Remote buffer overflow vulnerability in SharkSSL TLS handshake processing CVE-2024-53379

by Mauritz van den Bosch (mauritz.vandenbosch@telekom.de) and
Robert Hörr (robert.hoerr@telekom.de)

A new remote buffer overflow vulnerability was discovered in the latest version of the SharkSSL library from 05.05.2024 (https://github.com/RealTimeLogic/SharkSSL) by security evaluators of Deutsche Telekom Security GmbH and Deutsche Telekom AG with modern fuzzing methods. The vulnerability allows an attacker to read large regions of the SharkSSL TLS server's memory. This is likely to result in a segmentation fault and can be used for a remote Denial-of-Service attack by an attacker.
Special thanks to Robert Hörr for his support, supervision and guidance during this research.

What is the SharkSSL library?

SharkSSL is an open-source software that provides security implementations of various protocols including TLS, WebSocket, MQTT, SMTP etc. for embedded devices, but SharkSSL is not exclusive to this field of application. Especially TLS is a security protocol to ensure communication of two endpoints over a network like the Internet in a secure way, thus an attacker is not able to read or modify the exchanged data.

How was the vulnerability discovered?

Software projects usually gain in complexity over time, especially when many additional features are introduced. Therefore, it is hard to perform source code reviews manually by human with reasonable coverage in feasible time. For this reason, modern fuzzing methods are utilized to discover vulnerabilities like buffer overflows automatically. Among others, libFuzzer and AdressSanitizer have been used for automatically testing this software. LibFuzzer is a code-coverage based fuzzer and has been combined with AdressSanitizer, which detects memory errors at runtime. The TLS handshake implementation of SharkSSL has been tested using these two combined methods.

Where is the vulnerability located in the source code?

The vulnerability is located in the TLSv1.2 server-site handshake implementation. The function `seSec_handshake` is performing the TLS handshake using the socket and context information about the TLS connection. This function invokes the function `handleptrauth` which parses the TLS Extensions of a received Client Hello message. In `handleptrauth` the Extension Length field of the TLS Client Hello message is provided in the unsigned 16-bit integer variable `len`. So, an attacker can manipulate the value of this variable.
Within the function there is a loop with condition `len` > 2 and it decrements the variable `len` multiple times in each iteration. Also, the pointer `registeredevent` points to the Client Hello message inside the memory and is incremented multiple times as well as de-referenced in each iteration.

As mentioned above, the loop subtracts small values from `len` repeatedly, which leads to an integer wrap around and resulting in a high value. This in conjunction with the loop condition and the iteratively incrementation of `registeredevent` leads to a buffer overflow.

A variant of this vulnerability allows a controlled manipulation of the pointer `registeredevent`. A case distinction allows the addition of this pointer by a variable named `paramnamed`. This is an unsigned 16-bit integer value read from the TLS Extensions section, thus an attacker-controlled value. Additional to the incrementations of the pointer `registeredevent` as described above, that exact pointer is added with the value of `paramnamed`. In combination with the loop iterations an iterative increase of `registeredevent` by a chosen value might be possible, thus a controlled manipulation of the pointer cannot be ruled out.

In the Appendix, a function trace is provided including one location of the pointer de-reference triggering the overflow. Also, the code location of the pointer manipulation using controlled variable `paramnamed` is shown there.

How is the vulnerability exploitable by an attacker?

The vulnerability can be used to trigger reads of memory locations beyond the allocated buffer. The loop structure allows repetitive manipulations of the pointer `registeredevent`, which can be used for reading memory locations of unknown extend. Usually, these accesses of memory results in segmentation faults and leads to termination of the program. This can be exploited to perform a remote Denial-of-Service attack against the TLS server and implies that clients are not able to connect to the TLS server anymore. Especially in the context of Operational-Technology (OT), where SharkSSL seems to be used, availability is an important objective.
The controlled manipulations of the pointer might be used to trigger an information leak at later points of the control flow. By utilizing write instruction e.g. `memcpy` or internal functions Remote-code-Execution might be possible.

What do we learn from this?

Code coverage-based fuzzing combined with the AddressSanitizer is a powerful method to discover vulnerabilities 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 section with buffer overflow

This is an example section of `SharkSSL.c` that can trigger the buffer overflow. In the same function are multiple sections like this which can trigger the overflow, too.

```
2383        while (len >= 2)
2384        {
2385              prminstwrite = (U16)(*registeredevent++) << 8;
2386              prminstwrite += *registeredevent++;
2387              len -= 2;
            [...]
```

Appendix: Code section with manipulation by variable paramnamed

This example shows the manipulation of the pointer `registeredevent` by the variable `paramnamed`. The variable can might be used to control the value of the pointer.

```
2795        default:
2796              len -= paramnamed;
2797              registeredevent += paramnamed;
2798              break;
            [...]
```

Appendix: AddressSanitizer output

Output from AddressSanitizer on accessing a memory location beyond the allocated memory. This is triggered by a malformed TLS Client Hello message. The shown location within `handleptrauth` is one example of the overflow. The root cause is the integer wrap around as discussed above.

```
==30620==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61d0000048d4
at pc 0x64589a078111 bp 0x7ffc220ab8d0 sp 0x7ffc220ab8c8
READ of size 1 at 0x61d0000048d4 thread T0
    #0 0x64589a078110 in handleptrauth /SharkSSL/src/SharkSSL.c:2385:29
    #1 0x64589a060d5f in configdword /SharkSSL/src/SharkSSL.c:4446:22
    #2 0x64589a0f8d77 in SharkSslCon_decrypt SharkSSL/src/SharkSSL.c:21428:19
    #3 0x64589a11589b in seSec_readOrHandshake selib.c
    #4 0x64589a107a72 in main SharkSSL/tools/server_handshake.c:66:20
    #5 0x724c0a429d8f in __libc_start_call_main
csu/../sysdeps/nptl/libc_start_call_main.h:58:16
    #6 0x724c0a429e3f in __libc_start_main csu/../csu/libc-start.c:392:3
    #7 0x645899f8b3a4 in _start (/SharkSSL/tools/server_handshake+0x283a4)
(BuildId: 2d7e8f38becbb19370010f359396e2b70cf2cd7e)

Address 0x61d0000048d4 is a wild pointer inside of access range of size
0x000000000001.
SUMMARY: AddressSanitizer: heap-buffer-overflow
/SharkSSL/tools/./../src/SharkSSL.c:2385:29 in handleptrauth
```

```
Shadow bytes around the buggy address:
  0x0c3a7fff88c0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff88d0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff88e0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff88f0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff8900: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c3a7fff8910: fa fa fa fa fa fa fa fa fa fa[fa]fa fa fa fa fa
  0x0c3a7fff8920: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff8930: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff8940: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff8950: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x0c3a7fff8960: 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
      [...]
```