UM-SJTU Joint Institute
Project Two
OpenSSL and Vulnerability
(Ve475)

Group 7
OpenSSL

Ma Siyin	**517370910003**
Ming Xingyu	**517370910224**
Zhang Liqin	**517370910123**
Marina Chen	**516370990010**

Date: from 2020/7/19 to 2019/7/27

Abstract

The purpose of this report is to explore and analyze the OpenSSL library and understand the recent security issues. Given its popularity and widely applicable usage, security is essential for all its users. The development team has been actively updating the software library for the past 22 years. However, vulnerabilities in its code still exist. We will be analyzing seven large vulnerabilities types in our report, namely: Bypass Vulnerability, Denial of Service Vulnerability, Remote Code Execution Vulnerability, Buffer Overflow Vulnerability, Gain Information Vulnerability, Memory Corruption Vulnerability, and Heartbleed Security Vulnerability. Each vulnerability has its one unique properties, but many similarities occur. The vulnerabilities exists largely due to weakness in structures and leaks in address and memories. Any vulnerability of the code can be exploited in an attack giving hackers access to the code and the user's computers, in worst cases, to freely implement malicious codes or steal data. Our report will not only dive into descriptions of the vulnerabilities previously listed, but also provide real instances of how they occur in the OpenSSL library and examples to fix the bugs.

Contents

1 Introduction

A well known library, OpenSSL is commonly used in applications, such as HTTPS websites, to ensure network communication security. Available for Unix-like operating systems and Windows, the command-line tool is primarily written in C and a mix of assembly and Perl. The general-purpose cryptography library provides a selection of possible scenarios for the users from generating private keys to installing personalized SSL and TLS certificates. Given its wide applicability and free licensing, its popularity is only to be expected. Being first released nearly 22 years ago, the open-source toolkit is still regularly updated by the OpenSSL project team because vulnerabilities in its code exist until now.

2 Overview of Recent Security Vulnerabilities

Since it came out, OpenSSL has had multiple vulnerabilities. In this project, we will be focusing on the security vulnerabilities seen in last decade. According to the statistics provided by the CVE Details database[8], we notice that there are mainly 6 types of vulnerabilities in OpenSSL, which are illustrated in the following table.

Year	Occurrence	DoS	Code Execution	Overflow	Memory Corruption	Bypass	Gain information
2019	7	1					1
2018	11	2					1
2017	12	2		2			3
2016	34	24	2	9	5		8
2015	34	24		4	4		2
2014	24	17	1	3		1	3
2013	4	3					
2012	16	10		2	2		1
2011	4	2				1	1
2010	12	4	3	1		2	1

Table 1: Vulnerability Trends Over Last Ten Years

In the following sections, we will study specific codes, discussing the existence of the security vulnerabilities seen in the recent updates and correct implementations.

3 Bypass Vulnerability

3.1 Definition

Authentication bypass vulnerability allows attackers to perform attacks by bypassing the authentication mechanism. There are two possible causes

1. A weak authentication mechanism failed to enforce verified accesses
2. Unauthorized use of valid session identifications or cookies.

3.2 CCS Injection Vulnerability

In OpenSSL, a bypass vulnerability known as **CVE-2014-0224** exists due to the fragility in the keying materials method within the security process[1]. According to the bug reporter Masashi Kikuchi, OpenSSL accepts ChangeCipherSpec (CCS) incorrectly during a handshake. In a typical scenario, the client and the server should exchange messages as depicted below:

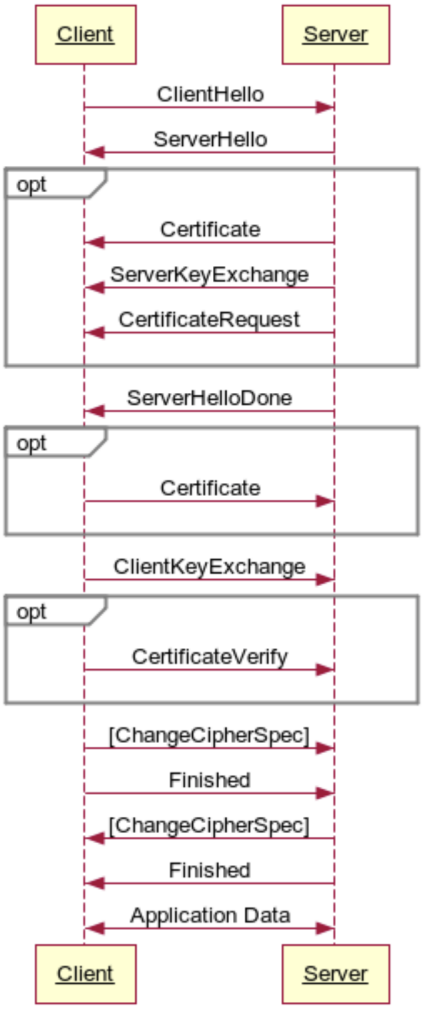

Figure 1: Message flow of Full Handshake.

From the graph, we can observe that ChangeCipherSpec has specific locations during the handshake and OpenSSL has a timing difference when sending and receiving CCS. By using the man-in-the-middle attack, an attacker can affect the CCS transit. With an uniquely

devised handshake, the attacker can force one to use a weak keying material from any location and at any authentication level. The vulnerability can be exploited, leading to a possible situation where the attacker is able to obtain any level of information. However, this attack can only occur if the client and server is vulnerable[2].

3.3 A correct implementation of CCS

To fix the bug as mentioned above, we have to make sure that the client sent a message and the server received it in the order shown in Figure 1. The code below gives a simple model of correctly sent handshake messages:

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case hello_request:       HelloRequest;
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        case certificate:         Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:   ServerHelloDone;
        case certificate_verify:  CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:            Finished;
    } body;
} Handshake;
```

4 Denial of Service(DoS) Vulnerability

4.1 Definition

Another type of attack is the Denial-of-service attack, abbreviated as DoS. As suggested by its name, the attack relies on the interrupted connection between users and the Internet. An attacker can force the interruption of services by overwhelming the target with excessive requests causing the system to overload. As a result, the user will experience a "denial of service" as legitimate requests for any resource from the computer and network are unable to be executed. [1]

As we can see in Table 1, DoS vulnerability is the most frequent vulnerability found on OpenSSL. This is one of two notable vulnerabilities we want to highlight in this project.

4.2 OCSP stapling Vulnerability

A type of DoS vulnerability found in the OpenSSL is **CVE-2011-0014**. This issue happens during the handshake process when the client sends an incorrectly formatted ClientHello message, resulting in OpenSSL to read a wrong memory address. The leaked contents of parsed OCSP extensions can be exploited as a DoS by attackers.[3]

We can test OCSP response using OpenSSL with the following command[5]

```
1 openssl s_client -connect holmesian.org:443 -servername holmesian.org
   ↪  -status -tlsextdebug < /dev/null 2>&1 | grep -i "OCSP response"
2
3 ************ Output **************
4 OCSP response: OCSP Response Data:
5 OCSP Response Status: successful (0x0)
6 Response Type: Basic OCSP Response
```

4.3 ClientHello sigalgs Vulnerability

CVE-2015-0291 is another DoS vulnerability verified as a high-severity issue by OpenSSL[4]. When a renegotiation occurs, a client can deny service remotely by abusing an invalid signature_algorithms extension in the ClientHello message, causing the NULL pointer to be dereferenced.

The following code is an actual patch for this bug[6]. As we can see, NULL pointer is assigned to *salgs* only when *nmatch* is true. Therefore, the DoS attack cannot be performed by creating mismatched *salgs*.

```
-        if (!nmatch)
-            return 1;
-        salgs = OPENSSL_malloc(nmatch * sizeof(TLS_SIGALGS));
-        if (!salgs)
-            return 0;
-        nmatch = tls12_shared_sigalgs(s, salgs, pref, preflen, allow,
 ↪  allowlen);
```

```
+        if (nmatch) {
+            salgs = OPENSSL_malloc(nmatch * sizeof(TLS_SIGALGS));
+            if (!salgs)
+                return 0;
+            nmatch = tls12_shared_sigalgs(s, salgs, pref, preflen, allow,
 ↪  allowlen);
+        } else {
+            salgs = NULL;
+        }
```

5 Remote Code Execution Vulnerability

5.1 Definition

Remote code execution (RCE) vulnerability is one of the most dangerous bugs of its kind. It allows the attacker to freely take control of a computer owned by another, and execute malicious code on it without authority regardless of the geographic location the computer is at.

5.2 Use After Free(UAF) Vulnerability

CVE-2016-6309 is a critical UAF security issue where a buffer storing the message could experience reallocation if the server receives a large message (one greater than 16 thousand bytes). As a result, a dangling pointer pointing to the previous direction is overlooked leaving room for any attempts to write or execute malicious codes in the previously freed location. Thus, it is listed as code execution vulnerability.

According to the commit [7], OpenSSL mainly changes two parts of codes to fix this bug.

```
+    static int grow_init_buf(SSL *s, size_t size) {
+         size_t msg_offset = (char *)s->init_msg - s->init_buf->data;
+         if (!BUF_MEM_grow_clean(s->init_buf, (int)size))
+             return 0;
+         if (size < msg_offset)
+             return 0;
+         s->init_msg = s->init_buf->data + msg_offset;
+         return 1;
+     }
```

In line 7, we notice that `s->init_msg` gets updated after calling `BUF_MEM_grow_clean`, which originally allocates and reallocates space after receiving the length of the string. Therefore, `str->data` is a dangling pointer, it will cause a `UAF`. With `grow_init_buf`, the value stored in `str->data` will be updated every time.

```
10          if (!SSL_IS_DTLS(s)
11                  && s->s3->tmp.message_size > 0
```

```
-               && !BUF_MEM_grow_clean(s->init_buf,
-                                      (int)s->s3->tmp.message_size
-                                      + SSL3_HM_HEADER_LENGTH)) {
```

```
+               && !grow_init_buf(s, s->s3->tmp.message_size
+                                    + SSL3_HM_HEADER_LENGTH)) {
```

With a regular change in line 15 and 16, `str->data` gets its buffer correctly updated.

6 Buffer Overflow Vulnerability

6.1 Definition

Buffer overflow is one of the most notable vulnerabilities. It exists when data exceeds the buffer boundary, or when data gets past the buffer to other memory areas. Since a buffer utilizes allocated memory to store its data like a string of characters or integer arrays, any action that violates the boundaries of the memory buffer could cause corruption of data, which can lead to the program crashing or executing malicious codes [11]. Vulnerabilities regarding overflow have been discovered 29 times, most of which are defined in 3 categories.

1. CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer.

2. CWE-787: Out-of-bounds Write.

3. CWE-190: Integer Overflow or Wraparound.

From the definition of the three vulnerabilities, we find that CWE-199 is the parent of CWE-787, and CWE-190 precedes CWE-119. We will introduce CWE-199 to help understand how buffer overflow threatens the security of the software.

6.2 CWE-119

CWE-119 occurs when some language does not forbid direct addressing of the position of memory and does not guarantee that the assigned locations are valid in terms of the referenced memory buffer. Therefore, some of the memory locations may be used for variables or data structures that are not designed to be there.

From these cases, we find that buffer overflow always happens with DoS, memory corruption, or gain information vulnerability. This is determined by the properties of the buffer overflow. Listed below are the consequences of CWE-199 along with other types of vulnerabilities.[9]

1. Execute Unauthorized Code or Commands, Modify Memory
   If an attacker can control the memory that is accessible to them, then they can execute some code. An example of an attack could be the redirection of pointers to their own codes.

2. Read Memory; DoS: Crash, Exit, or Restart; DoS: Resource Consumption (CPU); DoS: Resource Consumption (Memory)
   A probable consequence of memory access out of boundary is the corruption of relevant data causing crashes. Another attack that exists is the infinite loop.

3. Read Memory
   If an attacker applies reading operations out of memory's boundary, he or she could also obtain access to user's sensitive information, which can be used for future attacks. If the information contains the system details, the security of the entire system is threatened.

In our example below, the user's IP address is verified on its formation and then checked with the host name before being copied. [9]

```
1 void host_lookup(char *user_supplied_addr){
2 struct hostent *hp;
3 in_addr_t *addr;
4 char hostname[64];
5 in_addr_t inet_addr(const char *cp);
6
```

```
/*routine that ensures user_supplied_addr is in the right format for
    conversion */

validate_addr_form(user_supplied_addr);
addr = inet_addr(user_supplied_addr);
hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
strcpy(hostname, hp->h_name);
}
```

Even though, in our code, the buffer that is storing the host name has a size of 64 bytes, the host name is not guaranteed to be larger. With this information, any attacks using a host name that has a size larger than our buffer can cause user's data to be overwritten or even leakage of information about the flow of control.

Then, we are going to introduce the most famous security vulnerability in OpenSSL, which is heartbleed. It is closely related with overflow problems and will provide us with more insights about buffer overflow.

7 Gain Information

7.1 Definition

Most of the vulnerabilities about gaining information is defined as CWE-200. So we will introduce the properties of this vulnerability through this document.

Gain information vulnerability occurs when an unauthorized user gains access to one's sensitive information.[10] There are various mistakes that introduce the exposures of information. Also, the severity of the attack ranges widely, as it depends on the product's context, the type of information, and the benefit the attacker receives from the stolen information.

7.2 SSL_shutdown Behavior Vulnerability

CVE-2019-1559 comes from different behaviors after a fatal alert. The application will call the SSL_shutdown() function twice, where the first time is to send close_notify, and the second is to receive the notification. Depending on the situation, the software will illicit different responses. In one situation, the byte record acquired has an illogical padding. An in another, the byte record acquired has an illogical MAC. An attacker using a padding oracle can decrypt data. [14]

To fix the bug, the application must call SSL_shutdown() twice so that for any part of the protocol, if an error occurred, it can be ensured that the application is not vulnerable to a padding oracle. Originally, if the input is a fatal one, it would be removed from the cache. [15]

```
-     if ((level == 2) && (s->session != NULL))
-         SSL_CTX_remove_session(s->session_ctx, s->session);
```

Now if a fatal one exists, we remove it from cache and push code to go into the error state so that every kind of invalid input will behave the same.

```
+     if (level == SSL3_AL_FATAL) {
+         if (s->session != NULL)
+             SSL_CTX_remove_session(s->session_ctx, s->session);
+         s->state = SSL_ST_ERR;
+     }
```

8 Memory Corruption Vulnerability

8.1 Definition

In the recent decade, eleven memory corruption vulnerabilities have been discovered. Most of the vulnerabilities are defined in CWE-199. This kind of weakness happens when a software can perform its operations like read or write from outside of the memory buffer that was intended to be operated on.

This vulnerability deals with the consequences of exceeding the memory buffer's boundaries. Whenever something other than the intended copies sequential data from designated beginning points, issues may arise. Some problems like incorrect pointer arithmetic or invalid pointers retrieval would be result forms of inappropriate memory initialization.

Problems of memory corruption are always accompanied with DoS and overflow vulnerabilities. This is largely because memory corruption vulnerability is a consequence of DoS and buffer overflow vulnerabilities.

8.2 Negative Zero Vulnerability

CVE-2016-2108 is listed as high severity. In past renditions of OpenSSL, ASN.1 encodes zeros as negative integers. In i2c_ASN1_INTEGER when an out-of-bounds action is executed, the buffer can underflow. In most cases, because "negative zeroes" do not exist for ASN.1 parser, exploitation of this bug cannot be performed. [12]

Overtime, it was discovered that a large universal tag can be misinterpreted by the ASN.1 parser as a negative zero, despite not being present in ASN.1 structures like x509. However, ANY structure can contain large universal tags.

For an attacker to prompt an out-of-bounds write and thus memory corruption, the application must deserialize and re-serialize an ANY field encompassed in an ASN.1 structure

that is untrusted.

To fix the bug where i2c_ASN1_INTEGER mishandles zero if it is marked as negative, the following codes is added in the version(1.0.2). Since ret is the length of a and i is the first data index of a, we can use the judge sentence to tell where the data is true zero or something else. Only the real zero with length 1 and value 0 can be regarded as 0. [13]

```
        else {
         ret = a->length;
         i = a->data[0];

+    if (ret == 1 && i == 0)
+            neg = 0;
```

The following change allows the code to copy zeros to a destination as long as its source is zero, even if it is masked as a negative.

```
    p += a->length - 1;
        i = a->length;

-    while (!*n) {

+    while (!*n && i > 1) {

        *(p--) = 0;
        n--;
        i--;
```

9 Heartbleed Security Vulnerability

9.1 Background

Discovered in April 2014 by members from OpenSSL and Google's security team[16], the Heartbleed is the most severe and famous bug seen in OpenSSL. It is a security bug in the implementation process (OpenSSL cryptography library) of the transport layer heartbeat extension security protocols .[17] It is important to note that this is a problem of implementation and not an innate flaw in the design of OpenSSL.

Function of RFC6520

To have a better understanding of Heartbleed, we first must take a look at the function of the heartbeat extension (RFC6520).

The reason why RFC5620 is called a heartbeat extension because the custom structure will be sent to the other end in the connection and verifies whether the other end is still functional by receiving a replication structure from it. Ssince some of the firewall or other mechanism will cut off the connection with no data interaction for a certain period, the heartbeat structure can ensure the existence of the connection by transforming relatively small sizes of data.

Hence, to summarize, the function of RFC6520 is to ensure long-time connections and judge disconnections.

Problem in RFC6520

We already know that the heartbeat structure needs the two ends of the connection, so that a heartbeat message can be sent to each other. To be more specific, let's say one of the connection is a server of huge companies like Tencent, Alibaba or Baidu. To ensure a connection exists between the user and the server, the computer sends a message with a payload and a respective length of the payload (pl). The server will reply with the first bit (t_1) of the payload it received to the t_{pl}. Ideally, $(t_1, t_2, ..., t_{pl})$ should be exactly the same as the payload sent by the user. However, the problem occurred since there is an inconsistency in the payload length between the actual and message. The user can set the length of the payload, and if the length exceeds the actual length, the receiver can have more data, even some as secret keys, from the server. Theoretically each time a message is sent, a malicious user can illegally receive (at most) 64KB data from the server. The following figure is a demonstration of this bug.

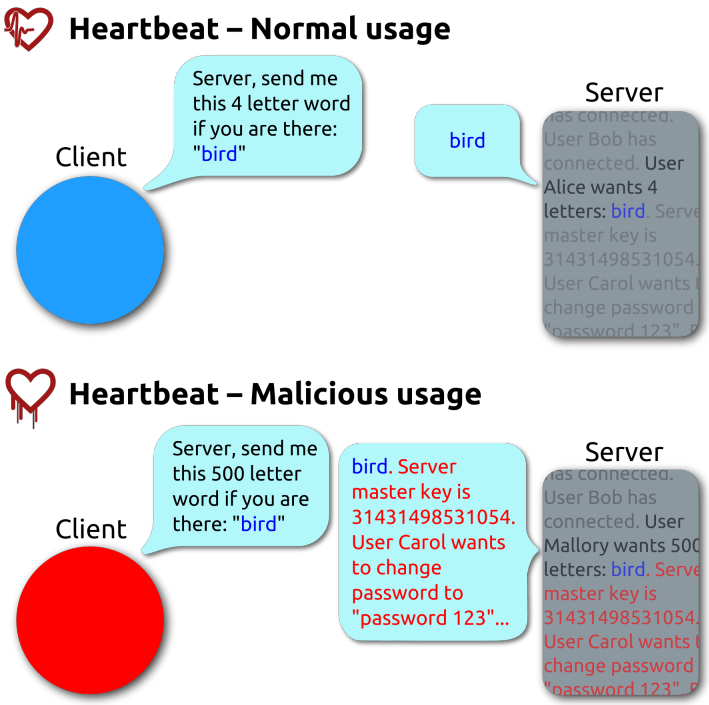

Figure 2: Demonstration of Heartbleed. [17]

We can see from the figure that the ”500 letter word” during the malicious usage message is much larger than ”bird”. Without checking the length, the server can send huge amounts of secrets back to the malicious user.

9.2 Technical Principle

To be more specific, we will examine the bug based on the code's deatil. [18]
The bug can be seen in *ssl/d1_both.c*

```
1  int dtls1_process_heartbeat(SSL *s){
2      usigned char *p = &s->s3->rrec.data[0], *pl;
3      usigned short hbtype;
4      usigned int payload;
5      unsigned int padding = 16; /* Use minimum padding */
6      ...
```

The pointer **p** in the above code is pointing to the data in the following structure.

```
1  typedef struct ssl3_record_st {
2      int type; /* type of record */
```

```
    unsigned int length; /* How many bytes available */
    unsigned int off; /* read/write offset into 'buf' */
    unsigned char *data; /* pointer to the record data */
    unsigned char *input; /* where the decode bytes are */
    unsigned char *comp; /* only used with decompression
                            - malloc()ed */
    unsigned long epoch; /* epoch number, needed by DTLS1 */
    unsigned char seq_num[8]; /* sequence number,
                                 needed by DTLS1 */
    } SSL3_RECORD;
```

Now we refer back to *dtls1_process_heartbeat*,

```
    /* Read type and payload length first */
    hbtype = *p++;
    n2s(p, payload);
    pl = p;
```

The above code simply read and then initialized the payload with *n2s*. After this, the server will give a response to the heartbeat request.

```
    unsigned char *buffer, *bp;
    int r;
    /* 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);
    bp = buffer;
    /* Enter response type, length and copy payload */
    *bp++ = TLS1_HB_RESPONSE;
    s2n(payload, bp);
    memcpy(bp, pl, payload);
```

In the last line of this code, there are *payload* length of data leaked.

9.3 The consequences of Heartbleed

A security research team from the University of Michigan used an internet scanning tool called ZMap to search the websites for Heartbleed bugs. Among the first million websites

ranked on Alexa, 32 persent of them support SSL; among all of the websites supporting HTTPS, 31.9 percent of them support OpenSSL TLS Heartbeat Extension.

Here is an example of this attack to hint its severity. On April 18th, 2014, a 19-year-old hacker was arrested by the Canadian police. He used this bug to attack the Canada Revenue Agency (CRA) website and illegally obtained 900 Canadian citizens' tax data.

Moreover, since many famous websites like Taobao, Yahoo, EaseNet and so on used OpenSSL, the bug can also impact them. Even though Taobao fixed the problem before April 8th, 2014, there still may be some data leaked.

Hence, the Heartbleed is a disastrous security vulnerability. In the next chapter, we will introduce some methods taken to handle this problem.

9.4 Treatment of Heartbleed

1. The OpenSSL officially fixed this bug on April 7th, 2014. According to the *OpenSSL Security Advisory [07 Apr 2014]*[19], they added a check on the boundaries during the TLS heartbeat extension process.

2. Technicians of the server can upgrade their OpenSSL or they can disable the heartbeat extension by using **-DOPENSSL_NO_HEARTBEATS**.

3. To check if Heartbleed exists locally on your computer or server, there are now many codes for testing available on the internet. The code provided is written by Derek Callaway. [20]

   The following is the results of testing:

   - **Sample testing result on a safe server:**

```
C:\Users\91381\AppData\Roaming\SPB_16.6>cd pacemaker

C:\Users\91381\AppData\Roaming\SPB_16.6\pacemaker>python pacemaker.py
Listening on :4433 for tls clients
Connection from: 127.0.0.1:55106
Possibly not vulnerable

Connection from: 127.0.0.1:55107
Possibly not vulnerable
```

- **Sample testing result on a vulnerable server[20]:**

```
Connection from: 127.0.0.1:40738
Client returned 65535 (0xffff) bytes
0000: 18 03 03 40 00 02 ff ff 2d 03 03 52 34 c6 6d 86  ...@....-..R4.m.
0010: 8d e8 40 97 da ee 7e 21 c4 1d 2e 9f e9 60 5f 05  ..@...~!.....`_.
0020: b0 ce af 7e b7 95 8c 33 42 3f d5 00 c0 30 00 00  ...~...3B?...0..
0030: 05 00 0f 00 01 01 00 00 00 00 00 00 00 00 00 00  ................
0040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
*
4000: 00 00 00 00 00 18 03 03 40 00 00 00 00 00 00 00  ........@.......
8000: 00 00 00 00 00 00 00 00 00 00 18 03 03 40 00 00  .............@..
...
e440: 1d 2e 9f e9 60 5f 05 b0 ce af 7e b7 95 8c 33 42  ....`_....~...3B
e450: 3f d5 00 c0 30 00 00 05 00 0f 00 01 01 00 00 00  ?...0...........
fff0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

10 Conclusion and Discussion

We have studied a few different types of vulnerabilities. Authentication bypass vulnerability exists due to vulnerability in client or server, whether its a weak authentication or unauthorized use of session identifications. Attacks that occur aim to exploit weakness in materials, but can be prevented with alterations to the handshake messages. Denial-of-service (DoS) vulnerability is one of the most frequent vulnerabilities occurring for OpenSSL, notably due to its respective simplistic attack style. From our research, DoS for OpenSSL largely occurs due to incorrect or mismatched (memory) addresses. To prevent attacks on this vulnerability, it is important to prevent leaks. Remote code execution vulnerability is an incredibly dangerous bug, as the attacker can freely take control of another computer and can perform actions such as remotely installing viruses. Following its name, attacks occur during the execution of code like the dangling pointer example we analyzed in our report. Buffer overflow vulnerability is another notable vulnerability with DoS. The vulnerability occurs when any type of overflow of buffer (past or present) exists, as writing outside the boundary of the buffer could corrupt the data allowing the program to crash or in worst case, execute malicious codes. To prevent buffer overflow vulnerability, proper tracing of memory locations to prevent leaks are needed. Gain information vulnerability transpires when sensitive information is exposed without authorization of accesses. Undoubtedly, the vulnerable could pose serious dangers given the type of information revealed. We studied the vulnerability that occurred due to the calling of the SSLshutdown() function. To prevent

exploitation of the padding oracle, calling the function twice and implementing a way to catch the error state can fix the issue. Memory corruption vulnerability has only occurred more recently. It occurs when software is executing operations on data outside of the memory buffer. Given its similarities, when memory corruption vulnerability occurs, DoS and buffer overflow vulnerabilities are often found to be in company. Its prevention method is comparable as well, focusing on reformulating the memory to prevent errors. Standing, the Heartbleed security vulnerability is the most severe and famous bug for OpenSSL. The threat lies in its resiliency, as even without connection to data interaction, the heartbeat structure can hold allowing the attacker's end to replicate a custom structure of the user's data. In 2014, OpenSSL fixed the bug by the addition of bounds to prevent heartbeat extensions. In addition, due to its regard, various codes are now available on the internet to check for Heartbleed in your own computer.

Through the research and analysis of the tool kit's recent vulnerabilities, we as students are able to build a deeper knowledge of cryptography and utilize knowledge we've learned from class.

References

[1] OpenSSL continues to bleed out more flaws – more critical vulnerabilities found". Cyberoam Threat Research Labs. 2014. Archived from the original on 2014-06-19. Retrieved 2020-07-21.

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0224

[3] "OpenSSL Updates Fix Critical Security Vulnerabilities [09 Aug 2014]". Retrieved 25 Aug 2014.

[4] "OpenSSL Patches Severe Denial-of-Service Vulnerability [20 March 2015]". Brandon Stosh. 2015.

[5] https://holmesian.org/OCSP-Stapling

[6] https://github.com/openssl/openssl/commit/34e3edbf3a10953cb407288101fd56a629af22f9

[7] https://github.com/openssl/openssl/commit/acacbfa7565c78d2273c0b2a2e5e803f44afefeb

[8] https://www.cvedetails.com/vendor/217/Openssl.html

[9] https://cwe.mitre.org/data/definitions/119.html

[10] https://cwe.mitre.org/data/definitions/200.html

[11] https://owasp.org/www-community/vulnerabilities/Buffer_Overflow

[12] https://www.openssl.org/news/secadv/20160503.txt

[13] https://github.com/openssl/openssl/commit/3661bb4e7934668bd99ca777ea8b30eedfafa871

[14] https://www.openssl.org/news/secadv/20190226.txt

[15] https://github.com/openssl/openssl/commit/e9bbefbf0f24c57645e7ad6a5a71ae649d18ac8e

[16] https://plus.google.com/+MarkJCox/posts/TmCbp3BhJma

[17] https://en.wikipedia.org/wiki/Heartbleedcite_note-24

[18] https://www.zhihu.com/search?type=contentq=heartbleed

[19] https://www.openssl.org/news/secadv/20140407.txt

[20] https://github.com/Lekensteyn/pacemaker