New critical remote buffer overflow vulnerability in axTLS TLS record size parsing
(CVE-2019-8981)

by Robert Hörr (e-mail: robert.hoerr@t-systems.com)
(Security Evaluator of the Telekom Security Evaluation Facility)

A new critical remote buffer overflow vulnerability (CVE-2019-8981) in the axTLS library for embedded devices (version 2.1.4, http://axtls.sourceforge.net) was discovered on 2019 February 20 by Telekom Security Experts with modern fuzzing methods, which possibly allows remote code executions. A new fixed version (2.1.5) countering this is now available for download (https://sourceforge.net/projects/axtls/files/2.1.5/).

What is the axTLS library?

The axTLS library is an open source project providing a TLS implementation (like the well-known open source project openssl) for embedded devices. The library has been downloaded 1,839 times since July 2017 and is used in commercial products.
The advantages of the library are the small size of the source code and the small number of features compared to openssl. These advantages do not automatically mean that axTLS has less vulnerabilities than openssl, because this depends on multiple factors like coding style, reviews, testing, distribution and so on.

Why was the axTLS library fuzzed?

Some security devices use the axTLS library to secure their network traffic with TLS. The task of the Security Evaluators of the Evaluation Facility is to check the IT-Security of security devices according to a security scheme like Payment Card Industry or Common Criteria. These security schemes require among other things testing the security devices with fuzzing methods to find vulnerabilities like buffer overflows. The reason for using fuzzing additionally to e.g. source code review is i.a. that source codes are becoming bigger and more complex.

How was the axTLS library fuzzed?

Our fuzzing approach is code coverage based fuzzing (which is also used by AFL and LibFuzzer) combined with the AdressSanitizer that was used to find all buffer overflows. The LibFuzzer was also used to fuzz the axTLS library. In order to use the LibFuzzer an interface between the fuzzer and the axTLS library is needed. This interface includes the functions of the axTLS library which one would like to fuzz.

Where is the vulnerability located in the source code?

The vulnerability is explained based on the source code sections in the appendix of this blog article. The bold marked source code lines are important for understanding. The following enumeration describes the vulnerability step by step. The number of the corresponding enumeration can be found in the bold marked source code lines as a comment to understand the enumeration (e.g. //step1).

1. The pointer *buf* of the function *basic_read* points to the array *bm_all_data* with the size 17408 (16384+1024).
2. The function *SOCKET_READ* writes new data to *buf* at index *ssl→bm_read_index* with size *ssl→need_bytes - ssl→got_bytes.*
3. Only if all of the needed bytes *ssl→need_bytes* are received, the function *basic_read* will be executed further (if(ssl->got_bytes < ssl→need_bytes)).

4. The variable *ssl->need_bytes* gets the record length, which can be greater than 17408 e.g. 65535, because the record length of the TLS protocol is coded in two bytes.
5. If the variable *ssl→need_bytes* is greater than 17403, SSL_ERROR_RECORD_OVERFLOW will be sent, but the variable *ssl→need_bytes* will not be reset. Hence the variable *ssl→need_bytes* keeps the value greater than 17408.
6. By the next function calls of *basic_read* the variable *ssl->got_bytes* tries to reach the value of the variable *ssl→need_bytes* (see 3.).
7. If the variable *ssl→got_bytes* reaches the value 17408, the buffer *buf* will be overflowed with up to 48127 bytes by the function SOCKET_READ (see 2.). AddressSanitizer detects a heap-buffer-overflow.

How is the vulnerability exploitable by an attacker?
An attacker sends a TLS packet with a record length of 65535 in the TLS record header. The first and next transmitted TLS packets are in sum 17408 bytes of e.g. random data. Now the buffer *buf* is completely full and the next received TLS data are stored outside of *buf*. Hence the attacker can send 48127 bytes to change the behavior of the system or TLS. Possibly the attacker can perform a remote code execution attack.

What do we learn from this?
Code coverage based fuzzing combined with the AddressSanitizer is a powerful method to find 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.

Appendix: source code sections of the axTLS library (version 2.4.1)

```
tls1.h:
#define RT_MAX_PLAIN_LENGTH      16384
#define RT_EXTRA                 1024
#define BM_RECORD_OFFSET            5

struct _SSL
{
...
    uint8_t bm_all_data[RT_MAX_PLAIN_LENGTH+RT_EXTRA];
...

tls1.c:
SSL *ssl_new(SSL_CTX *ssl_ctx, int client_fd)
{
...
    ssl->bm_data = ssl->bm_all_data+BM_RECORD_OFFSET; /* space at the start */
...

int basic_read(SSL *ssl, uint8_t **in_data)
{
    int ret = SSL_OK;
    int read_len, is_client = IS_SET_SSL_FLAG(SSL_IS_CLIENT);
    uint8_t *buf = ssl→bm_data; //step1

    if (IS_SET_SSL_FLAG(SSL_SENT_CLOSE_NOTIFY))
        return SSL_CLOSE_NOTIFY;
    //critical remote buffer overflow vulnerability
    read_len = SOCKET_READ(ssl->client_fd, &buf[ssl->bm_read_index],
                            ssl→need_bytes-ssl→got_bytes); //step2, step7

    if (read_len < 0)
    {
#ifdef WIN32
        if (GetLastError() == WSAEWOULDBLOCK)
#else
        if (errno == EAGAIN || errno == EWOULDBLOCK)
#endif
            return 0;
    }

    /* connection has gone, so die */
    if (read_len <= 0)
    {
        ret = SSL_ERROR_CONN_LOST;
        ssl->hs_status = SSL_ERROR_DEAD;  /* make sure it stays dead */
        goto error;
    }

    DISPLAY_BYTES(ssl, "received %d bytes",
            &ssl->bm_data[ssl->bm_read_index], read_len, read_len);

    ssl->got_bytes += read_len; //step6
    ssl->bm_read_index += read_len; //step6

    /* haven't quite got what we want, so try again later */
    if (ssl->got_bytes < ssl→need_bytes) //step3, step6
        return SSL_OK;

    read_len = ssl->got_bytes;
    ssl->got_bytes = 0;

    if (IS_SET_SSL_FLAG(SSL_NEED_RECORD))
    {
        /* check for sslv2 "client hello" */
        if ((buf[0] & 0x80) && buf[2] == 1)
        {
#ifdef CONFIG_SSL_FULL_MODE
            printf("Error: no SSLv23 handshaking allowed\n");
#endif
            ret = SSL_ERROR_NOT_SUPPORTED;
            goto error; /* not an error - just get out of here */
        }

        ssl->need_bytes = (buf[3] << 8) + buf[4]; //step4

        /* do we violate the spec with the message size?  */
       if   (ssl->need_bytes   >   RT_MAX_PLAIN_LENGTH+RT_EXTRA-BM_RECORD_OFFSET)
//step5
        {
            ret = SSL_ERROR_RECORD_OVERFLOW;
            goto error;
        }

...
```