New critical remote buffer overflow vulnerability in wolfSSL TLSv1.3 PSK extension parsing

(CVE-2019-11873)

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-2019-11873) was discovered in the wolfSSL library (versions 3.12.0-stable - 4.0.0-stable, http://www.wolfssl.com) 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 wolfSSL server with his data over the network. This might allow remote code execution. The wolfSSL developers have fixed the vulnerability (https://github.com/wolfSSL/wolfssl/pull/2239).

What is the wolfSSL library?

The wolfSSL library is an open source project providing implementations of the security network protocols SSL, TLS and DTLS for embedded devices. The wolfSSL library is employed in many commercially used systems, for example, MySQL and cURL. 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.

What is the TLS PSK extension?

The TLS PSK extension allows a secure communication with previously exchanged shared keys. Hence, both endpoints have the same keys and the key agreement process of the TLS handshake to negotiate a shared key is not necessary anymore. The advantages of the extension is that a PKI is not needed and public key operations can be avoided.

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 wolfSSL library was fuzzed using these fuzzing methods. The AddressSanitizer found the reported buffer overflow(stack overflow) in this article.

Where is the vulnerability located in the source code?

The vulnerability is located in the TLSv1.3 server parsing process of the PSK extension of the client hello packet. For a better understanding, the vulnerability is explained based on the source code sections in the appendix of this article. The source code lines marked in bold are important for understanding. Next, the vulnerability is described step by step. For each enumerated step, the corresponding source code line in the appendix is marked with the step number as a comment (e.g. //step1).

1. The size of the PSK extension is written to the variable *size*. Then the variable *size* is checked if it is greater than the total size of all extensions. So the variable *size* cannot be greater than the total size of all extensions.

2. The total size of all identities is written to the variable *len*. Then the variable *len* is checked if it is greater than the size of the PSK extension. So the variable *len* cannot be greater than the size of the PSK extension.
3. The size of one identity is written to the variable *identityLen*. Then the variable *identityLen* is checked if it is greater than the variable *len*. So the variable *identityLen* cannot be greater than the variable *len*.
4. The new identity (data: *identity*, size: *identityLen*) is written to the variable *ssl→extensions*. The value of the variable *identityLen* is not checked.
5. The data of the new identity (array: *current→identity*) is copied to the array *ssl→arrays→client_identity* with the size *current→identityLen*. But the size of the array *identity* can be greater than the size of the array *client_identity*. Hence, the first 128 bytes are written into the array *client_identity* and the rest is written into undefined memory. The size of the rest can be about 65 kilobyte, if the length fields of client hello packet: record length, client hello length, total extensions length, PSK extension length, total identities length and identity length contain their maximum value, which is 65535 (see steps 1-4).

How is the vulnerability exploitable by an attacker?

An attacker sends a special crafted hello client packet over the network to a TLSv1.3 wolfSSL server. The length fields of the packet: record length, client hello length, total extensions length, PSK extension length, total identity length and identity length contain their maximum value, which is 65535. The identity data field of the PSK extension of the packet contains the attack data, which the attacker wants to store in the undefined memory (RAM) of the server. The size of the data is about 65 kilobyte. Possibly the attacker can then 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 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 wolfSSL library (version 4.0.0)

```
internal.h:
OPAQUE16_LEN = 2;
OPAQUE32_LEN = 4;
MIN_PSK_ID_LEN = 6;
MAX_PSK_ID_LEN = 128;
NULL_TERM_LEN = 1;
char client_identity[MAX_PSK_ID_LEN + NULL_TERM_LEN];
WOLFSSL *ssl;         // contains all TLS data, e.g. the TLS extensions
```

```
tls.c:
int ret;
word16 len;
word16 idx = 0;       // the offset of the input variable
byte* identity;
word16 identityLen;
TLSX* extension;
PreSharedKey* psk = NULL;
byte* input;          // contains the received client hello packet
word16 length;
word16 offset = 0;    // the offset the input variable
```

```
TLSX_Pars(...) {  //the function is used to parse the TLS extensions
```

```
    ato16(input + offset, &size);  // size: size of the PSK extension //step1
    offset += OPAQUE16_LEN;
    if (offset + size > length)  // length: size of all extensions
        return BUFFER_ERROR;

    TLSX_PreSharedKey_Parse(…);
}

TLSX_PreSharedKey_Parse(...) { //the function is used to parse the PSK extension
    ato16(input + idx, &len);  // len: size of all identities //step2
    idx += OPAQUE16_LEN;
    if (len < MIN_PSK_ID_LEN || length - idx < len) // length: size of the PSK extension
        return BUFFER_E;

    ato16(input + idx, &identityLen); // identityLen: size of one identity //step3
    idx += OPAQUE16_LEN;
    if (len < OPAQUE16_LEN + identityLen + OPAQUE32_LEN)
    return BUFFER_E;
    identity = input + idx;
    idx += identityLen;

    TLSX_PreSharedKey_Use(…);
}

TLSX_PreSharedKey_Use(...) { //step4
    ret = TLSX_Push(&ssl->extensions, TLSX_PRE_SHARED_KEY, NULL, ssl->heap);
    extension = TLSX_Find(ssl->extensions, TLSX_PRE_SHARED_KEY);
    psk = (PreSharedKey*)extension->data;
    ret = TLSX_PreSharedKey_New((PreSharedKey**)&extension->data, identity, identityLen
    , ssl->heap, &psk);
}


tls13.c:
TLSX* ext;
PreSharedKey* current;


DoTls13ClientHello(...) { // the function is used to process the received client hello
    TLSX_Pars(…);
    DoPreSharedKeys(…);
}

DoPreSharedKeys(...) { // the function is used to process the PSK extension
    ext = TLSX_Find(ssl->extensions, TLSX_PRE_SHARED_KEY);
    current = (PreSharedKey*)ext->data;
    XMEMCPY(ssl->arrays->client_identity, current->identity, current->identityLen); //step5
}
```