Transport Layer Security

Chester Rebeiro
IIT Madras

Some of the slides borrowed from the book ‘Computer Security: A Hands on Approach’ by Wenliang Du
TLS: Protocol to achieve secure communication

TLS provides secure communication channel with 3 properties:

• Confidentiality
• Integrity
• Authentication

Two important components

• TLS Handshake
• Secure Data transmission
SSL vs TLS

• 1995: Netscape released SSL 2.0
• 1996: New version SSL 3.0
• 1999: TLS introduced as the new version of SSL
• 2011: SSL 2.0 deprecated by IETF
• 2015: SSL 3.0 deprecated by IETF

** Difference: Handshake protocols changes from SSL to TLS.
Encryption
• Between the network and Application layer.
  • Unprotected data is given to TLS by Application layer
  • TLS handles encryption, decryption and integrity checks
  • TLS gives protected data to Transport layer

---

**TLS in the Network Stack**

- Application Layer
  - TLS Layer
    - Transport Layer (TCP Protocol)
    - Network Layer (IP protocol)
    - Data Link Layer
    - Physical Layer
TLS Handshake

• Before a client and server can communicate securely, several things need to be set up first:
  • Encryption algorithm and key
  • MAC algorithm
  • Algorithm for key exchange

• These cryptographic parameters need to be agreed upon by the client and server
TLS Handshake

- **Client Hello**: (cipher suites, client random)
- **Server Hello**: (decisions on cipher suites, client random)
- **Server Certificate**
- **Server Hello Done**
- **Client Key Exchange**: (encrypted pre-master secret)
- **Change Cipher Spec**
- **Finished**: (message encrypted with session key)
Network Traffics During TLS Handshake

Since TLS runs top of TCP, a TCP connection needs to be established before the handshake protocol. This is how the packet exchange looks between a client and server during a TLS handshake protocol captured using Wireshark:

<table>
<thead>
<tr>
<th>No.</th>
<th>Source</th>
<th>Destination</th>
<th>Protocol</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10.0.2.45</td>
<td>10.0.2.35</td>
<td>TCP</td>
<td>59930 → 11110 [SYN] Seq=0 Win=14600 Len=0 MSS=1460...</td>
</tr>
<tr>
<td>2</td>
<td>10.0.2.35</td>
<td>10.0.2.45</td>
<td>TCP</td>
<td>11110 → 59930 [SYN, ACK] Seq=0 Ack=1 Win=14480...</td>
</tr>
<tr>
<td>3</td>
<td>10.0.2.45</td>
<td>10.0.2.35</td>
<td>TCP</td>
<td>59930 → 11110 [ACK] Seq=1 Ack=1 Win=14720 Len=0...</td>
</tr>
<tr>
<td>4</td>
<td>10.0.2.45</td>
<td>10.0.2.35</td>
<td>TLSv1.2</td>
<td>Client Hello</td>
</tr>
<tr>
<td>6</td>
<td>10.0.2.35</td>
<td>10.0.2.45</td>
<td>TLSv1.2</td>
<td>Server Hello, Certificate, Server Hello Done</td>
</tr>
<tr>
<td>8</td>
<td>10.0.2.45</td>
<td>10.0.2.35</td>
<td>TLSv1.2</td>
<td>Client Key Exchange, Change Cipher Spec, Finished</td>
</tr>
<tr>
<td>9</td>
<td>10.0.2.35</td>
<td>10.0.2.45</td>
<td>TLSv1.2</td>
<td>New Session Ticket, Change Cipher Spec, Finished</td>
</tr>
</tbody>
</table>
TLS Handshake: Client Hello

- Client Hello
  (cipher suites, client random)
- Server Hello
  (decisions on cipher suites, client random)
- Server Certificate
- Server Hello Done
- Client Key Exchange
  (encrypted pre-master secret)
- Change Cipher Spec
- Finished
  (message encrypted with session key)
Client sends Cipher Options

• A list of ciphers for data encryption and hashing
• Supported Groups:

```c
enum {
    /* Elliptic Curve Groups (ECDHE) */
    secp256r1(0x0017), secp384r1(0x0018), secp521r1(0x0019),
    x25519(0x001D), x448(0x001E),

    /* Finite Field Groups (DHE) */
    ffdhe2048(0x0100), ffdhe3072(0x0101), ffdhe4096(0x0102),
    ffdhe6144(0x0103), ffdhe8192(0x0104),

    /* Reserved Code Points */
    ffdhe_private_use(0x01FC..0x01FF),
    ecdhe_private_use(0x0FE0..0xFEFF),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<2..2^16-1>;
} NamedGroupList;
```
• A list of ciphers for data encryption

• Supported Groups

• Signature Algorithms:
  list of signature algorithms the client is ready to verify

```c
enum {
    /* RSASSA-PKCS1-v1_5 algorithms */
    rsa_pkcs1_sha256(0x0401),
    rsa_pkcs1_sha384(0x0501),
    rsa_pkcs1_sha512(0x0601),

    /* ECDSA algorithms */
    ecdsa_secp256r1_sha256(0x0403),
    ecdsa_secp384r1_sha384(0x0503),
    ecdsa_secp521r1_sha512(0x0603),

    /* RSASSA-PSS algorithms with public key OID rsaEncryption */
    rsa_pss_rsaes_sha256(0x0804),
    rsa_pss_rsaes_sha384(0x0805),
    rsa_pss_rsaes_sha512(0x0806),

    /* EdDSA algorithms */
    ed25519(0x0807),
    ed448(0x0808),

    /* RSASSA-PSS algorithms with public key OID RSASSA-PSS */
    rsa_pss_pss_sha256(0x0809),
    rsa_pss_pss_sha384(0x080a),
    rsa_pss_pss_sha512(0x080b),

    /* Legacy algorithms */
    rsa_pkcs1_sha1(0x0201),
    ecdsa_sha1(0x0203),

    /* Reserved Code Points */
    private_use(0xFE00..0xFFFF),
    (0xFFFF)
} SignatureScheme;
```

```c
struct {
    SignatureScheme supported_signature_algorithms<2..2^16-2>;
} SignatureSchemeList;
```
Pre-Shared Keys

• A list of ciphers for data encryption
• Supported Groups
• Signature Algorithms:
  list of signature algorithms the client is ready to verify
• Pre Shared Key Extensions
  list of key identities known to the client and a psk_key_exchange_mode

```c
struct {
    opaque identity<1..2^16-1>;
    uint32 obfuscated_ticket_age;
} PskIdentity;

opaque PskBinderEntry<32..255>;

struct {
    PskIdentity identities<7..2^16-1>;
    PskBinderEntry binders<33..2^16-1>;
} OfferedPsks;

struct {
    select (Handshake.msg_type) {
        case client_hello: OfferedPsks;
        case server_hello: uint16 selected_identity;
    }
} PreSharedKeyExtension;
```
Server Hello

Client Hello  
(cipher suites, client random)

Server Hello  
(decisions on cipher suites, client random)

Server Certificate

Server Hello Done

Client Key Exchange  
(encrypted pre-master secret)

Change Cipher Spec

Finished  
(message encrypted with session key)
Server Hello

struct {
    ProtocolVersion legacy_version = 0x0303;
    Random random;
    opaque legacy_session_id_echo<0..32>;
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>;
} ServerHello;

32 byte random number

Selected cipher suite
TLS Handshake

- **Client Hello** (cipher suites, client random)
- **Server Hello** (decisions on cipher suites, client random)
- **Server Certificate**
- **Server Hello Done**
- **Client Key Exchange** (encrypted pre-master secret)
- **Change Cipher Spec**
- **Finished** (message encrypted with session key)
Key Generation and Exchange

1. Pre-master Secret
   • After server’s certificate is verified, client generates random number, called **pre-master secret**.
   • Client encrypts pre-master secret using server’s public key and sends to the server.
   • Length of the key depends on the public key algorithm used.

2. Master Secret
   • Uses `client_random` and `server_random` (as nonce); along with the pre-master secret, generates a master secret key.
   • Master secret, is fixed length 48 bytes long

3. Session Keys
   • Master key used to generates 4 different session keys: `client_write_MAC_key`; `client_write_key`; `server_write_MAC_key`; `server_write_key`
     (each is of 32 bytes)
   MAC keys used for integrity; others are used for data encryption; each direction (client to server and server to client has a different key)
TLS Data Transmission

Record Format

- Content Type
- Version
- Length
- Data (may be compressed)
- MAC
- Padding

Type of protocol:
- 0x14 ChangeCipherSpec
- 0x15 Alert
- 0x16 Handshake
- 0x17 Application
- 0x18 Heartbeat

Type of protocol:
- 0x300 SSL 3.0
- 0x301 TLS 1.0
- 0x302 TLS 1.1
- 0x303 TLS 1.2
- TLS 1.3

<2^{14}
Sending Data with TLS Record Protocol

1. Application Layer
   - Get data from application

2. TLS Layer
   - Fragment
   - Compress (optional)
   - Add MAC and padding
   - Encrypt
   - Add TLS header

3. Transport Layer (TCP Stream)
   - TLS Header
   - Encrypted Data
   - TLS Header
   - Encrypted Data

SSL_write() function

write() function
Receiving Data with TLS Record Protocol
TLS Programming: Overall Picture

TLS Client
TLS context setup

connect() → TCP handshake

SSL_set_fd() → SSL_connect()

TLS handshake

SSL_read/SSL_write → TLS data transmission

TLS Server
TLS context setup

accept() → SSL_set_fd()

SSL_accept() → SSL_read/SSL_write
TLS Client Program: TLS Initialization

- TLS protocol is a stateful protocol
- Create a context data structure
- Create a SSL structure to hold state information

---

```c
// Step 1: SSL context initialization
SSL_METHOD *meth = (SSL_METHOD *)TLSv1_2_method();
SSL_CTX* ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_load_verify_locations(ctx, NULL, "./cert"),

// Step 2: Create a new SSL structure for a connection
SSL* ssl = SSL_new(ctx);
```
TLS Client Program: TLS Initialization (cont’d)

Should verify server’s certificate

Folder containing trusted CA’ certificates, such as root CA’s certificates.

Check whether the certificate’s subject field matches with hostname.

```
// Step 1: SSL context initialization
SSL_METHOD *meth = (SSL_METHOD *)TLSv1_2_method();
SSL_CTX* ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, NULL);
SSL_CTX_load_verify_locations(ctx, NULL, "./cert");

// Step 2: Create a new SSL structure for a connection
SSL* ssl = SSL_new (ctx);

// Step 3: Enable the hostname check
X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);
```
TLS Client Program: Set Up a TCP Connection

- TLS is primarily built on top of TCP.
- This part is standard.

```c
int setupTCPClient(const char* hostname, int port)
{
    struct sockaddr_in server_addr;

    // Get the IP address from hostname
    struct hostent* hp = gethostbyname(hostname);

    // Create a TCP socket
    int sockfd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    // Fill in the destination information (IP, port #, and family)
    memset(&server_addr, '\0', sizeof(server_addr));
    memcpy(&(server_addr.sin_addr.s_addr), hp->h_addr, hp->h_length);
    server_addr.sin_port = htons(port);
    server_addr.sin_family = AF_INET;

    // Connect to the destination
    connect(sockfd, (struct sockaddr*) &server_addr,
            sizeof(server_addr));

    return sockfd;
}
```
TLS Client Program: Initiate TLS Handshake

Establish the SSL session on top of an established TCP connection

```
SSL* ssl = setupTLSClient(hostname);
int sockfd = setupTCPClient(hostname, port);
SSL_set_fd(ssl, sockfd);
int err = SSL_connect(ssl);
```

Initiate the TLS Handshake protocol
TLS Client Program: Send/Receive Data

• We construct a simple HTTP GET request, and print out the reply from the web server.

```c
char buf[9000];
char sendBuf[200];
sprintf(sendBuf, "GET / HTTP/1.1\nHost: %s\n\n", hostname);
SSL_write(ssl, sendBuf, strlen(sendBuf));

int len;
do {
    len = SSL_read(ssl, buf, sizeof(buf) - 1);
    buf[len] = '\0';
    printf("%s\n", buf);
} while (len > 0);
```
TLS Server Program

Create a simple HTTPS server
TLS Server Program: Setup

// Step 1: SSL context initialization
meth = (SSL_METHOD *)&amp;TLSv1_2_method();
ctx = SSL_CTX_new(meth);
SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, NULL);

// Step 2: Set up the server certificate and private key
SSL_CTX_use_certificate_file(ctx, "./bank_cert.pem",
    SSL_FILETYPE_PEM);
/* SSL_CTX_use_certificate_chain_file(ctx,
   "./bank_chain_cert.pem"); */
SSL_CTX_use_PrivateKey_file(ctx, "./bank_key.pem",
    SSL_FILETYPE_PEM);

// Step 3: Create a new SSL structure for a connection
ssl = SSL_new (ctx);
This program creates a TCP socket, binds it to a TCP port (4433) and marks the socket as a passive socket. This is quite standard.

```c
int setupTCPServer()
{
    struct sockaddr_in sa_server;
    int listen_sock;

    listen_sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
    memset (&sa_server, '\0', sizeof(sa_server));
    sa_server.sin_family = AF_INET;
    sa_server.sin_addr.s_addr = INADDR_ANY;
    sa_server.sin_port = htons (4433);
    bind(listen_sock, (struct sockaddr*)&sa_server, sizeof(sa_server));
    listen(listen_sock, 5);
    return listen_sock;
}
```
TLS Server: Handshake & Data Communication

Conduct TLS handshake with the client

We can now use this established SSL session to conduct data communication

```
while (1) {
    int sock = accept(listen_sock, (struct sockaddr*)&sa_client, &client_len);
    if (fork() == 0) { // The child process
        close(listen_sock);
        SSL_set_fd(ssl, sock);
        int err = SSL_accept(ssl);
        CHK_SSL(err);
        printf("SSL connection established!\n");
        processRequest(ssl, sock);
        close(socket);
        return 0;
    } else { // The parent process
        close(sock);
    }
}
```
TLS Server Program: Data Transmission

- Logic for sending/receiving data is the same as the client program.
- We simply send an HTTP reply message back to the client.

```c
void processRequest(SSL* ssl, int sock)
{
    char buf[1024];
    int len = SSL_read (ssl, buf, sizeof(buf) - 1);
    buf[len] = '\0';
    printf("Received: %s\n",buf);

    // Construct and send the HTML page
    char *html = "... (omitted) ...");
    SSL_write(ssl, html, strlen(html));
    SSL_shutdown(ssl); SSL_free(ssl);
}
```
Padding Attack
Data Encryption (CBC mode)
CBC Mode Decryption

\[ \begin{align*} c_0 &\xrightarrow{d_K} p_0 \\
 c_1 &\xrightarrow{d_K} p_1 \\
 c_2 &\xrightarrow{d_K} p_2 \\
 c_3 &\xrightarrow{d_K} p_3 \\
 c_4 &\xrightarrow{d_K} p_4 \end{align*} \]
Recollect TLS Data Encryption

Application Data

Data Block 1

Data Block 2

Compressed

Compressed

MAC

Pad

Encrypted

Critical point: Pad is not protected by MAC (thus an attacker can modify the Pad, without being detected)
Receiver Checks (older TLS versions)

TEST(PAD)
Look at the last byte (pad length)
If it is 0x05, then the previous 5 bytes should contain 0x05.

Two different errors signaled
Padding Attack

Chosen Cipher text attack
Padding Attack

Let's try to decrypt $i_1^B$.
Attacker changes LSByte of $c_0$ to (say xx) and sends the modified ciphertext to the server.

$P_1^B = xx \oplus i_1^B$

(if $P_1^B$ holds a valid pad ( = 0x00), then pad test will pass
if $P_1^B$ holds an invalid pad ( ≠ 0x00), then pad test will fail)

There are 256 possible values of xx.
Vary the values of xx until, pad test passes.
Padding Attack

Let's try to decrypt $i_{1_{B-1}}$
Attacker changes LSByte of c0 to (say xx) and sends the modified ciphertext to the server.

Set xx such that, $P_{1_B} = 0x01$
$P_{1_{B-1}} = yy \oplus i_{1_{B-1}}$

(if $P_{1_{B-1}}$ holds an valid pad (= 01), then pad test will pass
if $P_{1_B}$ holds an invalid pad (≠ 01), then pad test will fail quite likely, the MAC test will fail, in this case)

There are 256 possible values of xx.
Vary the values of xx until, pad test passes.
Padding Attack in Practice

- Won’t work in all places
- When TLS detects a padding or MAC error, it renegotiates the key
- Certain scenarios where it will work
  - IMAP over TLS
    Every 5mins, IMAP will send the same encrypted string comprising of USERNAME and PASSWORD to the email server.
    
    Even with the key changes, the attacker would need at most $256 \times 8 \times 5$ minutes to capture the entire 8 byte (ASCII) password
  
  - Datagram TLS
Receiver Checks Modified

- Encrypted
  - decrypt
  - Compressed | MAC | Pad
    - TEST(PAD)
      - FAILED
      - TEST(MAC)
        - FAILED
        - Compressed
          - uncompressed
          - Data Block 1
    - Signal Error: Check failed
    - Signal Error: check failed

Same Error (so an attacker cannot distinguish between a PAD or MAC error)
Timing Attacks

Receiver Checks (Modification 2)

Encrypted

- decrypt

Compressed

- TEST(PAD)
  - PASSED / FAILED

MAC

- TEST(MAC)
  - PASSED
  - FAILED

Pad

- Signal Error: check failed

Data Block 1

- Compressed
  - uncompressed

Always do a MAC test. If PAD test failed, then assume 0 PAD and compute MAC.
Receiver Checks Modification 2

If PAD test fails, the server cannot identify the length of the PAD. Assuming 0 PAD, would imply that the data would be larger; hence, MAC computation would take longer.

This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.

Helps reduce attack surface ….. But not much 😐
Poodle Attack

(Padding Oracle Downgraded Legacy Encryption)
Recollect Client Hello

- A list of ciphers for data encryption and hashing
- Supported Groups
- Signature Algorithms: list of signature algorithms the client is ready to verify
- Pre Shared Key Extensions list of key identities known to the client and a psk_key_exchange_mode
Man in the Middle

Padding Oracle On Downgraded Legacy Encryption (POODLE) attack

Client:
- Hello. Do you support TLS 1.2? NO
- Do you support TLS 1.1? NO
- Do you support TLS 1.0? NO
- Do you support SSL 3.0? YES

Server:

Man In The Middle (Attacker)
Beast Attack (Man in the Middle)

Force Alice to execute something (for example using Javascript)

Sniff encrypted traffic

encrypted traffic
IVs used in CBC mode

Holds a password

Attacker can control

\[ e_K \]

\[ c_0 \]

\[ c_1 \]

\[ c_2 \]

\[ e_K \]

\[ p_0 \]

\[ p_1 \]

\[ p_2 \]

\[ q_0 \]

\[ q_1 \]

\[ d_0 \]

\[ d_1 \]
IVs used in CBC mode

$q_0 = c_2 \oplus c_0 \oplus p_1$
then
$c_1 = d_0$

Attacker knows $c_0, d_2$
Can control $q_0$

Needs to know $p_1$ ... this is not easy. 8 bytes $2^{64}$ possibilities
# Get and Post HTTP Requests

**GET /index.html HTTP/1.1\r\nHost: www.somesite.com\r\nCookie: Session=12345678**

- Constant HTTP Get request
- Secret information

---

## CBC Encryption with DES

<table>
<thead>
<tr>
<th>GET /index.html</th>
<th>ex.html</th>
<th>HTTP/1.1</th>
<th>\r\nHost</th>
<th>\r\nmysite</th>
<th>.com\r\n</th>
<th>Cookie:</th>
<th>Session: 12345678</th>
</tr>
</thead>
<tbody>
<tr>
<td>a7d25abbd67b2dbf</td>
<td>e2ade7246ea5ed5</td>
<td>e99063ebe430b75b</td>
<td>746ae5eca362bc3</td>
<td>f6f62f99a076056f</td>
<td>6b14704973a779ae</td>
<td>7fa9300d4e490cbb</td>
<td>b6040b9542a59ad5</td>
</tr>
</tbody>
</table>

http://commandlinefanatic.com/cgi-bin/showarticle.cgi?article=art027
Splitting Attack

GET /index.jsp HTTP/1.1
Host: mysite.com
Cookie: Session=12345678

One byte is unknown (256 guesses)

GET /index.js HTTP/1.1
Host: mysite.com
Cookie: Session=12345678

Rizzo and Duong exploited a security hole in the Java Applet of their browser (which has since been patched) to make this work
Taming the Beast

• Use a stream cipher such as RC4 (works)

• Use other modes of operation, like GCM mode

• Prepend each record with an empty plaintext fragment, just containing the MAC and padding.
  • The IV used for live data will not be known in advance
  • Buggy web-browsers and servers won’t tolerate empty packets

• Compression
CRIME

• The Deflate Compression Scheme
Force Alice to execute something (for example using Javascript)

Sniff encrypted traffic

Assumption: The Javascript program can inject known messages in the active TLS connection between the client and server
The Attack

• Consider that the following cookie is sent from client to server:
  \( \text{Cookie secret} = 345678 \)

• The attacker knows the session token “\( \text{Cookie secret} = \)“ and wants to obtain the secret token.

• The javascript, will start to inject strings as follows in the communication between Alice and Bob: “\( \text{Cookie secret} = a \)“
  • The attacker notes the size of the transmitted packet
  • If the size reduces, then, the guess is correct
Buffer Overread Example

```c
char some_data[] = "some data";
char secret_data[] = "TOPSECRET";

void main(int argc, char **argv)
{
    int i=0;
    int len = atoi(argv[1]);  // the length to be printed
    printf("%08x %08x %d\n", secret_data, some_data, (secret_data - some_data));
    while(i < len){
        printf("%c", some_data[i], some_data[i]);
        i++;
    }
    printf("\n");
}
```
Buffer Overread Example

```c
char some_data[] = "some data";
char secret_data[] = "TOPSECRET";

void main(int argc, char **argv)
{
    int i=0;
    int len = atoi(argv[1]); // the length to be printed
    printf("%08x %08x %d\n", secret_data, some_data, (secret_data - some_data));
    while(i < len){
        printf("%c", some_data[i], some_data[i]);
        i++;
    }
    printf("\n");
}
```

len used to specify how much needs to be read.
Can lead to an overread

```
chester@ahalya:~/sse/overread$ ./a.out 22
080496d2 080496c8 10
some dataTOPSECRET
```
Heartbleed: A buffer overread malware

- 2012 – 2014
  - Introduced in 2012; disclosed in 2014
- CVE-2014-0160
- Target: OpenSSL implementation of TLS – transport layer security
  - TLS defines crypto-protocols for secure communication
  - Used in applications such as email, web-browsing, VoIP, instant messaging,
  - Provide privacy and data integrity

https://www.theregister.co.uk/2014/04/09/heartbleed_explained/
Heartbeat

- A component of TLS that provides a means to keep alive secure communication links
  - This avoids closure of connections due to some firewalls
  - Also ensures that the peer is still alive
Heartbeat

- Client sends a heart beat message with some payload
- Server replies with the same payload to signal that everything is OK

<table>
<thead>
<tr>
<th>Heartbeat Message</th>
<th>type</th>
<th>length</th>
<th>payload</th>
<th>padding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello World; 12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SSL3 struct and Heartbeat

- Heartbeat message arrives via an SSL3 structure, which is defined as follows

```c
struct ssl3_record_st
{
    unsigned int D_length;    /* How many bytes available */
    [...]                     /* How many bytes available */
    unsigned char *data;    /* pointer to the record data */
    [...]                     /* How many bytes available */
} SSL3_RECORD;
```

length : length of the heartbeat message

data : pointer to the entire heartbeat message

<table>
<thead>
<tr>
<th>type</th>
<th>Length (pl)</th>
<th>payload</th>
</tr>
</thead>
</table>

Format of data (Heartbeat Message)
Payload and Heartbeat length

<table>
<thead>
<tr>
<th>type</th>
<th>Length (pl)</th>
<th>payload</th>
</tr>
</thead>
</table>

- **payload_length**: controlled by the heartbeat message creator
  - Can never be larger than D_length
  - However, this check was never done!!!
    - Thus allowing the heartbeat message creator to place some arbitrary large number in the payload_length
    - Resulting in overread
# Overread Example

**Heartbeat sent to victim**

**SSLv3 record:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Length</strong></td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

**HeartbeatMessage:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Payload data</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS1_HB_REQUEST</td>
<td>65535 bytes</td>
<td>1 byte</td>
</tr>
</tbody>
</table>

**Attacker** sends a heartbeat message with a single byte payload to the server. However, the pl_length is set to 65535 (the max permissible pl_length).

**Victim’s response**

**SSLv3 record:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Length</strong></td>
<td>65538 bytes</td>
</tr>
</tbody>
</table>

**HeartbeatMessage:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Length</th>
<th>Payload data</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLS1_HB_RESPONSE</td>
<td>65535 bytes</td>
<td>65535 bytes</td>
</tr>
</tbody>
</table>

**Victim ignores the SSL3 length (of 4 bytes),**

Looks only at the pl_length and returns a payload of 65535 bytes. In the payload, only 1 byte is victim’s data remaining 65534 from its own memory space.
`int tls1_process_heartbeat(SSL *s)`

```c
    unsigned char *p = &s->s3->rrec.data[0], *pl;
    unsigned short hbtpe;
    unsigned int padding;
    unsigned int padding = 16; /* Use minimum padding */

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

    if (s->msg_callback)
        s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
             &s->s3->rrec.data[0], s->s3->rrec.length,
             s, s->msg_callback_arg);

    if (hbtpe == TLS1_HB_REQUEST)
        {        // Allocate memory for the response, size is 1 bytes
            unsigned char *buffer, *bp;
            int r;

            /* 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, p, pl);
            bp += payload;
            /* Random padding */
            RAND_pseudo_bytes(bp, padding);
```
Broken OpenSSL code@victim

```c
/* 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);

r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
if (r >= 0 && s->msg_callback)
    s->msg_callback(1, s->version, TLS1_RT_HEARTBEAT,
                    buffer, 3 + payload + padding,
                    s, s->msg_callback_arg);

OPENSSL_free(buffer);
```

Add padding and send the response heartbeat message back to the attacker
65534 byte return payload may contain sensitive data

Further, invocations of similar false heartbeat will result in another 64KB of the heap to be read.
In this way, the attacker can scrape through the victim’s heap.
The patch in OpenSSL

```c
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

Discard the heartbeat response if it happens to be greater than the length in the SSL3 structure (i.e. D_length)