#### 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

#### TLS in the Network Stack

- 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

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**



#### 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:

	No.	Source	Destination	Protocol	Info
тср	1	10.0.2.45	10.0.2.35	TCP	59930 -> 11110 [SYN] Seq=0 Win=14600 Len=0 MSS=1460
ostablishmont	2	10.0.2.35	10.0.2.45	TCP	11110 -> 59930 [SYN, ACK] Seq=0 Ack=1 Win=14480
establistittetit	3	10.0.2.45	10.0.2.35	TCP	59930 -> 11110 [ACK] Seq=1 Ack=1 Win=14720 Len=0
	4	10.0.2.45	10.0.2.35	TLSv1.2	Client Hello
TLS	6	10.0.2.35	10.0.2.45	TLSv1.2	Server Hello, Certificate, Server Hello Done
handshake	8	10.0.2.45	10.0.2.35	TLSv1.2	Client Key Exchange, Change Cipher Spec, Finished
	9	10.0.2.35	10.0.2.45	TLSv1.2	New Session Ticket, Change Cipher Spec, Finished

#### **TLS Handshake: Client Hello**



#### **Client sends Cipher Options**

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

```
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(0xFE00..0xFEFF),
    (OxFFFF)
} NamedGroup;
struct {
    NamedGroup named group list<2..2<sup>16-1</sup>;
} NamedGroupList;
```

- A list of ciphers for data encryptic
- Supported Groups
- Signature Algorithms: list of signature algorithms the client is ready to verify

```
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_rsae_sha256(0x0804),
    rsa pss rsae sha384(0x0805),
    rsa pss rsae 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),
    (OxFFFF)
```

```
} SignatureScheme;
```

```
struct {
```

SignatureScheme supported\_signature\_algorithms<2..2^16-2>;
} SignatureSchemeList;

#### **Pre-Shared Keys**

- A list of ciphers for data encryptior
- 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

```
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



#### Server Hello



#### **TLS Handshake**



### 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**



#### Sending Data with TLS Record Protocol



#### Receiving Data with TLS Record Protocol



#### **TLS Programming : Overall Picture**



#### **TLS Client Program: TLS Initialization**

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

SSL Context: holding SSL → configuration	<pre>// 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");</pre>
Holding SSL states	<pre>// Step 2: Create a new SSL structure for a connection SSL* ssl = SSL_new (ctx);</pre>

#### TLS Client Program: TLS Initialization (cont'd)

// 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);

Should verify server's certificate

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

// Step 3: Enable the hostname check
X509\_VERIFY\_PARAM \*vpm = SSL\_get0\_param(ssl);
X509\_VERIFY\_PARAM\_set1\_host(vpm, hostname, 0);

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

# TLS Client Program: Set Up a TCP Connection

• TLS is primarily built on top of TCP.

• This part is standard.

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;

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.

```
char buf[9000];
                   char sendBuf[200];
                   sprintf(sendBuf, "GET / HTTP/1.1\nHost: %s\n\n", hostname);
 Send data
                   SSL_write(ssl, sendBuf, strlen(sendBuf));
                   int len;
                   do {
Receive data
                       Ien = SSL_read (ssl, buf, sizeof(buf) - 1);
                        buf[len] = ' \setminus 0';
                         printf("%s\n", buf);
                     while (len > 0);
```

### **TLS Server Program**

Create a simple HTTPS server

#### TLS Server Program: Setup



#### TLS Server Program: TCP Setup

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. 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

	<pre>while (1) {     int sock = accept(listen_sock, (struct sockaddr*)&amp;sa_client,     &amp;client_len);     if (fork() == 0) { // The child process</pre>
Conduct TLS handshake	close (listen_sock);
with the client	<pre>SSL_set_fd (ssl, sock); int err = SSL_accept (ssl); </pre>
	CHK_SSL(err); printf ("SSL connection established!\n"):
We can now	princi ( bbi connección ebcabilbhea. (n ),
use this	<pre>processRequest(ssl, sock);</pre>
established SSL	close(socket); return 0;
session to	} else { // The parent process
conduct data	close(sock);
communication	}

#### **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.

```
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);
```

#### Data Encryption (CBC mode)



#### **CBC Mode Decryption**



#### **Recollect TLS Data Encryption**





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

#### Receiver Checks (older TLS versions)



Chosen Cipher text attack



23456249xx 4372458815 c0 c1 d<sub>K</sub> d, i1 IV р1<sub>в</sub> **0**q p1 Compressed MAC Pad

Lets try to decrypt  $i1_B$ Attacker changes LSByte of c0 to (say xx) and sends the modified ciphertext to the server.

 $P1_B = xx \wedge i1_B$ 

(if  $P1_B$  holds an valid pad ( = 0x00), then pad test will pass if  $P1_B$  holds an invalid pad (  $\neq$  0x00), then pad test will fail)

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



Lets try to decrypt  $i1_{B-1}$ Attacker changes LSByte of c0 to (say xx) and sends the modified ciphertext to the server.

Set xx such that,  $P1_B = 0x01$  $P1_{B-1} = yy \wedge i1_{B-1}$ 

(if  $P1_{B-1}$  holds an valid pad ( = 01), then pad test will pass if  $P1_B$  holds an invalid pad (  $\neq$  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 x 8 x 5 minutes to capture the entire 8 byte (ASCII) password

• Datagram TLS

#### **Receiver Checks Modified**



#### **Timing Attacks**



https://www.iacr.org/cryptodb/archive/2003/CRYPTO/1069/1069.pdf

#### **Receiver Checks (Modification 2)**



Always do a MAC test. If PAD test failed, then assume 0 PAD and compute MAC.

#### **Receiver Checks Modification 2**



#### 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



#### Beast Attack (Man in the Middle)



#### IVs used in CBC mode



#### IVs used in CBC mode



 $q0 = c2 \oplus c0 \oplus p1$ 

Attacker knows c0, d2 Can control q0

then

c1 = d0

Needs to know p1 ... this is not easy. 8 bytes 2<sup>64</sup> possibilities

#### Get and Post HTTP Requests



 CBC Encryption with DES

 GET /ind
 ex.html
 HTTP/1.1
 \r\nHost
 : mysite
 .com\r\n
 Cookie:
 Session=
 12345678

 a7d25abbd67b2dbf
 e2ade7246ea5ed5
 e99063ebe430b75b
 746ae5eca36e2bc3
 f6f62f99a076056f
 6b14704973a779ae
 7fa9300d4e490cba
 b6040b9542a59ad5
 f9bb888ac3763722b

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

### **Splitting Attack**



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





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: *Cookie secret = 345678*
- The attacker knows the session token "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: "Cookie secret =a"
  - The attacker notes the size of the transmitted packet
  - If the size reduces, then, the guess is correct

#### **Buffer Overread Example**

```
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");
}</pre>
```

#### **Buffer Overread Example**



chester@aahalya:~/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



#### 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

#### SSL3 struct and Heartbeat

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

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

length : length of the heartbeat message

data : pointer to the entire heartbeat message

Format of data (Heartbeat Message)

|--|

#### Payload and Heartbeat length

Heartbeat Message

type	Length (pl)	payload	
		payload length (pl)	·/
·		D_length (pl)	/

- *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:

Length

4 bytes

HeartbeatMessage:

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)

Туре	Length	Payload data
TLS1_HB_REQUEST	65535 bytes	1 byte

Victim's response	Victim ignores the SSL3 length (of 4 bytes),
SSLv3 record:	Looks only at the pl_length and returns
Length	a payload of 65535 bytes. In the payload, only
65538 bytes	1 byte is victim's data remaining 65534 from
	its own memory space.

#### HeartbeatMessage:

Туре	Length	Payload data	
TLS1_HB_RESPONSE	65535 bytes	65535 bytes	

```
tls1 process heartbeat(SSL *s)
                                                                                Broken OpenSSL
       unsigned char *p = &s->s3->rrec.data[0], *pl; <-</pre>
       unsigned short hbtype;
                                                                                        code@victim
       unsigned int payload;
       unsigned int padding = 16; /* Use minimum padding */
       /* Read type and payload length first */
       hbtype = *p++;
                                                                           p points to the attackers heart
       n2s(p, payload);
       pl = p;
                                                                           beat packet which the victim
                                                                           just received.
       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);
                                                                           get the heartbeat type;
       if (hbtype == TLS1 HB REQUEST)
                                                                           fill payload with size of payload
                unsigned char *buffer, *bp;
                                                                           (pl in our notation)
                int r;
                                                                           This is picked up from the
                /* Allocate memory for the response, size is 1 bytes
                                                                           attackers payload and contains
                 * message type, plus 2 bytes payload length, plus
                 * payload, plus padding
                                                                           65535
                 */
                buffer = OPENSSL malloc(1 + 2 + payload + padding);
                bp = buffer;
                                                                                     Allocate buffer of 3 + 65535 +
                                                                                3
                /* Enter response type, length and copy payload */
                                                                                    16 bytes
                *bp++ = TLS1 HB RESPONSE;
                s2n(payload, bp);
                memcpy(bp, pl, payload);
                                                                                    memcpy grossly overreads
                bp += payload;
                /* Random padding */
                                                                                   from the victim's heap
                RAND pseudo bytes(bp, padding);
```

int

## Broken OpenSSL code@victim

Add padding and send the response heartbeat message back to the attacker

### 65534 byte return payload may contain sensitive data

Ĺ	1									Unti	itled	1 - k	lote	epa	d				×
F	ile Edit	For	mat	Vie	2W	Help	<b>)</b>												
	0700:	BC	9C	2D	61	5F	32	36	30	35	26	<b>2E</b>	73	61	76	65	3D	a_2605&.save=	~
	0710:	26	70	61	73	73	77	64	5F	72	61	77	3D	06	14	CE	6F	&passwd_raw=o	
	0720:	A9	13	96	CA	A1	35	1F	11	79	2B	20	BC	2E	75	3D	63	5y+u=c	
	0730:	6A	66	6A	6D	31	68	39	6B	37	6D	36	30	26	2E	76	3D	jfjm1h9k7m60&.v=	
	0740:	30	26	2E	63	68	61	6C	6C	65	6E	67	65	3D	67	7A	37	0&.challenge=gz7	
	0750:	6E	38	31	52	6C	52	4D	43	6A	49	47	4A	6F	71	62	33	n81R1RMCjIGJoqb3	
	0760:	75	69	72	61	2E	6D	6D	36	61	26	2E	79	70	6C	75	73	uira.mm6a&.yplus	
	0770:	3D	26	2E	65	6D	61	69	6C	43	6F	64	65	3D	26	70	6B	=&.emailCode=&pk	
	0780:	67	3D	26	73	74	65	70	69	64	3D	26	2E	65	76	3D	26	g=&stepid=&.ev=&	
	0790:	68	61	73	4D	73	67	72	3D	30	26	2E	63	68	6B	50	3D	hasMsgr=0&.chkP=	
	07a0:	59	26	2E	64	6F	6E	65	3D	68	74	74	70	25	33	41	25	Y&.done=http%3A%	
	07b0:	32	46	25	32	46	6D	61	69	6C	2E	79	61	68	6F	6F	2E	2F%2Fmail.yahoo.	
	07c0:	63	6F	6D	26	2E	70	64	ЗD	79	6D	5F	76	65	72	25	33	com&.pd=ym_ver%3	
	07d0:	44	30	25	32	36	63	25	33	44	25	32	36	69	76	74	25	D0%26c%3D%26ivt%	
	07e0:	33	44	25	32	36	73	67	25	33	44	26	2E	77	73	ЗD	31	3D%26sg%3D&.ws=1	
	07f0:	26	2E	63	70	3D	30	26	6E	72	3D	30	26	70	61	64	ЗD	&.cp=0&nr=0&pad=	
	0800:	36	26	61	61	64	3D	36	26	6C	6F	67	69	6E	3D	61	67	6&aad=6&login=ag	
	0810:	6E	65	73	61	64	75	62	6F	61	74	65	6E	67	25	34	30	nesaduboateng%40	
	0820:	79	61	68	6F	6F	2E	63	6F	6D	26	70	61	73	73	77	64	yahoo.com&passwd	
	0830:	3D	30	32	34													=024 &.pe	
<																			>

Further, invocations of similar false heartbleed 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

```
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)