Yubico libyubihsm Vulnerabilities

Oct 19, 2020 • Christian Reitter
ID: CVE-2020-24387, CVE-2020-24388

Related articles: Yubico libykpiv Vulnerabilities II • Yubico yubihsm-shell Vulnerability (CVE-2021-43399) • Yubico libyubihsm Vulnerabilities (CVE-2021-27217, CVE-2021-32489)

After my previous research had uncovered security issues in other Yubico smartcard libraries, I decided in July to take a closer look at the Yubico libyubihsm library. Libyubihsm is responsible for interacting with the YubiHSM2 Hardware Security Module that is used in enterprise systems with advanced cryptography requirements. After applying a few days of my private research time & fuzzing experience to it, I discovered multiple memory issues in the library code.

At least two of the issues are practical security vulnerabilities: a malicious HSM device or Man-In-The-Middle network attacker can trigger out of bounds write and read operations (CVE-2020-24387) and out of bounds read operations (CVE-2020-24388) that ultimately both lead to segmentation faults of the process which embeds libyubihsm. Notably, the built-in encryption and authentication of the regular HSM messages do not mitigate this attack since the issues are triggered before any effective cryptography.

This article will describe the issues and give some background information on how they were found.

Contents

Consulting

I'm a freelance Security Consultant and currently available for new projects. If you are looking for assistance to secure your projects or organization, contact me.

Fuzzing Methodology

The previous libykpiv article outlines the fuzzing approach and related strategies which I used.

To summarize:

- The libyubihsm library is fuzzed with libFuzzer through the yubihsm-shell CLI program.
- The modified yubihsm-shell operates on HSM2 messages that the fuzzer creates (via a custom USB backend).
- Yubihsm-shell is called with various command line parameters, covering a lot of the "shallow" HSM library functionality.
- The in-process design of libFuzzer and omission of actual network or USB I/O operations keeps the executions reasonably fast.
- The HTTP backend path was not fuzzed, but USB-related bugs mostly apply to it as well.

Given the complexity of the encrypted and authenticated HSM message contents, the focus of this research was to check for **low-level memory issues in the basic message handling code**, similar to previous libykpiv and libu2f-host issues. The custom fuzzing harness does not have an internal model of the HSM implementation details and performs no cryptographic steps such as encryption with hard-coded keys. While the resulting fuzzer setup is unable to reach a lot of the "complex" code paths, issues discovered via this unfocused approach have the nice property that they do not require specific knowledge of device secrets or special device states for the attacker. As mentioned in the following paragraphs, this strategy was successful in finding new low-level issues.

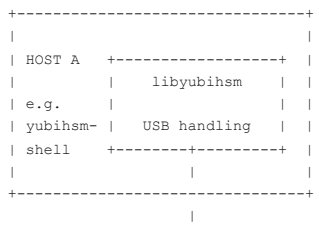
To reach additional in-depth coverage of the complex message handling functions (and potential bugs that remain there), a more concentrated approach with extensive code changes such as circumvention of the encryption + authentication of the messages or a special message generation harness will likely be necessary. This could be explored in future research.

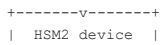
As with most fuzzing targets, there were a number of functional issues which had to be discovered, debugged and fixed to allow a stable fuzzing operation. This is common for fuzzing research, but can significantly increase the necessary time required to reach interesting results. Overhead due to essential stability bugfixing should be factored in when estimating how long it takes to analyze a specific software.

Technical Background

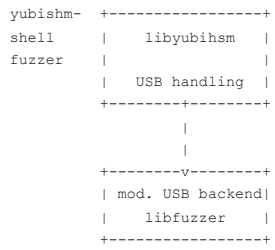
In the basic setup configuration, the HSM2 USB 2.0 device is directly attached to the host. The local software talks to the HSM via libyubihsm, the USB backend and operating system. This simple configuration is viable as a self-contained HSM setup and needs no network configuration.

Schematic diagram:



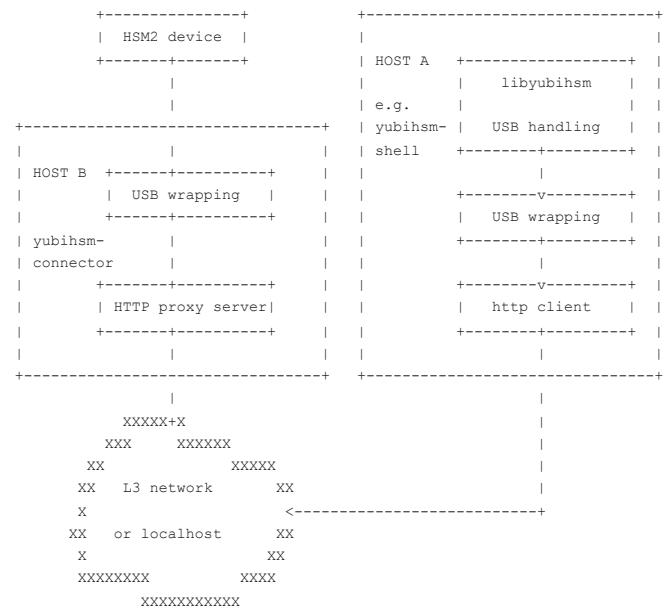


This structure was also the conceptual base of the fuzzing setup:



Larger and more complex HSM setups have requirements for redundancy and abstraction from a single host and local HSM. Yubico addresses this by providing an additional HTTP-based network layer that can be used to connect the client host and HSM either locally or remotely. The requests are proxied over the network to the yubihsm-connector server (written in Go) instead of direct USB communication. The connector service passes the raw USB messages over unencrypted HTTP to and from the HSM as a sort of USB proxy.

Example configuration:



Note that the network connection adds no encryption or authentication by default and usually relies on the basic TCP checksum for message integrity. This is intentional and well-known via the official documentation:

> *[...] 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.*

For the purpose of the discovered vulnerabilities, the default configuration provides no defense against network-level attackers that can mount a Man-In-The-Middle attack.

Some of the documentation suggests that the existing inner security layer is sufficient:

> *Sessions are established cryptographically between the application and the YubiHSM 2 using a symmetric mutual authentication scheme that is both encrypted and authenticated.*

However, as I will show, the cryptographic defenses on the HSM session level do not prevent the discovered attacks.

The Vulnerabilities

CVE-2020-24387

Interaction with the HSM2 device happens over individual logical **sessions** which represent open connections between an application and the HSM device. Session IDs and related decisions are coordinated by the HSM. This design allows concurrent access from multiple sources as well as long-running connections, but requires robust session management on all endpoints to avoid collisions or local issues. A noteworthy detail of the HSM message protocol design is that the initial session handshake between client and HSM is exchanged in cleartext and without message authentication.

CVE-2020-24387 consists of an essential flaw in the handshake parsing on the library side. This turns into a memory management and memory safety problem that terminates the running process.

The relevant code to open a new connection with the HSM includes the following section:

```
yrc = send_msg(connector, &msg, &response_msg, new_session->s.identifier);

[...] // basic failure handling and sanity checks
```

```
ptr = response_msg.st.data;

// Save sid
new_session->s.sid = (*ptr++);
```

yubihsm.c

The `new_session->s.sid` is an `uint8_t` integer with a maximum value of **255** that represents the session ID as decided by the HSM.

There are a maximum of **16** sessions running with the HSM at any given time. A genuine HSM2 device will constantly reuse session IDs **0** to **15** for new connections. Although the library code is aware of the session limits via `#define YH_MAX_SESSIONS 16`, this limited value range **is not actually enforced** in the library code and session IDs with values between **16** and **255** are also accepted. This represents the main bug.

The ID value range restrictions are important because the memory handling of libyubihsm directly uses the session ID parameter as an **index value** into the fixed session array `yh_session *sessions[YH_MAX_SESSIONS]` that keeps track of the open connections.

As it is common with C, the access happens without additional implicit or explicit bounds checking. Successfully opening sessions with session IDs **greater than 15 therefore cause out of bounds writes and reads behind this array**.

`yh_com_open_session()` is one code section with unsafe session index usage:

```
if (ctx->sessions[session_id] != NULL) {
  yrc = yh_destroy_session(&ctx->sessions[session_id]);
  if (yrc != YHR_SUCCESS) {
    fprintf(stderr, "Failed to destroy old session with same id (%d): %s\n",
            session_id, yh_strerror(yrc));
    return -1;
  }
}
ctx->sessions[session_id] = ses;
```

commands.c

The comparison check `ctx->sessions[session_id] != NULL` will evaluate to false in many cases, particularly early in the program execution. The `session` array is located on the global buffer which is initialized with `0x00` at startup by C specifications.

Corresponding sanitizer warning:

```
==19585==ERROR: AddressSanitizer: global-buffer-overflow on address 0x0000010c52b8 at pc 0x0000005a623a bp 0x7fffffff63d0 sp 0x
READ of size 8 at 0x0000010c52b8 thread T0
```

The `ctx->sessions[session_id] = ses` operation will then do an out of bounds write with the `ses` session pointer value into the global buffer. Under x86_64, this writes a 64 bit = 8 byte memory segment. An attacker can control **the offset** of the destination memory address via the session ID, but has no direct control over **the value** that is written.

Corresponding sanitizer warning:

```
==19585==ERROR: AddressSanitizer: global-buffer-overflow on address 0x0000010c52b8 at pc 0x0000005a6802 bp 0x7fffffff63d0 sp 0x
WRITE of size 8 at 0x0000010c52b8 thread T0
```

In order to trigger a successful session handshake and therefore practical code issues, the malicious HSM (or MITM network attacker) has to accept a response message that is addressed to the problematic session ID. This requires a HSM response with the `YHC_AUTHENTICATE_SESSION_R` flag. In the hypothetical case where the session ID accidentally became large as a result of message transmission errors, a genuine HSM will not send a positive response. This should prevent further issues.

In a targeted exploit, the required 2nd message is easily sent by the attacker and libyubihsm continues with problematic behavior.

According to my understanding and the assessment of the relevant Yubico engineers, the out of bounds writes impact **the general memory integrity** of the program. However, we are not aware at the moment of a way to leverage this into some direct control over the code flow due to the location of the out of bounds write and other constraints. Since the writes are happening from a position on the global buffer and not the stack, existing stack protection countermeasures will be ineffective and the attack is not detected.

In an attack scenario, the code flow continues without returning error codes and the CLI or library code will perform some additional function calls depending on the originally requested HSM command that invoked the session creation. The original command may fail or succeed depending on the HSM behavior.

At some point, the code will attempt to close the session via `yh_util_close_session()`, which sends an encrypted command via `_send_secure_msg()` towards the HSM to inform it of the session state change:

```
yrc = yh_send_secure_msg(session, YHC_CLOSE_SESSION, NULL, 0, &response_cmd,
                         response, &response_len);
```

yubihsm.c

Since the code is expecting an active session in the normal session array (but there is none), some steps of the parameter initialization fail and the session pointer given to the following functions is still in a strange state. Consequently, a lot of the message preparation routines in `_send_secure_msg()` silently fail. The first deadly error happens when preparing AES data for the message encryption:

```
aes_set_encrypt_key((uint8_t *) session->s.s_enc, SCP_KEY_LEN, &aes_ctx);
```

yubihsm.c

`session->s.s_enc` is not in a usable state, and it is not in a memory region that may be accessed:

```
(gdb) print &session->s
$5 = (Scp_ctx *) 0x2c
```

The access results in a segfault that kills the process:

```
==20041==ERROR: AddressSanitizer: SEGV on unknown address 0x00000000002d (pc 0x7f304ec6e192 bp 0x7fff50637810 sp 0x7fff5063772
==20041==The signal is caused by a READ memory access.
==20041==Hint: address points to the zero page.
    #0 0x7f304ec6e192 in AES_set_encrypt_key (/usr/lib/x86_64-linux-gnu/libcrypto.so.1.1+0x8f192)
```

For the analyzed library use case, this denial of service represents the primary impact.

CVE-2020-24388

When sending a secure message to the HSM via `_send_secure_msg()`, the code looks like this:

```
yrc = send_authenticated_msg(session, &msg, &response_msg);

// [...] error handling

// definition: uint16_t out_len;
out_len = response_msg.st.len;

memcpy(work_buf, session->s.mac_chaining_value, SCP_PRF_LEN);
response_msg.st.len = htons(response_msg.st.len);
memcpy(work_buf + SCP_PRF_LEN, response_msg.raw, 3 + out_len - SCP_MAC_LEN);
compute_full_mac(work_buf, SCP_PRF_LEN + 3 + out_len - SCP_MAC_LEN,
                 session->s.s_rmac, SCP_KEY_LEN, mac_buf);

// [...] MAC result verification
```

yubihsm.c

`send_authenticated_msg()` internally calls `send_msg()` to transmit the specified message and receive a response message from the HSM in `&response_msg`. If the HSM sends a response and does not flag `YHC_ERROR`, the error handling code is passed and the code continues towards the response parsing section.

Note that `out_len = response_msg.st.len` is fully controlled by the HSM. The essential flaw behind CVE-2020-24388 is the use of `out_len` as an unchecked length field. This becomes a problem a few lines down at

```
memcpy(work_buf + SCP_PRF_LEN, response_msg.raw, 3 + out_len - SCP_MAC_LEN);
```

Since `SCP_MAC_LEN` is **8**, the length calculation for the `memcpy()` can result in negative numbers, for example with `out_len = 0`:

```
==16960==ERROR: AddressSanitizer: negative-size-param: (size=-5)
    #0 0x528eb4 in __asan_memcpy
    #1 0x655456 in _send_secure_msg
```

`memcpy()` expects positive integers of type `size_t`. Negative parameter values get converted implicitly to a large positive number via an unsigned integer underflow. As a result, the memory copy attempts to significantly over-read from the target memory region and violates memory bounds:

```
==16649==ERROR: UndefinedBehaviorSanitizer: SEGV on unknown address 0x7ffc85183fe8
==16649==The signal is caused by a READ memory access.
    #0 0x7fcac423564d  (/lib/x86_64-linux-gnu/libc.so.6)
```

The result is a segmentation fault that kills the program. Notably, this happens before the message authentication code (MAC) on the response packet is checked, so the provided authentication code is irrelevant.

Out of Bounds Read in hex_decode()

This is a small problem in a custom utility function.

```
bool hex_decode(const char *in, uint8_t *out, size_t *len) {
  int pos = 0;
  size_t in_len = strlen(in);
  if (in[in_len - 1] == '\n') {
    in_len--;
  }
  if (in[in_len - 1] == '\r') {
    in_len--;
  }
[...]
```

util.c

For small inputs, `in[in_len - 1]` can perform stack out of bounds reads before the input buffer `in`:

```
==30564==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffdf03b8bef at pc 0x0000006e6bd2 bp 0x7ffdf03b7d10 sp 0x7
READ of size 1 at 0x7ffdf03b8bef thread T0
    #0 0x6e6bd1 in hex_decode /yubihsm-shell/common/util.c:568:7
    #1 0x616824 in get_input_data /yubihsm-shell/src/main.c:1446:11
[...]
```

```
    [1232, 5328) 'buf' (line 1911) <== Memory access at offset 1231 underflows this variable
```

Given the context, this does not look like a serious security problem to me.

Undefined Behavior in send_secure_msg()

When closing a session, `send_secure_msg()` is called with the parameters `data_len = 0` and `*data = NULL`.

`send_secure_msg()` has the following line:

```
memcpy(decrypted_data + 3, data, data_len);
```

yubihsm.c

The send function will therefore call `memcpy(decrypted_data + 3, NULL, 0)`.

This is undefined behavior:

```
yubihsm-shell/lib/yubihsm.c:303:30: runtime error: null pointer passed as argument 2, which is declared to never be null
```

The simple solution is to skip this operation if no data source is given. Although technically all kind of things can go wrong, it is unlikely that this causes a notable issue in practice.

Out of Bounds Read in yh_util_get_log_entries()

This bug appears to be a combination of multiple issues. The main result are multiple out of bounds reads on the stack since `n_items` is larger than intended when processing log entries at the following location:

```
  for (uint16_t i = 0; i < *n_items; i++) {
    out[i].number = ntohs(ptr[i].number);
```

yubihsm.c

Example warning for the resulting access:

```
==26697==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7ffe14a9eb80 at pc 0x000000528ed7 bp 0x7ffe14a9e270 sp 0x7
READ of size 16 at 0x7ffe14a9eb80 thread T0
    #0 0x528ed6 in __asan_memcpy (/yubihsm-shell/build/src/yubihsm-shell+0x528ed6)
    #1 0x5d2775 in yh_util_get_log_entries /yubihsm-shell/lib/yubihsm.c:2659:5
    #2 0x56dad9 in yh_com_audit /yubihsm-shell/src/commands.c:92:9

    [256, 2304) 'response' (line 2601) <== Memory access at offset 2304 overflows this variable
```

During fuzzing, this problem was not consistently reproducible. It appears that at least one code path that triggers this issue also depends on the state of uninitialized memory, so it is possible that this issue cannot be reached or fully controlled in a deterministic way by an attacker.

This section will be updated later with more information.

Attack Scenario and Security Implications

Note that the vulnerabilities

- will not be triggered by genuine HSMs (*when excluding transmission error events for CVE-2020-24388*)
- are in the host side code and do not affect the HSM2 firmware

If you use HSM2 devices via software stacks that exclude libyubihsm, you are not affected.

CVSS Score

ID	CVSS 3.1 Score	Parameters
CVE-2020-24387	6.5 (Medium)	`CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:H`
CVE-2020-24388	5.9 (Medium)	`CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:N/A:H`

Proof Of Concept

The following patches simulate the attack with limited code changes to a vulnerable library version. This can be tested with an expendable HSM2 device to validate the error behavior without a complex hardware setup.

WARNING: use the following code at your own risk. Although not intended, please assume that this will PERMANENTLY overwrite data on the HSM device that you have connected.

Procedure:

1. Rebuild libyubihsm and all relevant programs with the specified patch, example: vulnerable repo version
2. Insert a test HSM device in factory settings
3. Run a basic command to test the impact

- Variant A: yubihsm-connector is running on localhost, run `./yubihsm-shell -a blink-device -p password`
- Variant B: direct USB connection, run `./yubihsm-shell -C yhusb:// -a blink-device -p password`

If you observe connection problems, please make sure that your setup is working with the regular unmodified CLI.

CVE-2020-24387

```
diff --git a/lib/yubihsm.c b/lib/yubihsm.c
index 565679a..427fbc2 100644
--- a/lib/yubihsm.c
+++ b/lib/yubihsm.c
@@ -81,6 +81,12 @@ static yh_rc send_msg(yh_connector *connector, Msg *msg, Msg *response,
     return YHR_INVALID_PARAMETERS;
   }
   DBG_NET(msg, dump_msg);
+  printf("sending message with id %u\n", msg->st.data[0]);
+  if(msg->st.data[0] == 0x42) {
+    printf("faking successful auth response\n");
+    response->st.cmd = YHC_AUTHENTICATE_SESSION_R;
+    return YHR_SUCCESS;
+  }
   yrc = connector->bf->backend_send_msg(connector->connection, msg, response,
                                         identifier);
   if (yrc == YHR_SUCCESS) {
@@ -699,7 +705,9 @@ yh_rc yh_create_session(yh_connector *connector, uint16_t authkey_id,
   ptr = response_msg.st.data;

   // Save sid
-  new_session->s.sid = (*ptr++);
+  new_session->s.sid = 0x42;
+  printf("simulating malicious response with sid %u\n" , new_session->s.sid);
+  ptr++;

   // Save card challenge
   memcpy(new_session->context + SCP_HOST_CHAL_LEN, ptr, SCP_CARD_CHAL_LEN);
```

Observed log:

```
Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
sending message with id 0
simulating malicious response with sid 66
sending message with id 66
faking successful auth response
Created session 66
Segmentation fault
```

CVE-2020-24388

```
diff --git a/lib/yubihsm.c b/lib/yubihsm.c
index 565679a..cfc4c77 100644
--- a/lib/yubihsm.c
+++ b/lib/yubihsm.c
@@ -346,6 +346,9 @@ static yh_rc _send_secure_msg(yh_session *session, yh_cmd cmd,
     goto cleanup;
   }

+  printf("manipulating response to secure message\n");
+  response_msg.st.len = 0;
+
   // Response is MAC'ed and encrypted. Unwrap it
   out_len = response_msg.st.len;
```

Observed log:

```
Using default connector URL: http://127.0.0.1:12345
Session keepalive set up to run every 15 seconds
Created session 0
manipulating response to secure message
Segmentation fault
```

Coordinated Disclosure

The disclosure went fairly smoothly. All the essentials were already in place from previous vulnerability reports, which reduces the overhead.

Similar to the libykpiv disclosure, I was able to coordinate closely with a dedicated security contact person, evaluate the proposed security patches and make suggestions for technical reporting summaries such as the CVE descriptions. This process was again very positive and I appreciate the trust from the Yubico engineers.

One aspect that could be improved in my opinion is the patch response time. Yubico spent basically the full 90 days of disclosure duration before shipping the security patch release. I am aware that libyubihsm is used in large and complex software ecosystems and that the corresponding release preparations are not trivial, so I am sympathetic towards not rushing a fix which might cause regressions. Nevertheless, I think it will be beneficial in the long run to work towards a more flexible release process that can accommodate security fixes in for example a month or less.

Overall, I am satisfied with the disclosure process.

Relevant yubihsm-shell Sources

variant	source	fix	references
Yubico upstream	GitHub	patch to 2.0.3, bundled in SDK release 2020.10	YSA-2020-06
Fedora	Fedora	update to 2.0.3-1	Bug 1890204, Bug 1890205, Bug 1890206

Other Relevant Software

variant	source	notes
yubihsm.rs	GitHub	not affected (AFAIK), not included in the disclosure

Detailed Timeline

Date	info
2020-07-20	Disclosure of issue #1 to Yubico
2020-07-21	Disclosure of issue #2 to #7 to Yubico
2020-07-21	Yubico acknowledges receiving the reports
2020-08-05	Yubico confirms issue #1
2020-08-07	Yubico confirms issue #2 to #7
2020-08-12	Yubico sends preview of proposed fixes
2020-08-12	Positive feedback to Yubico on proposed fixes
2020-08-17	Discussion of planned CVE descriptions
2020-08-18	Yubico requests CVEs from MITRE
2020-08-19	MITRE assigns CVEs
2020-08-25	Yubico communicates planned disclosure date: 2020-09-29
2020-09-28	Yubico communicates planned disclosure date: 2020-10-14
2020-10-09	Yubico communicates planned disclosure date: 2020-10-19
2020-10-19	Release of patched HSM2 SDK version 2020.10
2020-10-19	Public disclosure via YSA-2020-06
2020-10-19	Publication of this blog post
2020-10-21	Fedora includes patched version

Bug Bounty

Yubico provided hardware as a bug bounty for this issue.

Christian Reitter

Information security and other interests.