OpenSSL Security Assessment

Technical report

Ref.	18-04-720-REP
Version	1.2
Date	2019
Prepared for	OSTIF
By	Quarkslab

Quarkslab SAS
13 rue Saint-Ambroise
75011 Paris
France

privateinternetaccess

Quarkslab

SECURING EVERY BIT OF YOUR DATA

Contents

1. Project Information

Version	Date	Status	Author
	August 31, 2018	Document Creation	Quarkslab
Version 1.0	November 28, 2018	First Version	
Version 1.1	January 11, 2019	Intermediate Version	
Version 1.2	January 16, 2019	Final Version	

Quarkslab	Position	Contact Information
Frédéric Raynal	CEO Quarkslab	fraynal@quarkslab.com

OSTIF	Position	Contact Information
Derek Zimmer	CEO OSTIF	derek@ostif.org

2. Executive Summary

This report describes the results of the security assessment of OpenSSL 1.1.1 made by Quarkslab, and funded by OSTIF. Two Quarkslab engineers worked on this audit for a total of 60 man days of study.

Scope of the audit :

- TLS 1.3
- PRNG
- SRP
- CAPI

2.1 Security/Bug concerns

Two client Denial of Service vulnerability and bug were found, allowing an attacker to crash the client:

2.1.1 Bug

- Ensure we send an alert on error when processing a ticket. Cf. https://github.com/openssl/openssl/pull/6852

2.1.2 Vulnerability

- Resolve some TLSv1.3 alert issues. Cf. https://github.com/openssl/openssl/pull/6887

All these bug and vulnerability were fixed before OpenSSL 1.1.1 was launched.

2.2 Security overview

2.2.1 TLS 1.3

- The implementation follows the RFC 8446.
- The quality of the source code is good.

2.2.2 PRNG

- The implementation follows the NIST standard SP800-90A (Rev. 1).
- The quality of the source code could be improved.

2.2.3 SRP

- The protocol seems to be correctly implemented.
- The quality of the source code could be improved.
- The inaccurate SRP comment (mentioned at the beginning of section 6.3, and then later in 6.3.4) was actually recently corrected (independently of this report) in commit 495a1e5c3 (master branch only).

2.2.4 CAPI

- The quality of the source code could be improved.

CAPI is the interface to the Microsoft Cryptographic API, it is used to get certificates and keys from Windows certificate store, encrypt/decrypt/sign with RSA/DSA...

CAPI is not directly exposed to the end user so there is no way to compromise OpenSSL with it.

Malicious users should have admin access to potentially add a corrupted certificate in order to compromise OpenSSL CAPI. We didn't find any solution to fuzz CAPI, so we manually reviewed the code. We didn't find any vulnerability during this review. The CAPI code (engines/e_capi.c) clearly lacks of comment.

2.2.5 Recommendations

We recommend to the OpenSSL Security Team to integrate the fuzzing methodology we have developed during this security assessment and to fuzz new features of TLS 1.3. This fuzzing methodology could be used to fuzz older TLS versions as well.

The quality of the source code should be improved with more pointers verification.

3. Introduction

3.1 Context and Scope

This report describes the security assessment made by Quarkslab on OpenSSL 1.1.1, an open source library used to secure network communications.

This audit has been carried out at the request of the Open Source Technology Improvement Fund. Its goal was to evaluate the security of OpenSSL 1.1.1.

Two engineers from Quarkslab worked on this audit, for a total of 60 man days of study.

The Internet Engineering Task Force has been in charge of defining the TLS protocol. The 21st of March, 2018, IETF finalized TLS 1.3.

This study focuses on the source code of OpenSSL 1.1.1 especially TLS 1.3, PRNG, SRP and CAPI.

The version of OpenSSL source code that we analyzed is available in OpenSSL's Github repository with the dc55e4f70f401c5869410d6a0c068c18c3fd53ec commit hash. At the moment of the analysis OpenSSL 1.1.1 supports the draft 26 of TLS 1.3.

3.1.1 Past Security Vulnerabilities

OpenSSL is a software library developed since 1998. A number of vulnerabilities in its source code have been published. Recent security advisories are referenced in the Security Announcements webpage of the project: https://www.openssl.org/news/vulnerabilities.html

We decided to consider only the vulnerabilities found after 2014, assuming that source code older than five years widely differs from the current one.

Memory Corruption :

CVE-2017-3733 : During a renegotiation handshake, if the Encrypt-Then-Mac extension is negotiated while it was not in the original handshake (or vice-versa) then this can cause OpenSSL to crash (dependent on cipher suites). Both clients and servers are affected.

CVE-2016-7054 : TLS connections using `*-CHACHA20-POLY1305` cipher suites are susceptible to a DoS attack by corrupting larger payloads. This can result in an OpenSSL crash. This issue is not considered to be exploitable beyond a DoS.

CVE-2016-6304 : A malicious client can send an excessively large OCSP Status Request extension. If that client continually requests renegotiation, sending a large OCSP Status Request extension each time, then there will be unbounded memory growth on the server. This will eventually lead to a Denial Of Service attack through memory exhaustion. Servers with a default configuration are vulnerable even if they do not support OCSP. Builds using the "no-ocsp" build time option are not affected. Servers using OpenSSL versions prior to 1.0.1g are not vulnerable in a default configuration, instead only if an application explicitly enables OCSP stapling support.

CVE-2015-0291 : ClientHello sigalgs DoS. If a client connects to an OpenSSL 1.0.2 server and renegotiates with an invalid signature algorithms extension a NULL pointer dereference will occur. This can be exploited in a DoS attack against the server.

CVE-2014-0160 : The TLS and DTLS implementations in OpenSSL 1.0.1 before 1.0.1g do not properly handle Heartbeat Extension packets, which allows remote attackers to obtain sensitive information from process memory via crafted packets that trigger a buffer overread, as

demonstrated by reading private keys, related to d1_both.c and t1_lib.c, aka the Heartbleed bug.

Cryptographic vulnerabilities :

CVE-2018-0732 : During key agreement in a TLS handshake using a DH(E) based cipher suites a malicious server can send a very large prime value to the client. This will cause the client to spend an unreasonably long period of time generating a key for this prime resulting in a hang until the client has finished. This could be exploited in a Denial Of Service attack.

CVE-2016-0703 : This issue only affected versions of OpenSSL prior to March 19th 2015 at which time the code was refactored to address vulnerability CVE-2015-0293. s2_srvr.c did not enforce that clear-key-length is 0 for non-export ciphers. If clear-key bytes are present for these ciphers, they *displace* encrypted-key bytes. This leads to an efficient divide-and-conquer key recovery attack: if an eavesdropper has intercepted an SSLv2 handshake, they can use the server as an oracle to determine the SSLv2 master-key, using only 16 connections to the server and negligible computation. More importantly, this leads to a more efficient version of DROWN that is effective against non-export cipher suites, and requires no significant computation.

CVE-2015-0285 : Under certain conditions an OpenSSL 1.0.2 client can complete a handshake with an unseeded PRNG. If the handshake succeeds then the client random that has been used will have been generated from a PRNG with insufficient entropy and therefore the output may be predictable.

CVE-2014-0224 : An attacker can force the use of weak keying material in OpenSSL SSL/TLS clients and servers. This can be exploited by a Man-in-the-middle (MITM) attack where the attacker can decrypt and modify traffic from the attacked client and server.

3.1.2 OpenSSL Source code architecture

OpenSSL is composed of two separate C libraries: libcrypto and libssl. Libcrypto is in charge of general-purpose cryptography. This library can be used alone, without using libssl. libssl is a SSL/TLS library using libcrypto methods. Code specific to TLS 1.3 is located under the `ssl`, `ssl/statem` and `ssl/record` directories.

crypto :

This directory contains the code related to the libcrypto library.

crypto/rand

The PRNG is implemented in this directory.

crypto/srp

This section of libcrypto contains the main part of the code in charge of SRP.

ssl :

In this directory we can find code managing the main part of libssl, communications with libcrypto, structures definition and SSL/TLS context initialization.

ssl/statem :

This directory contains the source code responsible for extensions parsing, messages parsing and client/server state machine for SSL/TLS sessions.

ssl/record :

Source code in charge of TLS record compression, decompression, encoding, decoding, read and write.

engines :

Source code in the engines directory provides interfaces to hardware implementation of specific cryptographic operations like CAPI.

4. TLS 1.3 implementation

4.1 Methodology

We chose to focus our analysis with fuzzing methods in order to detect memory errors and invalid behaviors. During the fuzzing phase, we reviewed the source code manually.

As OpenSSL is already fuzzed by OpenSSL itself and OSS-Fuzz, we decided to use a different methodology of fuzzing to discover bugs that were not revealed by already existing fuzzers. Two kinds of fuzzer were developed in order to cover the maximum of possibilities of OpenSSL.

For the fuzzing phase, the library was compiled with TLS 1.3, ASAN and debug mode activated. ASAN was used in order to detect memory corruption that doesn't lead to a crash in normal compilation.

```
$ ./config no-shared enable-tls1_3 enable-asan -debug
```

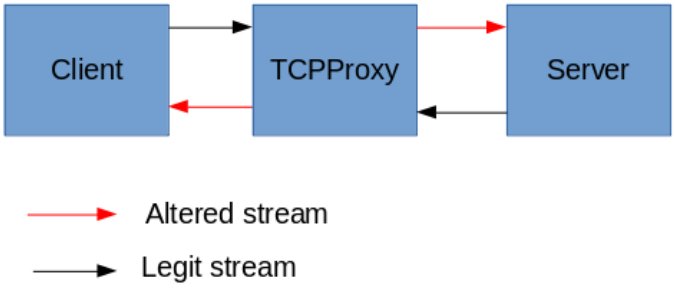
4.1.1 TCP Proxy

The first fuzzer we developed was a TCP proxy. This proxy was placed in a Man In The Middle position to alter TCP streams.

Alterations done by the fuzzer:

- Packet mutation (Bitflip)
- Packet injection (Alert, CSS...)
- Packet drop
- Packet duplication

A TLS 1.3 client and a ping server were developed. The fuzzer altered both the client and server streams to find vulnerabilities in both sides.



After some tests, we realized that this fuzzer had a low probability to find revelant bugs, due to a low speed of fuzzing and poor fuzzing strategies. We decided to stop this approach and move to code instrumentation.

4.1.2 Code instrumentation

In order to fuzz TLS 1.3 in a different manner than OpenSSL and OSS-Fuzz, we decided to modify the source code of OpenSSL to be able to save and restore a TLS session to and from a file.

To make it possible, hooks were placed in the OpenSSL source code. These hooks save and restore TLS Records and internal packets. Hooks were placed in functions `ssl3_read_n` of `ssl/record/rec_layer_s3.c`, `tls_get_message_header` of `ssl/statem/statem_lib.c` and `read_state_machine` of `ssl/statem/statem.c`.

To create an initial corpus of sessions from tls13ccstest and sslapitest, two tests from the OpenSSL test set were serialized using the installed hooks in the source code. These tests were chosen because they use the latest features of TLS 1.3 (Extensions, Early Data, HRR,...) and this is useful to have a good coverage of TLS 1.3.

Due to key generation and timestamp in `NewSessionTicket`, sessions generated by these programs were dynamic and therefore not replayable. To solve this problem, RNG and time were fixed, to have predictable TLS sessions that can be injected without error.

RNG was fixed using this code:

```
static void stdlib_rand_cleanup() {}
static void stdlib_rand_add(const void *buf, int num, double add_entropy) {}
static int stdlib_rand_status() { return 1; }

static void stdlib_rand_seed(const void *buf, int num)
{
        srand( *((unsigned int *) buf) );
}

static int stdlib_rand_bytes(unsigned char *buf, int num)
{
        for( int index = 0; index < num; ++index ){
                buf[index] = rand() % 256;
        }
        return 1;
}

RAND_METHOD stdlib_rand_meth = {
        stdlib_rand_seed,
        stdlib_rand_bytes,
        stdlib_rand_cleanup,
        stdlib_rand_add,
        stdlib_rand_bytes,
        stdlib_rand_status
};

RAND_METHOD *RAND_stdlib() { return &stdlib_rand_meth; }
```

And initialized with this code in the `main` function of each test:

```
RAND_set_rand_method(RAND_stdlib());
unsigned int seed = 0x00beef00;
RAND_seed(&seed, sizeof(seed));
```

Time was fixed by overloading the `time` function :

```
time_t time(time_t *t){
    time_t p_time_t;
    memset(&p_time_t, 0, sizeof(time_t));
    return p_time_t;
}
```

Two fuzzing engines were used during this analysis, HonggFuzz and AFL. HonggFuzz was used in association of ASAN because AFL gives false positives with ASAN.

Fuzzers ran over a week on a i7-8550U with 6 cores dedicated to the fuzzing process. We recommend running these fuzzers on more powerful stations and in a long-term process.

4.1.3 Code review

During the fuzzing phase, we reviewed the code of the state machine, extensions parsing and messages parsing. We didn't find anything relevant.

4.2 Internal Mechanisms of TLS 1.3

TLS stands for Transport Layer Security and is the successor of SSL. TLS brings secure communication between clients and servers. It is used by many protocols such as HTTP, SMTP, POP3, FTP, OpenVPN...

4.2.1 Handshake

1. The TLS 1.3 handshake starts with the `ClientHello` message. This message is constructed from cryptographic information such as supported protocols and supported CipherSuites. In this message there is also the client's key share.
2. The server replies with a `ServerHello` message that consists of the key agreement protocol it has chosen, the server's key share, its certificate and the `ServerFinished` message.
3. The client checks the server certificate, generates keys since it has the key share of the server, and sends the `ClientFinished` message. From here on, the encryption of the data begins.

4.2.2 0-RTT

0-RTT Resumption permits a zero-round trip handshake if the client has connected to the server before. This is done by saving secret information (`SessionID`) of a past session.

4.2.3 Hello Retry Request

HRR is a message sent by the server in case of a configuration mismatch between `ServerHello` and `ClientHello`. When a client receives an HRR, the client must resend a `ClientHello` to renegotiate the TLS connection.

4.2.4 Security

TLS 1.3 removes obsolete and insecure ciphers/hash functions from TLS 1.2 :

- SHA-1
- RC4
- DES
- 3DES
- AES-CBC
- MD5

All handshake messages after the `ServerHello` are now encrypted.

4.3 Security problem and bug identified in TLS 1.3 of OpenSSL

In this section, we are presenting two bugs we have found in our fuzzing campaign. These two bugs were found with AFL and our patch available in the appendix.

4.3.1 Client Denial of Service Bug (Debug mode)

If the server sends a `NewSessionTicket` extension that is larger than the size given in the TLS Record, this causes the `tls_process_new_sessions_ticket` function to return a `MSG_PROCESS_ERROR` state.

```
MSG_PROCESS_RETURN tls_process_new_session_ticket(SSL *s, PACKET *pkt)
...
    if (SSL_IS_TLS13(s)) {
        PACKET extpkt;
        if (!PACKET_as_length_prefixed_2(pkt, &extpkt)
                || PACKET_remaining(pkt) != 0
                || !tls_collect_extensions(s, &extpkt,
                                           SSL_EXT_TLS1_3_NEW_SESSION_TICKET,
                                           &exts, NULL, 1)
                || !tls_parse_all_extensions(s,
                                             SSL_EXT_TLS1_3_NEW_SESSION_TICKET,
                                             exts, NULL, 0, 1)) {

            /* SSLfatal() already called */
            goto err;
        }
    }


static SUB_STATE_RETURN read_state_machine(SSL *s)
...
            ret = process_message(s, &pkt);

            /* Discard the packet data */
            s->init_num = 0;

            switch (ret) {
            case MSG_PROCESS_ERROR:
                check_fatal(s, SSL_F_READ_STATE_MACHINE);
                return SUB_STATE_ERROR;
```

Debug :

```
ssl/statem/statem.c:664: OpenSSL internal error: Assertion failed: (s)->statem.in_
↪init && (s)->statem.state == MSG_FLOW_ERROR
```

Debug Stack Trace :

```
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:51
#1  0x00007ffff7632231 in __GI_abort () at abort.c:79
#2  0x00000000005350dc in OPENSSL_die (message=<optimized out>, file=<optimized out>,␣
↪line=<optimized out>) at crypto/cryptlib.c:355
#3  0x0000000000480da5 in ossl_assert_int (expr=<optimized out>, exprstr=<optimized␣
↪out>, file=<optimized out>, line=<optimized out>) at include/internal/cryptlib.h:39
```

(continued from previous page)

```
#4  read_state_machine (s=<optimized out>) at ssl/statem/statem.c:664
#5  state_machine (s=0xb21210, server=<optimized out>) at ssl/statem/statem.c:428
#6  0x00000000004141a6 in ssl3_read_bytes (s=0xb21210, type=<optimized out>, recvd_
↪type=0x0, buf=<optimized out>, len=<optimized out>, peek=0, readbytes=<optimized␣
↪out>) at ssl/record/rec_layer_s3.c:1631
#7  0x000000000042736b in ssl3_read_internal (s=0xb21210, buf=0x7fffffffe097, len=1,␣
↪peek=0, readbytes=0x7fffffffe078) at ssl/s3_lib.c:4414
#8  0x0000000000443558 in ssl_read_internal (s=0xb21210, buf=0x7fffffffe097, num=1,␣
↪readbytes=0x7fffffffe078) at ssl/ssl_lib.c:1740
#9  0x0000000000443ab5 in SSL_read_ex (s=0x2, buf=0x7fffffffdb70, num=0,␣
↪readbytes=0x7ffff7630e7b <__GI_raise+267>) at ssl/ssl_lib.c:1768
#10 0x0000000000406b35 in create_ssl_connection (serverssl=0xb1fa30,␣
↪clientssl=0xb21210, want=0) at test/ssltestlib.c:672
#11 0x00000000004071e3 in test_tls13ccs (tst=11) at test/tls13ccstest.c:308
#12 setup_tests () at test/tls13ccstest.c:542
#13 0x00000000004dfd53 in main (argc=<optimized out>, argv=0x7fffffffe2a8) at test/
↪testutil/main.c:50
```

4.3.2 Client Denial of Service Issue

If the server sends an invalid length for `SessionId` extension in the `ServerHello`, and the client and the server are in a HRR waiting state, or if the server sends an invalid cipher suite in the `ServerHello`, and the client and the server are in a HRR waiting state, then the client tries to send an encrypted alert. Because the secure channel is not established, this causes the generation of a new alert leading to a stack overflow.

Example 1 :

```
MSG_PROCESS_RETURN tls_process_server_hello(SSL *s, PACKET *pkt)
...
    /* Get the session-id. */
    if (!PACKET_get_length_prefixed_1(pkt, &session_id)) {
        SSLfatal(s, SSL_AD_DECODE_ERROR, SSL_F_TLS_PROCESS_SERVER_HELLO,
                 SSL_R_LENGTH_MISMATCH);
        goto err;
    }
```

Example 2 :

```
MSG_PROCESS_RETURN tls_process_server_hello(SSL *s, PACKET *pkt)
...
    if (!PACKET_get_bytes(pkt, &cipherchars, TLS_CIPHER_LEN)) {
        SSLfatal(s, SSL_AD_DECODE_ERROR, SSL_F_TLS_PROCESS_SERVER_HELLO,
                 SSL_R_LENGTH_MISMATCH);
        goto err;
    }
...
    if (!set_client_ciphersuite(s, cipherchars)) {
        /* SSLfatal() already called */
        goto err;
    }


static int set_client_ciphersuite(SSL *s, const unsigned char *cipherchars)
...
```

(continues on next page)

(continued from previous page)

```
    c = ssl_get_cipher_by_char(s, cipherchars, 0);
    if (c == NULL) {
        /* unknown cipher */
        SSLfatal(s, SSL_AD_ILLEGAL_PARAMETER, SSL_F_SET_CLIENT_CIPHERSUITE,
                 SSL_R_UNKNOWN_CIPHER_RETURNED);
        return 0;
    }
```

Debug :

```
tls13ccstest: ssl/statem/statem.c:122: void ossl_statem_fatal(SSL *, int, int, int,␣
↪const char *, int): Assertion `s->statem.state != MSG_FLOW_ERROR' failed.

ssl/record/ssl3_record_tls13.c:81: OpenSSL internal error: Assertion failed: s->s3->
↪tmp.new_cipher != NULL
```

Debug Stack Trace :

```
#0  __GI_raise (sig=sig@entry=6) at ../sysdeps/unix/sysv/linux/raise.c:51
#1  0x00007ffff7632231 in __GI_abort () at abort.c:79
#2  0x00007ffff76299da in __assert_fail_base (fmt=0x7ffff777cd48 "%s%s%s:%u: %s
↪%sAssertion `%s' failed.\n%n", assertion=assertion@entry=0x81cc3c "s->statem.state !
↪= MSG_FLOW_ERROR",
    file=file@entry=0x81cc5e "ssl/statem/statem.c", line=line@entry=122,␣
↪function=function@entry=0x81cc72 "void ossl_statem_fatal(SSL *, int, int, int,␣
↪const char *, int)") at assert.c:92
#3  0x00007ffff7629a52 in __GI___assert_fail (assertion=0x81cc3c "s->statem.state !=␣
↪MSG_FLOW_ERROR", file=0x81cc5e "ssl/statem/statem.c", line=122,
    function=0x81cc72 "void ossl_statem_fatal(SSL *, int, int, int, const char *, int)
↪") at assert.c:101
#4  0x000000000047de78 in ossl_statem_fatal (s=0xb33700, al=80, func=401, reason=68,␣
↪file=0x8184d0 "ssl/record/ssl3_record.c", line=1056) at ssl/statem/statem.c:122
#5  0x000000000041b7fd in tls1_enc (s=<optimized out>, recs=<optimized out>, n_recs=
↪<optimized out>, sending=<optimized out>) at ssl/record/ssl3_record.c:1055
#6  0x0000000000412485 in do_ssl3_write (s=0xb33700, type=21, buf=0xb1dd70 "\002/",␣
↪pipelens=0x7fffffffdda8, numpipes=<optimized out>, create_empty_fragment=0, written=
↪<optimized out>)
    at ssl/record/rec_layer_s3.c:1018
#7  0x00000000004290f1 in ssl3_dispatch_alert (s=0xb33700) at ssl/s3_msg.c:78
#8  0x0000000000429033 in ssl3_send_alert (s=0xb33700, level=2, desc=47) at ssl/s3_
↪msg.c:60
#9  0x000000000047ddf0 in ossl_statem_fatal (s=0xb33700, al=47, func=369, reason=999,␣
↪file=0x81d4b8 "ssl/statem/statem_clnt.c", line=1505) at ssl/statem/statem.c:127
#10 0x00000000004894f3 in tls_process_server_hello (s=<optimized out>, pkt=<optimized␣
↪out>) at ssl/statem/statem_clnt.c:1444
#11 0x00000000004888af in ossl_statem_client_process_message (s=0xb33700,␣
↪pkt=0x7fffffffdf60) at ssl/statem/statem_clnt.c:1016
#12 0x0000000000480903 in read_state_machine (s=<optimized out>) at ssl/statem/statem.
↪c:657
#13 state_machine (s=0xb33700, server=<optimized out>) at ssl/statem/statem.c:428
#14 0x0000000000442ea7 in SSL_do_handshake (s=0xb33700) at ssl/ssl_lib.c:3619
#15 0x00000000004067c6 in create_ssl_connection (serverssl=0xb40f30,␣
↪clientssl=0xb33700, want=0) at test/ssltestlib.c:634
#16 0x0000000000407555 in test_tls13ccs (tst=11) at test/tls13ccstest.c:364
#17 setup_tests () at test/tls13ccstest.c:542
```

(continues on next page)

(continued from previous page)

```
#18 0x00000000004dfd33 in main (argc=<optimized out>, argv=0x7fffffffe2a8) at test/
↪testutil/main.c:50
```

Stack Trace :

```
#0  0x00000000005aabc4 in EVP_MD_CTX_md (ctx=0x0) at crypto/evp/evp_lib.c:440
#1  0x000000000040fcfd in do_ssl3_write (s=0xb2f700, type=21, buf=0xb19d70 "\002P",␣
↪pipelens=0x7fffff800148, numpipes=<optimized out>, create_empty_fragment=0, written=
↪<optimized out>)
    at ssl/record/rec_layer_s3.c:716
#2  0x0000000000428091 in ssl3_dispatch_alert (s=0xb2f700) at ssl/s3_msg.c:78
#3  0x0000000000427fd3 in ssl3_send_alert (s=0xb2f700, level=2, desc=80) at ssl/s3_
↪msg.c:60
#4  0x000000000047c6ce in ossl_statem_fatal (s=0xb2f700, al=80, func=<optimized out>,␣
↪reason=<optimized out>, file=<optimized out>, line=<optimized out>) at ssl/statem/
↪statem.c:127
#5  0x000000000041a8ff in tls1_enc (s=<optimized out>, recs=0x7fffff800670, n_recs=
↪<optimized out>, sending=<optimized out>) at ssl/record/ssl3_record.c:1055
#6  0x0000000000411565 in do_ssl3_write (s=0xb2f700, type=21, buf=0xb19d70 "\002P",␣
↪pipelens=0x7fffff801798, numpipes=<optimized out>, create_empty_fragment=0, written=
↪<optimized out>)
...
#7332 0x0000000000428091 in ssl3_dispatch_alert (s=0xb2f700) at ssl/s3_msg.c:78
#7333 0x0000000000427fd3 in ssl3_send_alert (s=0xb2f700, level=2, desc=80) at ssl/s3_
↪msg.c:60
#7334 0x000000000047c6ce in ossl_statem_fatal (s=0xb2f700, al=80, func=<optimized out>
↪, reason=<optimized out>, file=<optimized out>, line=<optimized out>) at ssl/statem/
↪statem.c:127
#7335 0x000000000041a8ff in tls1_enc (s=<optimized out>, recs=0x7fffffffcc90, n_recs=
↪<optimized out>, sending=<optimized out>) at ssl/record/ssl3_record.c:1055
#7336 0x0000000000411565 in do_ssl3_write (s=0xb2f700, type=21, buf=0xb19d70 "\002P",␣
↪pipelens=0x7fffffffddb8, numpipes=<optimized out>, create_empty_fragment=0, written=
↪<optimized out>)
    at ssl/record/rec_layer_s3.c:1018
#7337 0x0000000000428091 in ssl3_dispatch_alert (s=0xb2f700) at ssl/s3_msg.c:78
#7338 0x0000000000427fd3 in ssl3_send_alert (s=0xb2f700, level=2, desc=50) at ssl/s3_
↪msg.c:60
#7339 0x000000000047c6ce in ossl_statem_fatal (s=0xb2f700, al=50, func=<optimized out>
↪, reason=<optimized out>, file=<optimized out>, line=<optimized out>) at ssl/statem/
↪statem.c:127
#7340 0x0000000000487ab3 in tls_process_server_hello (s=<optimized out>, pkt=
↪<optimized out>) at ssl/statem/statem_clnt.c:1444
#7341 0x0000000000486e6f in ossl_statem_client_process_message (s=0xb2f700,␣
↪pkt=0x7fffffffdf70) at ssl/statem/statem_clnt.c:1016
#7342 0x000000000047f195 in read_state_machine (s=<optimized out>) at ssl/statem/
↪statem.c:657
#7343 state_machine (s=0xb2f700, server=<optimized out>) at ssl/statem/statem.c:428
#7344 0x0000000000441b37 in SSL_do_handshake (s=0xb2f700) at ssl/ssl_lib.c:3619

#7345 0x00000000004057c6 in create_ssl_connection (serverssl=0xb3cf30,␣
↪clientssl=0xb2f700, want=0) at test/ssltestlib.c:634
#7346 0x0000000000406555 in test_tls13ccs (tst=11) at test/tls13ccstest.c:364
#7347 setup_tests () at test/tls13ccstest.c:542
#7348 0x00000000004de0d3 in main (argc=<optimized out>, argv=0x7fffffffe2b8) at test/
↪testutil/main.c:50
```

4.4 Conclusion

TLS 1.3 seems to be correctly implemented in OpenSSL and the related code has a quite good quality. The manual code review didn't reveal any vulnerability in TLS 1.3. On the other hand, our approach to fuzzing TLS 1.3 gave us some interesting results with two client denial of service issues, but we didn't find any bug on the server side. These bugs were disclosed to OpenSSL Security Team and patched before the release of OpenSSL 1.1.1.

5. PRNG implementation

The core internals of the random generator in OpenSSL have been reimplemented for the release of version 1.1.1. The implementation follows the NIST standard SP800-90A (Rev. 1) *[SP800-90A]*. In this section, we provide an expert opinion on this implementation and how it respects the standard.

In OpenSSL, the following parameters are chosen:

- the DRBG is used in CTR mode
- the chosen block cipher is AES, with AES-256 as default (AES-128 and AES-192 are also supported)
- by default, the generator uses a derivation function, but the code is constructed so that it can be used without a derivation function by using the flag `RAND_DRBG_FLAG_CTR_NO_DF`.

5.1 API

Using the DRBG is really easy and users can't make it wrong. Only the `RAND_bytes` or `RAND_priv_bytes`[1] functions have to be called.

These functions take two parameters, the buffer which will receive the cryptographically strong pseudo-random bytes and the requested size.

The initialization process of the DRBG is automatically done thanks to the call to `RUN_ONCE(&rand_init, do_rand_init)` in the `rand_lib.c:RAND_get_rand_method` function.

NB: Here is an abstract of the documentation : `RAND_bytes` and `RAND_priv_bytes` are equivalent, but `RAND_priv_bytes` uses a unique instance of the generator and so should be used for sensitive materials (keys for example). For values that should not remain secret, you can use `RAND_bytes` instead. This method does not provide 'better' randomness, it uses the same type of CSPRNG. The intention behind using a dedicated CSPRNG exclusively for private values is that none of its output should be visible to an attacker (e.g., used as salt value), in order to reveal as little information as possible about its internal state, and that a compromise of the "public" CSPRNG instance will not affect the secrecy of these private values.

5.2 Test Vectors

Test vectors for the standard SP800-90A are provided by NIST in the context of the *Cryptographic Algorithm Validation Program* (CAVP)[2]. These test vectors are included in the standard tests of OpenSSL, in the file `openssl/test/drbg_cavs_data.c`. The user can reproduce these tests using the binary `test/drbg_cavs_test` generated after compilation.

The test vectors for the DRBG in CTR mode using AES-128, AES-192 and AES-256, both using a derivation function or not, are all validated by the tests of OpenSSL (the vectors for 3KeyTDEA are not included).

It is interesting to note that other test vectors are included in the file `openssl/test/drbgtest.c`, but their origin could not be recovered.

[1] https://www.openssl.org/docs/manmaster/man3/RAND_bytes.html
[2] https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program/random-number-generators

5.3 Implementation Details

5.3.1 Description of the Internals

The internals of the DRBG mechanism are located in the file `crypto/rand/drbg_ctr.c`, where all the functions defined in the section 10.2 of [SP800-90A] can be found. We present in the following table the correspondence between the C functions and the functions defined in the standard.

Standard	Implementation
Block_Cipher_df (or **df**)	`ctr_df()`
CTR_DRBG_Update	`ctr_update()`
CTR_DRBG_Instantiate_algorithm	`drbg_ctr_instantiate()`
CTR_DRBG_Reseed_algorithm	`drbg_ctr_reseed()`
CTR_DRBG_Generate_algorithm	`drbg_ctr_generate()`
BCC	`ctr_BCC_update()`
Block_Encrypt	`EVP_Cipher_Update()`

The different parameters of the generator are stored in a `RAND_DRBG` object, which is a generalization for all types of DRBG. This object contains a `RAND_DRBG_CTR` object, which is defined as the following structure:

Listing 5.1: `openssl/crypto/rand/rand_lcl.h`

```
/*
 * The state of a DRBG AES-CTR.
 */
typedef struct rand_drbg_ctr_st {
    EVP_CIPHER_CTX *ctx;
    EVP_CIPHER_CTX *ctx_df;
    const EVP_CIPHER *cipher;
    size_t keylen;
    unsigned char K[32];
    unsigned char V[16];
    /* Temporary block storage used by ctr_df */
    unsigned char bltmp[16];
    size_t bltmp_pos;
    unsigned char KX[48];
} RAND_DRBG_CTR;
```

This object contains two `EVP_CIPHER_CTX`, which are used to encrypt a block with AES. Two encryption contexts are needed because two different keys are used, depending if the encryption is done in the derivation function (`ctr_df()`) or not. The `EVP_CIPHER` object holds the type of block cipher used and is affected in the `drbg_ctr_init()` function. The `keylen` parameter is the same as described in the standard, and is 256 bits by default. The buffers `K` and `V` correspond to the buffers *Key* and *V* in the standard. This structure is almost the equivalent of the *working state* described in [SP800-90A], but some elements are missing like the *reseed_counter*. These elements can be found in the more abstract structure `rand_drbg_st`.

In the following section, we describe in detail how the `ctr_update()` function correctly implements the update function of the standard.

5.3.2 Update Process

The parameters used in the function and their values are reminded here:

- *seedlen*: 384 bits
- *keylen*: 256 bits
- *blocklen*: 128 bits
- *ctr_len*: 128 bits (this is the case in the pseudo-code example of the standard, and can be inferred from the structure of `ctr_update()`)

When *seedlen* is 384 bits, three rounds of the "while" loop in the **CTR_DRBG_Update Process** (section 10.2.1.2 of *[SP800-90A]*) will be executed, as the output of **Block_Encrypt** is 128 bits, and 128*3 = 384. As *ctr_len* is equal to *blocklen*, each iteration will consist of:

- V = V+1
- Encryption of V with Key

At the end of the "while" loop, the variable *temp* will contain the concatenation of three successive encryptions of incremented versions of V. So if V_0 is the initial value of V before the loop, at the end of the loop we get:

$$temp = Encrypt(Key, V_0 + 1) \;||\; Encrypt(Key, V_0 + 2) \;||\; Encrypt(Key, V_0 + 3)$$

As 384 is exactly 128*3, *temp* does not need to be truncated with the instruction 3 of the process. Then, *temp* is XORed with the provided data, and *Key* and V are affected with part of *temp*, giving the final values for *Key* and V (we consider that the operator $[n : m]$ represents an access between bits n and m, *m-th* bit excluded):

$$Key = \Big(Encrypt(Key, V_0 + 1) \;||\; Encrypt(Key, V_0 + 2)\Big) \oplus provided\_data[0 : 256]$$
$$V = Encrypt(Key, V_0 + 3) \oplus provided\_data[256 : 384]$$

We reproduce here part of the code of the function ctr_update(), with custom comments to help understand the implementation.

```
__owur static int ctr_update(RAND_DRBG *drbg,
                             const unsigned char *in1, size_t in1len,
                             const unsigned char *in2, size_t in2len,
                             const unsigned char *nonce, size_t noncelen)
{
    RAND_DRBG_CTR *ctr = &drbg->data.ctr;
    int outlen = AES_BLOCK_SIZE;

    /* First iteration */
    inc_128(ctr);
    if (!EVP_CipherUpdate(ctr->ctx, ctr->K, &outlen, ctr->V, AES_BLOCK_SIZE)
        || outlen != AES_BLOCK_SIZE)
        return 0;

    /* Second iteration */
    if (ctr->keylen != 16) {
        inc_128(ctr);
```

(continues on next page)

(continued from previous page)

```
        if (!EVP_CipherUpdate(ctr->ctx, ctr->K+16, &outlen, ctr->V,
                              AES_BLOCK_SIZE)
            || outlen != AES_BLOCK_SIZE)
            return 0;
    }
        /* Third iteration */
    inc_128(ctr);
    if (!EVP_CipherUpdate(ctr->ctx, ctr->V, &outlen, ctr->V, AES_BLOCK_SIZE)
        || outlen != AES_BLOCK_SIZE)
        return 0;

    [...]

    /* In our case, we use the derivation function */
    if ((drbg->flags & RAND_DRBG_FLAG_CTR_NO_DF) == 0) {
        /* When coming from instantiate, generate or reseed, the condition is true */
        if (in1 != NULL || nonce != NULL || in2 != NULL)
            if (!ctr_df(ctr, in1, in1len, nonce, noncelen, in2, in2len))
                return 0;
        if (in1len)
            ctr_XOR(ctr, ctr->KX, drbg->seedlen);
    } [...]

    if (!EVP_CipherInit_ex(ctr->ctx, ctr->cipher, NULL, ctr->K, NULL, 1))
        return 0;
    return 1;
}
```

One can note that the `ctr_update()` function does not exhibit a "while" loop like in the standard. This is because the loop has been unrolled, with "if" conditions guarding the operations depending of the length of *seedlen*. In our case, as *seedlen* is 384 bits, we go through the three iterations. Each iteration contains two instructions: a call to `inc_128(ctr)`, which increments the buffer `V` by one, and a block encryption with `EVP_CipherUpdate`. This seems to fit the process of the standard. An interesting point is the absence of a *temp* variable: indeed, the result of the encryption is directly affected in K or V (we remind that the second parameter of `EVP_CipherUpdate` is the output buffer for the encryption), which leaves us with these values after the three iterations:

$$K = Encrypt(Key, V_0 + 1) \;||\; Encrypt(Key, V_0 + 2)$$
$$V = Encrypt(Key, V_0 + 3)$$

Then one just needs to check that the XOR with the provided data is performed. Another particularity of the implementation is the presence of the derivation function in the code of the update function. Indeed, in the standard, the **df** function is independent from the update function, and usually called just before. The return of **df** is then passed as an argument in the update function. This behavior can be witnessed in the instantiate, reseed and generate functions of the standard.

The implementation has taken another approach by executing the derivation function inside the update function just before computing the XOR (as the derivation function is applied on the input of the update function, which are the *provided_data*). A quick look at `ctr_df()` shows that the modified buffer is `ctr->KX`, it will thus contain the output of the derivation. After the derivation function, the XOR is performed with `ctr_XOR()` between `ctr->KX` (the *provided_data* after treatment by the derivation function) and the concatenation of K and V. Then the values of K and V at the end of the function are as expected by the standard.

The end of the function calls `EVP_CipherInit_ex` to update the cipher context with the new value of K.

5.3.3 Coding Style

Some improvements should be considered. Reading the files was quite difficult due to the lack of comments and the intensive use of macros.

Let's take an example and try to understand the following call: `RUN_ONCE(&rand_init, do_rand_init)`

The macro `RUN_ONCE` is defined by:

Listing 5.2: `openssl/include/internal/thread_once.h`

```
# define RUN_ONCE(once, init)                                                \
(CRYPTO_THREAD_run_once(once, init##_ossl_, #init) ? init##_ossl_ret_ : 0)
```

The second parameter `init` will be expanded thanks to the processor `init##_ossl_`. In our case, `RUN_ONCE(&rand_init, do_rand_init)` will call the function `do_rand_init_ossl_`.

This function is not declared directly but is defined through the call to the macro `DEFINE_RUN_ONCE_STATIC` (thread_once.h) with the parameter `do_rand_init`, which adds some glue.

Another encountered difficulty came from the use of `ifdef`, `ifndef` directives without comments at the ending part. So it was quite difficult to see which parts of code will be compiled or not. Here is an example with the end of the function `rand_pool_acquire_entropy`:

```
                [ ...]
                if (num == (int)bytes_needed)
                    bytes = bytes_needed;

                rand_pool_add_end(pool, bytes, 8 * bytes);
                entropy_available = rand_pool_entropy_available(pool);
            }
            if (entropy_available > 0)
                return entropy_available;
        }
    }
#    endif

    return rand_pool_entropy_available(pool);
#   endif
}
#  endif
# endif
```

It could be useful for the future developers to add a comment after each `endif` directive to indicate from which `if`, `ifdef`, `ifndef` directive it refers to.

Also, there is a lack of defensive programming style. Even if a function is declared static (i.e. internal to a file), its parameters should be checked. That is true especially with pointers. The functions contained in the file `drbg_ctr.c` which implements the internals of the DRBG never check their parameters. This practice is really dangerous and can lead to vulnerabilities. Even if we focused our review on the new DRBG, this practice is global to the project. One of the worst examples can be found in the TLS stack:

```
EXT_RETURN tls_construct_stoc_ec_pt_formats(SSL *s, WPACKET *pkt,
                                            unsigned int context, X509 *x,
                                            size_t chainidx)
{
    unsigned long alg_k = s->s3->tmp.new_cipher->algorithm_mkey;

[ ... ]
```

We have also noticed that some parts of code should be mutualized. An example is the function `static void inc_128(RAND_DRBG_CTR *ctr)` located in drbg_ctr.c, which is quite the same as the `static void ctr128_inc(unsigned char *counter)` function located in ctr128.c. For example, it could be possible to improve the code (in terms of speed) by using dedicated instructions but it will require to modify the code at (at least) two places.

5.4 Entropy Sources

Entropy from the system is used in order to seed the generator. This source depends on different characteristics (architecture, operating system...) of the hardware on which OpenSSL is installed. We studied the two most common sources, which are Linux (Debian 9 testing) and Windows on x86-64 architecture. The function providing this entropy is `rand_pool_acquire_entropy()` and can be found in `rand_unix.c` or `rand_win.c`.

5.4.1 Windows

On Windows, the `rand_pool_acquire_entropy()` gathers entropy from `BCryptGenRandom` on Windows 7 or higher and from legacy CryptoAPI `CryptGenRandom` for earlier versions of Windows. These two functions can be considered secure. We can note that, on Windows, the default random number provider implements an algorithm for generating random numbers that comply with the NIST SP800-90 standard, specifically the CTR_DRBG portion of the standard[3]. It means that two DRBG which implements the NIST SP800-90 standard are chained.

5.4.2 Linux

On Linux, the function `rand_pool_acquire_entropy()` provides different methods to gather entropy. The method can be chosen with the option `--with-rand-seed` during compilation. By default, the option "OS" is chosen. On Linux, this uses the system call `getrandom()`[4] that draws entropy from `urandom` by default. We consider that on modern systems, this provides enough entropy.

[3] https://docs.microsoft.com/en-us/windows/desktop/api/bcrypt/nf-bcrypt-bcryptgenrandom
[4] http://man7.org/linux/man-pages/man2/getrandom.2.html

6. SRP implementation

6.1 Description

The Secure Remote Password (SRP) protocol is an augmented password-authenticated key agreement. During this protocol, a client (or user) demonstrates to a server that it knows a password, without sending the password itself. The password never leaves the client and is unknown to the server. In the end, the SRP protocol creates a private key shared between the client and server, in a manner similar to Diffie-Hellman.

We recall briefly the different parameters of the protocol here. Further details can be found in *[RFC2945]*. This document describes the version 3 of the protocol, which is actually in version 6a. The differences are apparently small and focused on the value of k, as it can be seen in the Wikipedia article about SRP[1]. Unfortunately, the official documents (IEEE 1363.2-2008 and ISO/IEC 11770-4) describing the standard for version 6 are not publicly available. The version chosen by OpenSSL is 6a.

The different parameters of the SRP protocols are:

- q and $N = 2q + 1$ are both primes, and all arithmetic is performed modulo N.
- g is a generator of the multiplicative group $\mathbb{Z}_N^*$.
- $H()$ is a hash function.
- k is a parameter of both sides. In version 6a, $k = H(N, g)$.
- s is a small salt.
- I is an identifying username.
- p is the password of the user.
- v is the *password verifier*, $v = g^x$ with $x = H(s, p)$ at minimum.
- A and B are random ephemeral keys (computed from random numbers a and b with $A = g^a$ and $B = g^b$)

First, the client choses a small random salt s and computes x and v. The server stores s, v as well as g and N, all indexed by the username I. The password verifier v will allow the server to check that the client has the password corresponding to the username. We will not detail here the computations done in order to get a shared key and the authentication (which is based on a proof that the keys match).

The SRP protocol can be used for strong password authentication in SSL/TLS. The document describing how to include SRP in TLS is *[RFC5054]*.

6.2 Test Vectors

[RFC5054] provides test vectors to demonstrate calculation of the verifier and premaster secret (computed from B, k, g^x, a, and A). These test vectors can be found in `openssl/test/srptest.c`, and can be tested with the binary `srptest`. All test vectors passed.

[1] https://en.wikipedia.org/wiki/Secure_Remote_Password_protocol

6.3 Implementation

```
The inaccurate SRP comment was actually recently corrected in commit 495a1e5c3.
```

OpenSSL provides a set of cipher suites called TLS-SRP[2] that use the SRP protocol to provide an authenticated key exchange. The chosen version of the protocol is 6a.

The implementation is mainly organized in the following way:

- `openssl/crypto/srp/srp_lib.c` contains the core internals of the protocol (computations of A, B, k...).
- `openssl/crypto/srp/srp_vfy.c` contains more high-level functions, designed to create the verifier, parse the server file containing it, or set different fields.
- `openss/ssl/tls_srp.c` contains the interface with the TLS layer, managing the connection between the server and the client using the SRP parameters.

The different numbers used in the protocols are of type `BIGNUM`, which is a type for big numbers implemented by OpenSSL.

A first point of importance is that the code, while clear, is not well documented. Indeed, there are very few comments to explain the content of the different functions, and one needs to spend some time just to understand the interactions of the different parts of the code. A very demonstrative example is the format of the verifier file, described wrongly by a comment in the code (we will present the format later in section *Format*).

In order to setup a communication between a local client and servers, one first needs to create a verifier file with the following commands:

```
 $ touch password.srpv
 $ openssl srp -srpvfile password.srpv -add -gn 1536 user
 Enter pass phrase for user:
 Verifying - Enter pass phrase for user:
 $ cat password.srpv
 V   47/CC92kpj1TAuH6laYS1GAq3zRwd7gHiv9j8KxzaCJDwO1rlX4EVKHOpcjEeDjBGHqI4/ywF
J2WbJjZrLOBnoxSuEMyfCIhVZbkIiuLQHSno6DYj83qTAb3.l3DyrEZjgXz4G5xfpL24ZtHX1/2SBG
AhZYFLKrbYtnfoNzFT33X9AuAP5Ls9KJZ2gLEYZvPxt.0Yza8DyCqnrd734jX2JwTBFEU7Bxmyeyv9
WbIapQMkJncOGw2thquotGCTH/9  EsU3c/ApDnG.jFL7CdsfbjKVeot     user    1536
```

The file `password.srpv` contains the verifier v, the salt s, the username I and the index of the group parameters. It must be stored on the server. A quick way to setup a connection between a server and a client using the SRP protocol is to use the commands `s_server` and `s_client` provided by OpenSSL. Then the server is started with this command:

```
$ openssl s_server -nocert -cipher SRP -srpvfile password.srpv
```

The `-nocert` option deactivates the use of cipher suites that require a certificate for authentication (one can consider that the server is authenticated by its possession of the SRP verifier, as pointed in *[RFC5054]*.

The client is executed with this command (assuming that the server is on localhost):

```
$ /openssl s_client -srpuser user -cipher SRP -connect localhost
```

[2] https://en.wikipedia.org/wiki/TLS-SRP

6.3.1 Parameters

The parameters chosen by OpenSSL for the SRP protocol are:

- SHA1 for the hash function
- 20 random bytes for the salt s
- $x = H(s|H(I|\text{ ":" }|p))$ as defined in *[RFC2945]*.

[RFC5054] provides standard group parameters (N, g) to use for the protocol. OpenSSL's implementation uses these standard parameters groups, which are stored in the file `openssl/crypto/bn/bn_srp.c`. The server only accepts these standard parameters as they are identified by their id (see *Format*), and the client checks the parameters of the server in `srp_verify_server_param()`.

6.3.2 Quality of the code

Computations in `srp_lib.c` seem well implemented and following the standard of *[RFC2945]*. The code lacks comments but is quite straightforward to understand when it comes to computations on bignums. The same applies to the functions of `srp_vfy.c`: while understanding what each function does is not too difficult, the different interactions between the functions and the higher layers of TLS are often hard to picture. We also want to stress the importance of checking the return values of the functions computing bignums: for example, in the function `SRP_create_verifier_BN()`, the return value of `SRP_Calc_x()` is not checked. This means that if the computation of x fails (e.g. if an allocation or a hash digest has failed), the exponentiation afterwards with x is still performed with a null value for x, and will cause a segmentation fault during the creation of the verifier file. We reproduce the relevant piece of code below:

Listing 6.1: `openssl/crypto/srp/srp_vfy.c`

```
x = SRP_Calc_x(salttmp, user, pass);

*verifier = BN_new();
if (*verifier == NULL)
    goto err;

if (!BN_mod_exp(*verifier, g, x, N, bn_ctx)) {
    BN_clear_free(*verifier);
    goto err;
}
```

6.3.3 Security Checks

The *[RFC5054]* contains several security checks (detailed in section 3 of the RFC) that should be performed by the client and/or server:

- Check that $A \mod N$ and $B \mod N$ are not null: the function `SRP_Verify_A_mod_N()` is called by the server in `srp_generate_server_master_secret()` (in the file `tls_srp.c`) to check that $A \mod N$ is not null. Both the client and the server check $B \mod N$, in `srp_verify_server_param()` and `srp_generate_client_master_secret()` respectively.
- Check that a and b are at least 256-bit random numbers: while there is no explicit check, both parameters are instantiated with a call to `RAND_priv_bytes()` with a size of `SSL_MAX_MASTER_KEY_LENGTH` (384 bits by default) and the return error code is checked.

One can note the use of `RAND_priv_bytes()` instead of `RAND_bytes()`, which is fitting for sensitive buffers.

- Check that N is large enough: the client performs this check in `srp_verify_server_param()`.

A few other recommendations are given about the high-level layers of the protocol (mainly about resuming the session, preventing repeated connections from a malicious user...), but we had less time to study this part of the protocol and thus could not check the presence of these security measures.

6.3.4 Format

The format of the verifier file (in our setup, `password.srpv`) is described in a comment over the function `SRP_VBASE_init()` (in `srp_vfy.c`) that parses the file. However, we found that this description does not provide the right order for the parameters, and thus we detail here the format as we understand it.

We recall an example of the content of `password.srpv`:

```
 V   47/CC92kpj1TAuH6laYS1GAq3zRwd7gHiv9j8KxzaCJDwO1rlX4EVKHOpcjEeDjBGHqI4/ywF
J2WbJjZrLOBnoxSuEMyfCIhVZbkIiuLQHSno6DYj83qTAb3.l3DyrEZjgXz4G5xfpL24ZtHX1/2SBG
AhZYFLKrbYtnfoNzFT33X9AuAP5Ls9KJZ2gLEYZvPxt.0Yza8DyCqnrd734jX2JwTBFEU7Bxmyeyv9
WbIapQMkJncOGw2thquotGCTH/9  EsU3c/ApDnG.jFL7CdsfbjKVeot     user    1536
```

The "offsets" (indexing the words of the files, not the characters) used to parse the file can be found in `openssl/include/openssl/srp.h`:

Listing 6.2: `openssl/include/openssl/srp.h:95`

```
# define DB_srptype      0
# define DB_srpverifier  1
# define DB_srpsalt      2
# define DB_srpid        3
# define DB_srpgN        4
# define DB_srpinfo      5
```

So we understand that the file is composed of the following fields:

- a first word representing the type of the file (in our case, it is `DB_SRP_VALID`).
- the second being the verifier v encoded in base64 (it is indicated in the code of `srp_vfy.c` that SRP uses its own variant of base64 encoding).
- then the salt s used to compute the verifier, and encoded in base64.
- the identifying username I.
- the identifying number of the group parameters (N, g): this number is just the size of the parameters.
- optional info about the user than can be added when creating the file.

6.3.5 Fuzzing

In order to fuzz the TLS 1.3 part of SRP, we used the same technique we used to fuzz the core part of TLS 1.3 and the client and the server we developed. The TLS 1.3 part of SRP has a

very low number of lines of code. We didn't discover any vulnerability in SRP using fuzzing. During the SRP fuzzing, we reviewed the code of the TLS 1.3 part of SRP, and we didn't find any vulnerability in it.

6.4 Conclusion

The protocol seems to be correctly implemented: the underlying operations are correct, and we were able to successfully setup a connection between a server and a client. Nevertheless, the code lacks clarity, for example comments explaining how the different functions interact and when they are called. In particular, the higher levels of the protocols are the most difficult to understand. There also seems to be some oversights on checking the return values of some functions, as shown in section *Quality of the code*. A point we did not have time to study during the audit is the handling of secrets a and b.

7. Appendix

7.1 OpenSSL Patch

Listing 7.1: `fuzzing.patch`

```
diff --git a/ssl/packet_locl.h b/ssl/packet_locl.h
index 8e553e62b5..99f493466c 100644
--- a/ssl/packet_locl.h
+++ b/ssl/packet_locl.h
@@ -24,7 +24,10 @@ extern "C" {

 typedef struct {
     /* Pointer to where we are currently reading from */
-    const unsigned char *curr;
+    //const unsigned char *curr;
+    /*MYCODE*/
+    unsigned char *curr;
+    /*MYCODE*/
     /* Number of bytes remaining */
     size_t remaining;
 } PACKET;
diff --git a/ssl/record/rec_layer_s3.c b/ssl/record/rec_layer_s3.c
index 61010f4e72..591eac09d7 100644
--- a/ssl/record/rec_layer_s3.c
+++ b/ssl/record/rec_layer_s3.c
@@ -166,6 +166,9 @@ const char *SSL_rstate_string(const SSL *s)
 /*
  * Return values are as per SSL_read()
  */
+/*MYCODE*/
+extern int hook_flag;
+/*MYCODE*/
 int ssl3_read_n(SSL *s, size_t n, size_t max, int extend, int clearold,
                 size_t *readbytes)
 {
@@ -294,6 +297,23 @@ int ssl3_read_n(SSL *s, size_t n, size_t max, int extend, int␣
↪clearold,
             s->rwstate = SSL_READING;
             /* TODO(size_t): Convert this function */
             ret = BIO_read(s->rbio, pkt + len + left, max - left);
+
+            /*MYCODE*/
+            size_t len = (max - left);
+            if(len == 5){
+                uint8_t* u8_data = pkt + len + left;
+                printf("RECORD:\n");
+                dump_buffer(u8_data, len);
+                if(hook_flag == -1){
+                    save_to_file(u8_data, len);
+                }else{
+                    printf("NEW_RECORD :\n");
+                    read_from_file(u8_data, len);
+                    dump_buffer(u8_data, len);
+                }
```

(continues on next page)

(continued from previous page)

```
+            }
+            /*MYCODE*/
+
             if (ret >= 0)
                 bioread = ret;
         } else {

diff --git a/ssl/statem/statem.c b/ssl/statem/statem.c
index 1f221e7542..b1eceef496 100644
--- a/ssl/statem/statem.c
+++ b/ssl/statem/statem.c
@@ -529,6 +529,9 @@ static int grow_init_buf(SSL *s, size_t size) {
  * control returns to the calling application. When this function is recalled we
  * will resume in the same state where we left off.
  */
+/*MYCODE*/
+extern int hook_flag;
+/*MYCODE*/
 static SUB_STATE_RETURN read_state_machine(SSL *s)
 {
     OSSL_STATEM *st = &s->statem;
@@ -629,6 +632,15 @@ static SUB_STATE_RETURN read_state_machine(SSL *s)
                          ERR_R_INTERNAL_ERROR);
                 return SUB_STATE_ERROR;
             }
+            /*MYCODE*/
+            dump_packet(&pkt);
+            if(hook_flag == -1){
+                save_to_file(pkt.curr, pkt.remaining);
+            }else{
+                read_from_file(pkt.curr, pkt.remaining);
+                dump_packet(&pkt);
+            }
+            /*MYCODE*/
             ret = process_message(s, &pkt);

             /* Discard the packet data */

diff --git a/ssl/statem/statem_lib.c b/ssl/statem/statem_lib.c
index 44c9c2c856..631f032e2c 100644
--- a/ssl/statem/statem_lib.c
+++ b/ssl/statem/statem_lib.c
@@ -1093,6 +1093,12 @@ WORK_STATE tls_finish_handshake(SSL *s, WORK_STATE wst, int␣
↪clearbufs, int stop)
     return WORK_FINISHED_STOP;
 }

+/*MYCODE*/
+int hook_flag = -1;
+void set_hook_index(int index){
+    hook_flag = index;
+}
+/*MYCODE*/
 int tls_get_message_header(SSL *s, int *mt)
 {
     /* s->init_num < SSL3_HM_HEADER_LENGTH */
```

(continues on next page)

(continued from previous page)

```
@@ -1108,10 +1114,24 @@ int tls_get_message_header(SSL *s, int *mt)
                                           &p[s->init_num],
                                           SSL3_HM_HEADER_LENGTH - s->init_num,
                                           0, &readbytes);
+            /*MYCODE*/
+            if (i > 0) {
+                printf("HEADER:\n");
+                dump_buffer(&p[s->init_num], readbytes);
+                if(hook_flag == -1){
+                    save_to_file(&p[s->init_num], readbytes);
+                }else{
+                    read_from_file(&p[s->init_num], readbytes);
+                    printf("NEW_HEADER :\n");
+                    dump_buffer(&p[s->init_num], readbytes);
+                }
+            }
+            /*MYCODE*/
             if (i <= 0) {
                 s->rwstate = SSL_READING;
                 return 0;
             }
+
             if (recvd_type == SSL3_RT_CHANGE_CIPHER_SPEC) {
                 /*
                  * A ChangeCipherSpec must be a single byte and may not occur
diff --git a/ssl/statem/statem_srvr.c b/ssl/statem/statem_srvr.c
index 60e0bc7373..aca30a6b45 100644
--- a/ssl/statem/statem_srvr.c
+++ b/ssl/statem/statem_srvr.c
@@ -1081,6 +1081,78 @@ size_t ossl_statem_server_max_message_size(SSL *s)
     }
 }

+/*MYCODE*/
+#define MIN(a,b) (((a)<(b))?(a):(b))
+extern int hook_flag;
+FILE *input_file = NULL;
+void set_input_filename(char *filename)
+{
+    if(hook_flag == -2){
+        printf("use set_hook_index before !\n");
+        exit(0);
+    }
+    if(hook_flag == -1){
+        input_file = fopen(filename, "w");
+    }else{
+        input_file = fopen(filename, "r");
+    }
+}
+
+void save_to_file(uint8_t* data, size_t len)
+{
+    if(input_file == NULL){
+        return;
+    }
+    if(data == NULL){
```

(continues on next page)

(continued from previous page)

```
+        return;
+    }
+    fwrite(data, 1, len, input_file);
+}
+
+void read_from_file(uint8_t* data, size_t len)
+{
+    if(input_file == NULL){
+        return;
+    }
+    if(data == NULL){
+        return;
+    }
+    fread(data, sizeof(char), len, input_file);
+}
+
+void dump_buffer(char *buffer, size_t len)
+{
+       for(size_t i=0; i<len; i++){
+               printf("%02x ", buffer[i] & 0xFF);
+        if(i%8 == 7){
+            printf("   ");
+        }
+        if(i%16 == 15){
+            printf("\n");
+        }
+       }
+    if(len%16 != 15){
+        printf("\n");
+    }
+    printf("\n");
+}
+
+uint32_t check_buffer(uint8_t* buffer, size_t len)
+{
+    uint32_t check = 0;
+       for(size_t i=0; i<len; i++){
+        check += buffer[i];
+    }
+    return check;
+}
+
+void dump_packet(PACKET* pkt)
+{
+    printf("dump_packet (len:%d check:%08x):\n", pkt->remaining, check_buffer(pkt->
↪curr, pkt->remaining));
+    dump_buffer(pkt->curr, pkt->remaining);
+}
+/*MYCODE*/
+
 /*
  * Process a message that the server has received from the client.
  */


diff --git a/test/tls13ccstest.c b/test/tls13ccstest.c
```

(continues on next page)

(continued from previous page)

```
index 41e4896fa9..a14f217ce8 100644
--- a/test/tls13ccstest.c
+++ b/test/tls13ccstest.c
@@ -483,14 +484,61 @@ static int test_tls13ccs(int tst)
     return ret;
 }

+/*MYCODE*/
+#include <openssl/rand.h>
+
+static void stdlib_rand_cleanup() {}
+static void stdlib_rand_add(const void *buf, int num, double add_entropy) {}
+static int stdlib_rand_status() { return 1; }
+
+static void stdlib_rand_seed(const void *buf, int num)
+{
+        srand( *((unsigned int *) buf) );
+}
+
+static int stdlib_rand_bytes(unsigned char *buf, int num)
+{
+        for( int index = 0; index < num; ++index ){
+                buf[index] = rand() % 256;
+        }
+        return 1;
+}
+
+RAND_METHOD stdlib_rand_meth = {
+        stdlib_rand_seed,
+        stdlib_rand_bytes,
+        stdlib_rand_cleanup,
+        stdlib_rand_add,
+        stdlib_rand_bytes,
+        stdlib_rand_status
+};
+
+RAND_METHOD *RAND_stdlib() { return &stdlib_rand_meth; }
+
+time_t time(time_t *t){
+    time_t p_time_t;
+    memset(&p_time_t, 0, sizeof(time_t));
+    return p_time_t;
+}
+
 int setup_tests(void)
 {
+    RAND_set_rand_method(RAND_stdlib());
+    unsigned int seed = 0x00beef00;
+    RAND_seed(&seed, sizeof(seed));
+
+    char *input_filename = test_get_argument(2);
+    set_hook_index(atoi(test_get_argument(3)));
+    set_input_filename(input_filename);
+
+/*MYCODE*/
+
```

(continues on next page)

(continued from previous page)

```
     if (!TEST_ptr(cert = test_get_argument(0))
             || !TEST_ptr(privkey = test_get_argument(1)))
         return 0;

-    ADD_ALL_TESTS(test_tls13ccs, 12);
-
+    //ADD_ALL_TESTS(test_tls13ccs, 12);
+    test_tls13ccs(11);
     return 1;
 }
```

7.2 How to apply the patch and start fuzzing

Clone the openssl repository:

```
$ git clone https://github.com/openssl/openssl.git
```

Enter the openssl directory:

```
$ cd openssl
```

Reset to a specific revision:

```
$ git reset --hard dc55e4f70f401c5869410d6a0c068c18c3fd53ec
```

Apply the fuzzing patch:

```
$ patch -p1 < fuzzing.patch
```

Configure the compilation:

```
$ CC=afl-clang-fast ./config enable-fuzz-afl no-shared enable-tls1_3 -DPREDICT --debug
```

Compile:

```
$ make -j
```

Create the input directory:

```
$ mkdir /tmp/in
```

Create a serialized TLS 1.3 session:

```
$ ./test/tls13ccstest ~/certrsa.pem ~/keyrsa.pem /tmp/in/data.bin -1
```

Fuzz with afl:

```
$ afl-fuzz -i /tmp/in -o /tmp/out -t 50+ -S fuzzer02 ./test/tls13ccstest certrsa.pem␣
↪keyrsa.pem @@ 0
```

7. Bibliography

[SP800-90A] E. Barker, J. Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, 2015, https://csrc.nist.gov/publications/detail/sp/800-90a/rev-1/final

[RFC2945] T. Wu, *The SRP Authentication and Key Exchange System*, September 2000, https://tools.ietf.org/html/rfc2945

[RFC5054] D. Taylor, T. Wu, N. Mavrogiannopoulos, T. Perrin, *Using the Secure Remote Password (SRP) Protocol for TLS Authentication*, November 2007, https://tools.ietf.org/html/rfc5054