yubico
Trust the Net

YubiHSM

User Manual

Version: 1.5.0

April 6, 2015

Introduction

Yubico is the leading provider of simple, open online identity protection. The company's flagship product, the YubiKey®, uniquely combines driverless USB hardware with open source software. More than a million users in 100 countries rely on YubiKey strong two-factor authentication for securing access to computers, mobile devices, networks and online services. Customers range from individual Internet users to e-governments and Fortune 500 companies. Founded in 2007, Yubico is privately held with offices in California, Sweden and UK.

Disclaimer

Trademarks

Contact Information

Yubico Inc
228 Hamilton Avenue, 3rd Floor
Palo Alto, CA 94301
USA
info@yubico.com

Contents

1 Document Information

1.1 Purpose

This document describes the Yubico YubiHSM philosophy, concepts and modes of operation as well as a detailed protocol- and implementation specifications.

1.2 Audience

Systems integrators

1.3 Related documentation

- The YubiKey Manual – Usage, configuration and introduction of basic YubiKey concepts
- Web server API Validation Protocol Version 2.0
- RFC 3610 – Counter with CBC-MAC
- NIST Special Publication 800-90 – Recommendation for Random Number Generation Using Deterministic Random Bit Generators
- Yubico online forum – http://forum.yubico.com

1.4 Document History

Date	Version	Author	Activity
2011-02-23	0.1	JE	First release
2011-04-05	0.9	JE	Update prior to first release
2011-04-12	0.9	JE	Minor corrections
2011-09-14	1.0	JE	Release changes
2012-03-16	1.0.4	FT	Updates to describe version 1.0.4
2015-03-30	1.5.0	TM	Updates to images, logo

1.5 Definitions

Table Header 1	Table Header 2
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
CBC	Chaining Block Cipher
CCM	Counter with CBC-MAC
CDC	Communication Device Class
CTR	Counter mode encryption
ECB	Electronic Codebook
DRBG	Deterministic Random Bit Generator
HSM	Hardware Security Module
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
OTP	One Time Password
PRNG	Pseudo Random Number Generator

RNG	Random Number Generator
TRNG	True Random Number Generator
USB	Universal Serial Bus

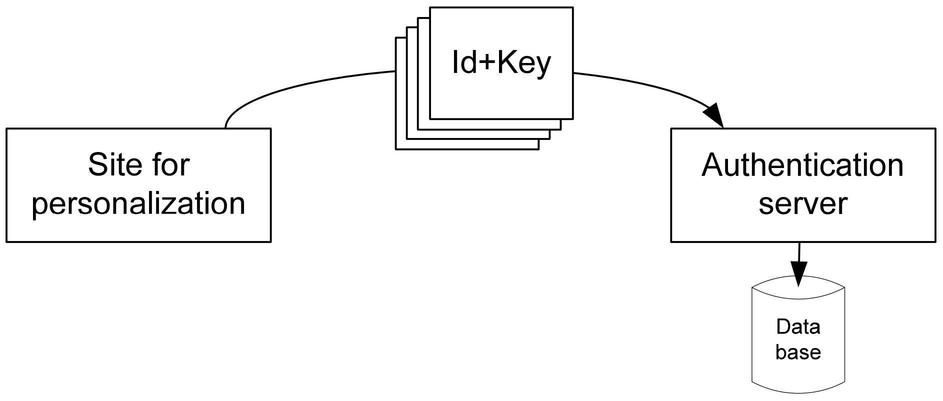
2 Introduction and concepts

The Yubico YubiHSM provides a low-cost way to move out sensitive information and cryptographic operations away from a vulnerable computer environment without having to invest in expensive dedicated Hardware Security Modules (HSMs). Although the main application area is for securing Yubico OTP authentication operations, several generic cryptographic primitives allows for a wider range of applications.

The hardware is a small USB device, entirely powered from the host computer.

No low-level drivers are needed as the YubiHSM implements the USB-CDC serial communication class. The YubiHSM is intended to operate in conjunction with a host application.

The YubiHSM supports several modes of operation, but the key concept is a symmetric scheme where one device at one location can generate a secure data element in a secure environment. This piece of data can then be transmitted over a potentially insecure channel to a second location where it is stored on a potentially vulnerable storage. With a minimal server application and a second device, Yubico OTPs can then be verified securely.



In this scenario, symmetric keys are generated at a personalization site. These are to be transferred over to an authentication server over an unsecure channel where the data is to be stored in a potentially vulnerable database. Although the protection of the computers as such is critical, the YubiHSM is solely designed to protect the data from being compromised.

In a more complex setting, there are multiple sites for personalization and multiple authentication servers, all applying to the same set of YubiKey tokens. The YubiHSM is designed to handle such settings as well.

Furthermore, several generic cryptographic primitives, such as AES ECB encryption, HMAC-SHA1, cryptographically secure random number generation are included that can be used for other purposes than YubiKey OTP validation.

It is however important to underline that the YubiHSM is not a direct replacement for a full-blown HSM and that some properties found in HSMs are not implemented.

2.1 What AEAD and nonce is

AEAD stands for authenticated encryption with associated data, which describes a cryptographic method for both providing confidentiality and authenticity.

An AEAD block consists of two parts – a ciphertext block and a message authentication code (MAC). In order to construct, decrypt or verify an AEAD, a symmetrical cryptographic key and a piece of associated data is required. This associated data, called a "nonce" can either be a uniquely generated "handle" or something that is uniquely related to the AEAD.

The YubiHSM uses the proven AES-CCM scheme for AEAD generation and absolutely critical to this algorithm is that the same nonce is never used twice together with the same cryptographic key.

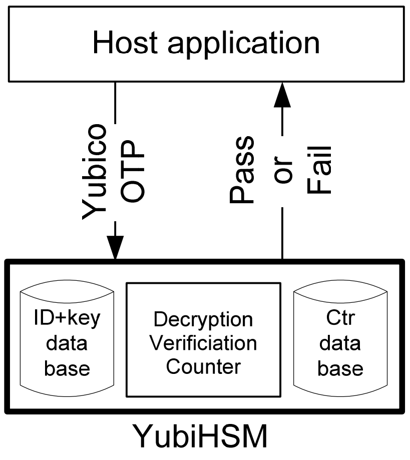
To never reuse the nonce is so important that there are several different YubiHSM permission flags made available to lock down the AEAD generation functionality for various use cases. See `YSM_AEAD_GENERATE`, `YSM_BUFFER_AEAD_GENERATE`, `YSM_RANDOM_AEAD_GENERATE`, `YSM_BUFFER_LOAD` and `YSM_USER_NONCE_ENABLE`.

In the case of an AEAD being used for Yubico OTP related operations, the public identity together with a unique key identifier is used as the nonce as there is a uniqueness guaranteed by the public identity and key used for a specific AEAD in this context.

In the case there is no uniqueness provided, the YubiHSM provides a method to generate a guaranteed (per YubiHSM) unique nonce.

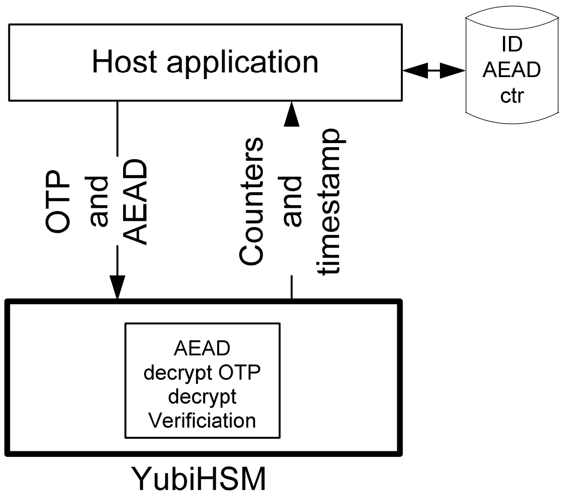
2.2 Internal- vs. external storage

The YubiHSM has an internal storage for YubiKey secrets, but this only holds a limited number of entries. The basic model currently has a capacity of 1024 entries maximum.



With internal storage, the complete OTP validation can be processed by the YubiHSM, including maintaining the values of the database counters which also are stored internally.

In order to support larger installations, YubiKey secrets can be generated and turned into AEADS internally. These AEADs can then be securely stored in an external database, without any obvious limitations in terms of size.

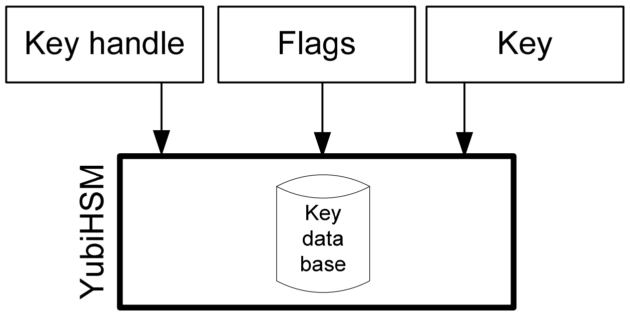


In this setting, host logic is required to locate an AEAD in the database based on the public ID received from an OTP. Additional host logic is then required to verify and maintain the counter values to prevent replay attacks.

Host logic can also be used to provide OTP counter replication to multiple validation servers if fault tolerance is desired.

2.3 Key storage

All cryptographic keys are identified by a unique 32-bit key handle. After keys have been generated and stored in the internal key store, all references to cryptographic keys uses the key handle.



For each key handle, a flag word defines which operations are allowed. With this setting, an “asymmetric” scheme can be made with two YubiHSM having the same key handle but with different operations enabled. For example, the first may be configured to generate AEADs only and the second to verify AEADs/OTPs only.

At initial configuration of the YubiHSM, at least one symmetric AEAD key has to be created and stored in the internal on-chip memory. The key database is write-only and can hold up to 40 entries.

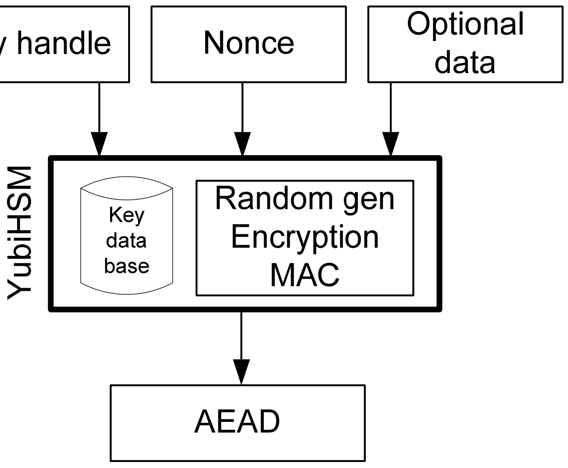
An additional “temporary key” is available which is reset on power-up or key storage unlock. This temporary key can be set by writing a valid AEAD and when successfully set, this key can be used like any other key handle. The “temporary key” has keyhandle 0xffffffff. See `YSM_TEMP_KEY_LOAD` for further information.

Access to the key storage can be protected with a HSM password (called “master key” during configuration) and possibly also require a YubiKey OTP. If set, this password (and OTP) has to be explicitly written after power-up.

The keystore is encrypted (AES-256) using the master key when stored in the internal non-volatile memory. The keystore can be locked again at any time by performing an unlock with the incorrect key.

2.4 AEAD generation

AEAD generation comprises generation of a secure data unit from two non-sensitive entities, i.e. a handle (reference) to a key and nonce (public identity in case of a YubiKey secret) to associate with the AEAD generated. Additional clear text data can optionally be provided to be included into the generated AEAD.



This mode allows generation of secrets with high-quality random numbers where these are instantly encrypted with a shared secret without ever leaving the secure environment of the embedded device. This encrypted and MACed AEAD can then be securely transmitted over open networks and stored on external media.

Note: The AEADs are generated using AES-CCM. Because of how AES-CCM works, it is absolutely necessary to never use the same nonce more than once!

If an attacker can trigger generation of AEADs using a specific keyhandle, an arbitrary nonce and an arbitrary plaintext the attacker can actually decrypt a previously generated AEAD by specifying the same nonce and the ciphertext of the previous AEAD as plaintext.

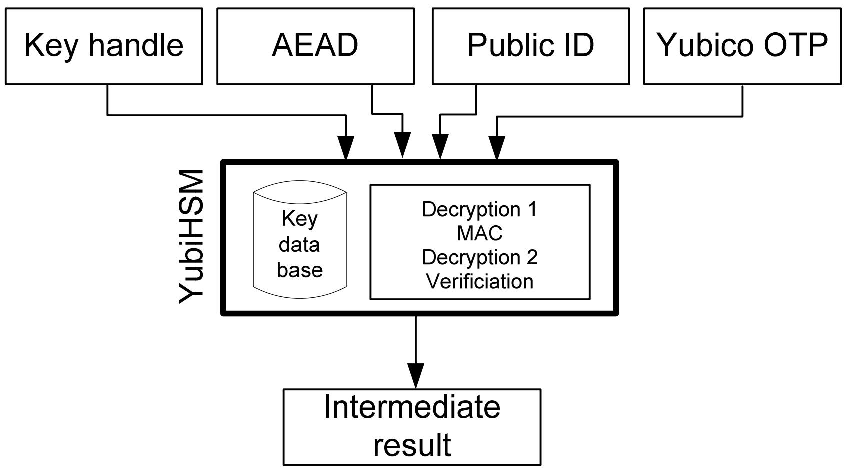
The following table describes some (believed safe) use cases and the smallest set of permission bits required on the keyhandle used to generate AEADs. If a use case requires "Yes" for both user supplied nonce and plaintext, external actions must be taken to ensure an attacker cannot control the input to the AEAD generation process.

User	Encrypt to multiple keyhandles	User supplied nonce	User supplied plaintext	Permission flags
Generate AEADs containing YubiKey secrets	Yes	No	Yes	`YSM_BUFFER_AEAD_GENERATE,` `YSM_BUFFER_LOAD`
	No	No	Yes	`YSM_AEAD_GENERATE`
Generate AEADs containing random secrets with known nonce	Yes	Yes	No	`YSM_BUFFER_AEAD_GENERATE,` `YSM_USER_NONCE`

Note: All keyhandles should have only the minimum set of required permission flags. Some combinations are inherently insecure, such as AEAD generation/validation and AES ECB commands. As a deterring example, a keyhandle with `YSM_AES_ECB_BLOCK_ENCRYPT` (note, *encrypt*) enabled can actually be used to *decrypt* AEADs used with that keyhandle due to the nature of how the CCM mode works.

2.5 OTP validation (external AEAD storage)

This mode allows verification of a received Yubico OTP with a supplied AEAD, stored on an external media.



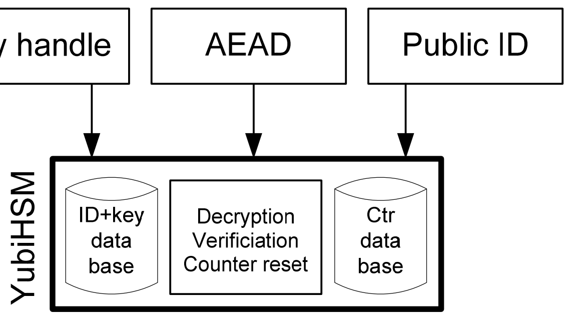
In detail, the principle for OTP verification is typically done as:

1. Receive a 6 + 16 bytes Yubico OTP
2. Use the public ID to retrieve the associated AEAD from a database
3. The key handle is typically a constant for a validating instance
4. Send key handle + AEAD + public ID + OTP to YubiHSM
5. The YubiHSM decodes the AEAD and then uses the decrypted secrets to decrypt the received OTP.
6. If successfully decoded, the counter- and timestamp values are sent back to the host application.
7. The counters are checked against the previously stored values to detect potential OTP replays. If received counters are higher, the database is updated with the newly decoded counters.

2.6 Yubico OTP validation AEAD store

The YubiHSM provides an internal storage for AEADs to allow full validation of Yubico OTPs, including detection of replay attempts.

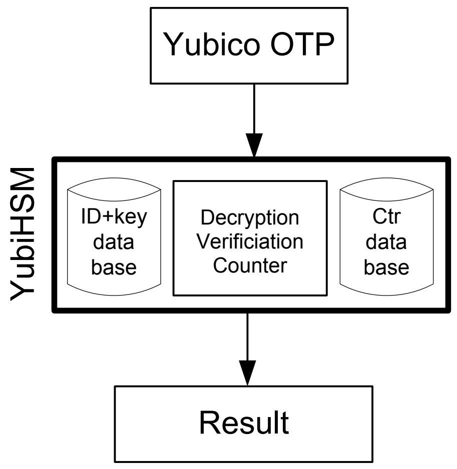
The AEAD store operation takes a supplied AEAD and decrypts and verifies its authenticity. If successful, the secrets are stored in the internal identity database together with the supplied public id as a unique identifier. Associated counters are reset.



The internal AEAD storage is limited by the amount of on-chip memory, which restricts this mode of operation to smaller installations.

2.7 Yubico OTP validation (internal AEAD storage)

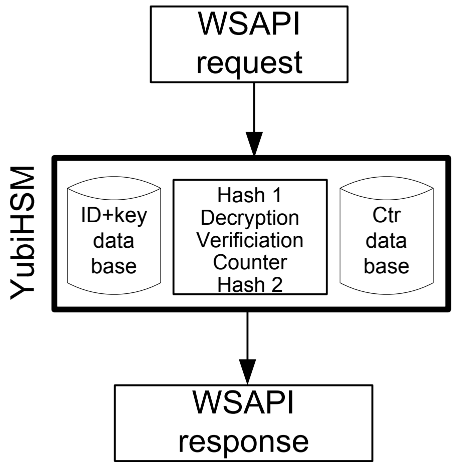
This mode allows complete verification of a received Yubico OTP using secrets stored in the internal database.



In this setting, the YubiHSM also verifies and maintains the OTP counter values.

2.8 Web Server API compatible mode

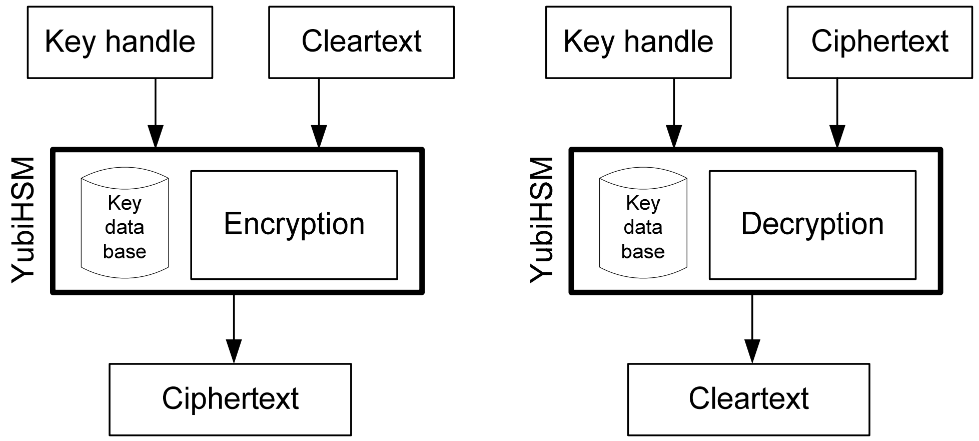
This mode allows full verification of received OTPs, where the secrets are stored internally. The data format and encoding is compatible with the Yubico Web Server API protocol, including secure hashing of requests and responses.



In this mode, no host application logic or database is required.

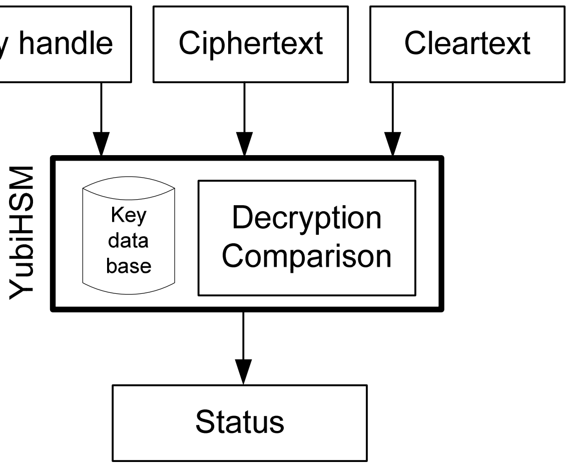
2.9 ECB mode encryption and decryption

The YubiHSM provides functions for arbitrary AES ECB encryption and decryption on a per-block basis.



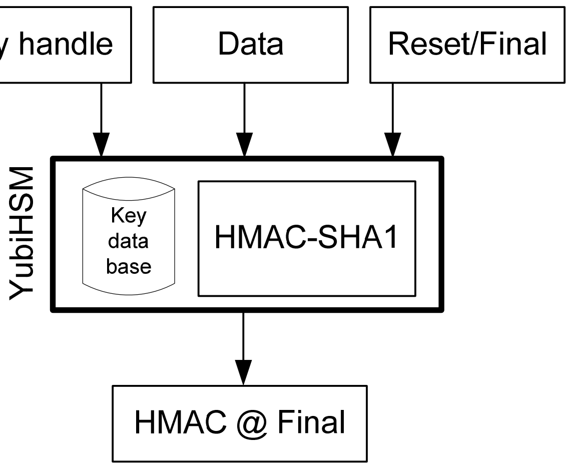
An additional function is provided to decrypt a ciphertext and internally compare with a supplied plaintext, returning the result of the comparison only.

This allows the decryption function above to be disabled in settings where it is not desired to allow generic decryption.



2.10 HMAC-SHA1 message hashing

The YubiHSM provide a function for HMAC-SHA1 hashing of a variable length input.



The reset/final bits control the state of the HMAC-SHA1 generation. The principle is as follows:

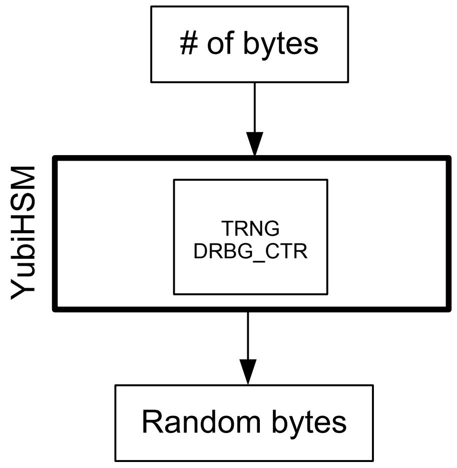
1. Write first block of data with the reset bit set
2. Write any number of data blocks with neither of the reset or final bits set
3. Write the final data block with the final bit set. This returns the HMAC-SHA1

When performing a HMAC-SHA1 on a block with a size of 64 bytes or less, both the reset and final bits shall be set. The HMAC is then returned with only one function call.

The generated 160-bit hash can either be returned or stored in the internal data buffer for subsequent AEAD operations.

2.11 Random number generation

The random number generation can be used to generate high quality random numbers.



The RNG is built on the DRBG_CTR scheme, described in NIST publication SP800-90. The DRBG is seeded using the on-chip TRNG on startup, and then continuously seeded using USB SOF jitter and further allows additional application initiated re-seeding.

2.12 HSM access control mechanisms

The YubiHSM provides two means of protecting access and use of AEAD key data.

- **Access control**
  If configured for this mode, the YubiHSM AEAD operations can be disabled by default at power up. A valid OTP must then be supplied to the YubiHSM in order to enable it.
- **Key store encryption**
  The non-volatile AEAD key store is encrypted using AES-256. In order to use the AEAD keys, the key store must be decrypted using a supplied key. If not specified during configuration, a default key is used and the key store will then be automatically decrypted at startup.

These functions are both optional and are used independently of each other.

2.13 Performance considerations

The in-system performance of the YubiHSM is dependent on several factors outside the device itself. The roundtrip time for sending a request from a host application through the operating system layers, through the USB stack and then over the USB bus and back again often takes longer time than the security operation itself.

AEAD operations as well as random number generation involve quite a few cryptographic operations (see section 4.1). Pre-expansion and caching of keys are done when a new key handle is selected. A 16-byte AEAD generation, comprising 5 AES encryption operations and some overhead takes about 650 µs without a cached key and about 550 µs with.

The USB-CDC stack is streamed by nature rather than block oriented, which means that performance can be degraded by data being sent as individual USB packets rather than full packets.

As a ball-park figure, a reasonably optimized OTP verification roundtrip in Windows with externally supplied AEAD takes about 1.0-1.2 milliseconds. A sustained throughput of at least 500 requests/seconds can be expected.

Operations involving updating of the internal database with OTP counter values are slower, adding a non-volatile memory write overhead of about 3-5 milliseconds.

WSAPI requests including write- and hashing overhead typically take about 5-6 milliseconds to complete.

A performance measurement tool is available, taking the entire roundtrip time into account. In order to optimize the throughput, an USB packet analyzer can be used to monitor the low-level transmission time and the actual processing time in the YubiHSM for various operations.

2.14 Security considerations

An obvious reflection of a HSM or HSM-like device like the YubiHSM is the physical protection of secrets stored inside. Can secret information be retrieved via the host port or is there a way to physically dissect the YubiHSM to gain access to memory contents? Can internal operations be "sniffed", by analyzing variations in current consumption or by measuring electromagnetic radiation?

Although the physical security is a part of the concept, it should be explicitly underlined that the main design objective for the YubiHSM is to protect symmetrical keys and other sensitive in transit and data stored on servers from being compromised by remote attacks.

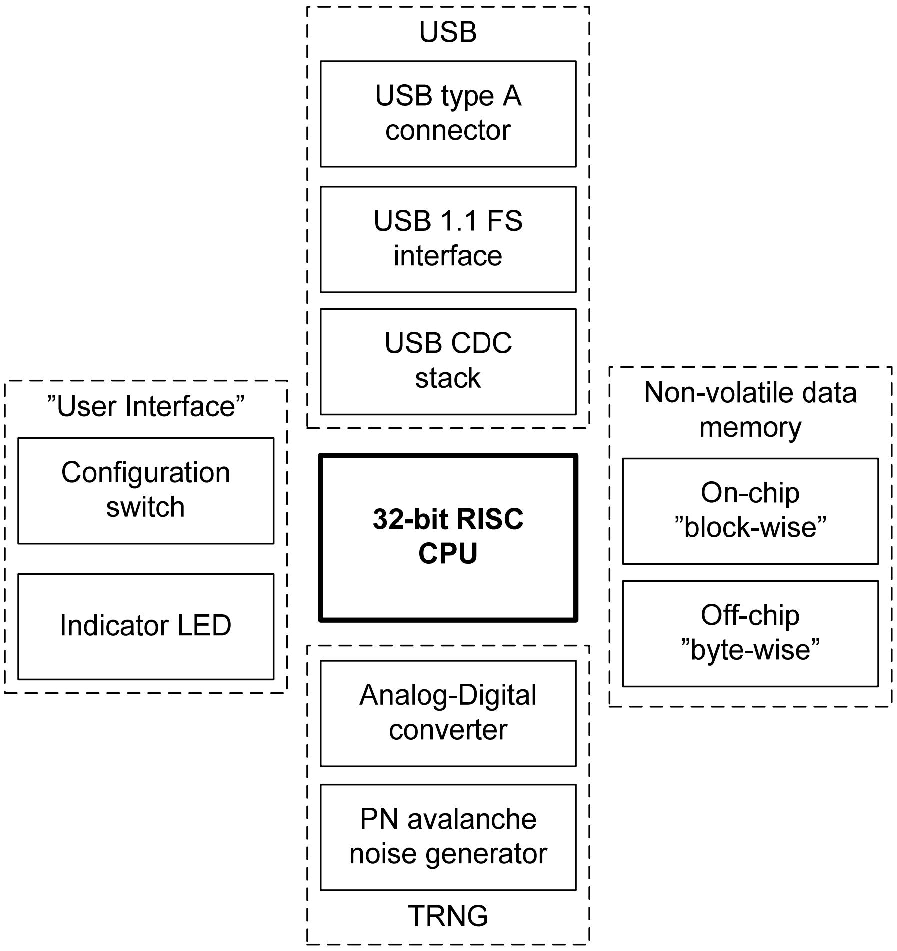
A deeper elaboration of this topic is beyond the scope of this document, but a few statements can be postulated:

- There is a very limited interface with a very small communication stack, supporting USB-CDC only.
- Malicious code and/or sensitive data cannot be uploaded or downloaded.
- Firmware update via USB is disabled for security reasons.
- The YubiHSM runs on a commercial embedded microcontroller without any specific security protection features, typically found in SmartCards and other security devices. Although there are code- and data protection features, there are typically no explicit statements from the manufacturers regarding the data protection.
- All sensitive data is stored in the processor on-chip non-volatile memory (FLASH).
- AEAD keys are stored in encrypted form in the on-chip FLASH using a master AES-256 key which needs to be supplied to start HSM mode of operation.
- Although very complicated and sophisticated, it is not denied that there are methods to strip off and locate individual bits of memory in a laboratory.
- The Basic version provides no physical protection. The Pro version is potted into a steel tube with reinforcements.
- A device with key storage in RAM with battery backup may be considered for certain applications in the future. Tamper detection circuits will be integrated that if affected, will erase the keys.

As a kind of final word on this subject, the reader may wish to bear in mind the practical and theoretical attacks in this realm must be soberly considered both rationally and practically and should neither be exaggerated nor neglected. The intention with YubiHSM is not the right product for all authentication needs, but to provide the most cost efficient vs. security compromise consistent with the YubiKey philosophy.

3 Block diagram

The high-level functional blocks of the YubiHSM are as follows:



USB subsystem
Communication with the host computer utilizes a full-speed (12 Mbit/s) USB 1.1 interface with a USB CDC stack.

Non-volatile data storage
Configuration-, AEAD- and key data is stored in two different non-volatile memory blocks. Security critical information is stored on-chip whereas counters for OTP validation are stored in a byte-wise off-chip memory.

Random number generator
A hardware PN-avalanche noise generator and USB jitter clock is implemented, which is used to seed a DRBG_CTR random number generator.

User interface
A switch and a LED are the only elements of the “user interface”.

4 Cryptographic internals

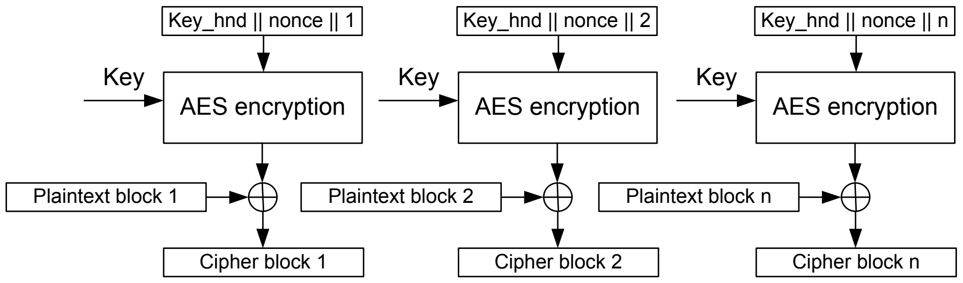
The purpose of this section is to describe the cryptographic internals, algorithms and rationales behind the secure operations. For command reference and usage, please refer to sections 5 and 6.

4.1 AEAD encryption and decryption

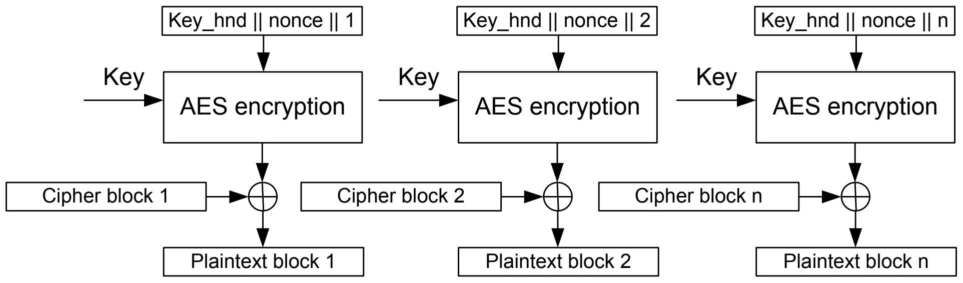
AEADs are encrypted and authenticated to provide both confidentiality and authenticity. The scheme used is AES-CCM as described in RFC 3610 with 128-, 192- or 256-bit key length. Please refer to RFC 3610 for a detailed description of AES-CCM. A basic primer is included below for reference.

Encryption and decryption is done using AES-CTR mode with a variable number of bytes, not limited to the fixed block length (flag fields omitted for clarity):

Encryption stage (CTR mode)



Decryption stage (CTR mode)



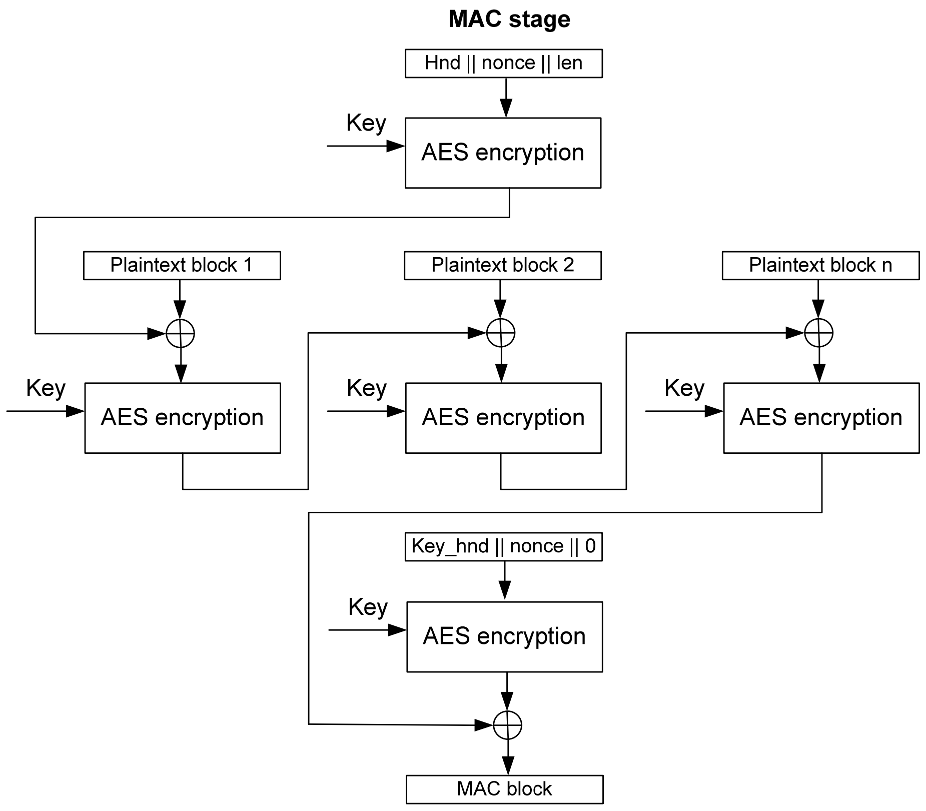
Important in AES-CTR mode is that two different plaintext messages must never be encrypted using the same key and nonce as this instantly compromises the confidentiality.

AES-CTR input block layout

AES-CTR input block layout

Flags (1 byte)	Key handle (4 bytes)	Nonce (6 bytes)	RFU (zeroes) (3 bytes)	Counter (2 bytes)

As encryption only ensures confidentiality, a MAC is included to provide authenticity.



The MAC block is truncated to 8 bytes and appended to the AES-CTR ciphertext, forming an AEAD block with a total length of payload + 8 bytes.

4.2 Generating an AEAD

AEADs can either be generated from clear text data provided externally, from random data generated internally or a combination of both. This approach allows maximum flexibility for combinations of secret key data that must never be revealed outside the YubiHSM and optional additional data associated with the AEAD.

Different primitives are implemented to support various scenarios:
(See section 6 for a detailed description of commands)

- **Generate an AEAD from random data for a single target**
  A given number of random bytes are generated internally and then encrypted to a single AEAD using a provided nonce and key handle. This operation can be done with the atomic function `YSM_RANDOM_AEAD_GENERATE` where no data is buffered for subsequent calls.

- **Generate an AEAD from external fixed data for a single target**
  A single AEAD is created using provided data, nonce and key handle. This operation can be done with the atomic function `YSM_AEAD_GENERATE` where no data is buffered for

subsequent calls. If it is required to allow for a user specified nonce (which might be a security risk), the keyhandle must have permission flag `YSM_USER_NONCE` set.

- **Generate AEADs from random data for multiple targets**
  A shared secret is first created by generating a given number of random bytes with the `YSM_BUFFER_RANDOM_LOAD` function internally in the YubiHSM. Subsequent AEAD creation calls with the `YSM_BUFFER_AEAD_GENERATE` function then creates AEADs for different targets, using the same shared random data but with their respective key handle. Nonces may be common or unique on a per-target basis. If it is required to allow for a user specified nonce (which might be a security risk), the keyhandle must have permission flag `YSM_USER_NONCE` set. Note that generating AEADs from random data generated internally does *not* require the `YSM_BUFFER_LOAD` permission flag on the keyhandle.

- **Generate AEADs from fixed- or mixed random/fixed data**
  AEAD data in this case is generated with `YSM_BUFFER_RANDOM_LOAD` function calls and/or loaded with the `YSM_BUFFER_LOAD` function calls into the shared data buffer of the YubiHSM on a per-target basis. AEAD(s) are then created with subsequent calls to the `YSM_BUFFER_AEAD_GENERATE` function. If it is required to allow for a user specified nonce (which might be a security risk), the keyhandle must have permission flag `YSM_USER_NONCE` set. Note that generating AEADs from random data generated internally does *not* require the `YSM_BUFFER_LOAD` permission flag on the keyhandle, while the `YSM_BUFFER_LOAD` flags *is* required if fixed data is loaded into the YubiHSM internal buffer. This requirement applies to HMAC-SHA-1 checksums generated into the internal buffer as well.

4.3 AEAD decryption

The YubiHSM intentionally does not provide any functions that decrypts an AEAD and returns it in clear text, either fully or partial.

Note that this refers to the command set of the YubiHSM, not necessarily including the cryptographic properties of generated AEADs. Two ways to effectively decrypt a generated AEAD (both a natural consequence of the design of AES CCM) is by using either the `AES_ECB_BLOCK_ENCRYPT` operation, or by asking the YubiHSM to generate another AEAD (with the `YSM_AEAD_GENERATE` function) based on an existing AEAD. The permission flags on all keyhandles should always be kept at a minimum. See section 4. AEAD generation for further details of safe and unsafe permission flags.

The implicit function `YSM_AEAD_DECRYPT_CMP` is however provided that does an internal decryption and comparison with a supplied clear text. Only the result of the comparison is returned to the host application.

4.4 Nonce usage and generation

Critical to the security of AES-CCM is a correct usage of a unique nonce / key handle for each AEAD being generated. In the case of Yubikey OTP, it is assumed that the public id used as nonce has only one single representation of its key/private id for each target key handle.

When AEADs are used in a generic setting, it is therefore important to ensure that the used nonce is unique. If there is no “natural” link to an associated unique property, supplying a zero nonce to any AEAD generation function causes the YubiHSM to generate a unique one.

Supplying a non-zero nonce to the AEAD generating functions requires the keyhandle to be explicitly configured to allow user supplied nonces through the `YSM_USER_NONCE` permission bit.

The YubiHSM has a non-volatile 16-bit counter that is incremented by one for each power-up event. Together with a RAM-based 32-bit counter which is cleared at power-up and incremented by one for each nonce being generated, this ensures a unique nonce.

Any given number of nonces can be generated by the host application by an explicit call to the `YSM_NONCE_GET` function. The volatile counter is then incremented with the argument given in the function call.

If the volatile counter would wrap, the non-volatile counter is automatically incremented.

4.5 Using AEADs to decode/validate Yubico OTPs

AEADs created with a specific format may be used to decode and validate Yubico OTPs.

The core of the secret in a Yubico OTP is the private ID and AES key and an AEAD holding these can be used to validate and decode an OTP without revealing any part of the secret.

```
typedef struct  {

      uint8_t key[KEY_SIZE];        // AES  key

      uint8_t uid[UID_SIZE];        // Unique  (secret)  ID

} YUBIKEY_SECRETS;
```

These 6 + 16 = 22 bytes when encoded into a 22 + 8 bytes MAC = exactly 30 bytes AEAD can be used to the AEAD OTP decode function `YSM_AEAD_YUBIKEY_OTP_DECODE`.

Conceptually, the function is performed as follows:

1. The YubiHSM receives a `YSM_AEAD_YUBIKEY_OTP_DECODE` command followed by a 4 byte key handle n, a 30-byte AEAD block, a 6-byte public ID and a 16-byte OTP.
2. The AEAD block is decrypted and verified. If the integrity check fails, the request fails with status code `YSM_AEAD_INVALID`
3. The OTP is decrypted using AES-128, using the decrypted key
4. The CRC16 value of the decrypted OTP is verified. If this verification fail, the request fails with status code `YSM_OTP_INVALID`
5. The private id (UID) is verified from the decrypted AEAD block. If this verification fail, the request fails with status code `YSM_OTP_INVALID`
6. The YubiHSM responds with a function `YSM_AEAD_YUBIKEY_OTP_DECODED` response and status code `YSM_STATUS_OK`. Given that (3, 4, 5, 6) all pass, the useCounter, sessionCounter and timestamp fields are returned. On failure in any of the steps, the returned values are all zeroes.

An external database application then has to verify the useCounter and sessionCounter has to be verified and maintained properly to prevent OTP replay. Optionally, the timestamp field can be verified as well.

4.6 AEAD load to internal storage

The YubiHSM allows a limited number of AEADs to be stored internally, thereby allowing Yubico OTP verification without the need for external logic or storage as even the counters are validated and maintained.

1. The YubiHSM receives a `YSM_DB_YUBIKEY_AEAD_STORE` (or a `YSM_DB_YUBIKEY_AEAD_STORE2`)command followed by a 4 byte key handle n, a 6-byte public ID and a 30-byte AEAD.
2. The YubiHSM decrypts and verifies the MAC of the AEAD using the supplied key handle. If the AEAD is valid, the private id and key are stored, otherwise the error code `YSM_AEAD_INVALID` is returned.
3. The YubiHSM responds with `YSM_STATUS_OK`

The `YSM_DB_YUBIKEY_AEAD_STORE2` command allows for AEADs generated with a nonce different from the public id, by allowing the nonce to be supplied separately.

4.7 OTP verification using internal storage

The YubiHSM can fully validate an OTP if the associated AEAD has been stored in the internal memory as described in section 4.6.

1. The YubiHSM receives a `YSM_DB_YUBIKEY_OTP_DECODE` command followed by a 6-byte public ID and a 16-byte OTP
2. The internal database is searched for a matching 6-byte public ID.
3. The OTP is decrypted using AES-128 with the stored key
4. The CRC16 value of the decrypted OTP is verified
5. The private id (UID) is verified against the stored value
6. The YubiHSM responds with `YSM_STATUS_OK` status code if the verification passes. Given that (3, 4, 5) all pass, the useCounter, sessionCounter and timestamp fields are returned. On failure in any of the steps, the status returned is `YSM_OTP_INVALID` and the values are all zeroes.

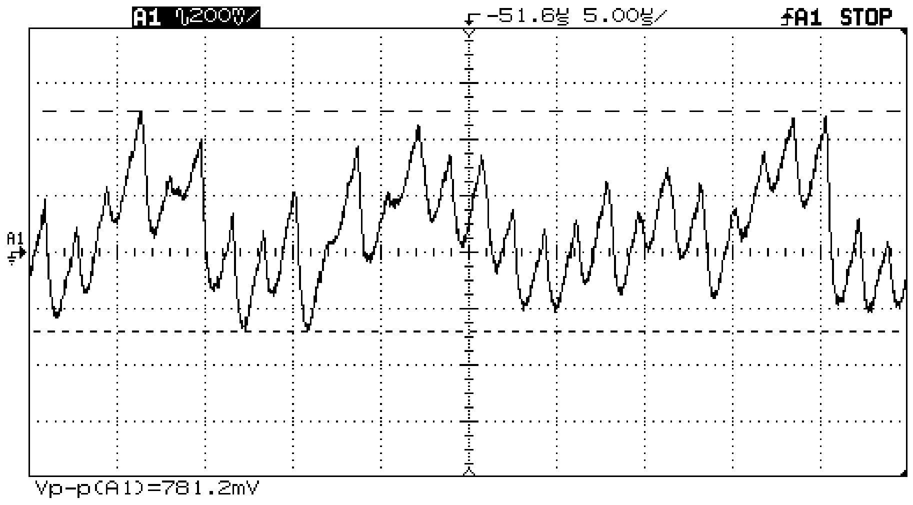
4.8 Random number generation

The random number generator comprises two steps

Seed generation

The first step is to generate a seed for the random number generator. This is done by applying a voltage over a reversed PN-junction in an ordinary silicon transistor. At a certain voltage, around 10-12 volts, a large amount of noise, known as avalanche noise starts to appear. This noise is amplified by a common emitter stage and then the DC bias is removed.

Typical avalanche noise looks like (200mV/div, 5µs/div):



The PN bias voltage is adjusted in small steps until the noise reaches a steady and high (but not too high = avoid clipping) amplitude. This allows optimal noise, independent on device- and temperature variations.

Then a fixed DC bias be 50% of the analog-digital converter span is added to avoid saturation.

The 12-bit converter has a reference voltage of 2.5V and with a noise amplitude as in this example of 780 mV, the span of the samples will be approximately 4095 * 0.78 / 2.5 = 1280.

32 sequential samples are taken, which are then used as seeds to a following DRBG_CTR operation. These operations are repeated 100 times while updating the DRBG_CTR.

It is always difficult, if not impossible to prove the quality and robustness of a hardware random number source, but given the dynamic bias approach, a 1.5µs conversion rate, > 500mV noise unrelated to the sampling rate, and 0.6mV/bit together with the properties of DRBG_CTR, the entropy of the random numbers are high.

The output of the RNG can be monitored in monitor mode. See section 8.21 for details.

Random number generation

Instead of providing proof for a continuously performing hardware TRNG, the known property of DRBG_CTR is used. Once the seed has been generated by the hardware above, this is used as the 256-bit seed for subsequent DRBG_CTR operations.

The RNG is continuously seeded with USB SOF jitter sampled with a 72 MHz counter. If desired, the counter can be explicitly re-seeded with a host provided seed.

5 HSM mode of operation

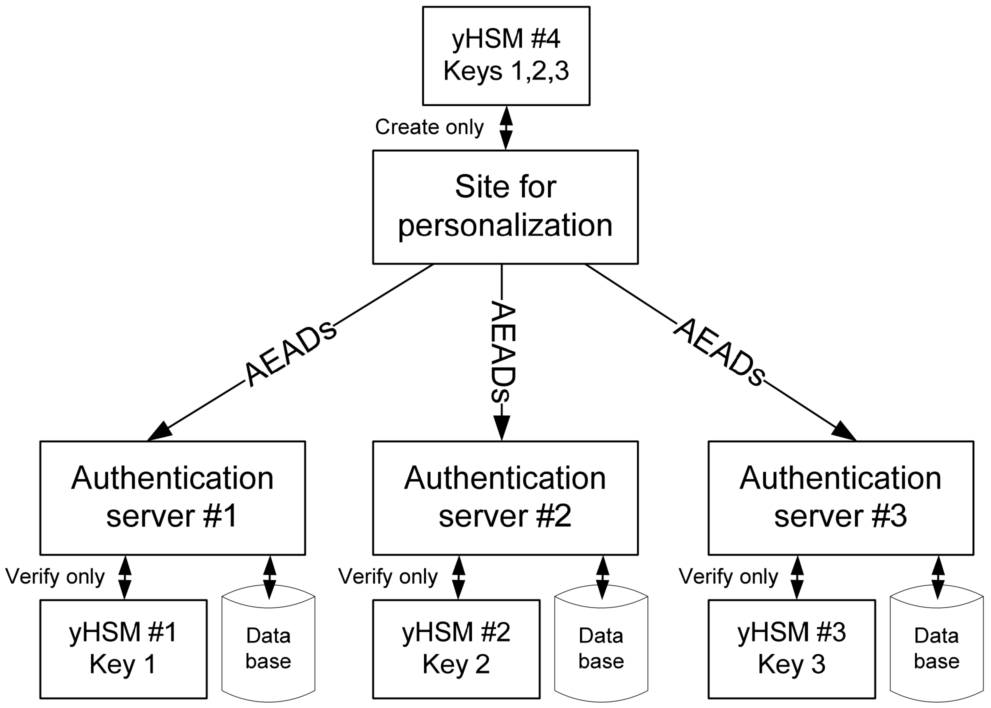
Each YubiHSM HSM command is here described in detail. Each data structure can be found in the **`YubiHSM_if.h`** header file.

Structures need to be packed according to the smallest data element present, i.e. 1 byte. Some compilers require a specific `#pragma` directive for setting the structure packing and data member alignment

To setup the YubiHSM for HSM operation, please refer to section 8.2.

5.1 Multi-device settings and the Flag word

A hypothetical scenario in a HSM mode setting may look like



Here, AEADs are generated by YubiHSM 4 and by command of the host there, each AEAD targeted for server 1, 2 and 3 are each encrypted with key 1, 2 and 3 respectively.

To prevent the devices at authentication servers 1-3 from generating AEADs, thereby "impersonating" the real site for personalization, each key can control which functions are allowed to use it. In the case above, devices 1-3 will then have the `YSM_AEAD_YUBIKEY_OTP_DECODE` flag enabled only whereas the device 4 will have any combination of the generation flags set only.

Flag bits can also be defined for the entire device at time of configuration. In order to use a specific key, both the device-wide flag and the key flags must be set.

5.2 Protocol arbitration and "de-streaming"

The YubiHSM communication is done in a request-response fashion where the host sends a `YSM_XXX_REQ` command and the YubiHSM responds with a `YSM_XXX_RESP` command. No ad-hoc information is sent from the YubiHSM while in HSM mode.

Serial protocols as the USB-CDC tend to be byte oriented whereas the YubiHSM HSM mode handles data units as blocks. Therefore, careful design is required not to lose synchronization in a serial byte stream.

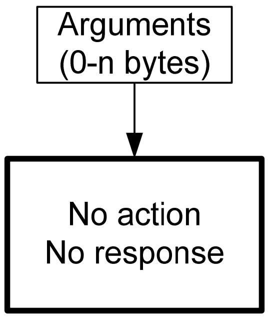
1. Make sure all host response data is flushed prior to a host request being sent
2. The first byte in a data packet specifies the number of bytes to follow (command + payload)
3. All response commands have response data that can and shall be verified against expected values.
4. Implement a transaction time-out to handle potential out-of-sync conditions / no response (`YSM_NULL` being the exception). A re-synchronization action is then needed.
5. Consider the typical non-realtime performance of the host so although all operations are completed well within 10 ms, including communication overhead, a longer time-out may be needed.
6. Make sure no other serious "bandwidth eaters" are present on the same host hub. As the USB CDC is built on USB bulk transfers, starvation may occur otherwise.
7. The maximum USB bulk packet length is 64 bytes. In order to maximize performance and minimize communications overhead, try to write transactions in blocks, thereby allowing the host operating system to group the transfers into as full packets as possible.

5.3 HSM mode command description

Below is a list of implemented commands. For a detailed description of commands, parameters and data, please refer to the `YubiHSM_if.h` file.

5.4 YSM_NULL

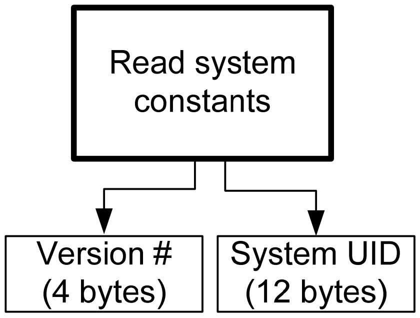
The NULL command does what it states – nothing, including not sending a response. This can be used to reset the protocol if needed.



By filling up a complete packet (defined size of up to `YSM_MAX_PKT_SIZE` bytes) of `YSM_NULL` bytes, the YubiHSM is ensured to get back in sync if communication has been lost, independent of if or where the synchronization was lost.

5.5 YSM_SYSTEM_INFO_QUERY

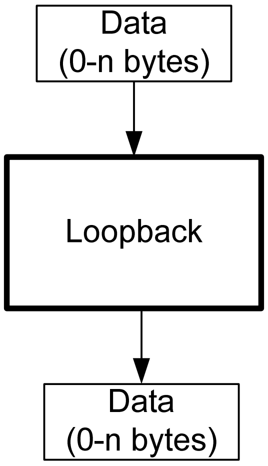
The `YSM_SYSTEM_INFO_QUERY` query returns information about the YubiHSM



The version number and protocol version defines the installed firmware whereas the system UID is factory programmed unique CPU identity.

5.6 YSM_ECHO

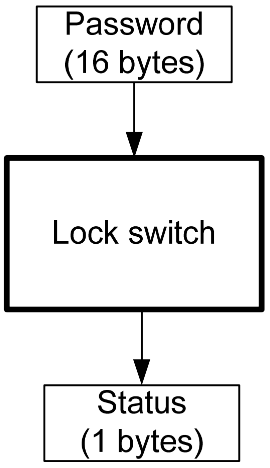
The `YSM_ECHO` command can be used to test the roundtrip performance. By sending an ECHO command, the YubiHSM responds with an identical packet with almost no execution overhead.



The application execution time for the ECHO command therefore gives the shortest roundtrip time that can be expected. The execution time for any other `YSM_XXX` command can therefore be said to be equal to the roundtrip time for that command subtracted with the roundtrip time for an ECHO command with the same data payload.

5.7 YSM_KEY_STORAGE_UNLOCK (Version 0.x only)

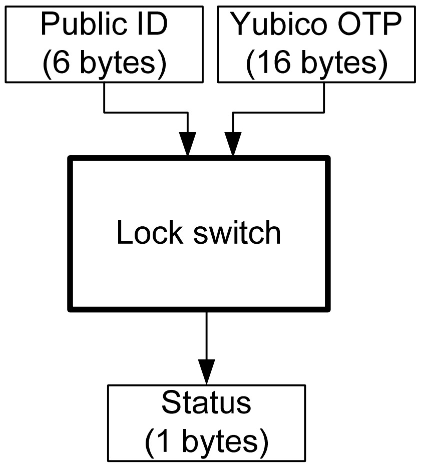
The `YSM_KEY_STORAGE_UNLOCK` command enables all operations using a key handle if a HSM configuration password is set. This function has been removed from firmware version 1.0 and is superseded by the `YSM_HSM_UNLOCK` (5.8) and `YSM_KEY_STORE_DECRYPT` (5.9) commands



If a HSM password is set, the lock switch is set to“locked” when the YubiHSM is powered up and a valid password must be written to unlock the switch, otherwise all key handle related operations will unconditionally return `YSM_KEY_STORAGE_LOCKED`.

5.8 YSM_HSM_UNLOCK

The `YSM_HSM_UNLOCK` command is used to enable HSM mode of operation using an OTP from one of the optionally pre-stored administrator keys assigned at configuration time.

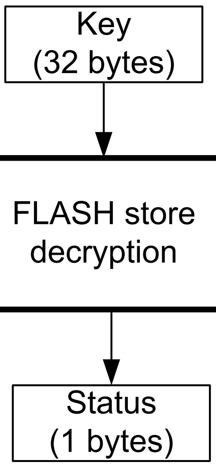


The YubiHSM uses the internal key database to verify the OTP. A positive OTP verification enables the YubiHSM AEAD operations. A negative OTP verification disables AEAD operations and returns `YSM_KEY_STORAGE_LOCKED`.

Note that this function operates independently of the `YSM_KEY_STORE_DECRYPT` command (5.9).

5.9 YSM_KEY_STORE_DECRYPT

The `YSM_KEY_STORE_DECRYPT` command is used to decrypt and verify the AEAD key store in FLASH over to volatile RAM storage.

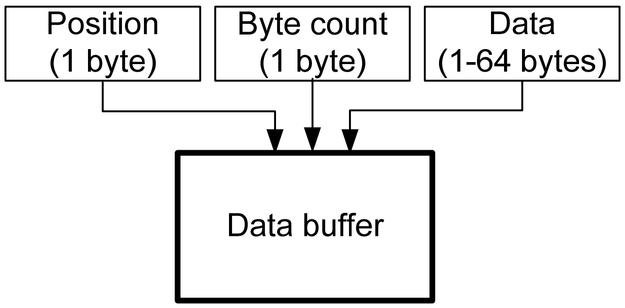


Note that this function operates independently of the `YSM_HSM_UNLOCK` command (5.8).

5.10 YSM_BUFFER_LOAD

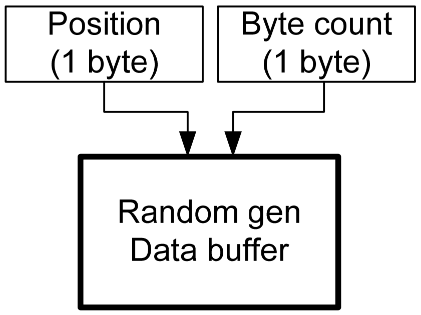
The `YSM_BUFFER_LOAD` command loads an arbitrary data block into a specific location of the AEAD data buffer. Writing to position 0 clears the entire buffer first.

If the data loaded into the buffer is to be used to generate an AEAD, the keyhandle used in the AEAD generation operation must have the YSM_BUFFER_LOAD flag set. If it does not, the AEAD generation command will fail with YSM_FUNCTION_DISABLED if the data in the buffer was loaded using YSM_BUFFER_LOAD.



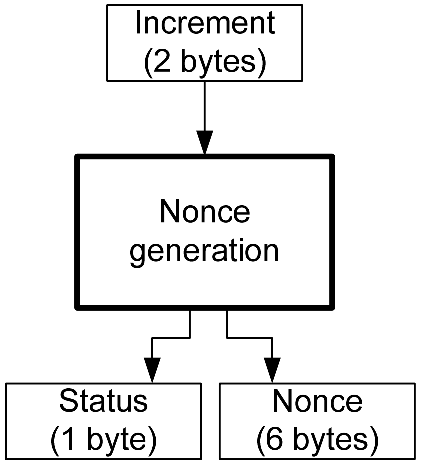
5.11 YSM_BUFFER_RANDOM_LOAD

The `YSM_BUFFER_RANDOM_LOAD` command generates a given number of random bytes into a specific location of the AEAD data buffer. Writing to position 0 clears the entire buffer first.



5.12 YSM_NONCE_GET

The `YSM_NONCE_GET` command returns the current unique nonce and then increments the nonce counter with a specified number.

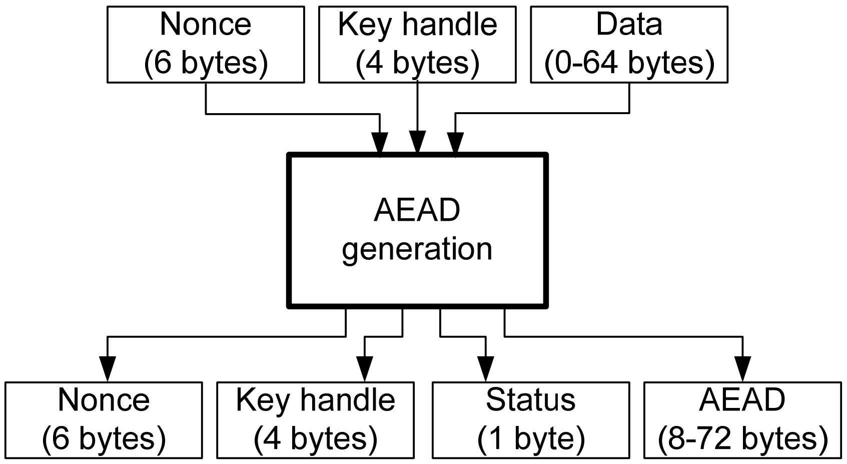


This approach allows the host application to read out the current nonce and then allocate a "pool" of nonces without having to make subsequent calls to the `YSM_NONCE_GET`.

Normally, an application which needs nonces to be generated would specify a null nonce to AEAD generation functions, thereby causing automatic unique nonce generation.

5.13 YSM_AEAD_GENERATE

The `YSM_AEAD_GENERATE` command generates an AEAD based on a specified data block.

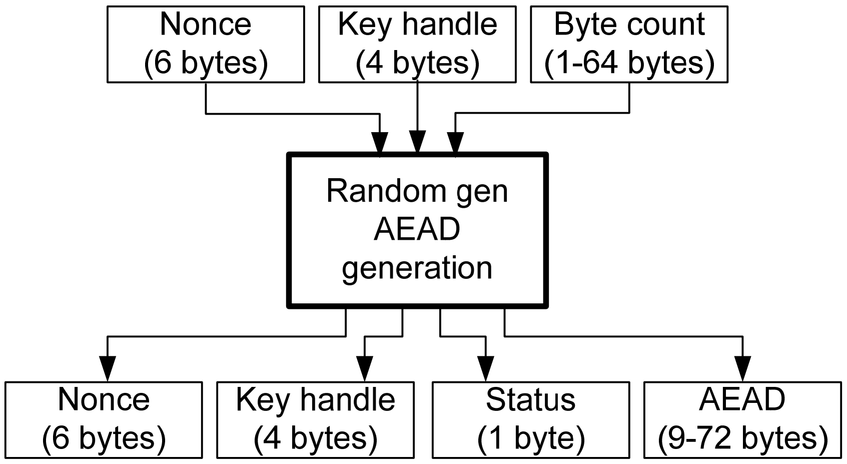


If the nonce is all zero, the system will generate a unique nonce which is returned in the response. The internal data buffer is flushed after the call.

A non-zero nonce will be rejected with YSM_FUNCTION_DISABLED unless the keyhandle used has the permission flag YSM_USER_NONCE set.

5.14 YSM_RANDOM_AEAD_GENERATE

The `YSM_RANDOM_AEAD_GENERATE` command generates an AEAD based on an internally generated random block with a specified number of bytes.

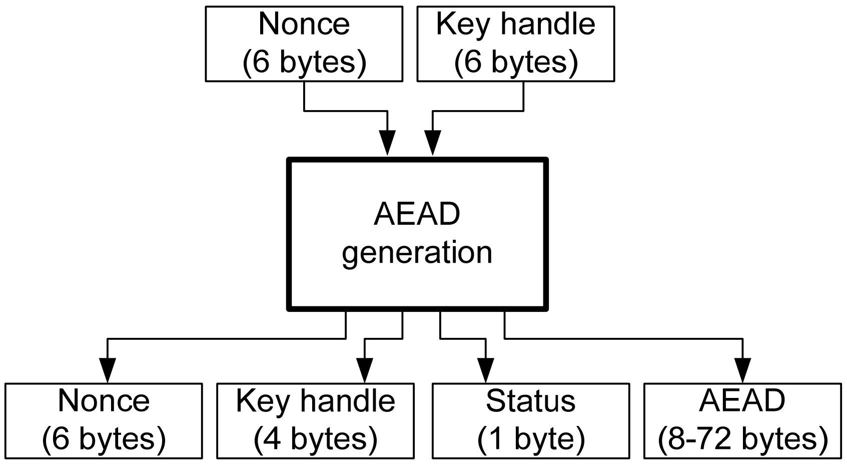


If the nonce is all zero, the system will generate a unique nonce which is returned in the response. The internal data buffer is flushed after the call.

A non-zero nonce will be rejected with YSM_FUNCTION_DISABLED unless the keyhandle used has the permission flag YSM_USER_NONCE set.

5.15 YSM_BUFFER_AEAD_GENERATE

The `YSM_BUFFER_AEAD_GENERATE` command generates an AEAD based on the current contents of the AEAD data buffer, loaded with prior calls to `YSM_BUFFER_LOAD` and `YSM_BUFFER_RANDOM_LOAD`.

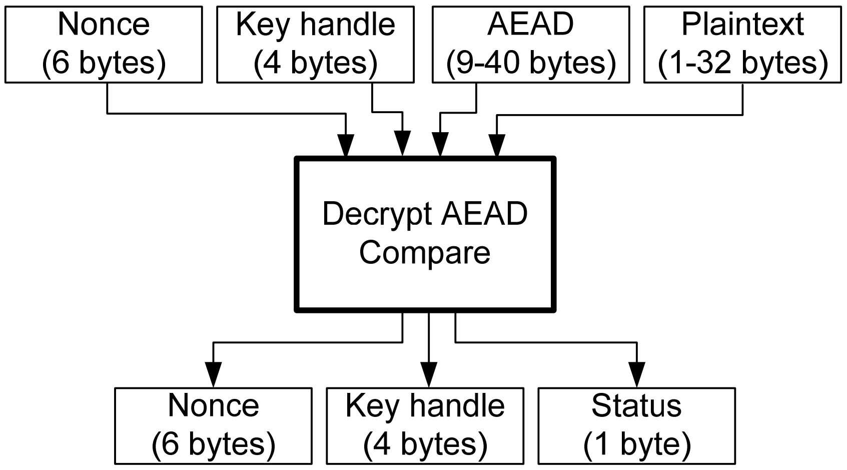


If the nonce is all zero, the system will generate a unique nonce which is returned in the response. The internal data buffer is not flushed after the call, allowing multiple calls with different nonces, key handles as well as a partially changed data buffer.

A non-zero nonce will be rejected with YSM_FUNCTION_DISABLED unless the keyhandle used has the permission flag YSM_USER_NONCE set.

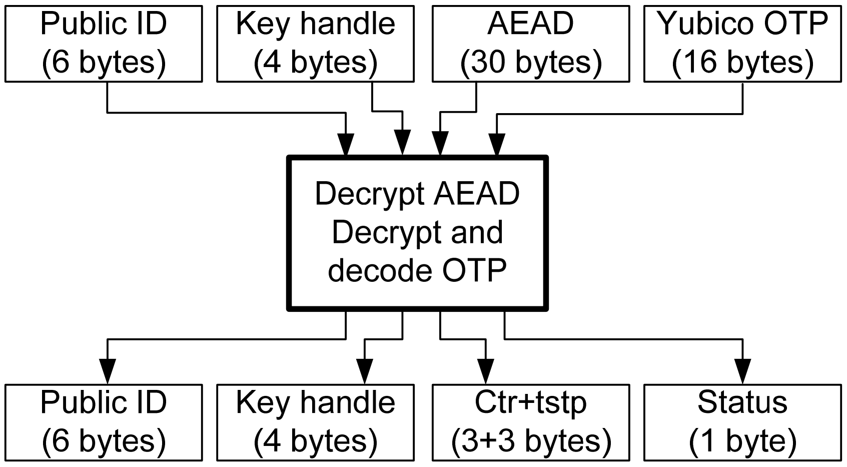
5.16 YSM_AEAD_DECRYPT_CMP

The `YSM_AEAD_DECRYPT_CMP` command internally decrypts a given AEAD and compares the result with a given plaintext block.



5.17 YSM_AEAD_YUBIKEY_OTP_DECODE

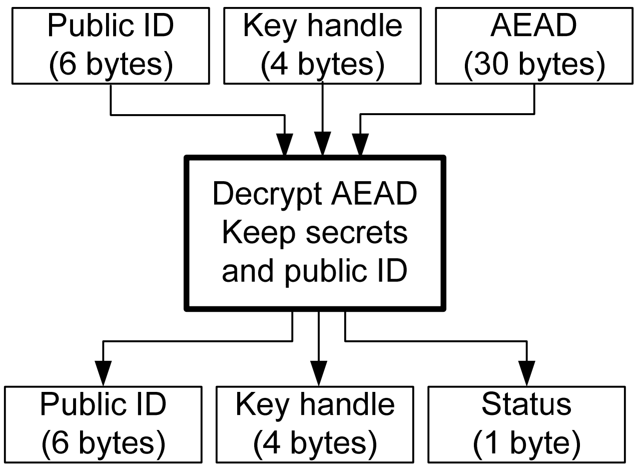
The `YSM_AEAD_YUBIKEY_OTP_DECODE` command decodes a supplied Yubikey secrets formatted AEAD which is then used to decode a supplied Yubico OTP.



Note that although the function decodes and verifies the OTP, the counter and timestamp fields have to be verified by the host application

5.18 YSM_DB_YUBIKEY_AEAD_STORE

The `YSM_DB_YUBIKEY_AEAD_STORE` command stores a Yubikey secrets formatted AEAD in the internal persistent store.

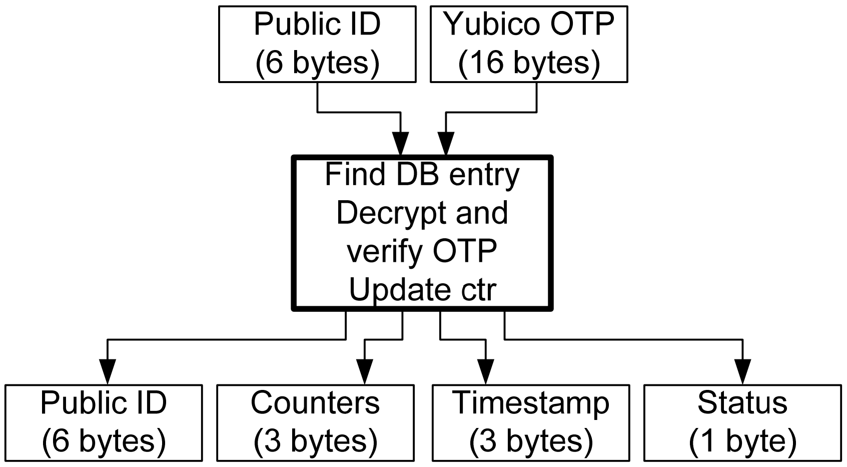


5.19 YSM_DB_YUBIKEY_AEAD_STORE2

The YSM_DB_YUBIKEY_AEAD_STORE2 command stores a Yubikey secrets formatted AEAD in the internal persistent store, with the ability to supply a nonce that is different than the YubiKey public id.

5.20 YSM_DB_YUBIKEY_OTP_VALIDATE

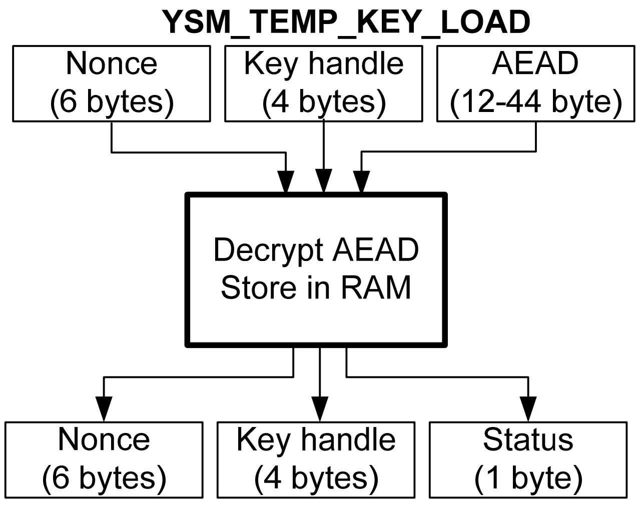
The `YSM_DB_YUBIKEY_OTP_VALIDATE` command verifies a supplied Yubico OTP using the internal persistent data store. The function also verifies the counter values with stored persistent values to prevent OTP replay.



The internal database is searched for the supplied public ID and if found, the OTP is decrypted and verified using the key in the database. If successful, the system returns `YSM_STATUS_OK` with the values of the counters- and timestamp fields. Then, the internal database counter registers are updated.

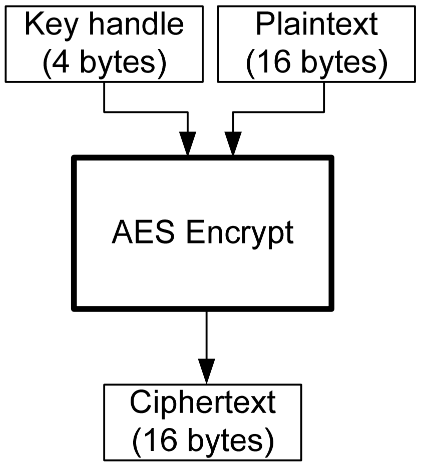
5.21 YSM_TEMP_KEY_LOAD

The `YSM_TEMP_KEY_LOAD` command decrypts a specified AEAD holding a key + 32-bit flag word and stores this as the temporary key in RAM. When loaded, the `TEMP_KEY_HANDLE` can be used just like any other key handle. By sending a null AEAD, the temporary key is erased.



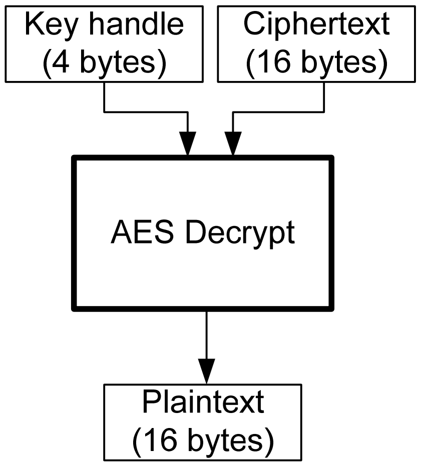
5.22 YSM_AES_ECB_BLOCK_ENCRYPT

The YSM_AES_ECB_BLOCK_ENCRYPT command encrypts a single plaintext block.



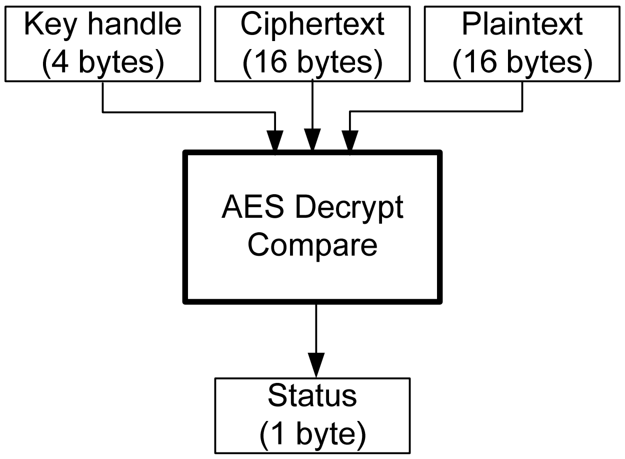
5.23 YSM_AES_ECB_BLOCK_DECRYPT

The YSM_AES_ECB_BLOCK_DECRYPT command decrypts a single ciphertext block.



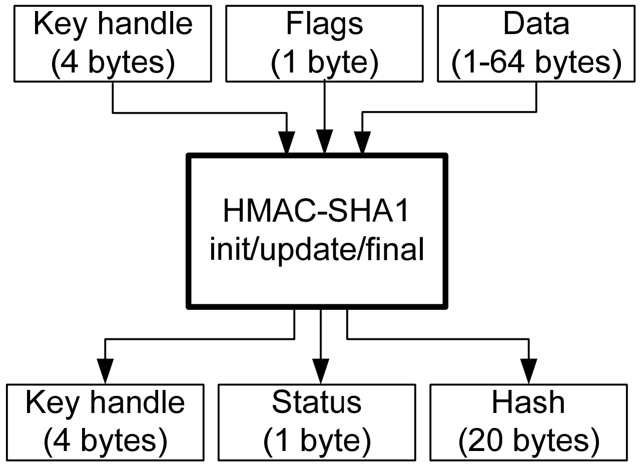
5.24 YSM_AES_ECB_BLOCK_DECRYPT_CMP

The YSM_AES_ECB_BLOCK_DECRYPT_CMP command decrypts a single ciphertext block and then internally compares the result with a provided plaintext. Only the result of the comparison is returned.



5.25 YSM_HMAC_SHA1_GENERATE

The `YSM_HMAC_SHA1_GENERATE` command creates a HMAC-SHA1 hash from data of arbitrary length.



The function allows an unlimited amount of data in the written

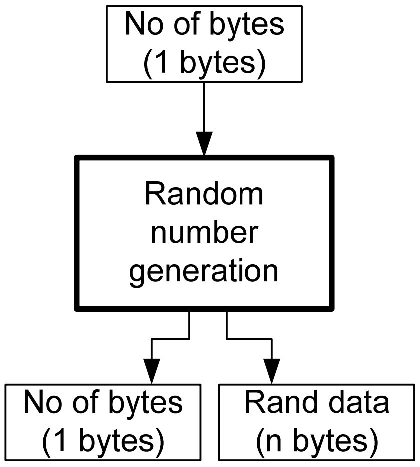
1. Write first block with flags set to `YSM_HMAC_SHA1_RESET`
2. Write any number of data blocks with flags set to zero
3. Write the final data block with flags set to `YSM_HMAC_SHA1_FINAL`

When the data to be hashed is 64 bytes or less, set the flags to `YSM_HMAC_SHA1_RESET | YSM_HMAC_SHA1_FINAL`

By setting the `YSM_HMAC_SHA1_TO_BUFFER` flag, the HMAC is written to the AEAD data buffer rather than being returned when the `YSM_HMAC_SHA1_FINAL` bit is set.

5.26 YSM_RANDOM_GENERATE

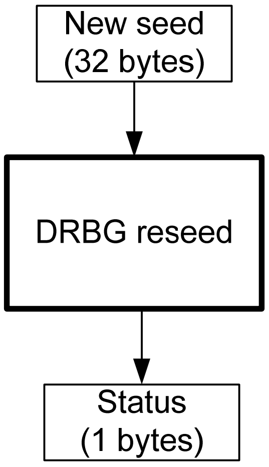
The `YSM_RANDOM_GENERATE` command uses the random number generator to generate a block of random bytes.



The random data is generated in 16-bytes blocks andis truncated if necessary.

5.27 YSM_RANDOM_RESEED

The `YSM_RANDOM_RESEED` command performs an explicit re-seed of the DRBG_CTR, using the provided seed material.

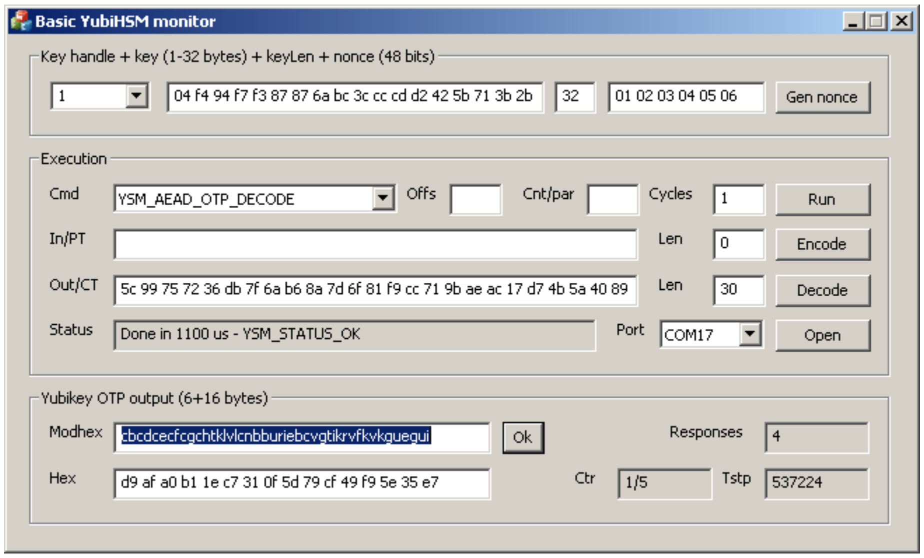


5.28 YSM_MONITOR_EXIT

This command is only available if the system wide flags have the debug flag set. It provides a way to exit HSM mode and enter configuration mode remotely. Naturally, the debug flag should never be enabled in a production setting.

5.29 Using the Windows HSM monitor

A basic test tool for evaluating the HSM functionality is available.


Basic YubiHSM monitor

Key handle + key (1-32 bytes) + keyLen + nonce (48 bits)

1 | 04 f4 94 f7 f3 87 87 6a bc 3c cc cd d2 42 5b 71 3b 2b | 32 | 01 02 03 04 05 06 | Gen nonce

Execution

Cmd: YSM_AEAD_OTP_DECODE | Offs: | Cnt/par: | Cycles: 1 | Run

In/PT: | Len: 0 | Encode

Out/CT: 5c 99 75 72 36 db 7f 6a b6 8a 7d 6f 81 f9 cc 71 9b ae ac 17 d7 4b 5a 40 89 | Len: 30 | Decode

Status: Done in 1100 us - YSM_STATUS_OK | Port: COM17 | Open

Yubikey OTP output (6+16 bytes)

Modhex: cbcdcecfcgchtklvlcnbburiebcvgtikrvfkvkguegui | Ok | Responses: 4

Hex: d9 af a0 b1 1e c7 31 0f 5d 79 cf 49 f9 5e 35 e7 | Ctr: 1/5 | Tstp: 537224

The tool further comprises time measurements, which can be used to verify throughput and round-trip times.

6 WSAPI mode of operation

The WSAPI mode implements a subset of the Yubico Web Service API for OTP validation, allowing easy migration to and froman existing WSAPI settings. Instead of routing the request to a validation service URL, the request is simply sent using serial communication routines to the YubiHSM, which also responds in a WSAPI compatible manner.

The communication protocol is entirely text based with commands and responses being terminated with carriage return. Aswith HSM mode, the protocol is based on a request-response scheme where a request is sent to the YubiHSM, which then sends a response string.Data is never sent ad-hoc.

The WSAPI mode supports multiple client IDs with individual shared secrets, allowing the YubiHSM to serve multiple clients.

A detailed description of the WSAPI can be found inthe Web Service API 2.0 reference.

6.1 Loading the WSAPI secrets and identities

The shared secrets and the Yubikey identities must be loaded into the YubiHSM internal database prior to sending authentication requests. Once the configuration mode has been left, the WSAPI supports authentication requests only.

To setup the YubiHSM for WSAPI operation and how to load keys and the identity database, please refer to section 8.5.

6.2 Sample request without client ID

The simplest request is to verify an OTP without a nonce or client ID.

This can easily be tested in a terminal program. Open the YubiHSM and type `otp=` and then output an OTP from a configured Yubikey which has been loaded into the YubiHSM database.

```
otp=cccccccbjhjvjjdbduhcuuikkgerkkvbffebktvdlubj
```

```
otp=cccccccbjhjvjjdbduhcuuikkgerkkvbffebktvdlubj&status=OK
```

The OTP and counters are validated and updated accordingly

6.3 Sample request with client ID

Assume the recently updated client id 5075 is used with its associated shared secret.

A complete sample request would then have the following items:

Client ID	5075
Shared secret	PXBv6EcyyX9zfopo8mynaPbEi5Y=
Sample nonce	BKCqmmPYnszTmjopHDHa
Sample OTP	Cccccccbjhjvejkldfuvcrurlefjccecbrnvidibcifr

HMAC hash	cgc/FnyquoSEJ9yRKaXuVh4eaFg=

The resulting request string would then look like

```
id=5075&nonce=BKCqmmPYnszTmjopHDHa&otp=cccccccbjhjvejkldfuvcrurlefjccec
brnvidibcifr&h=cgc/FnyquoSEJ9yRKaXuVh4eaFg=
```

The response will then be

```
id=5075&nonce=BKCqmmPYnszTmjopHDHa&otp=cccccccbjhjvejkldfuvcrurlefjccec
brnvidibcifr&status=OK&h=GY7zdvKXtr1H37pycRUYyr8o21Y=
```

Sending the same string again would cause a replay failure

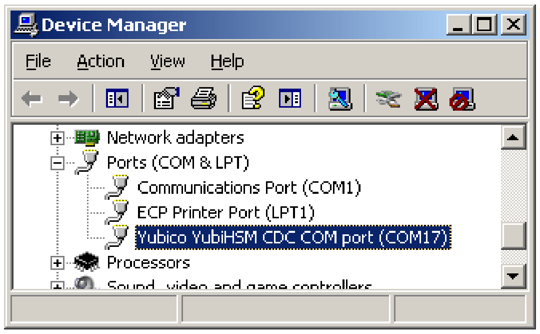
```
id=5075&nonce=BKCqmmPYnszTmjopHDHa&otp=cccccccbjhjvejkldfuvcrurlefjccec
brnvidibcifr&status=REPLAYED_OTP&h=hfwFmNzXRmKxY2rak9MjEhCDYCA=
```

7 Installing YubiHSM

The YubiHSM emulates a USB CDC (Communication Device Class) device, thereby allowing simple integration with serial communication libraries and terminal programs.

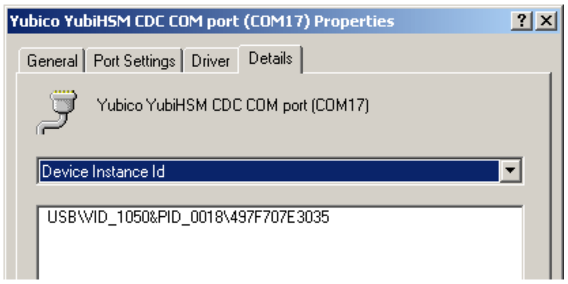
Both Linux- and MacOSX systems provide default configuration and drivers for the CDC class, installing these as communication ports found under /dev/ttyxxx

Windows system from WindowsXP and onwards provides WDM support for CDC devices. Use the supplied `YubicoVCP.inf` file during the installation. When successfully installed, the YubiHSM appears under Ports in the Device Manager.



Here it can be seen that the YubiHSM has been installed as COM17. Each YubiHSM has a unique USB serial number which allows the operating system to associate each unique YubiHSM with a specific port setting.

Select “Properties” and locate the “Details” tab.


Yubico YubiHSM CDC COM port (COM17) Properties
General
Port Settings
Driver
Details
Yubico YubiHSM CDC COM port (COM17)
Device Instance Id

USB\VID_1050&PID_0018\497F707E3035

This device’s unique serial number is 49F707E33035.

8 Configuration - and monitor mode

At power-up, the YubiHSM enters the selected mode of operation if properly configured. By activation of the configuration mode switch while the device is inserted, this is overridden and configuration mode is entered. While in configuration mode, the YubiHSM can be configured using any terminal program of choice.

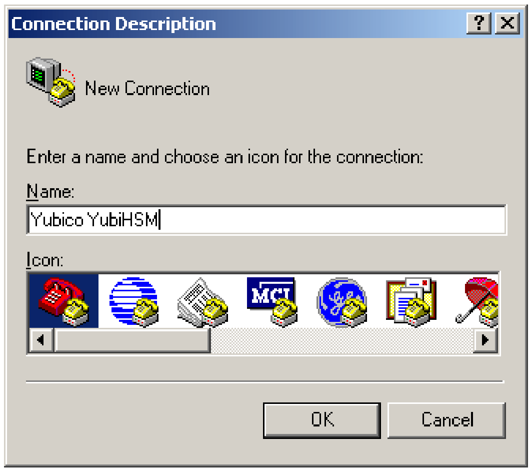
8.1 Entering configuration mode

If the YubiHSM is unconfigured, it enters configuration mode automatically without having to activate the switch.

If the YubiHSM has been configured, enter the configuration mode by inserting the device into the USB port in the server and holding the touch area for three seconds. When the LED starts to flash, you have entered the configuration mode.

If the YubiHSM is accidentally placed in configuration mode when inserting it into a USB port on a server, you can cancel the configuration mode in the Configuration Monitor using HyperTerminal.

Open a terminal program, such as HyperTerminal in Windows to enter the configuration monitor:

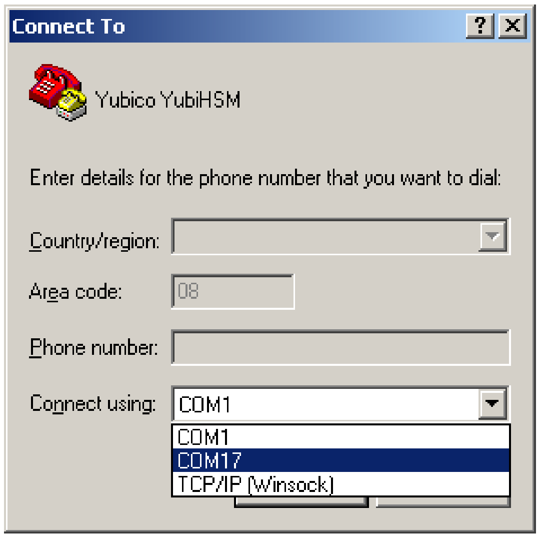

Connection Description

New Connection

Enter a name and choose an icon for the connection:

Name: Yubico YubiHSM

Enter a description for the connection, such as “Yubico YubiHSM” and press OK


Connect To

Yubico YubiHSM

Enter details for the phone number that you want to dial:

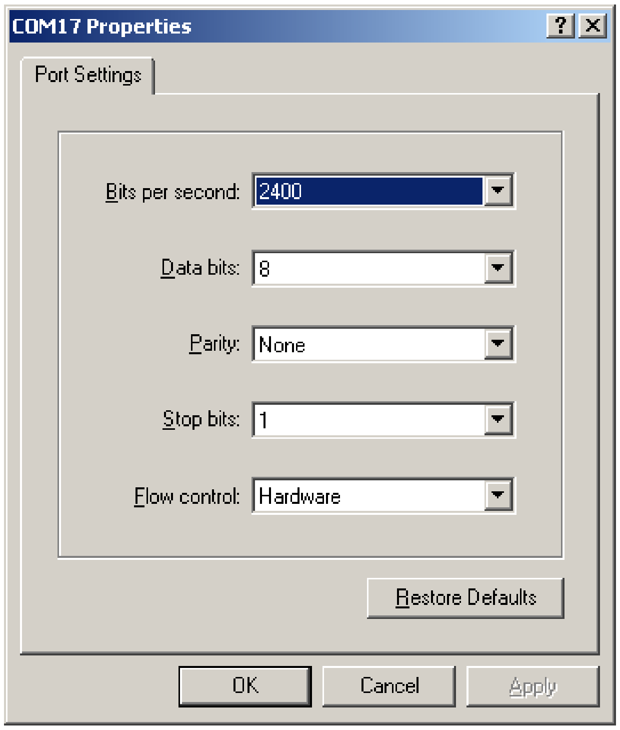
Country/region:

Area code: 08

Phone number:

Connect using: COM1

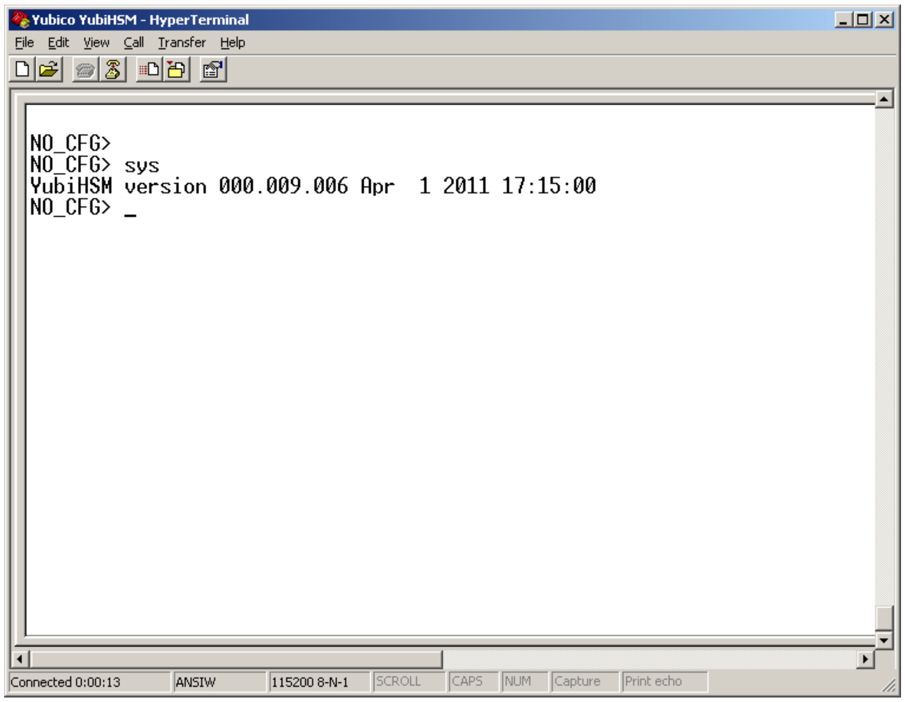
Select the COM port which corresponds to the inserted YubiHSM. In this case, select COM17 and press OK



Bits per second: 2400
Data bits: 8
Parity: None
Stop bits: 1
Flow control: Hardware

None of these settings are applicable for the YubiHSM, so just stick with default and press OK.

Now, commands can be entered by the prompt. Make sure the configuration mode has been enabled by tapping ENTER a few times


Yubico YubiHSM - HyperTerminal
File Edit View Call Transfer Help

```
NO_CFG>
NO_CFG> sys
YubiHSM version 000.009.006 Apr  1 2011 17:15:00
NO_CFG> _
```

The device is currently unconfigured, shown by the prompt

```
NO_CFG>
```

Verify the installation by typing `sys` and hit ENTER. The version information string is displayed.

Typing `?` or `help` shows the available commands, which are described below.

8.2 Understanding the indicator LED

The indicator LED shows the current operational state of the YubiHSM.

Off	Device is powered of or in suspend
Fast flash 50-50% (3 Hz)	HSM mode, waiting for command
Fast flash 25-75% (6 Hz)	HSM mode, waiting for payload
Medium flash 50-50%(0.8 Hz)	Config mode, waiting for command
Medium flash 10-90% (1.6 Hz)	Config mode, waiting for user input
Slow flash 50-50% (0.4 Hz)	WSAPI mode, waiting for input
Slow flash 1-99% (0.4 Hz)	USB not enumerated or failure

8.3 Debug mode

A special mode is available to allow debug and exploration of YubiHSM features. In this mode, all secrets are visible in clear text and the write protection feature is disabled. This mode shall obviously not be used in a configuration setting and if enabled, it can be permanently disabled. See section 8.19 for details.

8.4 Setting up for HSM mode of operation

The YubiHSM can either be configured for HSM- or WSAPI mode. For WSAPI mode, please refer to section 8.5.

At the prompt, type `hsm`

```
NO_CFG> hsm
Enabled flags 7fffffff = YSM_AEAD_GENERATE,YSM_BUFFER_AEAD_GENERATE,
YSM_RANDOM_AEAD_GENERATE,YSM_AEAD_DECRYPT_CMP,YSM_DB_YUBIKEY_AEAD_STORE,
YSM_AEAD_YUBIKEY_OTP_DECODE,YSM_DB_YUBIKEY_OTP_VALIDATE,YSM_AES_ECB_BLOCK_ENCR
YPT,YSM_AES_ECB_BLOCK_DECRYPT,YSM_AES_ECB_BLOCK_DECRYPT_CMP,YSM_HMAC_SHA1_GENE
RATE,YSM_TEMP_KEY_LOAD,YSM_BUFFER_LOAD

a:YSM_AEAD_GENERATE                    b:YSM_BUFFER_AEAD_GENERATE
c:YSM_RANDOM_AEAD_GENERATE             d:YSM_AEAD_DECRYPT_CMP
e:YSM_DB_YUBIKEY_AEAD_STORE            f:YSM_AEAD_YUBIKEY_OTP_DECODE
g:YSM_DB_YUBIKEY_OTP_VALIDATE          h:YSM_AES_ECB_BLOCK_ENCRYPT
i:YSM_AES_ECB_BLOCK_DECRYPT            j:YSM_AES_ECB_BLOCK_DECRYPT_CMP
k:YSM_HMAC_SHA1_GENERATE               l:YSM_TEMP_KEY_LOAD
m:YSM_USER_NONCE                       n:YSM_BUFFER_LOAD
o:FLAG_DEBUG

Toggle bit (space = all, enter = exit)
```

First step is to determine which functions that should be available in the current configuration. Default is all functions being enabled, except debug mode. Refer to section 5 for description of available HSM mode commands.

In order to toggle a function on or off, simply type the letter preceding it. In the case above, to enable debugging, press “o”.

When the desired functions are enabled, hit ENTER to proceed to next step

```
Enter cfg password (g to generate)
```

Here, an optional configuration password can be set. A configuration password allows the configuration to be changed after exit from configuration mode. Either enter ‘g’ and ENTER to generate a random password or paste in a 16 byte hexadecimal string. Keep this password stored securely for future reference.

```
Enter cfg password (g to generate) df5ba5701684e2e57baa907c043d9a1b
```

If this feature is not desired, just press ENTER to proceed.

Next step is to set up "adminstrative Yubikeys" to be used in conjunction with the `YSM_HSM_UNLOCK` command (see section 5.8).

```
Enter admin Yubikey public id (enter when done)
```

Now, enter the public IDs in modhex format for each administrative Yubikey, separated by ENTER.

```
Enter admin Yubikey public id 001/008 (enter when done)djccdbcccccdhnlnehbfgerudcgvtv
Enter admin Yubikey public id 002/008 (enter when done)cccccccbjhhutuujirlltjekvedgdc
Enter admin Yubikey public id 003/008 (enter when done)
```

Finish the list by press ENTER. Note that a complete Yubico OTP can be supplied, but only the first 12 characters are used. Furthermore, note that all used admin Yubikeys must be loaded into the internal OTP database as well using the `dbload` command (see section 8.13).

If this feature is not desired, just press ENTER to proceed.

Next step is to generate a AEAD key storage master key

```
Enter master key (g to generate)
```

Either enter ‘g’ and ENTER to generate a random master key or paste in a 32 byte hexadecimal string. Keep this key stored securely for future reference.

```
Enter master key (g to generate)
9128ba3096d6c0d72f0435d61fbe6cc735744fd625419a47550be13260119496
```

A default key of all zeroes is applied if only ENTER is pressed.

The final step is to confirm the selected configuration and thereby erase any previously stored data.

```
Confirm current config being erased (type yes)
```

Type yes and ENTER to proceed. Any other combination aborts.

```
Confirm current config being erased (type yes) yes - wait - done
HSM (keys changed)> keycommit - Done
HSM>
```

The keycommit command initializes the keystore in flash and is necessary in order to be able to unlock the HSM later on in case a master key was supplied.

The system responds with the `HSM>` prompt when complete. Next step is to load or generate at least one AEAD key and then optionally to load an identity database.

8.5 Setting up for WSAPI mode of operation

The YubiHSM can either be configured for HSM- or WSAPI mode. For HSM mode, please refer to section 8.4.

At the prompt, type `wsapi`

```
NO_CFG> wsapi
Enabled flags 00000000 = None set

a:YSM_AEAD_GENERATE                    b:YSM_BUFFER_AEAD_GENERATE
c:YSM_RANDOM_AEAD_GENERATE             d:YSM_AEAD_DECRYPT_CMP
e:YSM_DB_YUBIKEY_AEAD_STORE            f:YSM_AEAD_YUBIKEY_OTP_DECODE
g:YSM_DB_YUBIKEY_OTP_VALIDATE          h:YSM_AES_ECB_BLOCK_ENCRYPT
i:YSM_AES_ECB_BLOCK_DECRYPT            j:YSM_AES_ECB_BLOCK_DECRYPT_CMP
k:YSM_HMAC_SHA1_GENERATE               l:YSM_TEMP_KEY_LOAD
m:YSM_BUFFER_LOAD                      n:FLAG_DEBUG

Toggle bit (space = all, enter = exit)
```

In WSAPI mode, the only useful function is the debug flag which may be enabled. To toggle this flag, type ‘g’
When the desired functions are enabled, hit ENTER to proceed to next step

```
Enter cfg password (g to generate)
```

Here, an optional configuration password can be set. A configuration password allows the configuration to be changed after exit from configuration mode. Either enter ‘g’ and ENTER to generate a random password or paste in a 16 byte hexadecimal string. Keep this password stored securely for future reference.

```
Enter cfg password (g to generate) 3b427ee8408a59dafb32614749cc58e7
```

If this feature is not desired, just press ENTER to proceed.

The final step is to confirm the selected configuration and thereby erase any previously stored data.

```
Confirm current config being erased (type yes)
```

Type yes and ENTER to proceed. Any other combination aborts.

```
Confirm current config being erased (type yes) yes - wait - done WSAPI>
```

The system responds with the `WSAPI>` prompt when complete. Next step is to load an identity database and optionally one or more client keys if needed.

8.6 Erasing all stored data

The `zap` command erases the configuration, the key list and the id database. No secrets whatsoever are left on the device after a zap operation.

8.7 Setting the flags for subsequent load operations

The flags command is used to enable and disable flags for subsequent load operations. Note that this function does not alter the already assigned configuration flags, which after configuration cannot be changed.

At the prompt, type `flags`

```
HSM> flags
Enabled flags 7fffffff = YSM_AEAD_GENERATE,YSM_BUFFER_AEAD_GENERATE, YSM_RANDO
M_AEAD_GENERATE,YSM_AEAD_DECRYPT_CMP,YSM_DB_YUBIKEY_AEAD_STORE,YSM_AEAD_YUBIKE
Y_OTP_DECODE,YSM_DB_YUBIKEY_OTP_VALIDATE,YSM_AES_ECB_BLOCK_ENCRYPT,YSM_AES_ECB
_BLOCK_DECRYPT,YSM_AES_ECB_BLOCK_DECRYPT_CMP,YSM_HMAC_SHA1_GENERATE,YSM_TEMP_KEY_LOAD

a:YSM_AEAD_GENERATE                     b:YSM_BUFFER_AEAD_GENERATE
c:YSM_RANDOM_AEAD_GENERATE              d:YSM_AEAD_DECRYPT_CMP
e:YSM_DB_YUBIKEY_AEAD_STORE             f:YSM_AEAD_YUBIKEY_OTP_DECODE
g:YSM_DB_YUBIKEY_OTP_VALIDATE           h:YSM_AES_ECB_BLOCK_ENCRYPT
i:YSM_AES_ECB_BLOCK_DECRYPT             j:YSM_AES_ECB_BLOCK_DECRYPT_CMP
k:YSM_HMAC_SHA1_GENERATE                l:YSM_TEMP_KEY_LOAD
m:YSM_BUFFER_LOAD                       n:FLAG_DEBUG

Toggle bit (space = all, enter = exit)
```

Toggle individual function flags by typing the corresponding character above. Press ENTER when finished.

8.8 Generating AEAD- or client keys

AEAD- or client keys can either be generated or loaded to be used in various settings (see sections 5 and 6).

To generate and store a range of keys, type

```
keygen <start id> <count> <key_length>
```

Assume we want to create three keys - 5, 6 and 7 with 32 bytes length.

Type `keygen 5 3 20`

```
HSM> keygen 5 3 20 - using flags ffffffff and len 032
00000005,46e62142239340ce3914bdc65cf087a3273373dab28e0d54e519f4802a17989d
00000006,580ca38dbd36c4bab218042203a24538937b0e5d1169f18df1f2e06bb9b21f13
00000007,0fd96b730dc7a1a5dde51418403d8b60bf6de4beb5b00b5a46c51a3982bcc9f9
```

Note that the currently selected flags (see section8.7) are stored in conjunction with these generated keys.

Use cut-and-paste or input capture from the terminal program to keep a backup copy for these generated keys.

Note that a `keycommit` command (see section 8.11) must be given in order to encrypt and write the key buffer to the non-volatile storage in flash memory.

8.9 Loading AEAD keys

Previously - or externally generated keys can be loaded in batch. To load keys in WSAPI format, please refer to section 8.10.

Type `keyload`

Paste in your keys here or upload an externally stored file

```
HSM> keyload - Load key data now using flags ffffffff. Press ESC to quit
00000010 - stored ok
00000011 - stored ok
00000012 - stored ok
```

Note that the currently selected flags (see section8.7) are stored in conjunction with these loaded keys.

Note that a `keycommit` command (see section 8.11) must be given in order to encrypt and write the key buffer to the non-volatile storage in flash memory.

Example input to store 32 byte (256 bit) key with keyhandle 0x20 :
```
00000020,000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f
```

8.10 Loading client keys (WSAPI compatible)

With the online tool for generation of WSAPI clientids and keys, the ids are decimal and the keys are presented in base64 format. Therefore, a special key load feature is available to accept output from these tools without conversion, using simple cut-and-paste operations.

```
WSAPI> cli - Enter pairs of IDs and client keys. ESC to quit
5075
Client id 05075 - enter key now
PXBv6EcyyX9zfopo8mynaPbEi5Y=
Client ID 05075 stored ok
```

Client ids + keys can also be pasted in or loaded batch-wise. Then organize the ids + keys with a delimiter (space or comma), such as

```
5080 vaswFwqt1V3cSF1zUQGO0H1M5tA=
5081 zNMW4xOSSeAowD9udlY7QHeHywA=
5082 pX9GXOC30kyhJ54jeOKbxJHUGpY=
```

Uploading these is then done as

```
WSAPI> cli - Enter pairs of IDs and client keys. ESC to quit
5080 vaswFwqt1V3cSF1zUQGO0H1M5tA=
Client ID 05080 stored ok
5081 zNMW4xOSSeAowD9udlY7QHeHywA=
Client ID 05081 stored ok
5082 pX9GXOC30kyhJ54jeOKbxJHUGpY=
Client ID 05082 stored ok
```

Note that a `keycommit` command (see section 8.11) must be given in order to encrypt and write the key buffer to the non-volatile storage in flash memory.

8.11 Committing keys to non-volatile storage

Keys generated or loaded during a configuration session is stored in a temporary buffer only. In order to be able to use them, the current set of keys must be encrypted and committed to non-volatile storage in flash memory.

```
HSM (keys changed)> keycommit - Done
HSM>
```

Note that the key database must be committed even if no master key has been set. The keys are then encrypted with a default (all zero) key.

8.12 Showing list of keys

The current list of stored keys can be displayed with the `keylist` command. In non-debug mode, only the keys and flags(if applicable) are shown:

```
HSM> keylist
00000005,ffffffff
00000006,ffffffff
00000007,ffffffff
Entries 00003 invalid 00000 free 00039
```

In debug mode, even the secrets are displayed:

```
HSM> keylist
00000005,46e62142239340ce3914bdc65cf087a3273373dab28e0d54e519f4802a17989d,ffff
00000006,580ca38dbd36c4bab218042203a24538937b0e5d1169f18df1f2e06bb9b21f13,ffff
00000007,0fd96b730dc7a1a5dde51418403d8b60bf6de4beb5b00b5a46c51a3982bcc9f9,ffff
Entries 00003 invalid 00000 free 00039
```

The key id (handle, client id) is always shown as a32-bit hexadecimal number and the key is shown as a 1-32 bytes hexadecimal string.

Note that the key list will not be automatically available if a master key has been set and the configuration mode once has been exited. In order to make the keys available again in configuration mode, use the `keydrecrypt` command

8.13 Loading identities into the database

Identities + secrets can be loaded while in configuration mode. Output from the Yubico configuration programming station (default provisioning format) can be used as input.

```
HSM> dbload - Load id data now. Press ESC to quit
99950 - inserted ok
99951 - inserted ok
99952 - inserted ok
99953 - inserted ok
99954 - inserted ok
99955 - inserted ok
```

Upload can either be done by cut-and-paste or by utilizing the upload feature of the terminal program to send an externally stored text file.

Example input to store two YubiKey identities in the internal database:

```
00001,ftftftfteeee,f0f1f2f3f4f5,4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d4d,,,
00002,ftftftftcccc,f5f6f7f8f9f0,vevevevevevevevevevevevevevevevev,,,
```

Here, ftftftfteeee is public id of YubiKey 1, f0f1f2f3f4f5 is the private uid, 4d4d4d... is the AES-128 key.

8.14 Showing list of identities

The currently stored identities can be displayed with the `dblist` command

```
HSM> dbli
00000,cccccccbjhhu,00000/000
00001,cccccccbjhhv,00000/000
00002,cccccccbjhic,00000/000
00003,cccccccbjhib,00000/000
00004,cccccccbjhid,00000/000
00005,cccccccbjhie,00000/000
```

The supplied device identity is discarded (as it isnot used by the YubiHSM firmware) so this is replaced with a sequence number into this list. Last in the list are the stored OTP counter values which are updated for each successful OTP validation request.

In debug mode, all secrets are displayed as well:

```
HSM> dblis
00000,cccccccbjhhu,fe37a81ab606,46d917606afe4d81830b8822d65a70ca,00000/000
00001,cccccccbjhhv,f9fc0cf39012,bf0af3bf814bc8250b4fbb0e806ed8a0,00000/000
00002,cccccccbjhic,8b99376dc0af,ccf6e1e61b93f2cec00dc88cbd1c03bd,00000/000
00003,cccccccbjhib,9d1a6cbb695e,37f90e667417fb55aeb74c83e13c4e57,00000/000
00004,cccccccbjhid,f2a01d796353,2d84de46f3bba40c189784af134b0d7a,00000/000
00005,cccccccbjhie,0bf13c6bb1b1,6f38bc0d30372eb6f4cc4c2c3b1fac1e,00000/000
```

8.15 OTP verification test

When identities with associated secrets have been loaded, a quick function for verifying an identity is provided.

Type `otpverat` the command prompt:

```
HSM> otpv - Enter OTP 12+32 strings now. ESC to quit
cccccccbjhhuniruiulurbgkgululvvllkgrtkrrnjrh - Ok
cccccccbjhhutrhliercvccltnejnkhinetltujnfjch - Ok
cccccccbjhhuniruiulurbgkgululvvllkgrtkrrnjrh - OTP replay
```

The updated counters can then be checked with the dblist command:

```
HSM> dbli
00000,cccccccbjhhu,fe37a81ab606,46d917606afe4d81830b8822d65a70ca,00001/001
```

8.16 Nonce generation

The persistent unique nonce generation can be tested with the `nonce` command

`HSM> non – 000000000100`	Current nonce after startup
`HSM> non – 000000000100`	Same - no argument before
`HSM> non 1 – 000000000100`	Same - no argument before
`HSM> non 2 – 010000000100`	Incremented by 1
`HSM> non – 030000000100`	Incremented by 2

The first 4 bytes is a volatile 32-bit counter which is set to zero at startup or a new configuration being written. The next 2 bytes is a startup counter which is reset to 1 when a configuration is written and then incremented by 1 for each power up event.

8.17 Preset of the nonce

The nonce startup counter part is incremented by one at each startup where the session counter part is reset to zero. The nonce is not reset by mode change.

In cases where it is desired to set the nonce to a predefined value, use the
`npreset <startup-counter-value> <session-counter-value>` to set it.

```
NO_CFG> non – 000000001600
NO_CFG> npr 22 4711 – 114700002200
```

In this example, the startup counter which was at 16 was changed to 22 with the session counter set to 4711. Note the little-endian notation for the nonce.

The function is only available when the device is unconfigured to prevent nonces for a valid configuration from being destroyed

```
HSM> non – 000000001700
HSM> npre 1 2 - valid when unconfigured only
```

8.18 Removing write protection

The default action when not in debug mode is that the key- and identity databases become write protected at exit from configuration mode.

If a configuration password has been set during the configuration phase, the write protection can be removed with the `unprot` command.

```
HSM> dblo - Writes are disabled
HSM> unp - enter password df5ba5701684e2e57baa907c043d9a1b - ok
HSM> dblo - Load id data now. Press ESC to quit
```

The write protection will be disabled until exit from monitor mode.

8.19 Disabling debug mode

As all secrets are visible in debug mode and that the key- and identity databases are constantly open for writes, this modeshall obviously not be used in a production setting. If enabled, the debugmode can be permanently disabled with the ndebugcommand.

```
HSM> ndeb - debug flag removed
```

Note that once disabled, the debug flag cannot be set again unless a configuration is done from scratch.

8.20 Random number generation test

To test the randomness of the RNG or to generate a few blocks of random numbers, a random number function is available.

Type rng followed by the desired number of 16-bytesblocks. A value of zero keeps the rng running until terminated with a keystroke

```
HSM> rng 6
6716377dec2f18348a3169fc502daa9453aa91db3e2de70e6a432c2ed4fe
9a6a86665372f4f66838b5b7172eb96691a4368e473f7fdac25221f3ee7d
028c493fece497ade370dc9e49f5689f1e950c70a08d5d5a2d483418d0bd 6c6fce1b66f6
```

8.21 Random number generator diagnostics

A diagnostics function is available for the hardware random number generator, which provides seeds for the PRNG.

The command `rdiag` without arguments shows the startup result. Adding an argument re-seeds the TRNG the specified number of times.

```
HSM> rdi 5 - Current bias 005 levels 01559-02663 span 01104
status=ok bias 005 levels 01543-02719 span 01176
status=ok bias 005 levels 01564-02859 span 01295
status=ok bias 004 levels 01645-02652 span 01007
status=ok bias 005 levels 01536-02747 span 01211
status=ok bias 005 levels 01450-02681 span 01231
```

The bias should be a number between 3 and 7 and thespan should be in the range 1000 – 2000. The levels should be around 1000-3000.

The internals of the random number generator is explained in more detail in section 4.8.

8.22 Leaving configuration mode

By typing `exit` by the prompt, the selected mode of operation is entered. Unless being in debug mode, the configuration write protection is enabled at this stage

9 Appendices

Appendix 1: YubiHSM firmware ChangeLog

- 1.0.4
  - Security: Introduce permission flag YSM_USER_NONCE to allow some legitimate use cases that were insecure without it after the AEAD decryption possibilities discussed in the YubiHSM security advisory 2012-02-13.
- 1.0.3
  - Bugfix: The 'ndebug' configuration mode command could accidentally destroy the configuration data.
  - Bugfix: The 'dbload' configuration mode command did not properly validate it's input.
  - Bugfix: WSAPI mode did not unlock the keystore.
  - Usability: Added several checks to the configuration mode to warn if keys have not been written to keystore or admin YubiKeys have not been loaded into the internal database.
- 1.0.2
  - First commercially available version.

Appendix 2: Licensing information