

Repairing Critical Vulnerabilities and Flaws in OpenSSL library

Madanayaka B.P.W
Department of Cyber Security
Sri Lanka Information Technology
Malabe, Sri Lanka
it20208776@my.sliit.lk

Abstract— In this project, I'll look into some flaws in the OpenSSL 1.0.1 software library. I'll look into this code utilizing the manual code analysis method.

I was able to find and fix a heap overflow vulnerability, an error in the OSCP Status Request extension unbounded memory growth, a bug in the Sanity check ticket length, a problem with the DTLS buffered message, a problem with the DTLS replay protection, and a problem with OpenSSL error handling by using the manual code analysis method. I was also able to fix the Heartbleed bug in OpenSSL 1.0.1.

Keywords—Python, OpenSSL, vulnerability, Heartbleed, Script

I. INTRODUCTION

Operating systems, software libraries, and apps are all now connected to the internet and regularly updated. The bulk of the time, these updates are made to fix issues rather than provide new functionality. The system's ability to protect itself against new viruses and malware is improved as a result. however, most software, lacks this form of connectivity, making it accessible to attackers. You may secure your software in one of two ways: by accessing it on every system and altering the code whenever an attacker emerges, or by minimizing the susceptibility of your code during development. [1]

OpenSSL is a software library for applications that require to protect communications over computer networks from eavesdropping or identify the other side. The majority of HTTPS websites, as well as most Internet servers, employ it. The SSL and TLS protocols are implemented in OpenSSL, which is an open-source project. The core library, written in C, implements fundamental cryptographic functions as well as providing a variety of utility functions. I'll address certain code issues I've discovered in OpenSSL using the manual code analysis approach. The technique of analyzing source code line by line in order to uncover potential code vulnerabilities is known as manual secure code analysis.

The software's security is referred to as "code vulnerability." It's a flaw in the code that makes security breaches more likely. Hackers will be able to obtain data, mess with your program, or, in the worst-case scenario, completely destroy the code if you attach an endpoint to it. A significant number of unexpected development blunders that will surely come back to bite you. The unsafe code exposes both the user and the developer, and it will harm everyone if it is abused. [1]

II. METHODOLOGY

As I mention before, I am using static code analysis approaches throughout this project to identify flaws and vulnerabilities.

A. Fixing the Heartbleed vulnerability within the OpenSSL 1.0.1 version.

Heartbleed is a code flaw in the OpenSSL cryptography library. This is how it appears within the code, "Fig .1"

```
if (hotype == TLS1_HB_REQUEST)
{
    unsigned char *buffer, *bp;
    unsigned int write_length = 1 /* heartbeat type */ +
                                2 /* heartbeat length */ +
                                payload + padding;

    int r;

    if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
        return 0;

    /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(write_length);
    bp = buffer;

    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
    bp += payload;
    /* Random padding */
    RAND_pseudo_bytes(bp, padding);
```

FIG. 1- HEARTBLEED VULNERABILITY IN OPENSSL 1.0.1

As I previously stated, OpenSSL is used to build the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols, which offers developers with tools and resources. Websites, emails, instant messaging (IM) programs, and virtual private networks use the SSL and TLS protocols to offer security and anonymity for communication over the Internet (VPNs). As a result, when the Heartbleed vulnerability was discovered, any programs that used OpenSSL components were vulnerable.

The security of the most frequently used Internet communication protocols is compromised by Heartbleed (SSL and TSL). Heartbleed-affected websites' memory can be accessed by potential attackers. As a result, astute burglars may be able to decrypt the data. [3] [6]

If the encryption keys were leaked, a malicious person may get access to the credentials—such as names and passwords—needed to break into networks. Depending on the authorization level of the gained credentials, a malicious user can launch more attacks, listen in on conversations, impersonate people, and steal data from within the system.

The Heartbeat extension should, in theory, protect SSL and TLS protocols by verifying connections sent to the host. It enables one end of the communication to send a Heartbeat Request message to the other.

Each message contains a payload, which is a text string containing the data to be conveyed, as well as a memory length number, which is commonly a 16-bit integer. Before giving the necessary information, the heartbeat extension performs a bound check that validates the input request and delivers the exact payload length that was requested.

The vulnerability in the verification procedure was caused by a weakness in the OpenSSL heartbeat extension. The Heartbeat addon built a memory buffer without going through the validation procedure instead of completing a boundaries check. Threat actors can ask for any data in the memory buffer and get up to 64 kilobytes in response.

Memory buffers are temporary memory storage areas that are used to hold data while it is being sent. They may include a variety of data kinds that represent different sorts of information storage. A memory buffer, in essence, holds data until it is transported to its final destination.

Rather than arranging data, a memory buffer stores information in chunks. Sensitive and financial data, as well as passwords, cookies, website pages and pictures, digital assets, and data in transit, might all be stored in a single memory buffer. When threat actors use the Heartbleed flaw, they trick the Heartbeat extension into giving them full access to the memory buffer. [5]

To fix the Heartbleed vulnerability, I added some code to the heartbeat extension that directed it to ignore any Heartbeat Request messages that asked for more data than the payload allowed. “Fig. 2”

if (1 + 2 + payload + 16 > s->s3->rrec.length) return 0;
/* silently discard per RFC 6520 sec. 4 */

```
if (hbtype == TLS1_HB_REQUEST)
{
    unsigned char *buffer, *bp;
    unsigned int write_length = 1 /* heartbeat type */ +
        2 /* heartbeat length */ +
        payload + padding;

    int r;

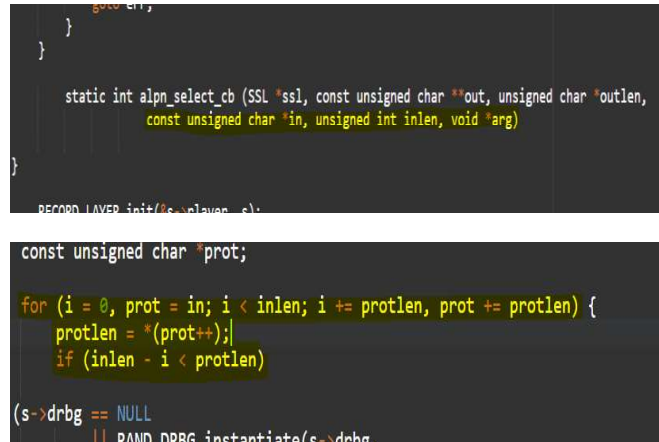
    if (write_length > SSL3_RT_MAX_PLAIN_LENGTH)
        return 0;

    /* Allocate memory for the response, size is 1 byte
     * message type, plus 2 bytes payload length, plus
     * payload, plus padding
     */
    buffer = OPENSSL_malloc(1 + 2 + payload + padding);
    buffer = OPENSSL_malloc(write_length);
    bp = buffer;

    /* Enter response type, length and copy payload */
}
```

FIG. 2- FIXING HEARTBLEED VULNERABILITY IN OPENSSL 1.0.1

B. I observed that OpenSSL is vulnerable to a heap overflow issue using the approach described above. “Fig. 3”



```
static int alpn_select_cb(SSL *ssl, const unsigned char **out, unsigned char *outlen,
    const unsigned char *in, unsigned int inlen, void *arg)
{
    RECORD_LAYER_init(&ssl->record_layer, c);
}

const unsigned char *prot;

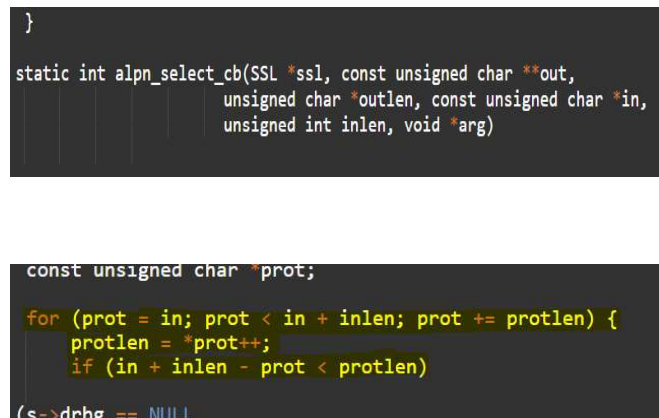
for (i = 0, prot = in; i < inlen; i += protlen, prot += protlen) {
    protlen = *(prot++);
    if (inlen - i < protlen)
        break;
}

(s->drbg == NULL
    || RAND_DRBG_instantiate(s->drbg,
        (const unsigned char *)prot,
        protlen))
```

FIG. 3- HEAP OVERFLOW VULNERABILITY.

Heap Overflows (CWE-122) are a subclass of the Buffer Overflow vulnerability that can affect applications written in a variety of programming languages. The name refers to any situation in which software attempts to move data from one location in memory to a fixed-length buffer allocated on the heap that is insufficient to hold the data. [3]

The solutions I used to solve the heap overflow issue in the software are shown in the screen figure below. “Fig. 4”



```
static int alpn_select_cb(SSL *ssl, const unsigned char **out,
    unsigned char *outlen, const unsigned char *in,
    unsigned int inlen, void *arg)
{
    const unsigned char *prot;

    for (prot = in; prot < in + inlen; prot += protlen) {
        protlen = *(prot++);
        if (in + inlen - prot < protlen)
            break;
    }

    (s->drbg == NULL
        || RAND_DRBG_instantiate(s->drbg,
            (const unsigned char *)prot,
            protlen))
```

FIG. 4- FIXING HEAP OVERFLOW VULNERABILITY.

- C. Using the manual code review approach, I discovered that a mistake in OpenSSL caused unbounded memory expansion in the OCSF Status Request extension. **CVE-2016-6304** is a vulnerability that has been identified in the past year. [2] “Fig.5”

```
if (!s->tlsext_ocsp_ids || !s->tlsext_ocsp_ids->sk_OCSP_RESPID_new_null()) { *al = SSL_AD_INTERNAL_ERROR;
OCSP_RESPID_free(id);
return 0;
}
if (item == NULL)
return
```

FIG. 5- UNBOUNDED MEMORY EXPANSION VULNERABILITY

A memory leak issue was discovered in the way OpenSSL processed TLS status request extension data during session renegotiation. If OCSF stapling support was enabled, a remote attacker might force an OpenSSL-based TLS server to consume excessive memory and, perhaps, terminate abruptly after exhausting all available memory. This might result in a denial-of-service attack. To avoid unbound memory loss, I tried to delete any OCSF RESPIDs from a prior handshake to solve this vulnerability.

The screen screenshots of the corrected vulnerability are shown below. “Fig. 6”

```
sk_OCSP_RESPID_pop_free(s->tlsext_ocsp_ids,
OCSP_RESPID_free);
if (dsize > 0) {
s->tlsext_ocsp_ids = sk_OCSP_RESPID_new_null();
if (s->tlsext_ocsp_ids == NULL) {
    "al = SSL_AD_INTERNAL_ERROR;
    return 0;
    while ((item = pqueue
}
} else {
    s->tlsext_ocsp_ids = NULL;
}
}
DTLS is capable Handshake comm many
```

FIG. 6- FIXING UNBOUNDED MEMORY EXPANSION VULNERABILITY

- D. After extensively studying the code, I discovered a flaw in the Sanity check ticket length. [2] “Fig. 7”
(CVE-2016-6302).

```
if
(eticklen < 48)
return 2;
```

FIG. 7- SANTY CHECK TICKET LENGTH VULNERABILITY

A DoS attack can occur if a server employs SHA512 for TLS session ticket HMAC. A faulty ticket will result in an OOB read, causing the server to fail. To get around this problem, make sure the ticket length is more than keyname + IV + HMAC.

The screenshots below show how the code was fixed. “Fig.8”

```
if (eticklen <= 16+ EVP_CIPHER_CTX_iv_length(&ctx) + mlen) {
HMAC_CTX_cleanup(&hctx);
EVP_CIPHER_CTX_cleanup(&ctx);
return 2;
}
```

FIG. 8- FIXING SANTY CHECK TICKET LENGTH VULNERABILITY

- E. During the code review, I discovered a problem with the DTLS buffered message. “Fig. 9”

```
item = pqueue_peek(s->d1->buffered_messages);
if (item == NULL)
return 0;

void dtls1_clear_record_buffer(SSL *s)
pitem
"item;
for (item = pqueue_pop(s->d1->sent_messages);
item != NULL; item = pqueue_pop(s->d1->sent_messages)) {
    dtls1_hm_fragment_free ((hm_fragment *) item->data); pitem_free(item);
}
}
```

```

while ((item= pqueue_pop(s->d1->buffered_app_data.q)) != NULL) { rdata (DTLS1_RECORD_DATA *)item->data;
if (rdata->rbuf.buf) {
OPENSSL_free(rdata->rbuf.buf);
}
OPENSSL_free(item->data); pitem_free(item);
}
}

```

FIG. 9- DTLS BUFFERED VULNERABILITY

DTLS is capable of delivering out-of-order records. Handshake communications can also be split throughout many records since they can be bigger than a single packet (as with normal TLS). That means the communications can come in parts, which we'll have to put back together. We have a backlog of buffered "from the future" messages, which are messages that we aren't ready to deal with right now but have come early. It's conceivable that the communications stored there are incomplete; they might be one or more pieces that are being rebuilt at the moment.

The algorithm anticipates that we will complete the reassembly at some point, and when that occurs, the full message is removed from the queue at the location where it is required.

DTLS, on the other hand, is packet loss tolerant. To get around this, DTLS conversations can be retransmitted. We ignore the message in the queue and only use the non-fragmented version if we get a complete (non-fragmented) message from the peer after receiving a fragment. At that point, the pending mail will never be deleted. [3]

To complete the handshake, the peer might potentially send "future" messages that we will never see. Each message is allocated a sequence number (starting from 0). A message fragment for the current message sequence number or any future sequence up to ten will be accepted. However, if the sequence number of the Finished message is 2, anything in the queue with a higher number is simply ignored.

We may end up with "orphaned" data in the queue as a result of these two ways, which will never be cleared unless the connection is canceled. At that moment, all queues are deleted.

An attacker might exploit this by filling the queues with large messages that will never be used, attempting a DoS through memory depletion.

The following are the steps made to address this issue. "Fig. 10"

```

do{
item= pqueue_peek(s->d1->buffered_messages);
SF (Sten == NULL)
return 0;
frag (hm_fragment *)item->data;

if (frag->msg_header.seq<s->d1->handshake_read_seq) {
/* This is a stale message that has been buffered so clear it */
pqueue_pop(s->d1->buffered_messages);
dtls1_hm_fragment_free(frag);

pitem_free(item);
item= NULL;
frag= NULL;
}
} while (item= NULL);

```

```

while ((item= pqueue_pop(s->d1->buffered_app_data.q)) != NULL) {
rdata (DTLS1_RECORD_DATA *)item->data;
if (rdata->rbuf.buf) {
OPENSSL_free(rdata->rbuf.buf);
} OPENSSL_free(item->data);
pitem_free(item);
}
dtls1_clear_received_buffer(s);
dtls1_clear_sent_buffer(s);
}
void dtls1_clear_received_buffer (SSL *s)
{
pitem "item= NULL;
hm_fragment frag = NULL;

```

FIG. 10- FIXING DTLS BUFFERED VULNERABILITY

F. Using manual code view procedure, I was able to find an issue within the DTLS replay protection. "Fig. 11"

```

static int dtls1_process_record(SSL *s);
if (!dtls1_process_record(s))
return (0);

```

FIG. 11- DTLS REPLAY PROTECTION VULNERABILITY

The window's "right" hand edge is set to the highest sequence number we've received so far, resulting in a sliding "window" of valid record sequence numbers. Records that fall outside the "left" edge of the window are discarded. The records in the window are compared to a list of records received thus far. If we've previously rejected it, we'll reject it again.

It verifies the record's MAC if the code has not yet received the record or if the sequence number is outside the right-hand border of the window. The record is discarded if the MAC verification fails.

If the record isn't marked as received, mark it as such. The window was moved along until the right-hand edge aligned with the freshly received sequence number if the sequence number was off the right-hand edge. [4]

If the packets are re-ordered, for example, records for future epochs may arrive before the CCS. Because we have not yet received the CCS, we are unable to decode or confirm the MAC of those data. OpenSSL places such records in an unprocessed records queue. Even if the MAC has not yet been confirmed, it changes the window immediately. This will happen if you're in the middle of a handshake/renegotiation.

An attacker might take advantage of this by transmitting a record for the next epoch with a very big sequence number (which does not need to decrypt or have a valid MAC). The right-hand boundary of the window is shifted extremely far to the right, leading in a denial of service and the discarding of all following legitimate packets.

The solutions for the provided issue inside the code are shown in the screenshots below. "Fig. 12"

```
do {
    item pqueue_peek(s->d1->buffered_messages);
    if (item NULL)
        return 0;

    frag (hm_fragment) item->data;
    if (frag->msg_header.seq<s->d1->handshake_read_seq) {
        /* This is a stale message that has been buffered so clear it */
        pqueue_pop(s->d1->buffered_messages);
        dtls1_hm_fragment_free(frag);
        pitem_free(item);
        item=NULL;
        frag= NULL;
    }
} while (item == NULL);
```

```
while ((item=pqueue_pop(s->d1->buffered_app_data.q)) != NULL) {
    rdata (DTLS1_RECORD_DATA) item->data;
    if (rdata->rbuf.buf) {
        OPENSSL_free(rdata->rbuf.buf);
    }
    OPENSSL_free(item->data);
    pitem_free(item);
}
dtls1_clear_received_buffer(s);
dtls1_clear_sent_buffer(s);
}

void dtls1_clear_received_buffer (SSL *s)
{
    pitem item NULL;
    hm_fragment "frag = NULL;
}
}
```

FIG. 12- FIXING DTLS REPLAY PROTECTION VULNERABILITY

G. I was able to discover that there is a flaw with OpenSSL error handling by utilizing the manual code view approach.

Screenshots of OpenSSL error handling flaws are shown below. "Fig. 13"

```
(const unsigned char *) SSL_version_str,
    sizeof(SSL_version_str) - 1) == 0) {
    CRYPTO_THREAD_LOCK_FREE(s->lock);
```

```
if (s->lock == NULL)
```

FIG. 13- OPENSSL ERROR HANDLING FLAWS

Inappropriate error handling may arise when a user receives an error message that provides information about how this library operates. Messages like this can assist attackers in obtaining information that can be used to gain access to secure places. For example, an error message containing information about the structure of a SQL database table might provide attackers with everything they need to launch a successful SQL injection attack. [4]

The screen photos below show the steps I took to resolve this issue within the program. "Fig. 14"

```
if (s->lock == NULL) {
    OPENSSL_free(s);
    s = NULL;
}
```

```
sizeof(SSL_version_str) - 1) == 0)
```

FIG. 14- FIXING OPENSSL ERROR HANDLING FLAWS

REFERENCES

- [1] O. Project, "OpenSSL Project Cryptography and SSL / TLS Toolkit," [Online]. Available: <https://www.openssl.org/news/vulnerabilities.html>.
- [2] "cvedetails.com," cvedetails, [Online]. Available: https://www.cvedetails.com/vulnerability-list/vendor_id-217/Openssl.html
- [3] A. P. C, "en.wikipedia.org," wikipedia, 19 May 2022. [Online]. Available: <https://en.wikipedia.org/wiki/OpenSSL>
- [4] s. dragon, "www.ssldragon.com," ssldragon, 5 Aug 2019. [Online]. Available: <https://www.ssldragon.com/blog/what-is-openssl-and-how-it-works/>.
- [5] Synopsys, "heartbleed.com," Synopsys, 03 June 2020. [Online]. Available: <https://heartbleed.com/>.
- [6] C. Contributing writer, "www.csoonline.com," csoonline, 13 Sep 2017. [Online]. Available: <https://www.csoonline.com/article/3223203/the-heartbleed-bug-how-a-flaw-in-openssl-caused-a-security-crisis.html>.