Adventures in fuzzing libolm

2021-06-14 — Security, Research — Denis Kasak

Introduction

Hi all! My name is Denis and I'm a security researcher. Six months ago, I started working for Element on doing dedicated security research on important Matrix projects. After some initial focus on Synapse, I decided to take a closer look at libolm. In this entry, I'd like to present an overview of that work, along with some early fruits that came out of it.

TL;DR: we found some bugs which had crept in since libolm's original audit in 2016, thanks to properly overhauling our fuzzing capability, and we'd like to tell you all about it! The bugs were not easily exploitable (if at all), and have already been fixed.

Update: CVE-2021-34813 has now been assigned to this.

To give a bit of a background, libolm is a cryptographic library implementing the Double Ratchet Algorithm pioneered by Signal and it is the cryptographic workhorse behind Matrix. The classic algorithm is called Olm in Matrix land, but libolm also implements Megolm which is a variant for efficient encrypted group chats between many participants.

Since libolm is currently used in all Matrix clients supporting end-to-end encryption, it makes for a particularly juicy target. The present state of libolm's monopoly on Matrix encryption is somewhat unfortunate -- luckily there are some exciting new developments on the horizon, such as the vodozemac implementation in Rust. But for now, we're stuck with libolm.

To start, I decided to do a bit of fuzzing. libolm already had a fuzzing setup using AFL, but it was written a while ago. The state of the art in fuzzing had advanced quite rapidly in the last few years, so the setup was missing many modern features and techniques. As an example, the fuzzing setup was configured to use the now ancient afl-gcc coverage mode, which can be slower than the more modern LLVM-based coverage by a factor of 2.

I also noticed that the fuzzing was done with non-hardened binaries (instead of using something like ASAN), so many memory errors could've gone unnoticed. There were also no corpora available from previous fuzzing runs and some of the newer code was not covered by the harnesses.

Preparation

I decided to tackle these one by one, adding ASAN and MSAN builds as a first step. I took the opportunity to switch to AFL++ since it is a drop-in replacement and contains numerous improvements, notably improved coverage modes which are either much faster (e.g. LLVM-PCGUARD) or guaranteed to have no collisions (LTO)[1]. AFL++ also optimizes mutation scheduling (by using scheduling algorithms from AFLFast) and mutation operator selection (through MOpt). All of this makes it much more efficient at discovering bugs.

After this, I changed the existing harnesses to use AFL's persistent mode (which lowers process creation overhead and thus increases fuzzing performance). This change, combined with the switch to a newer coverage mode, increased the fuzzing exec/s from ~2.5k to ~5.5k on my machine, so this is not an insignificant gain!

After this preparatory work, I generated a small initial corpus and ran a small fleet of fuzzers with varying parameters. Almost immediately, I started getting heaps of crashes. Luckily, after some investigation, these turned out not to be serious bugs in the library but a double-free in the fuzzing harness! The double-free only got triggered when the input was of size 0. It also only happened with AFL++ and not vanilla AFL, presumably due to differences in input trimming logic, which must be the reason no one noticed this earlier. I quickly came up with a patch and resumed.

The plot thickens

I let the fuzzers run for a while. Since ASAN introduces a bit of a performance overhead, I only run a single AFL instance with ASAN variant of the binary. This is okay because all fuzzer instances actually synchronize their findings, which means every instance gets to see every input which increases coverage. When I came back to check, there was another crash waiting. This time the crashing input wasn't being generated continually so it looked much more promising -- and only the ASAN instance was crashing. *A-ha*!

Running the offending input on the ASAN variant of the harness revealed it was an invalid read one byte past the end of a heap buffer. The read was happening in the base64 decoder:

```
❮ ./build/fuzzers/fuzz_group_decrypt_asan "" pickled-inbound-group-session.txt <
=================================================================
==1838065==ERROR: AddressSanitizer: heap-buffer-overflow on address 0xf4a00795 a
READ of size 1 at 0xf4a00795 thread T0
    #0 0x5656065f in olm::decode_base64(unsigned char const*, unsigned int, uns
    #1 0x565607b5 in _olm_decode_base64 src/base64.cpp:165
    #2 0x565d5a9e in olm_group_decrypt_max_plaintext_length src/inbound_group_se
    #3 0x56558e75 in main fuzzers/fuzz_group_decrypt.cpp:46
    #4 0xf7509a0c in __libc_start_main (/usr/lib32/libc.so.6+0x1ea0c)
    #5 0x5655a0f4 in _start (/home/dkasak/code/olm/build/fuzzers/fuzz_group_decr

0xf4a00795 is located 0 bytes to the right of 5-byte region [0xf4a00790,0xf4a007
```

```
    #2 0xf7509a0c in __libc_start_main (/usr/lib32/libc.so.6+0x1ea0c)

SUMMARY: AddressSanitizer: heap-buffer-overflow src/base64.cpp:124 in olm::decoc
Shadow bytes around the buggy address:
  0x3e9400a0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e9400b0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e9400c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e9400d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e9400e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x3e9400f0: fa fa[05]fa fa 05 fa fa fa fa fa fa fa fa fa
  0x3e940100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e940110: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e940120: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e940130: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x3e940140: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
  Shadow gap:              cc
==1838065==ABORTING
```

F awed, unconditionally a ne input is 1 (mod 4) in le are necessarily valid payloads. Specifically, if the payload was not 0 (mod 4) in length, the code was assuming it was at least 2 (mod 4) or more in length and immediately read the second byte. This spurious byte was then incorporated into the output value.

I examined the code in an attempt to find a way to have it leak more than a single byte, but it was impossible. As it turned out, not even the full byte of useful information was encoded into the output -- due to the way the byte is encoded, only about 6 bits of useful information ended up in the output value.

Still, even a single leaked bit is too much in a cryptographic context. Could we do some heap hacking so that something of interest is placed there and then have it be leaked to us?

I next tracked down all call sites of the vulnerable function `olm::decode_base64`. Most of them were immune to the problem si `olm::decode_base64_length` s that the base64 p here their base64 inputs come from. Promisingly, two of them received input from other conversation participants, but they either had no way of leaking the information back to the attacker or they hardcoded the number of bytes to be processed, after ensuring the input was of some minimum length. The output of the remaining function `olm_pk_decrypt` is never sent anywhere externally, so there was again no way of leaking the data to the attacker.

In conclusion, even though this invalid read is a valid bug, I was not able to find a working exploit for it.

But wait a second! Something was still bothering me about `olm_pk_decrypt`. It's a fairly complex function, *receives several string inputs from the homeserver* and it itself isn't tested by any of the harnesses. Furthermore, the reason I started looking at it in the first place is that it was missing the `olm::decode_base64_length` check. Perhaps it warrants a closer look?

It does

And sure enough, there was something amiss. As `olm_pk_decrypt` receives three base64 inputs from the homeserver: the ciphertext to decrypt, an ephemeral public key and a MAC. All three are eventually passed to `olm::decode_base64` to be decoded. Yet there was only a single length check there, to ensure the decrypted ciphertext would fit its output buffer. What would happen if the server returned a public key that was longer than expected?

```
struct _olm_curve25519_public_key ephemeral;
olm::decode_base64(
    (const uint8_t*)ephemeral_key, ephemeral_key_length,
    (uint8_t *)ephemeral.public_key
);
```

As can be seen from the snippet, the decoded version of public key gets written to `ephemeral.public_key`, which is an array allocated on the stack. If the input is longer than expected, this will become a stack buffer overflow.

homeserver. Your other devices can then retrieve those keys from the homeserver, making it possible to view all of your private conversations on each of your devices.

I decided to go for an end-to-end test to confirm the bug is triggerable by connecting with the latest Element Android from my test phone to my homeserver, with mitmproxy sitting in between. This allowed me to write a small mitmproxy script which intercepts HTTP calls fetching the E2E encryption keys from the homeserver and modifies the response so that the key is longer than expected.

```
import json

from mitmproxy import ctx, http


def response(flow: http.HTTPFlow) -> None:
    if ("/_matrix/client/unstable/room_keys/keys" in flow.request.pretty_url
            and flow.request.method == "GET"):

        response_body = flow.response.content.decode("utf-8")
        response_json = json.loads(response_body)

        rooms = response_json["rooms"]
        room_id = list(rooms.keys())[0]

        sessions = rooms[room_id]["sessions"]
        session = list(sessions.keys())[0]
        session_data = sessions[session]["session_data"]

        ephemeral = session_data["ephemeral"]
        ctx.log.info(f"Replacing ephemeral key '{ephemeral}' with '{ephemeral *
        session_data["ephemeral"] = ephemeral * 10

        modified_body = json.dumps(response_json).encode("utf-8")
        flow.response.content = modified_body
```

This longer value is then eventually passed by Element Android to libolm's `olm_pk_decrypt`, which triggers the buffer overflow. With all of that in place, I deleted the local encryption key backup on my device and asked for it to be restored from the server:

```
F libc    : stack corruption detected (-fstack-protector)
F libc    : Fatal signal 6 (SIGABRT), code -6 (SI_TKILL) in tid 24517 (DefaultD
F DEBUG   : *** *** *** *** *** *** *** *** *** *** *** *** *** *** *** ***
F DEBUG   : Build fingerprint: 'xiaomi/tissot/tissot_sprout:9/PKQ1.180917.001/V
F DEBUG   : Revision: '0'
F DEBUG   : ABI: 'arm64'
F DEBUG   : pid: 24459, tid: 24517, name: DefaultDispatch  >>> im.vector.app <<
F DEBUG   : signal 6 (SIGABRT), code -6 (SI_TKILL), fault addr --------
F DEBUG   : Abort message: 'stack corruption detected (-fstack-protector)'
F DEBUG   :     x0  0000000000000000  x1  0000000000005fc5  x2  000000000000000
F DEBUG   :     x4  0000000000000000  x5  0000000000000000  x6  000000000000000
F DEBUG   :     x8  0000000000000083  x9  7d545b4513138652  x10 000000000000000
F DEBUG   :     x12 0000000000000001  x13 0000000060b0f2a9  x14 0022ed916fede20
F DEBUG   :     x16 00000079e741b2b0  x17 00000079e733c9d8  x18 000000000000000
F DEBUG   :     x20 0000000000005fc5  x21 0000007940e3c400  x22 000000000000026
F DEBUG   :     x24 000000000000002f  x25 000000793d9653f0  x26 000000794830336
F DEBUG   :     x28 00000000000001d0  x29 0000007945dd37d0
F DEBUG   :     sp  0000007945dd3790  lr  00000079e732e00c  pc  00000079e732e03
```

Impact

This vulnerability is a server-controlled stack buffer overflow in Matrix clients supporting room key backup.

Of course, the largest fear stemming from any remotely controlled stack buffer overflow is code execution. This is perhaps even doubly so in a cryptographic library, where we have the additional worry of an attacker being able to leak our dearly protected conversations.

The federated architecture of Matrix may be somewhat of a mitigating circumstance in this case, since users are much more likely to know and trust the homeserver owner, but we don't want to have to rely on this trust.

Native binaries

Luckily, on its own, this bug is not enough to successfully execute code on native binaries. By default, libolm is compiled for all supported targets with stack canaries (also called stack protectors or stack cookies), which are magic values unknown to the attacker, placed just before the current function's frame on the stack. This value is checked upon returning from the function -- if its value is changed, the process aborts itself to prevent further damage. This is evident from the `Abort message: 'stack corruption detected (-fstack-protector)'` message above. Besides canaries, other system-level protections exist to make exploiting bugs such as this harder, such as ASLR.

WASM

With WASM, the analysis is much more complicated due to its very different memory and execution model. In WASM, the unmanaged stack is generally much more vulnerable due to it missing support for stack canaries. This implies a stack buffer overflow can not only overwrite the frame of the function in which the overflow occurred but also *all parent frames*.

On the other hand, due to typed calls and much stronger control-flow integrity techniques, it's much harder for the attacker to make the code do something that is (maliciously) useful. Notably, return addresses live outside unmanaged memory and are out of reach to the attacker. Because of this, the primary way of influencing code execution is by manipulating `call_indirect` instructions in such a way as to call.

The analysis of the impact of this bug on the WASM binary is thus left as an exercise to the reader. If you're interested, the 2020 USENIX paper Everything Old is New Again: Binary Security of WebAssembly is a great starting point.

The fix

Once the problems were identified, the patches were rather trivial and the issues were promptly resolved. The first libolm release that includes the fix is 3.2.3 which was released on 2021-05-25.

We reached out to all Matrix clients which were determined to be affected. The Element client versions which first fix the issue are as follows:

- Element Web/Desktop: v1.7.29
- Element Android: v1.1.9
- Element iOS: v1.4.0

For the mobile clients, these versions are already available in their respective application stores at the time of publishing this post. If you haven't already, please upgrade.

Future work

Even though the fuzzing setup is in a much better shape now (or rather will be, since I still have some PRs to merge upstream), there's still a lot that can be done to further improve it.

Right now, there are undoubtedly parts of the codebase that are not fuzzed well. The reasons for this range from the obvious, like some parts of the code simply not being called by any the existing harnesses, to more subtle ones such as the fact that cryptographic operations form a nearly-insurmountable natural barrier for naive fuzzing operations[3]. Finally, some of the existing harnesses accept additional parameters as command-line arguments, meaning we would have to re-run the same harness with different values of those parameters in order to reach full coverage of the code. This is suboptimal.

So the plan for future work is roughly as follows:

1. Write missing harnesses to cover more portions of the codebase.
2. Write starting corpus generators. These should generate believable, valid input for each of the harnesses. For example, for the decryption harness, we should generate a variety of encrypted messages: empty, short, long, text, binary, etc.
3. Modify the harnesses so that their extra parameters are determined from the fuzzed input. This will allow the fuzzer to vary these itself, which reduces the importance of the human in the loop and makes it harder to forget some combination.
4. Fuzz for some time until coverage stops increasing. The corpora generated should be saved so that future fuzzing attempts can resume from an earlier point so that this work is not wasted.
5. Use afl-cov to investigate which parts of the code are not covered well or at all. This should inform us what further changes are needed.
6. Write intelligent, custom mutators. These will allow the fuzzer to take a valid input and easily produce another valid input instead of only corrupting it with a high probability.
7. Design harnesses which test for wanted semantic properties instead of only memory errors.

It's very exciting that we're able to do full-time security research on Matrix these days (thanks to Element's funding), and going forwards we'll publish any interesting discoveries for the visibility and education of the whole Matrix community. We'd also like to remind everyone that we run an official Security Disclosure Policy for Matrix.org and we'd welcome other researchers to come join our Hall of Fame! (And hopefully we will get more bounty programmes running in future.)

---

1. In the context of fuzzing, collisions are situations where two different execution paths appear to the fuzzer as the same one due to technical limitations. Classically, AFL tracks coverage by tracking so-called "edges" (or "tuples"). Edges are really pairs of (A, B), where A and B represent basic blocks. Each edge is meant to represent a different execution "jump", but sometimes, as the number of basic blocks in a program grows, two different execution paths can end up being encoded as the same edge. LTO mode in AFL++ does some magic so that this is guaranteed not to happen. ↩
2. By *remainder byte*, I mean bytes which are not part of a group of 4. These can only occur at the end of a base64 payload and they're the ones that get suffixed with padding in padded base64. ↩
3. Classic fuzzers famously have a hard time circumventing magic values and checksums, and cryptography is full of these. This is further complicated by the fact that the double ratchet algorithm is very stateful and depends on the two ratchets evolving in lockstep. This means that even if, for example, the decryption harness is supplied with a corpus of valid encrypted messages, the mutations done by the fuzzer would only manage to produce corrupted versions of those messages which will fail to decrypt, but it will ~never manage to produce a different *valid* encrypted message. ↩

PREVIOUS
This Week in Matrix 2021-06-11

NEXT
Synapse 1.36.0 released

All posts