CS558 Network Security

Lecture 13: SSL and TLS (Part 2)

hello

BOSTON UNIVERSITY

Review: TLS Intuition/Goals

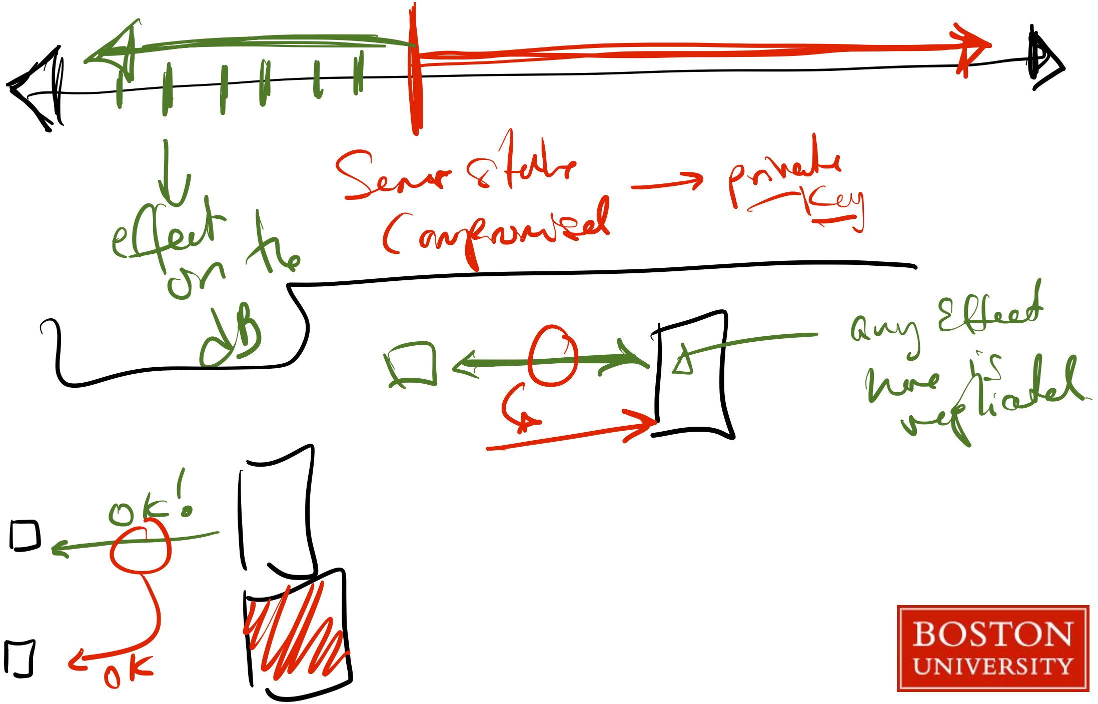
Confidentiality

Integrity

Authenticity

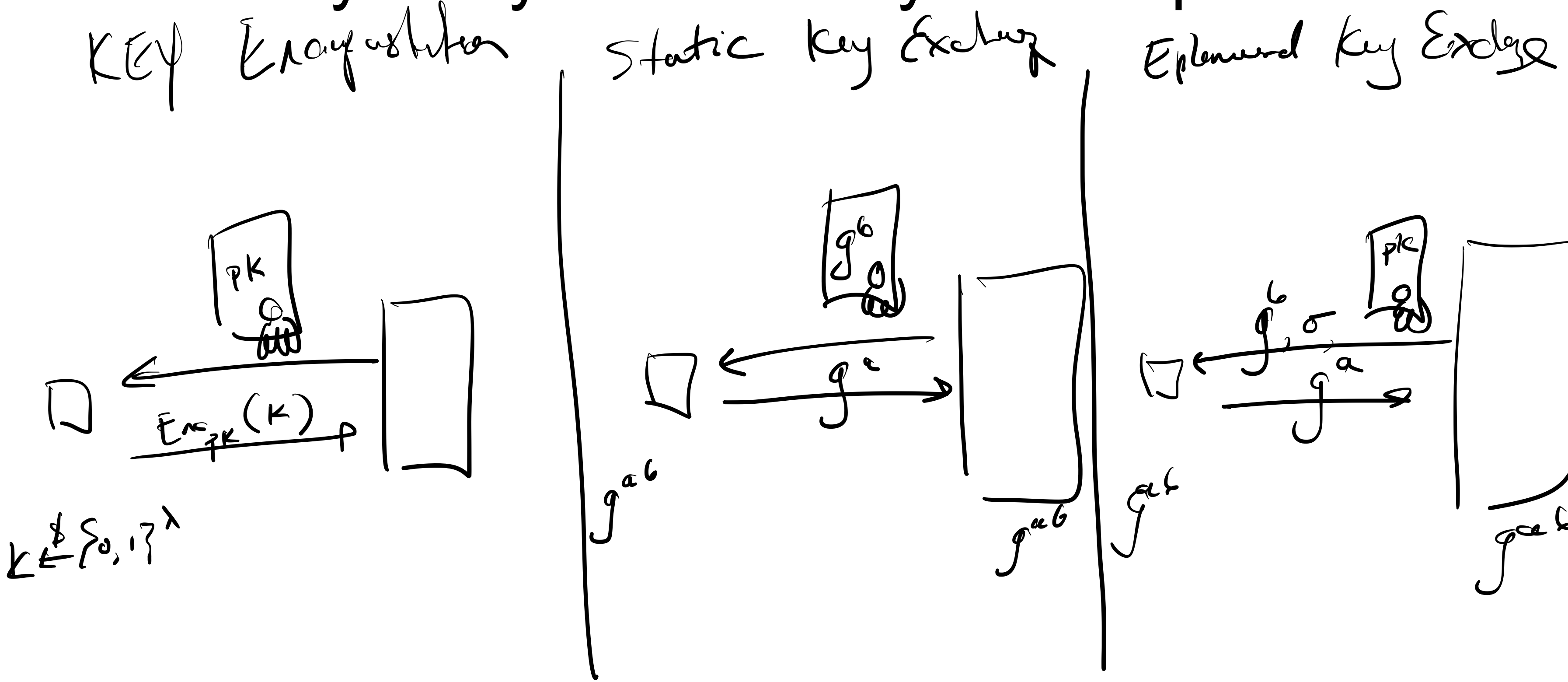
Forward Secrecy

Replay protection



BOSTON UNIVERSITY

Public Key to Symmetric Key: Two Options*



TLS Record Layer

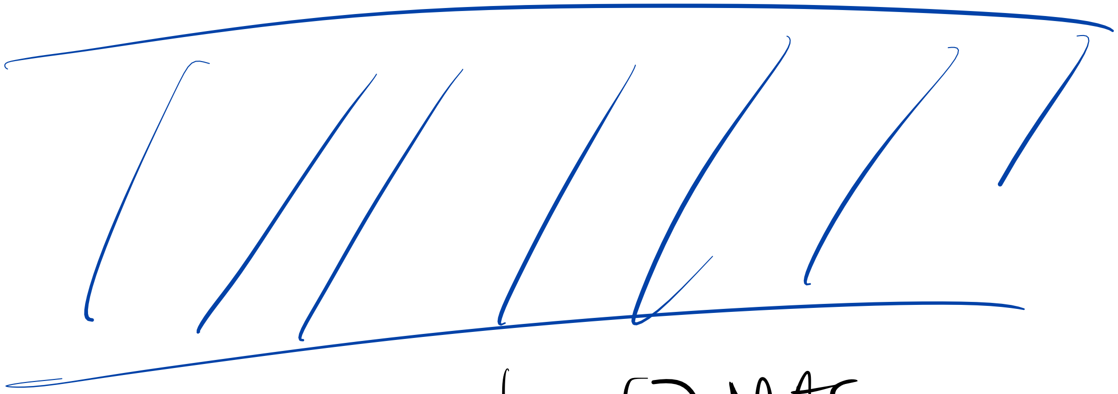
AES $\hookrightarrow$ $k \in \{0,1\}^\lambda$

k

$H(g^{ab})$

k

$H(g^{ab})$



① Encrypt ② MAC

6. The TLS Record Protocol

The TLS Record Protocol is a layered protocol. At each layer, messages may include fields for length, description, and content. The Record Protocol takes messages to be transmitted, fragments the data into manageable blocks, optionally compresses the data, applies a MAC, encrypts, and transmits the result. Received data is decrypted, verified, decompressed, reassembled, and then delivered to higher-level clients.

TLS(1.2) in Detail

RFC 5246 TLS August 2008

negotiating the options

Client -- Server

ClientHello -------->

ServerHello
Certificate*
ServerKeyExchange*
~~CertificateRequest*~~
<-------- ServerHelloDone

~~Certificate*~~
ClientKeyExchange
~~CertificateVerify*~~
[ChangeCipherSpec]
Finished -------->

[ChangeCipherSpec]
<-------- Finished

Application Data <-------> Application Data

Figure 1. Message flow for a full handshake

TLS_RSA_WITH_NULL_SHA256	NULL-SHA256
TLS_RSA_WITH_AES_128_CBC_SHA256	AES128-SHA256
TLS_RSA_WITH_AES_256_CBC_SHA256	AES256-SHA256
TLS_RSA_WITH_AES_128_GCM_SHA256	AES128-GCM-SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384	AES256-GCM-SHA384
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	DH-RSA-AES128-SHA256
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	DH-RSA-AES256-SHA256
TLS_DH_RSA_WITH_AES_128_GCM_SHA256	DH-RSA-AES128-GCM-SHA256
TLS_DH_RSA_WITH_AES_256_GCM_SHA384	DH-RSA-AES256-GCM-SHA384
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	DH-DSS-AES128-SHA256
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	DH-DSS-AES256-SHA256
TLS_DH_DSS_WITH_AES_128_GCM_SHA256	DH-DSS-AES128-GCM-SHA256
TLS_DH_DSS_WITH_AES_256_GCM_SHA384	DH-DSS-AES256-GCM-SHA384
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	DHE-RSA-AES128-SHA256
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	DHE-RSA-AES256-SHA256
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	DHE-RSA-AES128-GCM-SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	DHE-RSA-AES256-GCM-SHA384
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	DHE-DSS-AES128-SHA256
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	DHE-DSS-AES256-SHA256
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	DHE-DSS-AES128-GCM-SHA256
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	DHE-DSS-AES256-GCM-SHA384
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	ECDH-RSA-AES128-SHA256
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	ECDH-RSA-AES256-SHA384
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	ECDH-RSA-AES128-GCM-SHA256
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	ECDH-RSA-AES256-GCM-SHA384
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	ECDH-ECDSA-AES128-SHA256
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	ECDH-ECDSA-AES256-SHA384
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	ECDH-ECDSA-AES128-GCM-SHA256
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	ECDH-ECDSA-AES256-GCM-SHA384
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	ECDHE-RSA-AES128-SHA256
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	ECDHE-RSA-AES256-SHA384
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	ECDHE-RSA-AES128-GCM-SHA256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	ECDHE-RSA-AES256-GCM-SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	ECDHE-ECDSA-AES128-SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	ECDHE-ECDSA-AES256-SHA384
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	ECDHE-ECDSA-AES128-GCM-SHA256
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	ECDHE-ECDSA-AES256-GCM-SHA384
TLS_DH_anon_WITH_AES_128_CBC_SHA256	ADH-AES128-SHA256
TLS_DH_anon_WITH_AES_256_CBC_SHA256	ADH-AES256-SHA256
TLS_DH_anon_WITH_AES_128_GCM_SHA256	ADH-AES128-GCM-SHA256
TLS_DH_anon_WITH_AES_256_GCM_SHA384	ADH-AES256-GCM-SHA384

https://www.openssl.org/docs/man1.0.2/man1/ciphers.html

Key Exchange Authentication Cipher (algorithm, strength, mode) Hash or MAC

ECDHE-ECDSA-AES128-GCM-SHA256

https://blog.keyfactor.com/cipher-suites-explained

BOSTON UNIVERSITY

TLS(1.2) in Detail

```
RFC 5246                        TLS                        August 2008

      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data

             Figure 1.  Message flow for a full handshake
```

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-2>;
    CompressionMethod compression_methods<1..2^8-1>;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ClientHello;
```

Client Random

BOSTON UNIVERSITY

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
    select (extensions_present) {
        case false:
            struct {};
        case true:
            Extension extensions<0..2^16-1>;
    };
} ServerHello;
```

Server Random

TLS(1.2) in Detail

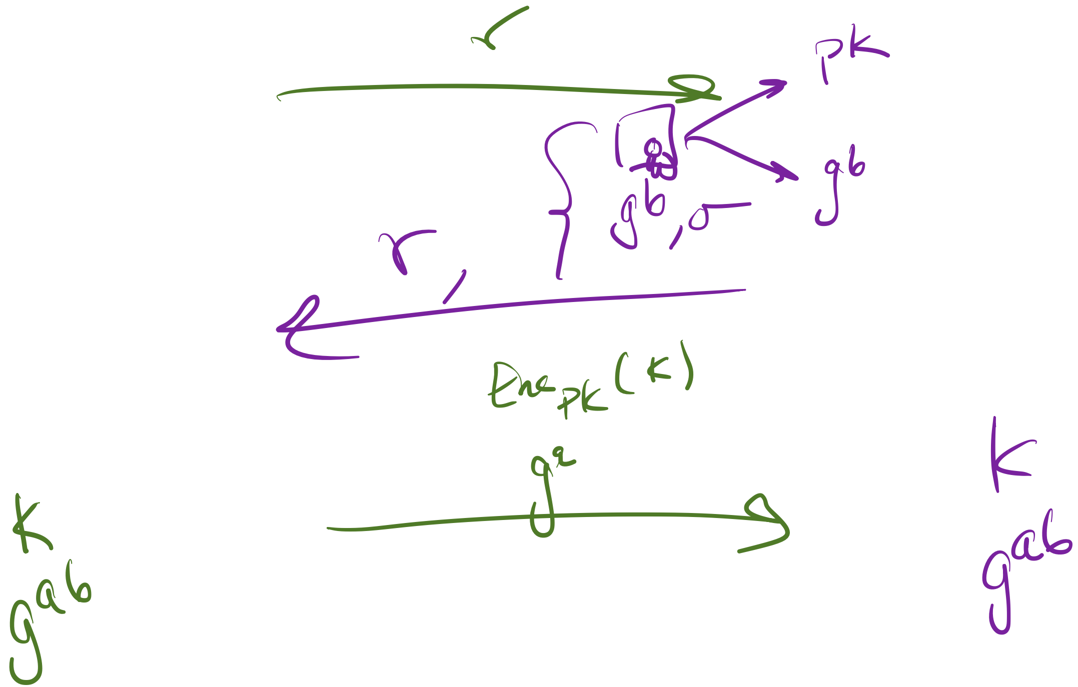
```
RFC 5246                          TLS                        August 2008

      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                               [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data

             Figure 1.  Message flow for a full handshake
```

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHParams params;              g^b
        case dhe_dss:
        case dhe_rsa:
            ServerDHParams params;
            digitally-signed struct {
                opaque client_random[32];       r
                opaque server_random[32];       r
                ServerDHParams params;          g^b
            } signed_params;
                                      σ
        case rsa:
        case dh_dss:
        case dh_rsa:
            struct {} ;
           /* message is omitted for rsa, dh_dss, and dh_rsa */
        /* may be extended, e.g., for ECDH -- see [TLSECC] */
    };
} ServerKeyExchange;
```

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa:
            EncryptedPreMasterSecret;
        case dhe_dss:
        case dhe_rsa:
        case dh_dss:
        case dh_rsa:
        case dh_anon:
            ClientDiffieHellmanPublic;
    } exchange_keys;
} ClientKeyExchange;
```

$Enc_{pk}(k)$

g^a


r
pk
g^b
r, g^b, σ
Enc_pk(K)
g^a
K
g^ab
K
g^ab

K , g^{ab}

```
master_secret = PRF(pre_master_secret, "master secret",
                    ClientHello.random + ServerHello.random)
                    [0..47];
```

$\xrightarrow{r}$

$\xleftarrow{r}$

To generate the key material, compute

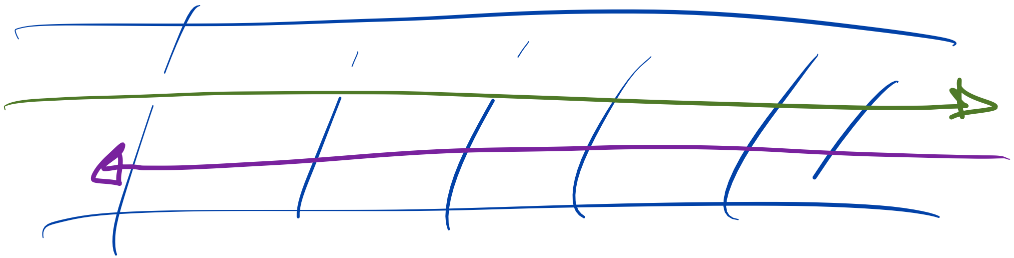
```
key_block = PRF(SecurityParameters.master_secret,
                "key expansion",
                SecurityParameters.server_random +
                SecurityParameters.client_random);
```

$\xrightarrow{r}$

$\xleftarrow{r'}$

until enough output has been generated. Then, the key_block is partitioned as follows:

```
client_write_MAC_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```



```
struct {
    opaque verify_data[verify_data_length];
} Finished;

verify_data
   PRF(master_secret, finished_label, Hash(handshake_messages))
      [0..verify_data_length-1];
```

Heartbeat Extension

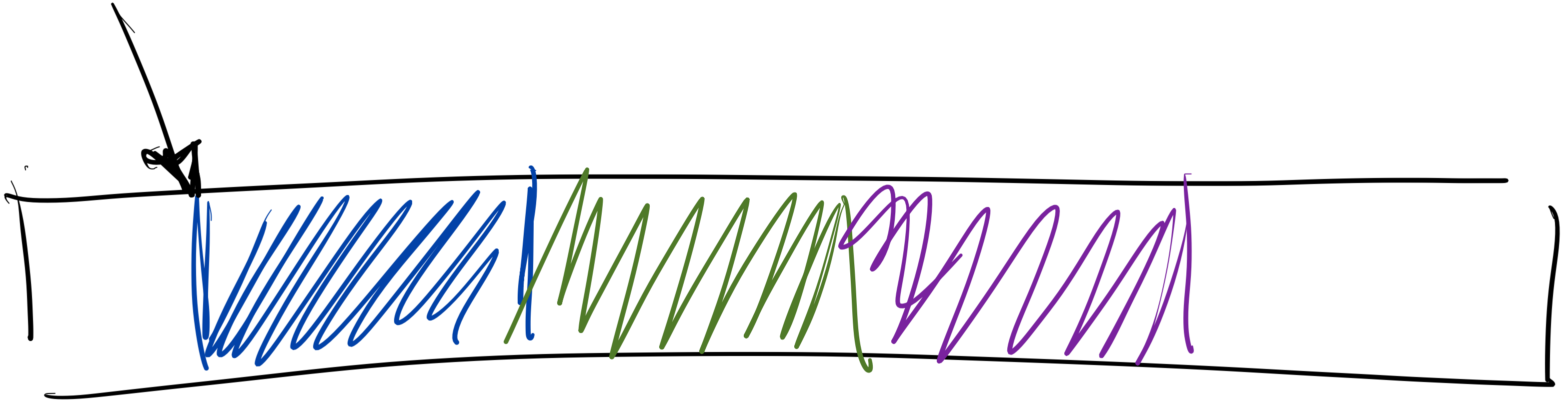
↳ makes things more efficient when you have long gaps

BOSTON UNIVERSITY

```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[HeartbeatMessage.payload_length];
    opaque padding[padding_length];
} HeartbeatMessage;
```

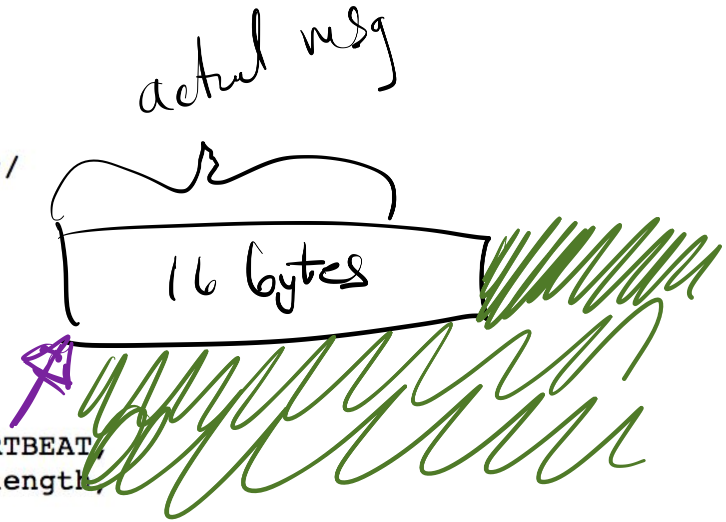
BOSTON UNIVERSITY

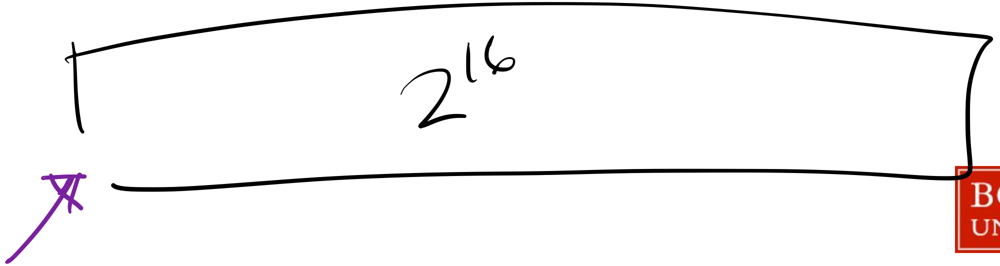
Memory Management Reminder



Heartbleed Attack

```
tls1_process_heartbeat(SSL *s)
        {
        unsigned char *p = &s->s3->rrec.data[0], *pl;
        unsigned short hbtype;
        unsigned int payload;
        unsigned int padding = 16; /* Use minimum padding */

        /* Read type and payload length first */
        hbtype = *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);
```





BOSTON UNIVERSITY

Heartbleed Attack

```
tls1_process_heartbeat(SSL *s)
        {
        unsigned char *p = &s->s3->rrec.data[0], *pl;
        unsigned short hbtype;
        unsigned int payload;
        unsigned int padding = 16; /* Use minimum padding */

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

        if (s->msg_callba
                s->msg_ca
                        &
                        s
```

```
        if (hbtype == TLS1_HB_REQUEST)
                {
                unsigned char *buffer, *bp;
                int r;

                /* Allocate memory for the response, size is 1 bytes
                 * 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);
                bp += payload;
                /* Random padding */
                RAND_pseudo_bytes(bp, padding);

                r = ssl3_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Heartbleed Attack

HB {
payload_length = 16
Payload[ ] $\underbrace{\qquad\qquad}_{16}$

HB {
payload_length = 2^{16}
payload[ ] $\underset{16}{\underline{\qquad\qquad}}$

```
struct {
   HeartbeatMessageType type;
   uint16 payload_length;
   opaque payload[HeartbeatMessage.payload_length];
   opaque padding[padding_length];
} HeartbeatMessage;
```

How widespread is this?

The most notable software using OpenSSL are the open source web servers like Apache and nginx. The combined market share of just those two out of the active sites on the Internet was over 66% according to Netcraft's April 2014 Web Server Survey. Furthermore OpenSSL is used to protect for example email servers (SMTP, POP and IMAP protocols), chat servers (XMPP protocol), virtual private networks (SSL VPNs), network appliances and wide variety of client side software. Fortunately many large consumer sites are saved by their conservative choice of SSL/TLS termination equipment and software. Ironically smaller and more progressive services or those who have upgraded to latest and best encryption will be affected most. Furthermore OpenSSL is very popular in client software and somewhat popular in networked appliances which have most inertia in getting updates.