca66b554
item:       48 -- cmd: 0x00 -- length:      0 -- session key: 0xffff -- target key:␣
↪0x0000 -- second key: 0x0000 -- result: 0x00 -- tick: 0 -- hash:␣
↪14ac7747ba9bbb243cfc70befeb5349b
item:       49 -- cmd: 0x03 -- length:   10 -- session key: 0xffff -- target key: 0x0001␣
↪-- second key: 0xffff -- result: 0x83 -- tick: 139 -- hash:␣
↪b20a8f25c025e693a8e869b433294a20
item:       50 -- cmd: 0x04 -- length:   17 -- session key: 0xffff -- target key: 0x0001␣
↪-- second key: 0xffff -- result: 0x84 -- tick: 139 -- hash:␣
↪ebfae425c319ac7a0afbb8b92597de7c
item:       51 -- cmd: 0x67 -- length:      2 -- session key: 0x0001 -- target key:␣
↪0xffff -- second key: 0xffff -- result: 0xe7 -- tick: 697 -- hash:␣
↪2e395d1b706668737e1d2215813db47e
```

31.30.4 Protocol Details

Command

```
Tc = 0x4d
Lc = 0
Vc = Ø
```

Response

```
Tr = 0xcd
Lr = 2 + 2 + 1 + (N * 32)
Vr = B || O || N || E1 || E2 || ... || EN
```

where –

B = Number of unlogged boot events (if the log buffer is full and audit enforce is set) (2 bytes)

O = Number of unlogged authentication events (if the log buffer is full and audit enforce is set) (2 bytes)

N = Number of elements in the list (1 byte)

`Ei` = Generic log entry composed of:

- Command number (two bytes)
- Command ID (one byte)
- Command length (two bytes)
- ID of the originating session's authentication key (two bytes)
- Target key affected by the command (two bytes)
- Secondary key if the command affected more than one key (two bytes)
- Result of the command on success or an error code if unsuccessful (one byte)
- Systick when the command was processed (4 bytes)
- Digest (16 bytes)

The digest is computed as `trunc(16, SHA256(Ei.Data || trunc(16, Ei-1.Digest)))`. For the initial log entry, a random string of 32 bytes is used, instead of the digest of the previous message.

When the device initializes after a reset, a log entry with all fields set to `0xff` is logged.

When the device boots up, a log entry with all fields set to `0x00` is logged.

31.31 GET OBJECT INFO Command

Fetch all metadata about an *Objects*.

31.31.1 Shell Example

Get Object info for Asymmetric Key with ID `0x1e15`.

```
yubihsm> get objectinfo 0 0x1e15 asymmetric-key
id: 0x1e15, type: asymmetric-key, algorithm: rsa2048, label: "rsakey",
length: 896, domains: 1, sequence: 0, origin: imported, capabilities:
  sign-pkcs
```

31.31.2 Interactive Mode

```
yubihsm> get objectinfo e:session, w:id, t:type
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Value: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **id** Required.

  Object ID of the object to delete. Object ID is a 2 bytes integer. Can be specified in hex or decimal

- **type** Required.

Type of the object to delete.

Possible Values: asymmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key

Example

Get Object info for Asymmetric Key with ID `0x1e15`.

```
yubihsm> get objectinfo 0 0x1e15 asymmetric-key
id: 0x1e15, type: asymmetric-key, algorithm: rsa2048, label: "rsakey", length: 896,␣
↪domains: 1, sequence: 0, origin: imported, capabilities: sign-pkcs
```

31.31.3 Command Line Mode

```
$ yubihsm-shell -a get-object-info -i <id> -t <type> [ --authkey <authKeyID> -p
↪<password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the object to delete. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-t, --object-type=STRING` Required.

  Type of the object to delete.

  Possible Values: asymmetric-key, authentication-key, hmac-key, opaque, otp-aead-key, template, wrap-key

Example

Get Object info for Asymmetric Key with ID `0x1e15`.

```
$ yubihsm-shell -a get-object-info -i 0x1e15 -t asymmetric-key
```

31.31.4 Protocol Details

Command

```
Tc = 0x4e
Lc = 2 + 1
Vc = I || T
```

where –

`I` = *Object ID* (2 bytes)

`T` = Type, *Objects* (1 byte)

Response

```
Tr = 0xce
Lr = 8 + 2 + 2 + 2 + 1 + 1 + 1 + 1 + 40 + 8
Vr = C || I || N || D || T || A || S || O || L || DC
```

where –

`C` = *Capability* (8 bytes)

`I` = *Object ID* (2 bytes)

`N` = Object Length (2 bytes)

`D` = *Domain* (2 bytes)

`T` = Type, *Objects* (1 byte)

`A` = *ALGORITHMS* (1 byte)

`S` = *Sequence* (1 byte)

`O` = Origin (1 byte)

`L` = *Label* (40 bytes)

`DC` = Delegated *Capability* (8 bytes)

31.32 GET OPAQUE Command

Retrieve an Opaque Object (like an X.509 certificate) from the device.

31.32.1 Shell Example

Fetch Opaque Object `0xe255` and store in the file `cert.der`.

```
yubihsm> get opaque 0 0xe255 cert.der
```

31.32.2 Interactive Mode

```
yubihsm> get opaque e:session, w:object_id, F:file=-
```

Parameters

- **Session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **object_id** Required.

  Opaque Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **File**

  Value of Opaque object. Default output format: binary (DER). If object algorithm is opaque-x509-certificate, the output will be an X509Certificate.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

Example

Fetch Opaque Object `0xe255` and store in the file `cert.der`.

```
yubihsm> get opaque 0 0xe255 cert.der
```

31.32.3 Command Line Mode

```
$ yubihsm-shell -a get-opaque -i <object_id> [--out <file> --outformat <format> --
↪authkey <authKeyID> -p <password>]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Opaque Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--out=STRING`

Value of Opaque object. Default output format: binary (DER). If object algorithm is opaque-x509-certificate, the output will be an X509Certificate.

Possible Values: Path to file or stdout

Default Value: stdout

- `--outformat=ENUM`

  Output data format.

  Possible Values: binary, PEM

Example

Fetch Opaque Object `0xe255` and store in the file `cert.pem`.

```
$ yubihsm-shell -a get-opaque -i 0xe255 --out cert.pem
```

31.32.4 Protocol Details

Command

```
Tc = 0x43
Lc = 2
Vc = I
```

where –

`I` = *Object ID* (2 bytes)

Response

```
Tr = 0xc3
Lr = LD
Vr = D
```

where –

`D` = Data

31.33 GET OPTION Command

Get device-global *Options*. Each invocation of this command retrieves a single Option, which is selected by its represented TAG (see *SET OPTION Command*).

31.33.1 Shell Example

```
yubihsm> get option 0 force-audit
Option value is: 00
```

31.33.2 Interactive Mode

```
yubihsm> get option e:session, o:option
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **Option** Required.

  Device option. fips-mode option is only applicable in FIPS compatible YubiHSMs.

  Possible Values: algorithm-toggle, command-audit, force-audit, fips-mode

Example

```
yubihsm> get option 0 force-audit
2Option value is: 00
```

31.33.3 Command Line Mode

```
$ yubihsm-shell -a get-option --opt-name <option> [ --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--opt-name=STRING` Required.

  Device option name. `fips-mode` option is only applicable in FIPS compatible YubiHSMs.

  Possible Values: algorithm-toggle, command-audit, force-audit, fips-mode

Example

```
$ yubihsm-shell -a get-option --opt-name force-audit
Option value is: 00
```

31.33.4 Protocol Details

Command

```
Tc = 0x50
Lc = 1
Vc = T
```

where –

T = The tag of the selected option (1 byte)

Response

```
Tr = 0xd0
Lr = LO
Vr = O
```

where –

O = The option-specific value (LO bytes)

31.34 GET PSEUDO RANDOM Command

Extract a fixed number of pseudo-random bytes from the device, using the internal PRNG.

31.34.1 Shell Example

```
yubihsm> get random 0 16
bd50979da2d1bca13d8d735abf419556
```

31.34.2 Interactive Mode

```
yubihsm> get random e:session, w:count, F:out=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **Count** Required.

  Number of bytes to request.

- **File**

  Pseudo random number.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

  Default Output Format: hexf

Example

```
yubihsm> get random 0 16
bd50979da2d1bca13d8d735abf419556
```

31.34.3 Command Line Mode

```
$ yubihsm-shell -a get-pseudo-random [--count <count> --out <file>  --authkey <authKeyID>
↪ -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `--p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--count=INT`

  Number of bytes to request

  Default Value: 256

- `--out=STRING`

  Pseudo random number.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Default Output Format: hex

- `--outformat=ENUM`

  Output data format

  Possible Values: base64, binary, PEM, hex

Example

```
$ yubihsm-shell -a get-pseudo-random --count=16
81a0060782bc2386cdf7df597035c6d1
```

31.34.4 Protocol Details

Command

```
Tc = 0x51
Lc = 2
Vc = B
```

where –

B = Number of pseudo-random bytes to extract (2 bytes)

Response

```
Tr = 0xd1
Lr = B
Vr = R
```

where –

R = Random data (B bytes)

31.35 GET PUBLIC KEY Command

Fetch the public key of an object. With YubiHSM firmware version prior to 2.4, only public key of Asymmetric Key are returned. With firmware version 2.4 or later, public keys of RSA Wrap Keys can also be returned.

31.35.1 Interactive Mode

```
yubihsm> get pubkey e:session, w:key_id, t:key_type=asymmetric-key, F:file=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Asymmetric key Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **key_type**

  Object type.

  Possible Values: asymmetric-key or wrap-key

  Default Value: asymmetric-key

- **File**

  Public key.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

  Default Format: PEM

Example

Fetch the public key of RSA Wrap Key `0x2846`.

```
yubihsm> get pubkey 0 0x2846 wrap-key
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE85fayPHTvCrv0RRcyCsHv0hTKAM7
xHiU2I3NgO61RTRQumGDeBnQZIITykK/0PWKLGDANfBVrmKkWWxB47ze9A==
-----END PUBLIC KEY-----
```

31.35.2 Command Line Mode

```
$ yubihsm-shell -a get-public-key -i <key_id> [ -t <key_type> --out <file> --outformat
↪<format>  --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

Object ID of an asymmetric key. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-t, --object-type=STRING`

  Object type.

  Possible Values: asymmetric-key or wrap-key

  Default Value: asymmetric-key

- `--out=STRING`

  Public key.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Default Format: PEM

- `--outformat=ENUM`

  Output data format.

  Possible Values: binary, PEM

Example

Fetch the public key of RSA Wrap Key `0x2846`.

```
$ yubihsm-shell -a get-public-key -i 0x2846 -t wrap-key
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAE85fayPHTvCrv0RRcyCsHv0hTKAM7
xHiU2I3NgO61RTRQumGDeBnQZIITykK/0PWKLGDANfBVrmKkWWxB47ze9A==
-----END PUBLIC KEY-----
```

31.35.3 Protocol Details

Command

```
Tc = 0x54
Lc = 2 | /{ + 1 /}
Vc = I /{ || T /}
```

where –

`I` = *Object ID* (2 bytes)

`T` = Type, *Objects* (1 byte)

Response

```
Tr = 0xd4
Lr = 1 + LP1 /{ + LP2 /}
Vr = A || P1 /{ || P2 /}
```

where –

AA = *ALGORITHMS*

P1 =

- For RSA: Public modulus N (256, 384 or 512 bytes)
- For ECC: Public point X (32, 48, 64 or 66 bytes)
- For EDC: Public point A, compressed (32 bytes)

P2 =

- For RSA: NOT DEFINED
- For ECC: Public point Y (32, 48, 64 or 66 bytes)
- For EDC: NOT DEFINED

31.36 GET STORAGE INFO Command

Report currently free storage. This is reported as currently free records, free pages and page size. Each object takes a record slot and will use as many pages as needed.

31.36.1 Shell Example

```
yubihsm> get storage 0
free records: 255/256, free pages: 1023/1024 page size: 126 bytes
```

31.36.2 Interactive Mode

```
yubihsm> get storage e:session
```

Parameters

session Required,

The ID of the authenticated session to send the command over.

Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

```
yubihsm> get storage 0
free records: 255/256, free pages: 1023/1024 page size: 126 bytes
```

31.36.3 Command Line Mode

This command is not available in command line mode.

31.36.4 Protocol Details

Command

```
Tc = 0x41
Lc = 0
Vc = Ø
```

Response

```
Tr = 0xc1
Lr = 10
Vr = Rtotal || Rfree || Ptotal || Pfree || S
```

where –

`Rtotal` = Total number of records (2 bytes)

`Rfree` = Currently free storage records (2 bytes)

`Ptotal` = Total number of pages (2 bytes)

`Pfree` = Currently free storage pages (2 bytes)

S = Page size in bytes (2 bytes)

31.37 GET TEMPLATE Command

Retrieve a Template Object from the device.

31.37.1 Shell Example

Fetch Template Object `0x7b19` and store in the file `template.dat`.

```
yubihsm> get template 0 0x7b19 template.dat
```

31.37.2 Interactive Mode

```
yubihsm> get template e:session, w:object_id, F:out=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **object_id** Required.

  Object ID of a template object. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **File**

  Template value.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

Example

Fetch Template Object `0x7b19` and store in the file `template.dat`.

```
yubihsm> get template 0 0x7b19 template.dat
```

31.37.3 Command Line Mode

```
$ yubihsm-shell -a get-template -i <object_id> [--out <file> --outformat <format>  --
↪authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of a template object. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--out=STRING`

Template value.

Possible Values: Path to file or stdout

Default Value: stdout

- `--outformat=ENUM`

  Output data format.

  Possible Values: base64, hex, PEM

Example

Fetch the public key of Asymmetric Key `0x2846`.

```
$ yubihsm-shell -a get-template -i 0x7b19 --out template.dat
```

31.37.4 Protocol Details

Command

```
Tc = 0x5f
Lc = 2
Vc = I
```

where –

`I` = *Object ID* of the Template to retrieve (2 bytes)

Response

```
Tr = 0xdf
Lr = LD
Vr = D
```

where –

`D` = Data

31.38 IMPORT WRAPPED Command

Import a wrapped/encrypted Object that was previously exported by an YubiHSM 2 device into the device. The imported object will retain its metadata (Object ID, Domains, Capabilities, etc), however, the object's origin will be marked as *imported* instead of *generated*.

31.38.1 Shell Example

Import the Object stored in `key.enc` and unwrap it using Wrap Key `0xcf94`.

```
yubihsm> put wrapped 0 0xcf94 key.enc
Object imported as 0x997e of type asymmetric
```

31.38.2 Interactive Mode

```
yubihsm> put wrapped e:session, w:wrapkey_id, i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **wrapkey_id** Required.

  Object ID of the wrap key to decrypt/unwrap the data. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **File**

  Encrypted/wrapped object.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Default Format: base64

Example

Import the Object stored in `key.enc` and unwrap it using Wrap Key `0xcf94`.

```
yubihsm> put wrapped 0 0xcf94 key.enc
Object imported as 0x997e of type asymmetric
```

31.38.3 Command Line Mode

```
$ yubihsm-shell -a put-wrapped --wrap-id <wrapkey_id> [--in <file> --authkey <authKeyID>␣
↪-p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-p, --password=STRING`

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--wrap-id=INT` Required.

  Object ID of the wrap key to decrypt/unwrap the data. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--in=STRING`

  Encrypted/wrapped object.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Format: base64

Example

Import the Object stored in `key.enc` and unwrap it using Wrap Key `0xcf94`.

```
$ yubihsm-shell -a put-wrapped --wrap-id 0xcf94 --in key.enc
```

31.38.4 Protocol Details

Command

```
Tc = 0x4b
Lc = 2 + 13 + LO
Vc = I || N || O
```

where –

`I` = *Object ID* of the Wrap Key (2 bytes)

`N` = Nonce associated with this wrapped Object (13 bytes)

`O` = Wrapped *Objects* (Length dependant on Object)

Response

```
Tc = 0xcb
Lc = 3
Vc = T || I
```

where –

T = Type, *Objects* of imported Object (1 byte)

I = *Object ID* of imported Object (2 bytes)

31.39 IMPORT RSA WRAPPED Command

Available on YubiHSM devices with firmware version 2.4 or higher.

Import a wrapped/encrypted Object that was previously exported by an YubiHSM 2 device into the device. The imported object will retain its metadata (Object ID, Domains, Capabilities, etc), however, the object's origin will be marked as *imported* instead of *generated*.

31.39.1 Interactive Mode

```
yubihsm> put rsa_wrapped e:session, w:wrapkey_id, a:hash=rsa-oaep-sha256, a:mgf1=mgf1-
↪sha256, i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **wrapkey_id** Required.

  Object ID of the wrap key to decrypt/unwrap the data. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **File**

  Encrypted/wrapped object.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Format: Binary

- **hash**

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- **mgf1**

Hash algorithm to use for MGF1.

Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

Default Value: mgf1-sha256

Example

Import the Object stored in `object.enc` and unwrap it using Wrap Key `0xcf94`.

```
yubihsm> put rsa_wrapped 0 0xcf94 rsa-oaep-sha384 mgf1-sha1 object.enc
Object imported as 0x997e of type asymmetric
```

31.39.2 Command Line Mode

```
$ yubihsm-shell -a put-rsa-wrapped --wrap-id <wrapkey_id> [ --oaep <oaep> --mgf1 <mgf1> -
↪-in <file> --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-p, --password=STRING`

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--wrap-id=INT` Required.

  Object ID of the wrap key to decrypt/unwrap the data. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--in=STRING`

  Encrypted/wrapped object.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Format: Binary

- `--oaep=STRING`

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- `--mgf1=STRING`

  Hash algorithm to use for MGF1.

  Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

  Default Value: mgf1-sha256

Example

Import the Object stored in `object.enc` and unwrap it using Wrap Key `0xcf94`.

```
$ yubihsm-shell -a put-rsa-wrapped --wrap-id 0xcf94 --oaep rsa-oaep-sha384 --mgf1 mgf1-
↪sha1 --in object.enc
```

31.39.3 Protocol Details

Command

```
Tc = 0x77
Lc = 2 + 1 + 1 + Lw + LHL
Vc = I || H || M || W || HL
```

where –

`I` = *Object ID* of the Wrap Key (2 bytes)

`H` = *ALGORITHMS* to use for OAEP label (1 byte)

`M` = *ALGORITHMS* to use for MGF1 (1 byte)

`W` = The wrapped object (Length dependent on object)

`LH` = The label digest (Length dependent on OAEP algorithm)

Response

```
Tc = 0xf7
Lc = 3
Vc = T || I
```

where –

`T` = Type, *Objects* of imported Object (1 byte)

`I` = *Object ID* of imported Object (2 bytes)

31.40 IMPORT RSA WRAPPED KEY Command

Available on YubiHSM devices with firmware version 2.4 or higher.

Import a wrapped/encrypted an (a)symmetric key object. Only asymmetric and symmetric key objects are valid targets. Asymmetric keys are expected to have been serialized as PKCS#8. Since the object properties are not part of the wrapped object, they must be provided separately.

31.40.1 Interactive Mode

```
yubihsm> put rsa_wrapped_key e:session, w:wrapkey_id, t:type, w:key_id, a:algorithm,␣
↪s:label, d:domains, c:capabilities, a:hash=rsa-oaep-sha256, a:mgf1=mgf1-sha256,␣
↪i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **wrapkey_id** Required.

  Object ID of the wrap key to decrypt/unwrap the data. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **type** Required.

  Type of the object to import.

  Possible Values: asymmetric-key or symmetric-key

- **key_id** Required.

  Object ID of the imported key. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **algorithm** Required.

  Algorithm of the imported key.

  Possible values:

  For RSA keys: rsa2048, rsa3072, rsa4096

  For EC keys: eck256, ecp224, ecp256, ecp384, ecp521, ecbp256, ecbp384, ecbp512

  For ED keys: ed25519

  For AES keys: aes128, aes192, aes256

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **data**

  Encrypted/wrapped object.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Format: Binary

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by `,` or `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between.

  Possible Values: none, all, decrypt-oaep, decrypt-pkcs, derive-ecdh, exportable-under-wrap, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-pkcs, sign-pss, sign-ssh-certificate

- **hash**

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- **mgf1**

  Hash algorithm to use for MGF1.

  Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

  Default Value: mgf1-sha256

Example

Import the Object stored in `key.enc` and unwrap it using Wrap Key `0xcf94`.

```
yubihsm> put rsa_wrapped_key 0xcf94 asymmetric-key 0 rsa2048 "" 1,2,3 sign-pkcs,
↪exportable-under-wrap rsa-oaep-sha384 mgf1-sha1 key.enc
Object imported as 0x97df of type asymmetric
```

31.40.2 Command Line Mode

```
$ yubihsm-shell -a put-rsa-wrapped-key --wrap-id <wrapkey_id> -t <key_type> -A <key_
↪algorithm> -d <domains> [ -i <key_id> -l <key_label> -c <capabilities> --oaep <oaep> --
↪mgf1 <mgf1> --in <file> --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-p, --password=STRING`

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--wrap-id=INT` Required.

  Object ID of the wrap key to decrypt/unwrap the data. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--in=STRING`

Encrypted/wrapped object.

Possible Values: Path to file or stdin

Default Value: stdin

Format: Binary

- `-t, --object-type=STRING` Required.

  Type of the object to import.

  Possible Values: asymmetric-key or symmetric-key

- `-i, --object-id=SHORT`

  Object ID of the imported key. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-A, --algorithm=STRING` Required.

  Algorithm of the imported key.

  Possible values:

  For RSA keys: rsa2048, rsa3072, rsa4096

  For EC keys: eck256, ecp224, ecp256, ecp384, ecp521, ecbp256, ecbp384, ecbp512

  For ED keys: ed25519

  For AES keys: aes128, aes192, aes256

- `-l, --label=STRING`

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by `,` or `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING`

  Capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between.

  Possible Values: none, all, decrypt-oaep, decrypt-pkcs, derive-ecdh, exportable-under-wrap, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-pkcs, sign-pss, sign-ssh-certificate

- `--oaep=STRING`

  Hash algorithm to use for OAEP label.

  Possible Values: rsa-oaep-sha1, rsa-oaep-sha256, rsa-oaep-sha384 or rsa-oaep-sha512

  Default Value: rsa-oaep-sha256

- `--mgf1=STRING`

  Hash algorithm to use for MGF1.

  Possible Values: mgf1-sha1, mgf1-sha256, mgf1-sha384 or mgf1-sha512

  Default Value: mgf1-sha256

Example

Import the Object stored in `key.enc` and unwrap it using Wrap Key `0xcf94`.

```
$ yubihsm-shell -a put-rsa-wrapped-key --wrap-id 0xcf94 -t asymmetric-key -d 1,2,3 -A␣
↪rsa2048 --oaep rsa-oaep-sha384 --mgf1 mgf1-sha1 --in key.enc
```

31.40.3 Protocol Details

Command

```
Tc = 0x75
Lc = 2 + 1 + 2 + 40 + 2 + 8 + 1 + 1 + 1 + Lw + LHL
Vc = I || To || Ti || L || D || C || A || H || M || W || HL
```

where –

`I` = *Object ID* of the Wrap Key (2 bytes)

`To` = *Objects* of the imported key (1 byte)

`Ti` = *Object ID* of the imported key (2 bytes)

`L` = *Label* of the imported key (40 bytes)

`D` = *Domain* of the imported key (2 bytes)

`C` = *Capability* of the imported key (8 bytes)

`A` = *ALGORITHMS* of the imported key (1 byte)

`H` = *ALGORITHMS* to use for OAEP label (1 byte)

`M` = *ALGORITHMS* to use for MGF1 (1 byte)

`W` = The wrapped object (Length dependent on object)

`LH` = The label digest (Length dependent on OAEP algorithm)

Response

```
Tc = 0xf5
Lc = 3
Vc = T || I
```

where –

`T` = Type, *Objects* of imported Object (1 byte)

`I` = *Object ID* of imported Object (2 bytes)

31.41 LIST OBJECTS Command

Get a filtered list of *Objects* from the device.

31.41.1 Shell Example

Get a list of all Asymmetric Keys for Session 0.

```
yubihsm> list objects 0 0 asymmetric-key
Found 4 object(s)
id: 0x3479, type: asymmetric-key, sequence: 0
id: 0x7df6, type: asymmetric-key, sequence: 0
id: 0x9602, type: asymmetric-key, sequence: 0
id: 0xd6cd, type: asymmetric-key, sequence: 0
```

31.41.2 Interactive Mode

```
yubihsm> list objects e:session, w:id=0, t:type=any, d:domains=0, c:capabilities=0,␣
↪a:algorithm=any, s:label=
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **Id**

  Object ID. `0` returns all Object IDs. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 0

- **Type**

  Object type. `any` returns all types>

  Possible Values: any, opaque, authentication-key, asymmetric-key, wrap-key, hmac-key, template, otp-aead-key

  Default Value; any

- **domains**

  Domains where the key will be accessible. `all` returns all domains.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

  Default Value: all

- **capabilities**

  Capabilities of the key. `all` returns all capabilities.

  Possible Values: all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap,

generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

Default Value: all

- **Algorithm**

  Key algorithm. `any` returns all algorithms.

  Possible Values: any, rsa2048, rsa3072, rsa4096, ecp256, ecp384, ecp521, eck256, ecbp256, ecbp384, ecbp512, ed25519, ecp224, hmac-sha1, hmac-sha256, hmac-sha384, hmac-sha512, aes128-ccm-wrap, opaque-data, opaque-x509-certificate, aes128-yubico-otp, aes128-yubico-authentication, aes192-yubico-otp, aes256-yubico-otp, aes192-ccm-wrap, aes256-ccm-wrap

  Default Value: any

- **Label**

  Object label. Empty value means all labels.

  Possible Values: Maximum of 40 characters string.

  Default Value: Empty

Example

Get a list of all Asymmetric Keys for Session `0`.

```
yubihsm> list objects 0 0 asymmetric-key
Found 4 object(s)
id: 0x3479, type: asymmetric-key, sequence: 0
id: 0x7df6, type: asymmetric-key, sequence: 0
id: 0x9602, type: asymmetric-key, sequence: 0
id: 0xd6cd, type: asymmetric-key, sequence: 0
```

31.41.3 Command Line Mode

```
$ yubihsm-shell -a list-objects -t <type> -A <algorithm> [-i <key_id> -l <label> -d
↪<domains> -c <capabilities>  --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT`

  Object ID. `0` returns all Object IDs. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 0

- `-t, --object-type=STRING` Required.

  Object type. Use `any` to return all types.

  Possible Values: any, opaque, authentication-key, asymmetric-key, wrap-key, hmac-key, template, otp-aead-key

- `-l, --label=STRING`

  Object label.

  Possible Values: Maximum of 40 characters string

  Default Value: Empty

- `-d, --domains=STRING`

  Domains where the key will be accessible.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

  Default Value: all

- `-c, --capabilities=STRING`

  Capabilities of the key.

  Possible Values: all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

  Default Value: all

- `-A, --algorithm=STRING` Required.

  Key algorithm. Use `any` to return all algorithms.

  Possible Values: any, rsa2048, rsa3072, rsa4096, ecp256, ecp384, ecp521, eck256, ecbp256, ecbp384, ecbp512, ed25519, ecp224, hmac-sha1, hmac-sha256, hmac-sha384, hmac-sha512, aes128-ccm-wrap, opaque-data, opaque-x509-certificate, aes128-yubico-otp, aes128-yubico-authentication, aes192-yubico-otp, aes256-yubico-otp, aes192-ccm-wrap, aes256-ccm-wrap

Example

Generate a new key using `secp256r1` in the device.

```
$ yubihsm-shell -a list-objects -t any -A any
Found 4 object(s)
id: 0x3479, type: asymmetric-key, sequence: 0
id: 0x7df6, type: asymmetric-key, sequence: 0
id: 0x9602, type: asymmetric-key, sequence: 0
id: 0xd6cd, type: asymmetric-key, sequence: 0
```

31.41.4 Protocol Details

Command

```
Tc = 0x48
Lc = LF
Vc = F
```

where –

`F` = List of Tag-Value pairs describing a filter to apply. Possible tags to use for filtering are described in the table below.

Name	Identifier	Length
ID, *Object ID*	0x01	2 bytes
TYPE, *Objects*	0x02	1 byte
Domain	0x03	2 bytes
Capability	0x04	8 bytes
ALGORITHMS	0x05	1 byte
Label	0x06	40 bytes

Response

```
Tr = 0xc8
Lr = 4 * N
Vr = R1 || R2 || ... || RN
```

where –

`Ri` = *Object ID* (2 bytes), Type, *Objects* (1 byte) and *Sequence* (1 byte).

31.42 PUT ASYMMETRIC KEY Command

Import an Asymmetric Key into the device.

31.42.1 Shell Example

Store an RSA key from `key.pem` into the device.

```
yubihsm> put asymmetric 0 0 rsakey 1 sign-pkcs key.pem
Stored Asymmetric key 0x1e15
```

31.42.2 Protocol Details

Command

```
Tc = 0x45
Lc = 2 + 40 + 2 + 8 + 1 + LP1 /{ + LP2 /}
Vc = I || L || D || C || A || P1 /{ || P2 /}
```

The key parameters vary according to the chosen algorithm. Each parameter has a fixed length and the order is compulsory.

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`P1` =

For RSA: secret prime p (128, 192 or 256 bytes)

For ECC: private key integer d (32, 48, 64 or 66 bytes)

For EDC: private key integer k (32 bytes)

`P2` =

For RSA: secret prime q (128, 192 or 256 bytes)

For ECC: NOT DEFINED

For EDC: NOT DEFINED

Response

```
Tr = 0xc5
Lr = 2
Vr = I
```

where –

`I` = ID of created Object (2 bytes)

31.43 PUT ASYMMETRIC AUTHENTICATION KEY Command

Available with firmware version 2.3.1 or later.

Store an Asymmetric Authentication Key in the device.

31.43.1 Interactive Mode

```
yubihsm> put authkey_asym e:session,w:key_id,s:label,d:domains,c:capabilities,
↪c:delegated_capabilities,i:pubkey=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use '0' to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by `,` or `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque,

delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- **delegated_capabilities** Required.

  Delegated capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data. Use `all` to include all capabilities.

- **pubkey**

  The public key of the clien. When using stdin, click CTRL-D to mark end of input. Input format for a password string is `password`. If `password` format is used, the tool will derive an ec-p256 private key from the input string and calculate the public key from that. The private key is not used for anything else.

  Possible Values: File containing the client's public key as an uncompressed ec-p256 public key, password or - for stdin

  Default Value: stdin

  Default format: PEM

  Possible format for public key file: PEM, HEX, binary.

Example

Store a new Asymmetric Authentication Key using a client's public key:

```
yubihsm> put authkey_asym 0 0 asym_authkey 1,2,3 generate-asymmetric-key,sign-pkcs sign-
↪pkcs
-----BEGIN PUBLIC KEY-----
MFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEKIfzuX9uJ2gsNgXcFYtNkP30aBp+
e0f9mhpy+lQbvbbD72y5HiMIkbNkqBXHOwSPu/suD/f1BoN8xcP4FHk4iw==
-----END PUBLIC KEY-----
Stored Authentication key 0xe599
```

31.43.2 Command Line Mode

Asymmetric authentication keys cannot be added using the command line.

31.43.3 Protocol Details

Command

```
Tc = 0x44
Lc = 2 + 40 + 2 + 8 + 1 + 8 + 64
Vc = I || L || D || C || A || DC || Key
```

where –

`I` = Object ID of the Authentication Key (2 bytes)

`L` = Label (40 bytes)

`D` = Domains (2 bytes)

`C` = Capabilities (8 bytes)

`A` = Algorithm (1 bytes)

`DC` = Delegated Capabilities (8 bytes)

`Key` = Uncompressed EC-P256 public key (64 bytes)

Response

```
Tr = 0xc4
Lr = 2
Vr = I
```

where –

`I` = Object ID of created Authentication Key (2 bytes)

Note: This command will return ERROR_INV_DATA if Key is not a valid EC-P256 key.

31.44 PUT AUTHENTICATION KEY Command

Store an Authentication Key in the device.

31.44.1 Shell Example

Store a new Authentication Key derived from the password `newpassword`.

```
yubihsm> put authkey 0 0 authkey 1 generate-asymmetric-key,sign-pkcs
  sign-pkcs newpassword
Stored Authentication key 0xbb72
```

31.44.2 Interactive Mode

```
yubihsm> put authkey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪c:delegated_capabilities, i:password=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data Use `all` to include all capabilities.

- **delegated_capabilities** Required.

  Delegated capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data. Use `all` to include all capabilities.

- **password**

  The password used to derive the session keys from this authentication key.

  Possible Values: The password or - for stdin

  Default Value: stdin

  Input Format: password

Example

Store a new Authentication Key derived from the password `newpassword`.

```
yubihsm> put authkey 0 0 authkey 1 generate-asymmetric-key,sign-pkcs sign-pkcs␣
↪newpassword
Stored Authentication key 0xbb72
```

31.44.3 Command Line Mode

```
$ yubihsm-shell -a put-authentication-key -i <key_id> -l <label> -d <domains> -c
↪<capabilities> --delegated <delegated_capabilities> [--new-password <new_authkey_
↪password> --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the asymmetric key. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string.

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Value: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- `--delegated=STRING` Required.

  Delegated capabilities of the key. Use `all` to include all delegated capabilities. Use `none` to include no delegated capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- `--new-password=STRING`

  The password used to derive the session keys from this authentication key.

  Possible Values: The password or stdin

  Default ValueL: stdin

  Input Format: password

Example

Fetch the public key of Asymmetric Key `0x2846`.

```
$ yubihsm-shell -a put-authentication-key -i 0 -l authkey -d 1 -c generate-asymmetric-
↪key,sign-pkcs --delegated sign-pkcs --new-password newpassword
Stored Authentication key 0xbb72
```

31.44.4 Protocol Details

Command

```
Tc = 0x44
Lc = 2 + 40 + 2 + 8 + 1 + 8 + 16 + 16
Vc = I || L || D || C || A || DC || Ke || Km
```

where –

`I` = *Object ID* of the Authentication Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`DC` = Delegated *Capability* (8 bytes)

`Ke` = Encryption Key (16 bytes)

`Km` = Mac Key (16 bytes)

Response

```
Tr = 0xc4
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of created Authentication Key (2 bytes)

31.45 PUT HMAC KEY Command

Store an HMAC Key in the device.

31.45.1 Shell Example

Store an HMAC Key with the binary value `666f6f` in the device.

```
yubihsm> put hmackey 0 0 hmackey 1 sign-hmac, verify-hmac hmac-sha256 666f6f
Stored HMAC key 0x7cf2
```

31.45.2 Interactive Mode

```
yubihsm> put hmackey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪a:algorithm, i:key
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, sign-hmac, verify-hmac, exportable-under-wrap

- **Algorithm** Required.

  Key algorithm.

  Possible Values: hmac-sha1, hmac-sha256, hmac-sha384, hmac-sha512

- **key** Required.

  The HMAC key.

  Format: hex

Example

Store an HMAC Key with the binary value `666f6f` in the device.

```
yubihsm> put hmackey 0 0 hmackey 1 sign-hmac, verify-hmac hmac-sha256 666f6f
Stored HMAC key 0x7cf2
```

31.45.3 Command Line Mode

This command is not available in command line mode.

31.45.4 Protocol Details

Command

```
Tc = 0x52
Lc = 2 + 40 + 2 + 8 + 1 + LP
Vc = I || L || D || C || A || P
```

where –

`I` = *Object ID* of the HMAC Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`P` = Key (Minimum 1 byte)

For HMAC-SHA1 and HMAC-SHA256: maximum 64 bytes

For HMAC-SHA384 and HMAC-SHA512: maximum 128 bytes

Response

```
Tr = 0xd2
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of created HMAC Key (2 bytes)

31.46 PUT OPAQUE Command

Stores Opaque data (like an X.509 certificate) in the device. The size of the object is currently limited to what will fit into one message to the YubiHSM 2 (2028 bytes, including the headers).

31.46.1 Shell Example

Store the certificate in file `cert.der` in the device.

```
yubihsm> put opaque 0 0 certificate 1 none opaque-x509-certificate cert.der
Stored Opaque object 0xe255
```

31.46.2 Interactive Mode

```
yubihsm> put opaque e:session, w:object_id, s:label, d:domains, c:capabilities,␣
↪a:algorithm, i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Object label. Can be empty.

  Possible Values: Maximum of 40 characters string.

- **domains** Required.

  Domains where the object will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **Capabilities** Required.

  Capabilities of the data

  Possible Values: none, exportable-under-wrap

- **Algorithm** Required.

  Key algorithm. If `opaque-x509-certificate`, the value of the object will be treated as an X509Certificate.

  Possible Values: opaque-data, opaque-x509-certificate

- **data**

  Opaque data value (e.g. X509Certificate).

  Possible Values: Path to file or `-` for stdin

  Defaul Value: stdin

  Default Format: binary (DER).

Example

Store the certificate in file `cert.pem` in the device.

```
yubihsm> put opaque 0 0 certificate 1 none opaque-x509-certificate cert.pem
Stored Opaque object 0xe255
```

31.46.3 Command Line Mode

```
$ yubihsm-shell -a put-opaque -i <key_id> -l <label> -d <domains> -c <capabilities> [--
↪in <key> --informat <informat> --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the asymmetric key. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Object label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING`

  Domains where the opaque object will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING`

  Capabilities of the key.

  Possible Values: none, exportable-under-wrap

  Default Value: none

- `--in=STRING`

  Opaque data value (e.g. X509Certificate).

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Format: binary (DER)

- `--informat=ENUM`

  Input data format

  Possible Values: PEM, binary

Example

Store the certificate in file `cert.der` in the device.

```
$ yubihsm-shell -a put-opaque -i 0 -l certificate -d 1 -A opaque-x509-certificate --in␣
↪cert.der
```

31.46.4 Protocol Details

Command

```
Tc = 0x42
Lc = 2 + 40 + 2 + 8 + 1 + LO
Vc = I || L || D || C || A || O
```

where –

`I` = *Object ID* (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`O` = Opaque data

Response

```
Tr = 0xc2
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of created Opaque Object (2 bytes)

31.47 PUT OTP AEAD KEY Command

Import an OTP AEAD Key used for Yubico OTP Decryption.

31.47.1 Shell Example

Import OTP AEAD Key with Nonce ID `0x01020304` and key value `000102030405060708090a0b0c0d0e0f` (AES-128).

```
yubihsm> put otpaeadkey 0 0 otpaeadkey 1 decrypt-otp 0x01020304␣
↪000102030405060708090a0b0c0d0e0f
Stored OTP AEAD key 0xe34f
```

31.47.2 Interactive Mode

```
yubihsm> put otpaeadkey e:session, w:key_id, s:label, d:domains, c:capabilities, u:nonce_
↪id, i:key
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **Capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, decrypt-otp, create-otp-aead, randomize-otp-aead, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, exportable-under-wrap

- **nonce_id** Required.

  OTP nonce. 4 bytes.

- **key** Required.

The AEAD key.

Format: hex

Example

Import OTP AEAD Key with Nonce ID `0x01020304` and key value `000102030405060708090a0b0c0d0e0f` (AES-128).

```
yubihsm> put otpaeadkey 0 0 otpaeadkey 1 decrypt-otp 0x01020304␣
↪000102030405060708090a0b0c0d0e0f
Stored OTP AEAD key 0xe34f
```

31.47.3 Command Line Mode

This command is not available in command line mode.

31.47.4 Protocol Details

Command

```
Tc = 0x65
Lc = 2 + 40 + 2 + 8 + 1 + 4 + LK
Vc = I || L || D || C || A || N || K
```

where –

`I` = *Object ID* (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`N` = Nonce ID (4 bytes)

`K` = Key (16, 24 or 32 bytes depending on algorithm)

Response

```
Tr = 0xe5
Lr = 2
Vr = I
```

where –

`I` = ID of created OTP AEAD Key (2 bytes)

31.48 PUT SYMMETRIC KEY Command

Available with firmware version 2.3.1 or later.

Import a symmetric Key into the device.

31.48.1 Interactive Mode

```
yubihsm> put symmetric e:session,w:key_id,s:label,d:domains,c:capabilities,a:algorithm,
↪i:key
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use '0' to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by `,` or `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between.

  Possible Values: none, encrypt-ecb, decrypt-ecb, encrypt-cbc, decrypt-cbc, exportable-under-wrap

- **algorithm** Required.

  Key algorithm.

  Possible Values: aes128, aes192,aes256

- **key** Required.

  Symmetric key.

  Possible Values: Value of the symmetric key

  Input format: HEX

Example

Store an AES128 key into the device:

```
yubihsm> put symmetric 0 0 aeskey 1 encrypt-cbc,decrypt-cbc aes128␣
↪0a8a7ecc862b3d42b5dc127c111da0f4
Stored symmetric key 0x71c9
```

31.48.2 Command Line Mode

```
$ yubihsm-shell -a put-symmetric-key -i <key_id> -l <label> -d <domains> -c
↪<capabilities> -A <algorithm> --in <key> [--authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password will be prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the symmetric key. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal

- `-l, --label=STRING` Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string.

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by `,` or `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by `,` or `:` with no spaces between.

  Possible Values: none, encrypt-ecb, decrypt-ecb, encrypt-cbc, decrypt-cbc,exportable-under-wrap

- `-A, --algorithm=STRING` Required.

  Key algorithm.

  Possible Values: aes128, aes192, aes256

- `--in=STRING`

Symmetric key.

Possible Values: Value of the symmetric key

Input format: HEX

Example

Store an AES128 key into the device:

```
$ yubihsm-shell -a put-symmetric-key -l aeskey -d 1 -c encrypt-cbc, decrypt-cbc -A␣
↪aes128 --in 0a8a7ecc862b3d42b5dc127c111da0f4
```

31.48.3 Protocol Details

Command

```
Tc = 0x6d
Lc = 2 + 40 + 2 + 8 + 1 + Lk
Vc = I || L || D || C || A || K
```

The key parameters vary according to the chosen algorithm. Each parameter has a fixed length and the order is compulsory.

where –

`I` = *Object ID* of the symmetric Key (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`K` = The key value (16, 24 or 32 bytes)

Response

```
Tr = 0xed
Lr = 2
Vr = I
```

where –

`I` = ID of created Object (2 bytes)

31.49 PUT TEMPLATE Command

Stores a Template in the device. The size of the object is currently limited to what will fit into one message to the YubiHSM (2021 bytes, including the headers).

31.49.1 Shell Example

Store the SSH Template in file `template.dat` in the device.

```
yubihsm> put template 0 0 ssh_template 1 none template-ssh template.dat
Stored Template object 0x7b19
```

31.49.2 Interactive Mode

```
yubihsm> put template e:session, w:object_id, s:label, d:domains, c:capabilities,␣
↪a:algorithm, i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Object label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the object will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **Capabilities** Required.

  Capabilities of the data.

  Possible Values: none, exportable-under-wrap

- **Algorithm** Required.

  Key algorithm.

  Possible Values: template-ssh

- **data**

Template value.

Possible Values: Path to file or - for stdin

Default Value: stdin

Default Format: base64

Example

Store the SSH Template in file `template.dat` in the device.

```
yubihsm> put template 0 0 ssh_template 1 none template-ssh template.dat
Stored Template object 0x7b19
```

31.49.3 Command Line Mode

```
$ yubihsm-shell -a put-template -i <key_id> -l <label> -d <domains> -c <capabilities> [--
↪in <key> --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session.The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Object label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING` Required.

  Domains where the opaque object will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING`

  Capabilities of the key.

  Possible Values: none, exportable-under-wrap

  Default Value: none

- `--in=STRING`

Template value.

Possible Values: Path to file or stdin

Default Value: stdin

Default Format: base64

Example

Store the SSH Template in file `template.dat` in the device.

```
$ yubihsm-shell -a put-template -i 0 -l ssh_template -d 1 -c none -A template-ssh --in␣
↪template.dat
```

31.49.4 Protocol Details

Command

```
Tc = 0x5e
Lc = 2 + 40 + 2 + 8 + 1 + LD
Vc = I || L || D || C || A || D
```

where –

`I` = *Object ID* of the Template (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`D` = Template data

Response

```
Tr = 0xde
Lr = 2
Vr = I
```

where –

`I` = *Object ID* of created Template (2 bytes)

31.50 PUT WRAP KEY Command

Import a key for wrapping into the device.

31.50.1 Interactive Mode

Use `put wrapkey` command to import AES Wrap Key and `put rsa_wrapkey` command to import RSA Wrap Key

```
yubihsm> put wrapkey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪c:delegated_capabilities, i:key
yubihsm> put rsa_wrapkey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪c:delegated_capabilities, i:key
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, wrap-data, unwrap-data, export-wrapped, import-wrapped, exportable-under-wrap

- **delegated_capabilities** Required.

  Delegated capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key,

set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- **key** Required.

  The wrap key.

  Default Format: hex

Example

Import an AES-128 Wrap Key able to export and import, with some Delegated Capabilities set.

```
yubihsm> put wrapkey 0 0 wrapkey 1 export-wrapped,import-wrapped exportable-under-wrap,
↪sign-pkcs,sign-pss 000102030405060708090a0b0c0d0e0f
Stored Wrap key 0xaff7
```

31.50.2 Command Line Mode

Use `put-wrap-key` subcommand to import AES Wrap Key and `put-rsa-wrapkey` subcommand to import RSA Wrap Key

```
$ yubihsm-shell -a put-wrap-key -i <key_id> -l <label> -d <domains> -c <capabilities> --
↪delegated <delegated_capabilities> --in <key> [ --authkey <authKeyID> -p <password> ]
$ yubihsm-shell -a put-rsa-wrapkey -i <key_id> -l <label> -d <domains> -c <capabilities>␣
↪--delegated <delegated_capabilities> --in <key> [ --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, wrap-data, unwrap-data, export-wrapped, import-wrapped, exportable-under-wrap

- `--delegated=STRING` Required.

  Delegated capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- `--in=STRING` Required.

  The wrap key.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Format: hex

Example

Import an AES-128 Wrap Key able to export and import, with some Delegated Capabilities set.

```
$ yubihsm-shell -a generate-wrap-key -i 0 -l wrapkey -d 1 -c export-wrapped,import-
↪wrapped --delegated exportable-under-wrap,sign-pkcs,sign-pss --in wrap.key
Stored Wrap key 0xaff7
```

31.50.3 Protocol Details

Command

```
Tc = 0x4c
Lc = 2 + 40 + 2 + 8 + 1 + 8 + LW
Vc = I || L || D || C || A || DC || W
```

where –

`I` = *Object ID* (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

A = *ALGORITHMS* (1 byte)

DC = Delegated *Capability* (8 bytes)

W = Wrap Key (16, 24 or 32, 256, 384, 512 bytes)

For AES128_CCM_WRAP: 16 bytes

For AES192_CCM_WRAP: 24 bytes

For AES256_CCM_WRAP: 32 bytes

For RSA2048: 256 bytes

For RSA3072: 384 bytes

For RSA4096: 512 bytes

Response

```
Tc = 0xcc
Lc = 2
Vc = I
```

where –

I = ID of created Wrap Key (2 bytes)

31.51 PUT PUBLIC WRAP KEY Command

Import a public RSA Wrap Key exporting wrapped objects.

31.51.1 Interactive Mode

```
yubihsm> put pub_wrapkey e:session, w:key_id, s:label, d:domains, c:capabilities,␣
↪c:delegated_capabilities, i:key=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID. Use 0 to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **label** Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- **domains** Required.

Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- **capabilities** Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, wrap-data, unwrap-data, export-wrapped, import-wrapped, exportable-under-wrap

- **delegated_capabilities** Required.

  Delegated capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- **key** Required.

  A Public RSA key.

  Default Format: PEM

Example

Import the public RSA key from `rsa2048_pubkey.pem` as a Public Wrap Key able to export and import, with some Delegated Capabilities set.

```
yubihsm> put pub_wrapkey 0 0 rsa_wrapkey 1 export-wrapped,import-wrapped exportable-
↪under-wrap,sign-pkcs,sign-pss rsa2048_pubkey.pem
Stored Wrap key 0xadf8
```

31.51.2 Command Line Mode

```
$ yubihsm-shell -a put-public-wrapkey -i <key_id> -l <label> -d <domains> -c
↪<capabilities> --delegated <delegated_capabilities> --in <key> [ --authkey <authKeyID>␣
↪-p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID. Use `0` to generate Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-l, --label=STRING` Required.

  Key label. Can be empty.

  Possible Values: Maximum of 40 characters string

- `-d, --domains=STRING` Required.

  Domains where the key will be accessible. Use `all` to indicate all domains. Multiple domains can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: all,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16

- `-c, --capabilities=STRING` Required.

  Capabilities of the key. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, wrap-data, unwrap-data, export-wrapped, import-wrapped, exportable-under-wrap

- `--delegated=STRING` Required.

  Delegated capabilities of the key. Use `all` to include all capabilities. Use `none` to include no capability. Multiple capabilities can be separated by comma `,` or colon `:` with no spaces between.

  Possible Values: none, all, change-authentication-key, create-otp-aead, decrypt-oaep, decrypt-otp, decrypt-pkcs, delete-asymmetric-key, delete-authentication-key, delete-hmac-key, delete-opaque, delete-otp-aead-key, delete-template, delete-wrap-key, derive-ecdh, export-wrapped, exportable-under-wrap, generate-asymmetric-key, generate-hmac-key, generate-otp-aead-key, generate-wrap-key, get-log-entries, get-opaque, get-option, get-pseudo-random, get-template, import-wrapped, put-asymmetric-key, put-authentication-key, put-mac-key, put-opaque, put-otp-aead-key, put-template, put-wrap-key, randomize-otp-aead, reset-device, rewrap-from-otp-aead-key, rewrap-to-otp-aead-key, set-option, sign-attestation-certificate, sign-ecdsa, sign-eddsa, sign-hmac, sign-pkcs, sign-pss, sign-ssh-certificate, unwrap-data, verify-hmac, wrap-data

- `--in=STRING` Required.

  The file containing an RSA public key.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Format: PEM

Example

Import the public RSA key from `rsa2048_pubkey.pem` as a Public Wrap Key able to export and import, with some Delegated Capabilities set.

```
$ yubihsm-shell -a put-public-wrapkey -i 0 -l wrapkey -d 1 -c export-wrapped,import-
↪wrapped --delegated exportable-under-wrap,sign-pkcs,sign-pss --in rsa2048_pubkey.pem
Stored Wrap key 0xadf8
```

31.51.3 Protocol Details

Command

```
Tc = 0x73
Lc = 2 + 40 + 2 + 8 + 1 + 8 + LN
Vc = I || L || D || C || A || DC || N
```

where –

`I` = *Object ID* (2 bytes)

`L` = *Label* (40 bytes)

`D` = *Domain* (2 bytes)

`C` = *Capability* (8 bytes)

`A` = *ALGORITHMS* (1 byte)

`DC` = Delegated *Capability* (8 bytes)

`N` = RSA public key (256, 384 or 512 bytes)

For RSA2048: 256 bytes

For RSA3072: 384 bytes

For RSA4096: 512 bytes

Response

```
Tc = 0xf3
Lc = 2
Vc = I
```

where –

`I` = ID of created Public Wrap Key (2 bytes)

31.52 RANDOMIZE OTP AEAD Command

Create a new OTP AEAD using random data for key and private ID.

31.52.1 Shell Example

Generate a new OTP AEAD using OTP AEAD Key `0xc5f4` and put the result in file `aead`.

```
yubihsm> otp aead_random 0 0xc5f4 aead
```

31.52.2 Interactive Mode

```
yubihsm> otp aead_random e:session, w:key_id, F:aead
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of an OTP AEAD key. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **aead** Required.

  The generated OTP AEAD.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

Example

Generate a new OTP AEAD using OTP AEAD Key `0xc5f4` and put the result in file `aead`.

```
yubihsm> otp aead_random 0 0xc5f4 aead
```

31.52.3 Command Line Mode

```
$ yubihsm-shell -a randomize-otp-aead -i <key_id> [--out <aead> --authkey <authKeyID> -p
↪<password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of an OTP AEAD key. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--out=STRING`

  The generated OTP AEAD.

  Possible Values: Path to file or stdout

  Default Value: stdout

Example

Generate a new OTP AEAD using OTP AEAD Key `0xc5f4` and put the result in file `aead`.

```
$ yubihsm-shell -a randomize-otp-aead -i 0xc5f4 --out aead
```

31.52.4 Protocol Details

Command

```
Tc = 0x62
Lc = 2
Vc = I
```

where –

`I` = *Object ID* for the OTP AEAD Key (2 bytes)

Response

```
Tr = 0xe2
Lr = 36
Vr = A
```

where –

`A` = Nonce concatenated with AEAD (36 bytes)

31.53 RESET DEVICE Command

Resets and reboots the device, deletes all Objects and restores the default Options and Authentication Key.

31.53.1 Shell Example

Send reset over Session `0`.

```
yubihsm> reset 0
Device successfully reset
```

31.53.2 Interactive Mode

```
yubihsm> reset e:session
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Example

Send reset over Session `0`.

```
yubihsm> reset 0
Device successfully reset
```

31.53.3 Command Line Mode

```
$ yubihsm-shell -a reset [ --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

Example

Send reset over Session 0.

```
$ yubihsm-shell -a reset
Device successfully reset
```

31.53.4 Protocol Details

Command

```
Tc = 0x08
Lc = 0
Vc = Ø
```

Response

```
Tr = 0x88
Lr = 0
Vr = Ø
```

31.54 REWRAP OTP AEAD Command

Re-encrypt a Yubico OTP AEAD from one OTP AEAD Key to another OTP AEAD Key.

31.54.1 Shell Example

N/A

31.54.2 Interactive Mode

```
yubihsm> otp rewrap e:session, w:id_from, w:id_to, i:aead_in, F:aead_out
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **id_from** Required.

  Object ID of the OTP AEAD used to unwrap. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **id_to** Required.

Object ID of the OTP AEAD used to wrap. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **aead_in** Required.

  OTP AEAD to unwrap.

  Default Format: binary

- **aead_out** Required.

  OTP AEAD to wrap

31.54.3 Command Line Mode

This command is not available in command line mode.

31.54.4 Protocol Details

Command

```
Tc = 0x63
Lc = 2 + 2 + 36
Vc = I1 || I2 || A
```

where –

`I1` = Key ID from (2 bytes)

`I2` = Key ID to (2 bytes)

`A` = Nonce concatenated with AEAD (36 bytes)

Response

```
Tr = 0xe3
Lr = 36
Vr = A
```

where –

`A` = Nonce concatenated with AEAD (36 bytes)

31.55 SESSION MESSAGE Command

Sends a wrapped command for a previously established session. The command is encrypted and authenticated.

31.55.1 Example

Send an echo over Session `0`:

```
yubihsm> echo 0 0xff 1
Response (1 bytes):
ff
```

31.55.2 Protocol Details

Command

```
T~c~ = 0x05
L~c~ = 1 + L~inner_c~ + 8
V~c~ = S | | I~c~ | | M~c~
```

where –

S = Session ID (1 byte)

L~inner_c/inner_r~ = Length of the encrypted inner command / response (2 bytes)

M~c/r~ = CMAC of the outer command / response (8 bytes)

Response

```
T~r~ = 0x85
L~r~ = 1 + L~inner_r~ + 8
V~r~ = S | | I~r~ | | M~r~
```

31.56 SET INFORMAT Command

Set global input format. When set to something other than `default`, all future input is expected to have the set format.

31.56.1 Interactive Mode

```
yubihsm> set informat I:format
```

Parameters

- **format** Required.

  Input format. `default` resets the default expected input format, which can be different for different commands.

  Possible Values: default, base64, binary, PEM, password, hex, ASCII

Example

Set input format to PEM.

```
yubihsm> set informat PEM
```

31.56.2 Command Line Mode

Setting global input format is not possible in command line mode. However, individual commands can be set to expect a certain input format by using the `--informat=ENUM` flag.

31.57 SET LOG INDEX Command

Inform the device what the last extracted log entry is so logs can be reused. Mostly of practical use when forced auditing is enabled.

31.57.1 Shell Example

Set log index `41` as the last extracted entry.

```
yubihsm> audit set 0 41
```

31.57.2 Interactive Mode

```
yubihsm> audit set e:session, w:index
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **index** Required.

  Log index.

  Possible Values: 1-60

Example

Set log index `41` as the last extracted entry.

```
yubihsm> audit set 0 41
```

31.57.3 Command Line Mode

```
$ yubihsm-shell -a set-log-index --log-index <index> [ --authkey <authKeyID> -p
↪<password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--log-index=INT` Required.

  Log index.

  Possible Values: 1-60

Example

Set log index `41` as the last extracted entry.

```
$ yubihsm-shell -a set-log-index --log-index 41
```

31.57.4 Protocol Details

Command

```
Tc = 0x67
Lc = 2
Vc = I
```

where –

`I` = Index to set as last read log (2 bytes)

Response

```
Tr = 0xe7
Lr = 0
Vr = Ø
```

31.58 SET OPTION Command

Set device-global options that affect general behavior. Each invocation of this command sets a single option, which is represented as a `TAG-LENGTH-VALUE (TLV)`.

31.58.1 Shell Example

Turn off audit logging for Sign HMAC (command `53`) and Verify HMAC (command `5c`).

```
yubihsm> put option 0 command_audit 53005c00
```

31.58.2 Interactive Mode

```
yubihsm> put option e:session, o:option, i:data
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **option** Required.

  Device option name. `fips-mode` option is only applicable in FIPS compatible YubiHSMs.

  Possible Value: algorithm-toggle, command-audit, fips-mode, force-audit

- **data** Required.

  Value of option.

  Default Input Format: hex

Example

Turn off audit logging for Sign HMAC (command `53`) and Verify HMAC (command `5c`).

```
yubihsm> put option 0 command-audit 53005c00
```

31.58.3 Command Line Mode

```
$ yubihsm-shell -a put-option --opt-name <option> --opt-value <value> [ --authkey
↪<authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `--opt-name=STRING` Required.

  Device option name. `fips-mode` option is only applicable in FIPS compatible YubiHSMs.

  Possible Values: algorithm-toggle, command-audit, fips-mode, force-audit

- `--opt-value=STRING` Required.

  Device option value.

  Default input format: hex

Example

Set log index `41` as the last extracted entry.

```
$ yubihsm-shell -a put-option --opt-name command-audit --opt-value 53005c00
```

31.58.4 Protocol Details

Command

```
Tc = 0x4f
Lc = 3 + Lo
Vc = TO
```

where –

`To` = The TLV encoding of the selected option

`Lo` = The option-specific length in bytes

The options currently supported are the following:

`TAG` is 1 byte

`LENGTH` is 2 bytes

`VALUE` is Lo bytes

Tags.

```
force-audit = 0x01
command-audit = 0x03
algorithm-toggle = 0x4 (>=2.2.0)
fips-mode = 0x05 (>=2.2.0)
```

Values.

```
OFF = 0x00 (Disabled)
ON  = 0x01 (Enabled)
FIX = 0x02 (Enabled, only possible to turn off through factory reset)
```

The defined options are as follows:

With `Force audit` set, the device will refuse operations as long as the *Logs* Store is full. It takes a 1 byte value option.

`Command audit` can be used to toggle whether a specific command should be logged, this takes tuples of command number and option value.

`Algorithm toggle` allows the user to selectively disable individual algorithms for the whole device. This option can only be toggled on a freshly reset device, i.e. one with only the default Authentication Key. This takes a tuple of algorithm number and option value.

`FIPS mode` is only available on FIPS devices and can only be toggled on a freshly reset device, i.e. one with only the default Authentication Key present. It disables algorithms that are not allowed by FIPS 140. This step is required as part of setting the device in the approved mode of operation, together with deleting the default Authentication Key (see Section 3.2 of the YubiHSM FIPS Security Policy).

Response

```
Tr = 0xcf
Lr = 0
Vr = Ø
```

31.59 SET OUTFORMAT Command

Set global output format. When set to something other than `default`, all future output will be in the set format.

31.59.1 Interactive Mode

```
yubihsm> set outformat I:format
```

Parameters

- **format** Required.

  Output format. `default` resets the default output format, which can be different for different commands

  Possible Values: default, base64, binary, PEM, password, hex, ASCII

Example

Set output format to PEM.

```
yubihsm> set outformat PEM
```

31.59.2 Command Line Mode

Setting global output format is not possible in command line mode. However, individual commands can be set to output in a certain format by using the `--outformat=ENUM` flag.

31.60 SIGN ATTESTATION CERTIFICATE Command

Get attestation of an Asymmetric Key, output is an X.509 certificate.

31.60.1 Shell Example

Attest Asymmetric Key `0x79c3` using attestation key `0` (builtin).

```
yubihsm> attest asymmetric 0 0x79c3 0
-----BEGIN CERTIFICATE-----
MIIDeTCCAmGgAwIBAgIQaa8FkvRhqntp5HjyyCfilzANBgkqhkiG9w0BAQsFADAn
MSUwIwYDVQQDDBxZdWJpSFNNIEF0dGVzdGF0aW9uICgxMjM0NTYpMCAXDTE3MDEw
MTAwMDAwMFoYDzIwNzExMDA1MDAwMDAwWjAoMSYwJAYDVQQDDB1ZdWJpSFNNIEF0
dGVzdGF0aW9uIGlkOjB4NzljMzCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoC
ggEBAMYpAzHar0syanQEiRqWy8WDO5qETjDulo2txNBDwyMCNgeEYzo/uglUXLEm
Zj6Dd8EcdY9upHoqVpLduB+GIt+UEq5DeMN5Rzj2QZ/lQMELMdaD9ODc707aPvKT
/oAuj1aZ89vfg7jEVWBTPWquyFaxaCBoz8WWta9j5JxRppQpR27ub43950fX3wpW
btvlNLMx0QAQdDqEm2V3TEhnbo6T5XsgC78OdOikyJw2TP062rQXSY7GRuXob/Qa
INsJRXbbydqUXDHFNq8GnSkL8dHsNdf7bOSdAV6Vlw30JFbJ2uoW2EkGmF9qYWnt
EVyyPMMQwF09r9HVpLF83TBaYoMCAwEAAaOBnTCBmjATBgorBgEEAYLECgQBBAUE
AwIAADATBgorBgEEAYLECgQCBAUCAx6EgDASBgorBgEEAYLECgQDBAQDAgABMBMG
CisGAQQBgsQKBAQEBQMDAAABMBkGCisGAQQBgsQKBAUECwMJAAAAAAAAAARAMBIG
CisGAQQBgsQKBAYEBAICecMwFgYKKwYBBAGCxAoECQQIDAZyc2FrZXkwDQYJKoZI
hvcNAQELBQADggEBABRReYze+KRfevrgyI3C2aLAWSiQRjJ6vvaP1Fh4bOw4X2HC
rLAI150h4O5eH/aXVNv+368FWlQhcY68jKDgDoeckrlt9thFxaphasd/Wt1Pbqzj
trnEillYjjP6rddyCR1yitmnQ3Qnsk3w1mTE/AtzmDOi7V/wNymilB79OFDGmB6P
d1VI7zGUHtLlj1qeyY4/ETqKuPDzZY5RUPYrO8/iPzy64AdtDXt1e39n9pTcohp2
PSQQe36gU7vt9+5SebEj0CF/qTk317L1R42TfeHFSJlgBTHSWcuvDORNJxDHTcco
bI+wE2dCcnjyLU9dr5tkNsD3k5pscuTmpBGFDlg=
-----END CERTIFICATE-----
```

31.60.2 Protocol Details

Command

```
Tc = 0x64
Lc = 2 + 2
Vc = I || A
```

where –

`I` = *Object ID* of the Asymmetric Key to attest (2 bytes)

`A` = *Object ID* of the Asymmetric Key used for attestation (2 bytes)

If `A` is `0` the internal attestation key is used.

Response

```
Tr = 0xe4
Lr = LX
Vr = X
```

where –

`X` = DER encoded X.509 attestation

31.61 SIGN ECDSA Command

Computes a digital signature using ECDSA on the provided data.

31.61.1 Shell Example

Sign data in file `data` using key `0x52b6` and put the result in file `sig`.

```
yubihsm> sign ecdsa 0 0x52b6 ecdsa-sha256 data sig
```

31.61.2 Protocol Details

Command

```
Tc = 0x56
Lc = 2 + LD
Vc = I || D
```

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`D` = H

The DSI for ECDSA is a possibly zero-left-padded hash of the data, `H`.

Response

```
Tr = 0xd6
Lr = LDS
Vr = DS
```

where –

DS = Resulting signature

> The length of DS, LDS, depends on the *ALGORITHMS* used and equals the length of the signature plus its DER encoding.

31.62 SIGN EDDSA Command

Computes a digital signature using EdDSA on the provided data.

31.62.1 Example

Perform an EdDSA signature with key *0xddf6* of the content of file `data`:

```
yubihsm> sign eddsa 0 0xddf6 ed25519 data
wZljrOstOLPuMHGrXDnpAb5Wxo79+wX/vQkb/6K34tOd8se
QfLNRVTonfErttkWUAz/UlNtaG4XJYnY8vabCQ==
```

31.62.2 Protocol Details

Command

```
T~c~ = 0x6a
L~c~ = 2 + L~D~
V~c~ = I \|\| D
```

where –

I = *Object ID* of the Asymmetric Key (2 bytes)

> The DSI for EdDSA is the raw data *D*.

DSI = D

> For a given DSI, the command will generate a digital signature DS. The length of DS, L~DS~, depends on the Algorithm used. At this time only Ed25519 is implemented.

DS = EdDSA(DSI). Key is omitted

L~DS~ = 0x0040 bytes

Response

```
T~r~ = 0xea
L~r~ = L~DS~
V~r~ = DS
```

where –

DS = Resulting signature

31.63 SIGN HMAC Command

Perform an HMAC operation in device and return the result.

31.63.1 Shell Example

Perform an HMAC operation using the HMAC Key `0x7cf2`.

```
yubihsm> hmac 0 0x7cf2 666f6f626172
4c17e17300a51a3f8aeeba131e9c680e4e40b429aa1d547807efd8e3d95ccd39
```

31.63.2 Protocol Details

Command

```
Tc = 0x53
Lc = 2 + LD
Vc = I || D
```

where –

I = *Object ID* of the HMAC Key (2 bytes)

D = Data to HMAC

Response

```
Tr = 0xd3
Lr = LR
Vr = R
```

where –

R = HMAC Response, 20, 32, 48 or 64 bytes depending on the Algorithm.

31.64 SIGN PKCS1 Command

Computes a digital signature using RSA-PKCS1v1.5 on the provided data.

31.64.1 Shell Example

Sign the data in the file `test` using `rsa-pkcs1-sha256`.

```
yubihsm> sign pkcs1v1_5 0 0x1e15 rsa-pkcs1-sha256 test
eu9HQceSs0zsUogVloovRRcDGtkBj5AIp2Nnk6LWT4KbQZX8ac+vmFtVotjDIF9PkQ9MA8K
sfUGvXAxpnvUyin3BjGvzENu5XRi+ZOGP4m8777zbDi1v7FKQSx8/KdZf4tulIsL4rM4M+uH
/QoQ83vWty4c63QjcSlZJQDsdHn9r3E5or3QgBo06yK2Rd8W3WYGloSPvDaGu7L87CDFy
MniAQB//Sw7bYr4hbVpKIWi6q4VPhBKdaB6+FzTmYrqsSv1vwek0V4LbvyelTHlh9PpFuSF
ZeGJ/i1gkIeSO2XlKNLa4+AO+H+TYUOP3b6Qlhs3f7e4AFFWKE6lPpDHJA==
```

31.64.2 Interactive Mode

```
yubihsm> sign pkcs1v1_5 e:session, w:key_id, a:algorithm, i:data=-, F:out=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the asymmetric key to sign with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **Algorithm** Required.

  Signing algorithm.

  Possible Values: rsa-pkcs1-sha1, rsa-pkcs1-sha256, rsa-pkcs1-sha384, rsa-pkcs1-sha512

- **data**

  Data to sign.

  Possible Values:Path to file or - for stdin

  Default Value: stdin

  Default Input Format: binary

- **out**

  Signed data.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

  Default Input Format: PEM

Example

Sign the data in the file `test` using `rsa-pkcs1-sha256`.

```
yubihsm> sign pkcs1v1_5 0 0x1e15 rsa-pkcs1-sha256 test
eu9HQceSs0zsUogVloovRRcDGtkBj5AIp2Nnk6LWT4KbQZX8ac+vmFtVotjDIF9PkQ9MA8KlsfUGvXAxpnvUyin3BjGvzENu5XRi+ZO
↪KdZf4tulIsL4rM4M+uH/
↪QoQ83vWty4c63QjcSlZJQDsdHn9r3E5or3QgBo06yK2Rd8W3WYGloSPvDaGu7L87CDFyMniAQB//
↪Sw7bYr4hbVpKIWi6q4VPhBKdaB6+FzTmYrqsSv1vwek0V4LbvyelTHlh9PpFuSFZeGJ/
↪i1gkIeSO2XlKNLa4+AO+H+TYUOP3b6Qlhs3f7e4AFFWKE6lPpDHJA==
```

31.64.3 Command Line Mode

```
$ yubihsm-shell -a sign-pkcs1v15 -i <key_id> -A <algorithm> [--in <data> --informat
↪<informat> --out <out> --outformat <outformat> --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the asymmetric key to sign with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-A, --algorithm=STRING` Required.

  Signing algorithm.

  Possible Values: rsa-pkcs1-sha1, rsa-pkcs1-sha256, rsa-pkcs1-sha384, rsa-pkcs1-sha512

- `--in=STRING`

  Data to sign.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Input Format: binary

- `--informat=ENUM`

  Input data format.

  Possible Values: binary, base64, hex, PEM

- `--out=STRING`

Signed data.

Possible Values: Path to file or stdout

Default Value: stdout

Default Output Format: PEM

- `--outformat=ENUM`

  Output data format.

  Possible Values: binary, base64, hex, PEM

Example

Sign the data in the file `test` using `rsa-pkcs1-sha256`.

```
$ yubihsm-shell -a sign-pkcs1v15 -i 0x1e15 -A rsa-pkcs1-sha256 --in test
eu9HQceSs0zsUogVloovRRcDGtkBj5AIp2Nnk6LWT4KbQZX8ac+vmFtVotjDIF9PkQ9MA8KlsfUGvXAxpnvUyin3BjGvzENu5XRi+ZO
↪KdZf4tulIsL4rM4M+uH/
↪QoQ83vWty4c63QjcSlZJQDsdHn9r3E5or3QgBo06yK2Rd8W3WYGloSPvDaGu7L87CDFyMniAQB//
↪Sw7bYr4hbVpKIWi6q4VPhBKdaB6+FzTmYrqsSv1vwek0V4LbvyelTHlh9PpFuSFZeGJ/
↪i1gkIeSO2XlKNLa4+AO+H+TYUOP3b6Qlhs3f7e4AFFWKE6lPpDHJA==
```

31.64.4 Protocol Details

Command

```
Tc = 0x47
Lc = 2 + LD
Vc = I || D
```

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`D` = Digest

The Digest can be either a raw hash of data, where DigestInfo will be applied in the device, or DigestInfo + hash. Hashes supported are SHA-1, SHA-256, SHA-384 and SHA-512.

Response

```
Tr = 0xc7
Lr = LDS
Vr = DS
```

where –

`DS` = Resulting signature

31.65 SIGN PSS Command

Computes a digital signature using RSA-PSS on the provided data.

31.65.1 Shell Example

Sign what is in file `data` using key `0x79c3` and put the resulting signature in `sig`.

```
yubihsm> sign pss 0 0x79c3 rsa-pss-sha256 data sig
```

31.65.2 Interactive Mode

```
yubihsm> sign pss e:session, w:key_id, a:algorithm, i:data=-, F:out=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the asymmetric key to sign with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **Algorithm** Required.

  Signing algorithm.

  Possible Values: rsa-pss-sha1, rsa-pss-sha256, rsa-pss-sha384, rsa-pss-sha512

- **data**

  Data to sign.

  Possible Values: Path to file or - for stdin

  Devault Value: stdin

  Default Input Format: binary

- **out**

  Signed data.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

  Default Input Format: PEM

Example

Sign what is in file data using key 0x79c3 and put the resulting signature in sig.

```
yubihsm> sign pss 0 0x79c3 rsa-pss-sha256 data sig
```

31.65.3 Command Line Mode

```
$ yubihsm-shell -a sign-pss -i <key_id> -A <algorithm> [--in <data> --informat <informat>
↪ --out <out> --outformat <outformat> --authkey <authKeyID> -p <password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified

- `-i, --object-id=SHORT` Required.

  Object ID of the asymmetric key to sign with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-A, --algorithm=STRING` Required.

  Signing algorithm.

  Possible Values: rsa-pss-sha1, rsa-pss-sha256, rsa-pss-sha384, rsa-pss-sha512

- `--in=STRING`

  Data to sign.

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Input Format: binary

- `--informat=ENUM`

  Input format.

  Possible Values: binary, base64, hex, PEM

- `--out=STRIN`

  Signed data.

  Possible Values: Path to file or stdout

  Default Value: stdout

  Default Output Format: PEM

- `--outformat=ENUM`

Output format.

Possible Values: binary, base64, hex, PEM

Example

Sign what is in file `data` using key `0x79c3` and put the resulting signature in `sig`.

```
$ yubihsm-shell -a sign-pss -i 0x79c3 -A rsa-pss-sha256 --in data --out sig
```

31.65.4 Protocol Details

Command

```
Tc = 0x55
Lc = 2 + 1 + 2 + LD
Vc = I || M || S || D
```

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`M` = Hash *ALGORITHMS* to use for MGF1

`S` = Salt len (2 bytes)

`D` = Hashed data (20, 32, 48 or 64 bytes)

The DSI of EMSA-PSS is as defined in RFC 3447.

`DSI` = EMSA-PSS-ENCODE(M, emBits, Hash, MGF, sLen).

`Hash` is a supported hash Algorithm

`MGF` is a supported masking function

`sLen` is the length of the Salt

The DSI is generated internally and only the Hash of the data and the Salt length are provided.

Response

```
Tr = 0xd5
Lr = LDS
Vr = DS
```

where –

`DS` = Resulting signature

31.66 SIGN SSH CERTIFICATE Command

Produce an SSH Certificate signature. The certificate can then be used to login to hosts.

31.66.1 Shell Example

Produce a new SSH Certificate.

```
yubihsm> certify 0 0xabcd 0x1234 rsa-pkcs-sha256 req.dat cert.dat
```

31.66.2 Interactive Mode

```
yubihsm> certify e:session, w:key_id, w:template_id, a:algorithm, i:infile=-, F:outfile=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the asymmetric key to sign with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **template_id** Required.

  Template Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **algorithm** Required.

  Signing algorithm.

  Possible Values: rsa-pkcs1-sha1, rsa-pkcs1-sha256, rsa-pkcs1-sha384, rsa-pkcs1-sha512

- **data**

  Certificate request.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Default Input Format: binary

- **out**

  Signed SSH certificate.

  Possible Values: Path to file or - for stdout

  Default Value: stdout

  Default Input Format: binary

Example

Produce a new SSH Certificate.

```
yubihsm> certify 0 0xabcd 0x1234 rsa-pkcs-sha256 req.dat cert.dat
```

31.66.3 Command Line Mode

```
$ yubihsm-shell -a sign-ssh-certificate -i <key_id> --template-id <template_id> -A
↪<algorithm> [--in <data> --informat <informat> --out <out> --authkey <authKeyID> -p
↪<password> ]
```

Parameters

- `--authkey=INT`

  The ObjectID of the authentication key used to open a session. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

  Default Value: 1

- `-p, --password=STRING` Required.

  The password to authentication key used to open a session. The password is prompted for if not specified.

- `-i, --object-id=SHORT` Required.

  Object ID of the asymmetric key to sign with. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `--template-id=INT` Required.

  Template Object ID. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- `-A, --algorithm=STRING` Required.

  Signing algorithm.

  Possible Values: rsa-pkcs1-sha1, rsa-pkcs1-sha256, rsa-pkcs1-sha384, rsa-pkcs1-sha512

- `--in=STRING`

  Certificate request

  Possible Values: Path to file or stdin

  Default Value: stdin

  Default Input Format: binary

- `--informat=ENUM`

  Input data format.

  Possible Values: binary, base64, hex, PEM

- `--out=STRING`

Signed SSH certificate.

Possible Values: Path to file or stdout

Default Value: stdout

Default Output Format: binary

Example

Produce a new SSH Certificate.

```
$ yubihsm-shell -a sign-ssh-certificate -i 0xabcd --template-id 0x1234 -A rsa-pkcs-
↪sha256 --in req.dat --out cert.dat
```

31.66.4 Protocol Details

Command

```
Tc = 0x5d
Lc = 2 + 2 + 1 + 4 + 256 + LR
Vc = I || T || A || N || S || R
```

Sign and SSH Certificate by using the given Asymmetric Key and SSH Template.

where –

`I` = *Object ID* of the Asymmetric Key (2 bytes)

`T` = *Object ID* of the SSH Template (2 bytes)

`A` = *ALGORITHMS* (1 byte)

`N` = Timestamp with the definition of `Now` (4 bytes)

`S` = Signature over the request and timestamp (256 bytes)

`R` = Request (LR bytes)

Response

```
Tr = 0xd6
Lr = LS
Vr = S
```

where –

`S` = Certificate Signature (LS bytes)

31.67 UNWRAP DATA Command

Decrypt (unwrap) data using a Wrap Key.

31.67.1 Shell Example

```
yubihsm> decrypt aesccm 0 0x5b3a MRkj6B0AAAAAAAAAAoO4dkIeAYoPvwTV/M/JX1dwKnLqnERO1hSW4wPS
Hello world!
```

31.67.2 Interactive Mode

```
yubihsm> decrypt aesccm e:session, w:key_id, i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Required.

  Object ID of the wrap key. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **data**

  Data to decrypt/unwrap.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Default Input Format: base64

Example

```
yubihsm> decrypt aesccm 0 0x5b3a MRkj6B0AAAAAAAAAAoO4dkIeAYoPvwTV/M/JX1dwKnLqnERO1hSW4wPS
Hello world!
```

31.67.3 Command Line Mode

This command is not available in command line mode.

31.67.4 Protocol Details

Command

```
Tc = 0x69
Lc = 2 + 13 + LD + 16
Vc = I || N || D || M
```

where –

`I` = *Object ID* of a Wrap Key (2 bytes)

`N` = Nonce (13 bytes)

`D` = Data to be unwrapped

`M` = Mac (16 bytes)

Response

```
Tr = 0xe9
Lr = LD
Vr = D
```

where –

`D` = Unwrapped data

31.68 VERIFY HMAC Command

Verify a generated HMAC.

31.68.1 Shell Example

N/A

31.68.2 Protocol Details

Command

```
Tc = 0x5c
Lc = 2 + LH + LD
Vc = I || H || D
```

where –

`I` = *Object ID* of the HMAC Key (2 bytes)

`H` = HMAC (20, 32, 48 or 64 bytes)

`D` = Data

Response

```
Tr = 0xdc
Lr = 1
Vr = V
```

where –

`V` = Verified (1 byte)

`V` will have the value `1` if verification succeeded and `0` otherwise.

31.69 WRAP DATA Command

Encrypt (wrap) data using a Wrap Key.

31.69.1 Shell Example

Using Wrap Key `0x5b3a` encrypt the string `Hello world!`.

```
yubihsm> encrypt aesccm 0 0x5b3a "Hello world!"
MRkj6B0AAAAAAAAAAoO4dkIeAYoPvwTV/M/JX1dwKnLqnERO1hSW4wPS
```

31.69.2 Interactive Mode

```
yubihsm> encrypt aesccm e:session, w:key_id, i:data=-
```

Parameters

- **session** Required.

  The ID of the authenticated session to send the command over.

  Possible Values: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

- **key_id** Requiredc.

  Object ID of the wrap key. Object ID is a 2 bytes integer. Can be specified in hex or decimal.

- **data**

  Data to encrypt/wrap.

  Possible Values: Path to file or - for stdin

  Default Value: stdin

  Default Input Format: binary

Example

Using Wrap Key `0x5b3a` encrypt the string `Hello world!`.

```
yubihsm> encrypt aesccm 0 0x5b3a "Hello world!"
MRkj6B0AAAAAAAAAAoO4dkIeAYoPvwTV/M/JX1dwKnLqnERO1hSW4wPS
```

31.69.3 Command Line Mode

This command is not available in command line mode.

31.69.4 Protocol Details

Command

```
Tc = 0x68
Lc = 2 + LD
Vc = I || D
```

where –

`I` = *Object ID* of the Wrap Key (2 bytes)

`D` = Data to be wrapped

Response

```
Tr = 0xe8
Lr = 13 + LD + 16
Vr = N || D || M
```

where –

`N` = Nonce (13 bytes)

`D` = Wrapped data (L~W~ = 1 + L~D~ bytes)

> The wrapped data includes a leading encrypted nul byte that is added automatically by the YubiHSM2. This byte is checked by UNWRAP DATA and therefore must be added if manually generating an encrypted message offline.

`M` = Mac (16 bytes)

CHAPTER

THIRTYTWO

GLOSSARY

A

Application authentication key AES key used to authenticate to the device. Performs operations according to its defined capabilities.

Audit key AES authentication key with rights to access audit log.

authentication key Performs operations according to its defined capabilities.

authentication key: Default Factory-installed Advanced Encryption Standards (AES) key used when initializing the device. Possesses all capabilities.

C

Capability A description of what operations are allowed on or with an object such as a key.

Column Encryption Key (CEK) CEKs are content-encryption keys used to encrypt data in a Microsoft SQL Server Always Encrypted database.

Column Master Key (CMK) CMKs are key-protecting keys used to encrypt CEKs for a Microsoft SQL Server Always Encrypted database.

Cryptographic API Next Generation (CNG) A CNG is Microsoft's cryptographic architecture, which allows developers to implement applications with features for encryption, electronic signatures, certificate management, etc.

D

Delegated capability An operation that an object is allowed to perform by virtue of receiving those permissions from the authentication key or wrap key that was used to create it.

Domain A logical "container" for objects that can be used to control access to objects on the device.

G

Guarded Host This is an attested Hyper-V host machine with a Trusted Platform Module (TPM) that can run shielded Hyper-V VMs.

H

Host Guardian Services (HGS) This is a Windows Server role that is composed of the Attestation Service and Key Protection Services.

Hyper-V Virtual Machine (VM) Microsoft Hyper-V is a native hypervisor that can create VMs on x86-64 systems running Windows.

K

Key custodian Holder of a wrap key share.

Key Storage Provider (KSP) This is a Dynamic Link Library (DLL) that is loaded by Microsoft CNG. KSPs can be used to create, delete, export, import, open and store keys.

M

M of n Scheme where a Wrap key is split into a total number of shares (n) held by key custodians, where a minimum number of shares (m) (sometimes called a quorum and sometimes a privacy threshold) is needed to regenerate and use the key.

O

Object ID (OID) These are unique identifiers for any kind of object stored on YubiHSM2. An ID can range from 1 to 65535; however, the device can only hold a maximum of 256 unique objects.

S

Shielded VM This is a Hyper-V VM with a virtual TPM; it is encrypted using BitLocker, and can run only on attested guarded hosts in a guarded fabric.

SQL Server Management Studio (SSMS) SQL Server Management Studio (SSMS) is a software application that is used for configuring, managing, and administering all components within Microsoft SQL Server.

T

Trusted Computing Group (TCG) This is a group formed by AMD, Hewlett-Packard, IBM, Intel and Microsoft to implement Trusted Computing concepts across personal computers.

Trusted Platform Module (TPM) This is a cryptographic chip on a device that stores RSA encryption keys specific to the host system for hardware authentication.

W

Wrap key An AES key used to protect key material when exporting to file from device and when importing from file to device. Key material exported under wrap will be encrypted and can only be decrypted using the wrap key.

CHAPTER

THIRTYTHREE

COPYRIGHT

33.1 Trademarks

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

33.2 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.

33.3 Contact Information

Yubico AB
Gävlegatan 22
113 30 Stockholm
Sweden

33.4 License

33.5 Getting Help

Documentation is continuously updated on https://docs.yubico.com/ (this site). Additional support resources are available in the Yubico Knowledge Base.

Click the links to:

- Submit a support request
- Contact our sales team

33.6 Feedback

Yubico values and welcomes your feedback. If you think you may have discovered a flaw in our product, please submit a support request at https://support.yubico.com/hc/en-us and provide as much detail as you can.

33.7 Document Updated

2025-03-14 19:04:46 UTC