Yubico libyubihsm Vulnerabilities (CVE-2021-27217, CVE-2021-32489)

Mar 4, 2021 • Christian Reitter
ID: CVE-2021-27217, CVE-2021-32489

Related articles: Yubico libykpiv Vulnerabilities II • Yubico yubihsm-shell Vulnerability (CVE-2021-43399) • Tamper Evident Seals - Part I

I've continued my research on weaknesses in the Yubico HSM library and recently found additional vulnerabilities in it through fuzzing.
A malicious HSM device with authentication can trigger out of bounds read operations (CVE-2021-27217) on the host that may lead to segmentation faults and crash the program.

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

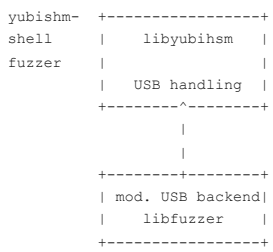
Please see the previous yubihsm-shell article for a summary of the automated testing approach and technical background.

Notably, the necessary fuzzer progress for the new discovery was achieved by weakening the normal message authentication checks in the code sufficiently to make the fuzzer pass them with some inputs. This allowed the fuzzer + sanitizer combination to discover problematic code located behind that barrier.

Without this strategic adjustment, passing the authentication checks with fuzzer-provided input would otherwise be equal to a brute-force attack on the valid message authentication code (*a specific 64 bit value*), which is very hard for a generic fuzzer to do.

Note that this step also corresponds to a change in the attacker model from "man-in-the-middle without knowledge of secrets" to "malicious HSM device with knowledge of secrets", since the latter is required to accurately compute the required MAC. More on this in later sections.

Conceptual overview of the fuzzing setup:



The Vulnerabilities

CVE-2021-27217

The underlying code issue is near the location of CVE-2020-24388 and triggered similarly, so I recommend reading the original issue description to understand this issue.

In short, the libyubihsm code and the HSM are interacting via USB messages, which are commonly transported over other network protocols in remote setups via the yubihsm-connector. After sending a secure message to the HSM via `_send_secure_msg()`, the host program expects the HSM to respond with an authenticated and encrypted message that contains some response data related to the request, for example a status code and response data for the requested HSM action.

The response message is initially checked as follows:

```
yrc = send_authenticated_msg(session, &msg, &response_msg);
// [...] error handling for send_authenticated_msg()

// Response is MAC'ed and encrypted. Unwrap it
out_len = response_msg.st.len;
if (out_len < SCP_MAC_LEN - 3 ||
    (size_t)(3 + out_len - SCP_MAC_LEN) >= sizeof(work_buf)) {
  DBG_ERR("Received invalid length %u", out_len);
  yrc = YHR_BUFFER_TOO_SMALL;
```

```
  goto cleanup;
}
```

yubihsm.c

For the previous CVE-2020-24388 issue, messages with `response_msg.st.len < SCP_MAC_LEN - 3` caused dangerous `memcpy()` problems during the early parsing steps. In particular, the problematic `memcpy()` happened **before** the message authentication field was checked, so an unauthenticated man-in-the-middle attacker could trigger it. This problem was fixed in libyubihsm **2.0.3** by - among other things - requiring `response_msg.st.len >= SCP_MAC_LEN - 3` as seen above.

Overall, the steps towards the MAC comparison now appear to be safe from this kind of attack based on problematic length fields.

However, it turns out that the **2.0.3** patches failed to prevent length-related issues that happen **after** the MAC comparison. Let's assume that libyubihsm is interacting with a malicious HSM device that **has access** to the correct secrets for authentication & encryption measures between the HSM and host.

The program looks as follows after the authentication stage:

```
DBG_INFO("Response MAC successfully verified");

out_len -= SCP_MAC_LEN;

if (session->s.sid != response_msg.st.data[0]) {
  DBG_ERR("Session ID mismatch, expected %d, got %d", session->s.sid,
          response_msg.st.data[0]);
  yrc = YHR_GENERIC_ERROR;
  goto cleanup;
}
out_len -= 1;

// Recompute IV, apparently OpenSSL's CBC destroys it
aes_set_encrypt_key((uint8_t *) session->s.s_enc, SCP_KEY_LEN, &aes_ctx);
aes_encrypt(session->s.ctr, encrypted_ctr, &aes_ctx);

aes_set_decrypt_key(session->s.s_enc, SCP_KEY_LEN, &aes_ctx);

DBG_CRYPTO(response_msg.st.data + 1, out_len,
           "CBC decrypting (%3d Bytes): ", out_len);
DBG_CRYPTO(encrypted_ctr, SCP_PRF_LEN, "IV: ");

aes_cbc_decrypt(response_msg.st.data + 1, decrypted_data, out_len,
                encrypted_ctr, &aes_ctx);
aes_remove_padding(decrypted_data, &out_len);
```

yubihsm.c

Since the libyubihsm <> HSM interaction happens in several logical message flows called **sessions**, the code first checks if the response message is actually meant for the current session to prevent mixups between concurrent sessions due to race conditions. The main message payload within the data section of the message is then decrypted and any remaining AES padding is removed.

Unfortunately, the `aes_remove_padding()` function that is called at the end of the code excerpt makes dangerous assumptions about the memory contents and length variable that it operates on:

```
void aes_remove_padding(uint8_t *in, uint16_t *len) {

  while (in[(*len) - 1] == 0) {
    (*len)--;
  }

  (*len)--;
}
```

yubihsm.c

As you can see, the function mainly cares about the memory contents at `in`, and not at all about the boundary conditions related to `len`.

If the function is called with `len = 0` or a pointer to a buffer filled with `0x00` bytes, then it will continue reading data and walking over memory **in front of the provided buffer** until it encounters a non-null data byte. This type of out-of-bounds memory access will not leak a lot of information, but ignores any memory boundaries that are in place along the way. Depending on the memory contents in front of the buffer and general memory layout, this can cause the libyubihsm program to be terminated with a segmentation fault.

Due to the way that the message parsing logic is written, it is in fact possible for a crafted message to reach `aes_remove_padding()` with an `out_len = 0` value. Messages with `response_msg.st.len = 9` are accepted and `SCP_MAC_LEN + 1 = 9` is subtracted from that length value during the parsing steps.

A problematic message would therefore consist of

- an acceptable response code such as `YHR_SUCCESS`
- the length value `0x0009`
- the correct session ID
- a valid 8-byte MAC code

During parsing, the code will then call `aes_remove_padding()` on a 0-byte inner data payload segment with length 0, which triggers the out-of-bounds read operations.

Since the attack scenario assumes that the HSM is malicious and also knows the encryption keys, it is also possible for the HSM to send a longer message of size `response_msg.st.len > 9` where the *encrypted* inner data payload section is chosen so that it decodes to all `0x00` bytes. This causes the `aes_remove_padding()` call to step over the provided payload buffer and then perform the same out of bounds reads as with the previous case. This means that both additional length checks as well as a better `aes_remove_padding()` have to be put in place to avoid this.

During execution, there are three possible outcomes:

1. Depending on compiler settings and other factors, the program memory can be such that `aes_remove_padding()` walks until reaching external memory boundaries of the program, causing a segmentation fault. This is the problematic outcome that represents a **denial of service** impact.

2. If the conditions are not right for a segmentation fault (*the attacker has no influence over this*), the program execution continues as `aes_remove_padding()` returns with some `out_len` value. Due to the unsigned integer overflow, this is usually a large value which is exceeding the allocated buffer size. In this case, a defense condition that was meant for another fault condition is triggered. The function then returns with an explicit `YHR_BUFFER_TOO_SMALL` error, halting further processing and preventing any practical impact:

```
out_len -= 3;
if (out_len > *response_len) {
  yrc = YHR_BUFFER_TOO_SMALL;
  goto cleanup;
}
```

yubihsm.c

3. Like 2), but with `0 < out_len < *response_len`. This has not been observed in practice and was not investigated further, but may cause additional memory problems.

CVE-2021-32489

While documenting CVE-2021-27217, I noticed a curious edge case. `response_msg.st.len = 9` looks like the shortest message length that can ever be accepted. After all, the library code wants 1 byte of valid channel ID and 8 bytes for the MAC data, so shorter messages are rejected automatically, right?

Actually, no. Looking closely at the code showed that the two fields are read from different sides of the payload:

The `sid` is read from the lowest memory position of the buffer:

```
if (session->s.sid != response_msg.st.data[0]) {
```

The `MAC` is read from the highest memory position of the buffer:

```
if (memcmp(response_msg.st.data + response_msg.st.len - SCP_MAC_LEN, mac_buf,
           SCP_MAC_LEN) != 0) {
```

The two fields may actually overlap, meaning that the first MAC byte can be a valid session ID, making the message valid!

Let's assume for a minute that both checks **do pass** on a message that was received with `response_msg.st.len = 8`. In this case, the `uint16_t out_len` variable is decremented too far by the combination of `out_len -= 8` and `out_len -= 1`. As a result, it overflows to `65535`.

This becomes a significant problem a few steps further into the parsing when running the AES decryption step:

```
aes_cbc_decrypt(response_msg.st.data + 1, decrypted_data, out_len,
                encrypted_ctr, &aes_ctx);
```

yubihsm.c

Through `aes_cbc_decrypt()`, OpenSSL is now told to work on a buffer of size 65535 while the buffer handed to it is actually far smaller. Unsurprisingly, this leads to a segmentation fault in OpenSSL, here shown in `gdb`:

```
Thread 1 "yubihsm-shell" received signal SIGSEGV, Segmentation fault.
0x00007ffff7d35ee7 in AES_decrypt () from /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
(gdb) bt
#0  0x00007ffff7d35ee7 in AES_decrypt () from /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
#1  0x00007ffff7e36796 in CRYPTO_cbc128_decrypt () from /usr/lib/x86_64-linux-gnu/libcrypto.so.1.1
#2  0x00007ffff7c14a17 in aes_cbc_decrypt (
  in=in@entry=0x7fffffff9b14 "\353[020\374\.032\214\r\rN\315~S\306\316)\030:\355\206\n.",
  out=out@entry=0x7fffffffa340 <incomplete sequence \353>,
  len=65535,
  iv=iv@entry=0x7fffffffa320 "\373\031\031N\035\243\036\345\353\351\036\221j\005\354\227:\236f\020\370c\016ä\373\346\344\241D\2
  <incomplete sequence \353>,
  ctx=ctx@entry=0x7fffffff9200)
at ../aes_cmac/aes.c:307
```

Unlike the distinct `aes_remove_padding()` based problem, this code path will always crash and does not depend on certain memory layouts.

This leaves us with the question: how can an attacker pass the session ID and message authentication checks in order to reach this case?

The slow and "naive" attack variant consists of the HSM behaving "harmlessly" by generally responding to request messages with non-malicious response messages. At the same time, it silently computes the potential MAC of a short attack message based on the current cryptographic parameters of the communication. Only if the first byte of this MAC matches the session ID of the request, it launches the attack and sends a short reply message.

Due to the cryptographic properties of the MAC, the relevant first MAC byte is basically changing in a random fashion and the average chance of a match with the current `sid` value is 1/256 for each request. So on average there is an attack opportunity every few hundred messages. This is plausible to reach in an automated setup where the HSM is queried regularly over some time. However, this situation does put some limits on the attack for an HSM that is queried rarely.

After thinking about this problem for a bit, I figured out an optimized attack variant that does this with significantly less message request opportunities. Our crafted message looks somewhat like this when plotted with the individual fields and offsets:

```
   1      2      3      4      5      6      7      8      9      10     11    |    byte position
[cmd]                                                                         |
     [length field]                                                           |    field usage
                    [ sid ]                                                   |
                    [                    MAC field                      ]     |
```

Normally, the MAC is computed over `[cmd] [length field] [payload data without MAC section]`, but since the remaining payload data has shrunk to zero and the length is fixed in our attack, the MAC is effectively just computed over the meta section `[cmd] [0x0008]`.

This appears somewhat hopeless at first since there apparently is no room for variations of the message, until one looks at the limits imposed on `cmd` during the parsing:

```
if (response_msg.st.cmd == YHC_ERROR) {
  // [...] error handling, returns with error
}
```

yubihsm.c

We can change `cmd` to any value except the error code `YHC_ERROR = 0x7f` when trying to trigger the bug!

Since message authentication codes are supposed to change wildly for each variation of the message, trying out the 255 different accepted variants for `cmd` results in the excellent *average* chance of 255/256 = `~99,6%` for a valid `sid <> first MAC byte` match that allows the attack. At worst, the malicious HSM has to wait out a few requests for an attack opportunity, but not a few hundred. That's a decent improvement.

Attack Scenario and Security Implications

Please note that the vulnerabilities are in libyubihsm on the host side. They can only be triggered by a trusted but malicious HSM or a man-in-the-middle attacker with similar knowledge of secrets. This is a very specific attack subclass that requires a lot of preparation and sophistication by the attacker. However, given that supply chain attacks are harder to defend against than most people realize (*see enclosure and tamper evident seal issues of the YubiHSM2 packaging*), it is beneficial to research and improve the defenses in this scenario as well.

If you are confident that your physical HSM2 modules are genuine or that your software stacks exclude libyubihsm then the described issues do not affect you.

CVSS Score

Yubico has rated the issues as follows:

ID	CVSS 3.1 Score	Parameters
CVE-2021-27217	4.4 (Medium)	AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:H
CVE-2021-32489	4.4 (Medium)	AV:N/AC:H/PR:H/UI:N/S:U/C:N/I:N/A:H

Note that the initial YSA-2021-01 advisory included inaccurate CVSS score information, but has been updated.

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-2021-27217

```
diff --git a/lib/yubihsm.c b/lib/yubihsm.c
index 647eada..7e4cfc5 100644
--- a/lib/yubihsm.c
+++ b/lib/yubihsm.c
@@ -360,8 +360,15 @@ static yh_rc _send_secure_msg(yh_session *session, yh_cmd cmd,
     goto cleanup;
   }

+
+  // POC: simulate 9-byte payload response
+  if(response_msg.st.len == 25) {
+    response_msg.st.len = 9;
+  }
+
   // Response is MAC'ed and encrypted. Unwrap it
   out_len = response_msg.st.len;
@@ -376,6 +383,8 @@ static yh_rc _send_secure_msg(yh_session *session, yh_cmd cmd,
                   session->s.s_rmac, SCP_KEY_LEN, mac_buf);
   response_msg.st.len = ntohs(response_msg.st.len);

+
+  // POC: assume MAC is correct
+  /*
   if (memcmp(response_msg.st.data + response_msg.st.len - SCP_MAC_LEN, mac_buf,
              SCP_MAC_LEN) != 0) {
     DBG_DUMPERR(response_msg.st.data + out_len - SCP_MAC_LEN, SCP_MAC_LEN,
@@ -383,7 +392,7 @@ static yh_rc _send_secure_msg(yh_session *session, yh_cmd cmd,
     yrc = YHR_MAC_MISMATCH;
     goto cleanup;
   }
-
+  */
   DBG_INFO("Response MAC successfully verified");
```

```
    out_len -= SCP_MAC_LEN;
```

As described previously, whether or not this leads to a segmentation fault is affected by compiler settings such as `-O0` and not expected to happen on all systems and configurations.

Second Issue Variant

```
diff --git a/lib/yubihsm.c b/lib/yubihsm.c
index 647eada..5c7da2e 100644
--- a/lib/yubihsm.c
+++ b/lib/yubihsm.c
@@ -360,6 +360,14 @@ static yh_rc _send_secure_msg(yh_session *session, yh_cmd cmd,
     goto cleanup;
   }

+  // POC: simulate malicious message
+  if(response_msg.st.len == 25) {
+    // copy the original MAC to the front of the data section
+    memcpy(response_msg.st.data, response_msg.st.data + response_msg.st.len - SCP_MAC_LEN, SCP_MAC_LEN);
+    // simulate the problematic length
+    response_msg.st.len = 8;
+  }
+
   // Response is MAC'ed and encrypted. Unwrap it
   out_len = response_msg.st.len;
   if (out_len < SCP_MAC_LEN - 3 ||
@@ -376,6 +384,8 @@ static yh_rc _send_secure_msg(yh_session *session, yh_cmd cmd,
                   session->s.s_rmac, SCP_KEY_LEN, mac_buf);
   response_msg.st.len = ntohs(response_msg.st.len);

+  // POC: assume MAC is correct, a malicious HSM can compute valid codes
+  /*
   if (memcmp(response_msg.st.data + response_msg.st.len - SCP_MAC_LEN, mac_buf,
              SCP_MAC_LEN) != 0) {
     DBG_DUMPERR(response_msg.st.data + out_len - SCP_MAC_LEN, SCP_MAC_LEN,
@@ -383,17 +393,22 @@ static yh_rc _send_secure_msg(yh_session *session, yh_cmd cmd,
     yrc = YHR_MAC_MISMATCH;
     goto cleanup;
   }
-
+  */
   DBG_INFO("Response MAC successfully verified");

   out_len -= SCP_MAC_LEN;

+  // POC: the contents at data[0] have to match the sid,
+  // which a malicious HSM can prepare for by mutations of `cmd`.
+  // Assume that this check is passed.
+  /*
   if (session->s.sid != response_msg.st.data[0]) {
     DBG_ERR("Session ID mismatch, expected %d, got %d", session->s.sid,
             response_msg.st.data[0]);
     yrc = YHR_GENERIC_ERROR;
     goto cleanup;
   }
+  */
   out_len -= 1;

   // Recompute IV, apparently OpenSSL's CBC destroys it
```

Example log:

```
./yubihsm-shell --connector=yhusb:// -a blink-device -p password
Session keepalive set up to run every 15 seconds
Created session 1
Segmentation fault
```

Coordinated Disclosure

Due to previous disclosures on this product with Yubico, there was a lot of familiar ground, although the communication was not as fluid as with the last disclosures.

Time-wise, the patch response time remains close to the 90 day mark. This is likely connected to the YubiHSM2 SDK release process that focuses on major releases. Although faster patch times would be preferable to me as a researcher, in this particular case the vulnerabilities are not really grave enough as to require an urgent release.

Overall, the disclosure process was okay, but there is some room for improvement.

Relevant yubihsm-shell Sources

variant	source	fix	references
Yubico upstream	GitHub	version **2.1.0**, bundled in SDK release 2021.03	YSA-2021-01
Fedora	Fedora	TBD	Bug 1936041

Detailed Timeline

Date	info
2020-12-14	Disclosure of issue #1 to Yubico
2020-12-15	Yubico acknowledges receiving the report
2020-12-15	POC for issue #1 provided to Yubico
2021-01-04	Yubico confirms issue #1
2021-01-18	Yubico shares proposed fix
2021-01-18	Positive feedback on fix, recommendation of additional parsing changes
2021-02-05	Yubico communicates planned disclosure date: 2021-02-22
2021-02-13	Disclosure of issue #2 to Yubico
2021-02-16	Yubico communicates CVE-2021-27217 as ID for the primary issue
2021-02-19	Yubico postpones planned disclosure date
2021-02-25	Yubico communicates planned disclosure date: 2021-03-04
2021-03-04	Release of patched HSM2 SDK version and Yubico advisory YSA-2021-01
2021-03-04	Publication of this blog post
2021-03-10	Yubico updates YSA-2021-01
2021-05-10	Yubico obtains second CVE

Note that the initial Yubico YSA-2021-01 advisory contained an incorrect publication date.

Bug Bounty

Yubico provided hardware as a bug bounty for this issue.

Christian Reitter

Information security and other interests.