New critical remote buffer overflow vulnerability in matrixssl TLSv1.3 server message processing

(CVE-2022-43974)

by Robert Hörr (e-mail: robert.hoerr@telekom.de) and
Alissar Ibrahim (e-mail: alissar.ibrahim@telekom.de)
(Security Evaluators of the Telekom Security Evaluation Facility)

A new critical remote buffer overflow vulnerability (CVE-2022-43974) was discovered in the matrixssl library (versions 4.5.1- 4.0.0, https://github.com/matrixssl/matrixssl) by Security Evaluators of Telekom Security with modern fuzzing methods. The vulnerability allows an attacker to overwrite a large part of the RAM of a matrixssl server with his data over the network. This might allow remote code execution. Additionally, the issue can be used to get RAM information from the server. The matrixssl developers have fixed the vulnerability (https://github.com/matrixssl/matrixssl/releases/tag/4-6-0-open). Special thanks to mrs. Ibrahim for the motivation boost to test matrixssl in more detail.

What is the matrixssl library?

The matrixssl library is an open source project providing implementations of the security network protocols SSL, TLS and DTLS for embedded devices. The matrixssl library is employed in many commercially used systems. The security protocols ensure that two endpoints can communicate in a secure way over a network like the internet, so that an attacker is not able to read or modify the exchanged data.

How was the vulnerability discovered?

Computer software is becoming more complex. So, it is almost impossible to perform a complete source code review with reasonable coverage. For this reason, modern fuzzing methods are used to discover vulnerabilities. The fuzzing methods include, among other things, AFL, libFuzzer and AdressSanitizer. The tools AFL and libFuzzer are code coverage based fuzzer which are the next generation of fuzzing tools. The matrixssl library was fuzzed using these fuzzing methods. The AddressSanitizer found the reported buffer overflow in this article.

Where is the vulnerability located in the source code?

The vulnerability is located in the TLSv1.3 server message processing. The function matrixSslReceivedData() processes the socket input data from the client. This function includes the function matrixSslDecode(), which parses a TLSv1.3 message with the function matrixSslDecodeTls13(). This function changes the variable uint32 len with a subtraction only. At this point, an integer wrap around can happen, which is not avoided by a length check. In worst case, the variable len gets the value 4294967295.

In the function matrixSslReceivedData() the memcpy(ssl->outbuf + ssl->outlen, ssl->inbuf, len) operation copies len bytes from the socket input data to the output data to be sent (TLS packet). Hence, an overflow will occur.

The appendix shows some code sections in more detail. The data array contains an example of a crafted TLS packet.

How is the vulnerability exploitable by an attacker?

The issue can be used to perform two attack scenarios. At the first attack, an attacker sends a special crafted TLS packet with e.g. a reverse shell over the network to a

TLSv1.3 matrixssl server to perform a code execution. This maximum size of this crafted TLS packet is 65 kb.

At the second attack, an attacker sends a special crafted TLS packet over the network to a TLSv1.3 matrixssl server to get RAM information from the server.

What do we learn from this?

Code coverage based fuzzing combined with the AddressSanitizer is a powerful method to discover e.g. buffer overflows. With increasingly complex source codes, it is a resource-efficient alternative to source code reviews, because this fuzzing approach can be done mainly automatically. As there exist many approaches for fuzzing, it is the art of fuzzing to find the best approach. We have already discovered several vulnerabilities with our fuzzing approach.

Appendix: code sections of the matrixssl library (version 4.5.1)

```
matrixsslAPI.c:
    matrixSslReceivedData()
        uint32 len;
        matrixSslDecode()
        Memcpy(ssl->outbuf + ssl->outlen, ssl->inbuf, len);

sslDecode.c:
    matrixSslDecode()
        matrixSslDecodeTls13()

tls13Decode.c:
    matrixSslDecodeTls13():
         len -= parsedBytes; (wrap around)


unsigned char data[] = {
  0x16, 0x00, 0x08, 0x00, 0x5a, 0x01, 0x00, 0x00, 0x45, 0x00, 0x00, 0x01,
  0x05, 0xe5, 0xff, 0xff, 0xea, 0xff, 0x32, 0xff, 0xea, 0x05, 0x00, 0x00,
  0x02, 0x02, 0x74, 0x00, 0x1b, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x04,
  0x00, 0x01, 0xb3, 0xff, 0xff, 0x00, 0x08, 0x00, 0x00, 0x10, 0x06, 0x03,
  0x55, 0x1d, 0x13, 0x02, 0x00, 0x5f, 0x13, 0x02, 0x00, 0x00, 0x00, 0x00,
  0x00, 0x40, 0x00, 0x04, 0x00, 0x01, 0xb3, 0xff, 0x50, 0x08, 0x00, 0x00,
  0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0x00,
  0x30, 0xff, 0x45, 0x00, 0x10, 0x06, 0x03, 0x55, 0x1d, 0x13, 0xff, 0x14,
  0x00, 0xff, 0x00, 0x01, 0x01, 0x14, 0xff, 0xf7, 0x00, 0x01, 0x01, 0x14,
  0x00, 0xff, 0x00, 0x01, 0x01, 0x14, 0x01, 0xff, 0x00, 0x01, 0x01, 0x14,
  0x00, 0xff, 0x00, 0x01, 0x01, 0x14, 0x00, 0xff, 0x00, 0x01, 0x01, 0x14,
  0xff, 0xf7, 0x00, 0x01, 0x01, 0x14, 0x00, 0xff, 0x00, 0x01, 0x01, 0x14,
  0x01, 0xff, 0x00, 0x01, 0x01, 0x14, 0x00, 0xff, 0x00, 0x01, 0x01, 0x14,
  0x00, 0xff, 0x00, 0x01, 0x01, 0x14, 0x00, 0xff, 0x00, 0x01, 0x01
};
unsigned int data_size = 167;
```