Remote buffer overflow vulnerability in SharkSSL TLS Client Key Exchange handshake processing CVE-2024-48075

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 09.09.2024 (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 trigger the reading of unallocated memory in 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. Specifically, TLS is a security protocol to ensure communication between 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 complexity over time, especially when many additional features are introduced. Therefore, it is hard to perform source code reviews manually by humans with reasonable coverage in a 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 for processing a received TLS Client Key Exchange message. The function `seSec_handshake` performs the TLS handshake using the socket and context information about the TLS connection. This function invokes the function `SharkSslCon_decrypt` which parses the received TLS Client Key Exchange message. That function uses the pointer `registeredevent` for parsing the content of the received message. In the case of a fragmented TLS Client Key Exchange message, the code location is reached, where the pointer `registeredevent` is manipulated, and this sets up the overflow:

There, the unsigned 16-bit integer variable `backupdata` is set to the current value referenced by `registeredevent`. Afterwards this variable is subtracted by 4, and the pointer `registeredevent` is

added to the current value of `backupdata`.
Since the pointer is referring to the received TLS Client Key Exchange message, the value of the variable `backupdata` is dependent on the message data. This means the TLS Client Key Exchange message data is attacker-controlled and can lead to a high value of `backupdata`. This high value is added to the pointer `registeredevent` which most likely references an unallocated memory region. At a later point within `SharkSslCon_decrypt`'s control-flow, `registeredevent` is dereferenced, and the overflow is triggered. In the Appendix, the described code locations are presented.

How is the vulnerability exploitable by an attacker?

The vulnerability can be used to trigger one or many reads of memory locations beyond the allocated buffer. Usually, these accesses to memory result in segmentation faults and lead to the termination of the program. This can be exploited to perform a remote Denial-of-Service attack against the SharkSSL 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 `registeredevent` might be used to trigger an information leak at later points of the control flow. By utilizing write instructions, 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, like 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 are many approaches to 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 pointer addition

This is the section of `SharkSSL.c` that prepares the buffer overflow. The pointer `registeredevent` is not dereferenced here, but it is added by a value controlled by the TLS Client Key Exchange message.

```
21221 if (o->flags & SHARKSSL_FLAG_FRAGMENTED_HS_RECORD){
[…]
21230 backuppdata = ((U16)(* registeredevent++)) << 8;
21231 backuppdata += *registeredevent++ - 4;
21232 registeredevent += backuppdata;
21233 }
```

Appendix: Code section with pointer dereference

This example shows the dereference of the pointer `registeredevent` that is triggering the buffer overflow.

```
21317 if ((o->major) || (0 == (*registeredevent & 0x80)) || SharkSsl_isClient(o→sharkSsl))
21318 {
21319 regsetcopyin = *registeredevent++;
[…]
```

Appendix: AddressSanitizer output

Below is the output from AddressSanitizer on accessing a memory location beyond the allocated memory. This is triggered by a fragmented TLS Client Key Exchange message. The shown location within `SharkSSLCon_decrypt` might not be the only location triggering the overflow. The root cause is the attacker-controlled addition of the pointer `registeredevent` as discussed above.

```
==511345==ERROR: AddressSanitizer: SEGV on unknown address 0x621000010121 (pc
0x65435ade3750 bp 0x7ffcd2da81f0 sp 0x7ffcd2da7680 T0)
==511345==The signal is caused by a READ memory access.
#0 0x65435ade3750 in SharkSslCon_decrypt SharkSSL/src/SharkSSL.c:21319:22
#1 0x65435ae05994 in seSec_readOrHandshake selib.c
#2 0x65435adf72db in LLVMFuzzerTestOneInput
#3 0x65435ac47212 in fuzzer::Fuzzer::ExecuteCallback
#4 0x65435ac31090 in fuzzer::RunOneTest
#5 0x65435ac36d57 in fuzzer::FuzzerDriver
#6 0x65435ac60372 in main
#7 0x716fab829d8f in __libc_start_call_main
#8 0x716fab829e3f in __libc_start_main csu/../csu/libc-start.c:392:3
#9 0x65435ac2ba34 in _start
```