

# **TCP AND TLS**

**EN.600.424**

**Fall 2018**

**Lecture Notes**

# TCP PROTOCOL

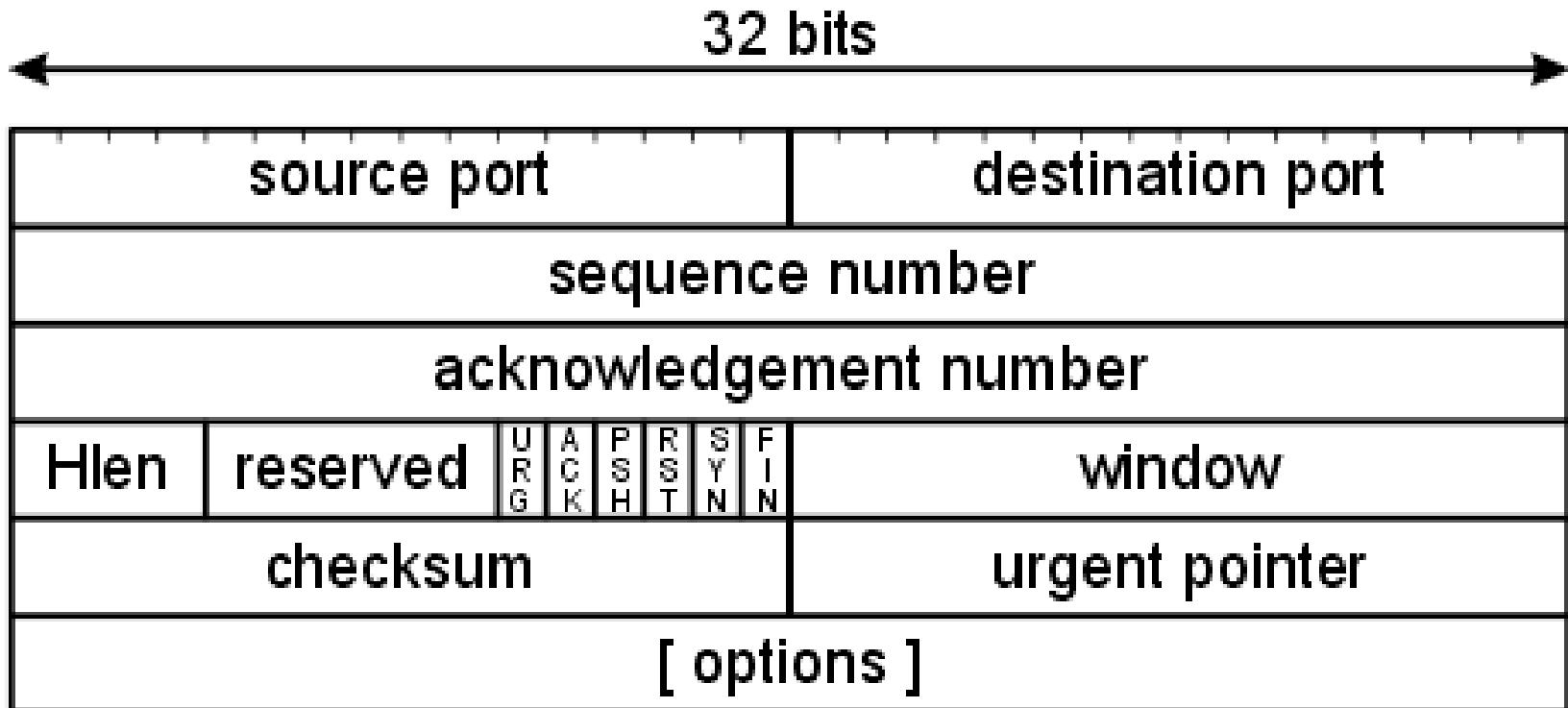
- Layer 4 Protocol
- Multiplexing (ports)
- Reliable Delivery (ack/resend)
- Congestion control
- Units called \*segments\*

# REVIEW

- Ethernet *frame*
- IP *packet*
- TCP *segment*

# TCP HEADER

TCP header format



# TCP HEADER FIELDS

- Source Port, Destination Port for multiplexing
- Sequence Number of the first data byte
- Acknowledgement Number of next expected seq. no.
- Hlen number of 32-byte words in the TCP header
- Window number of bytes willing to receive
- Checksum over the header and data

# TCP FLAGS

- URG The URGENT POINTER field contains valid data
- ACK The acknowledgement number is valid
- PSH The receiver should pass this data to the application as soon as possible
- RST Reset the connection
- SYN Synchronize sequence numbers to initiate a connection.
- FIN Sender is finished sending data

# ECN FLAGS ADDED IN RFC 3168

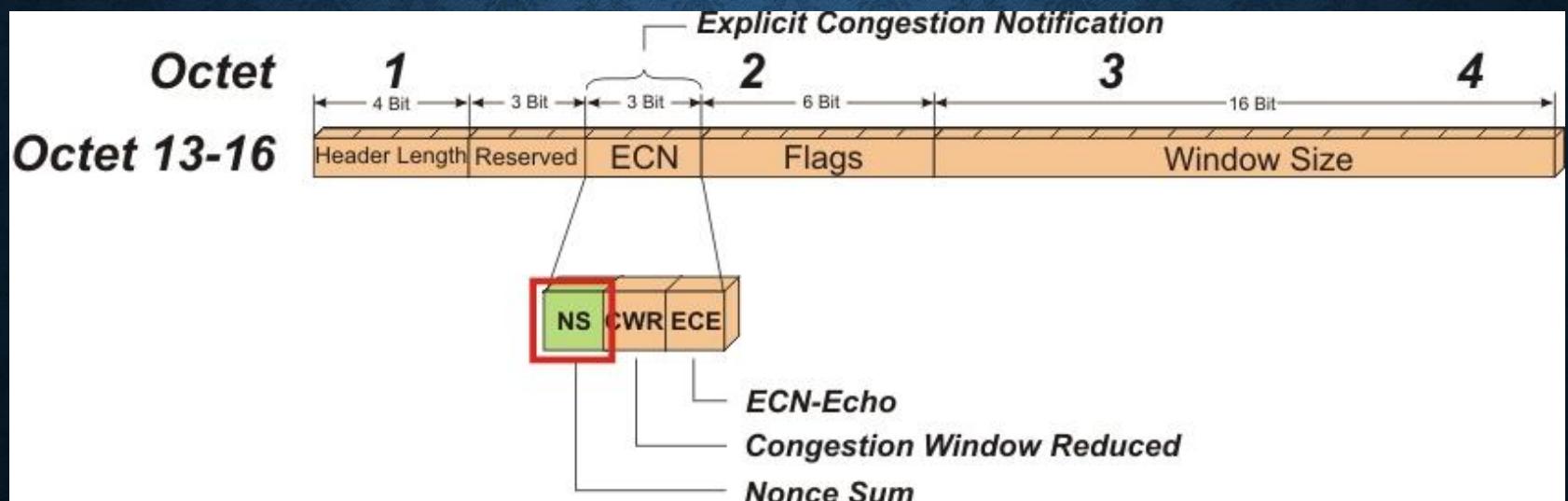
- Standard in 2001

TCP Header Fields														
Source Port					Destination Port									
Sequence Number														
Acknowledgement Number														
Header Length	Reserved Field	C W R	E C E	U R G	A C K	P S H	R S T	S Y N	F I N					
TCP Checksum					Urgent Pointer									

# TCP FLAGS (CONT)

- CWR – Acknowledges that congestion notification received
- ECN – indicates congestion notification via IP layer
  - (NOTE! Requires ECN capable IP layer!)
  - Sent until CWR received
- Only used if negotiated using TCP options during handshake

# OPTIONAL NS FLAG



# ONE-BIT NONCE

- NS is a parity bit used to catch changes to a packet
- Because it's only one bit, a cheater can guess it 50% right
- But, over repeated trials (frequent congestion) will get caught

# TCP OPTIONS

- Header can be “extended” with options
- Each option can have up to three fields:
  - Option Type (1 byte)
  - Option Length (1 byte)
  - Option data (variable)
- Examples include
  - Selective acknowledgement
  - ECN

# TCP SEQUENCE NUMBERS

- TCP header has a value for seq num and ack num every time
- SYN Sequence Number random between 0-4,294,967,295
- SYN Ack Num should be 0 (but any value should be ignored)
- SYN-ACK Seq Num also random
- SYN-ACK Ack Num is SYN Seq Num + 1
- (SYN-ACK) ACK Seq Num is SYN Seq Num + 1
- (SYN-ACK) ACK Ack Num is SYN-ACK Seq Num + 1

# DATA SEQUENCE NUMBERS

- Technically, there is only one packet type in TCP
- Flags simply indicate how the values can be used
- Sequence number is set every time
- But only increased by the length of the data
- (or increased by +1 for SYN and FIN)
- Ack field indicates that the ACK number is valid

# WIRESHARK TRACE

Time	192.168.1.2	Comment
0.000	(54841) → SYN (80)	Seq = 0 Ack = 94856056
0.047	(54841) ← SYN, ACK (80)	Seq = 0 Ack = 1
0.047	(54841) → ACK (80)	Seq = 1 Ack = 1
0.047	(54841) → PSH, ACK - Len: 725 (80)	Seq = 1 Ack = 1
0.097	(54841) ← ACK (80)	Seq = 1 Ack = 726
0.100	(54841) ← ACK - Len: 1448 (80)	Seq = 1 Ack = 726
0.100	(54841) → ACK (80)	Seq = 726 Ack = 1449
0.100	(54841) ← ACK - Len: 1448 (80)	Seq = 1449 Ack = 726
0.100	(54841) → ACK (80)	Seq = 726 Ack = 2897
0.100	(54841) ← ACK - Len: 1448 (80)	Seq = 2897 Ack = 726
0.100	(54841) → ACK (80)	Seq = 726 Ack = 4345
0.150	(54841) ← ACK - Len: 1448 (80)	Seq = 4345 Ack = 726
0.150	(54841) → ACK (80)	Seq = 726 Ack = 5793
0.152	(54841) ← ACK - Len: 1448 (80)	Seq = 5793 Ack = 726
0.152	(54841) → ACK (80)	Seq = 726 Ack = 7241
0.152	(54841) ← ACK - Len: 1448 (80)	Seq = 7241 Ack = 726
0.152	(54841) → ACK (80)	Seq = 726 Ack = 8689

# TCP SHUTDOWN



# SECURITY

- Obviously, TCP is not designed with security in mind
  - No confidentiality!
  - No authentication!
  - No integrity (checksum is not cryptographic)
- No secure availability either!
  - End any connection with RST

# **SSLV2**

- SSL = Secure Sockets Layer
- SSLv1 was never published
- SSLv2 was published 11/29/1994, last updated 2/9/1995

# SSL RECORD LAYER

- All data encapsulated in a “Record”
  - Header
  - Data
- Header includes record length, and some meta information
- Data portion includes
  - Mac data
  - Actual data
  - Padding data

# **SSLV2 RECORD MAC**

- **MAC-DATA = HASH[ SECRET, ACTUAL-DATA, PADDING-DATA, SEQUENCE-NUMBER ]**
- Sequence is 32bit number, starts at 0 and increments
- SECRET is the CLIENT-WRITE-KEY for the client.
- SECRET is the SERVER-WRITE-KEY for the server
- CLIENT-READ-KEY = SERVER-WRITE-KEY

# SSLV2 HANDSHAKE

- client-hello      C -> S: challenge, cipher\_specs
- server-hello      S -> C: connection-id,server\_certificate,cipher\_specs
- client-master-key   C -> S: {master\_key}server\_public\_key
- client-finish      C -> S: {connection-id}client\_write\_key
- server-verify      S -> C: {challenge}server\_write\_key
- server-finish      S -> C: {new\_session\_id}server\_write\_key

# KEY DERIVATION

- KEY-MATERIAL-0 = MD5[ MASTER-KEY, "0", CHALLENGE, CONNECTION-ID ]
- KEY-MATERIAL-1 = MD5[ MASTER-KEY, "1", CHALLENGE, CONNECTION-ID ]
- CLIENT-READ-KEY = KEY-MATERIAL-0[0-15]
- CLIENT-WRITE-KEY = KEY-MATERIAL-1[0-15]

# SSLV2 SESSION RECOVERY

- client-hello      C -> S: challenge, session\_id, cipher\_specs
- server-hello      S -> C: connection-id, session\_id\_hit
- client-finish      C -> S: {connection-id}client\_write\_key
- server-verify      S -> C: {challenge}server\_write\_key
- server-finish      S -> C: {session\_id}server\_write\_key

# CLIENT AUTH

- Server may request client cert
- Send a challenge to authenticate

# BULK DATA TRANSMISSION

- Use write key to create data
  - $C = \{P\}write-key$
- Use same write key to MAC data
  - $MAC = \text{Hash}(\text{write-key}, C, \text{Padding}, \text{Sequence})$

# SSLV3

- Netscape Memo March 1996
- “Layered Protocol” (e.g., record layer, message layer)
- “Stateful”

“SSL 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, and reassembled, then delivered to higher level clients”

# **SSLV3 STATE**

- Session identifier – arbitrary bytes
- Peer certificate – (may be null)
- Compression method
- Cipher spec – specifics bulk data, MAC, hash size, etc
- Master secret - 48-byte secret shared
- Is resumable

# **SSLV3 CONNECTION STATE**

- There can be multiple connections within the same session
- Server and client random
- Client write key
- Client write MAC secret
- Server write key
- Server write MAC secret
- IV's and sequence numbers

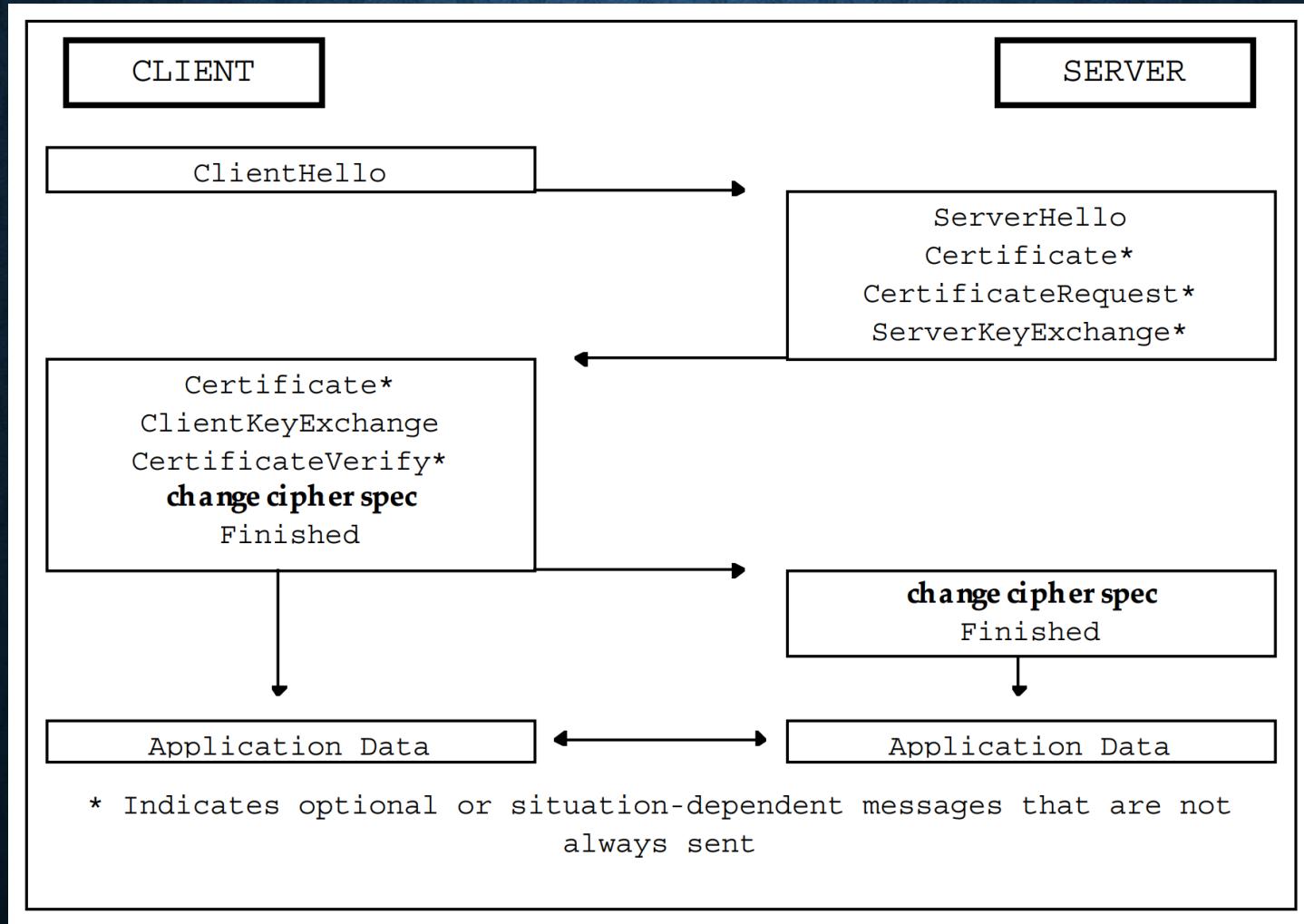
# RECORD LAYER

- Fragmentation
  - The record layer fragments information blocks into SSLPlaintext records of  $2^{14}$  bytes or less
- Record layer now includes:
  - Version
  - Higher SSL type
- Supports compression

# NEW MAC

- $\text{hash}(\text{MAC\_write\_secret} + \text{pad\_2} + \text{hash}(\text{MAC\_write\_secret} + \text{pad\_1} + \text{seq\_num} + \text{length} + \text{content}))$ ;
- pad\_1 - 0x36 repeated 48x for MD5 this or 40x for SHA.
- pad\_2 - 0x5c repeated the same number of times.
- hash – chosen by cipher suite

# HANDSHAKE CHANGES



# SSLV3 HANDSHAKE CHANGES

- Certs can now be a chain (before, only 1!)
- Many key exchange options including DH
- Hash of all handshake messages at the end
  - `md5_hash MD5(master_secret + pad2 +  
MD5(handshake_messages + Sender + master_secret + pad1));`
  - `sha_hash SHA(master_secret + pad2 +  
SHA(handshake_messages + Sender + master_secret + pad1));`

# MASTER SECRET

- Derived from pre-master secret
- Pre-master secret computed directly from DH
- Or, can exchange pre-master secret encrypted with pub key
- $\text{master\_secret} = \text{MD5}(\text{pre\_master\_secret} + \text{SHA}('A' + \text{pre\_master\_secret} + \text{ClientHello.random} + \text{ServerHello.random})) + \text{MD5}(\text{pre\_master\_secret} + \text{SHA}('BB' + \text{pre\_master\_secret} + \text{ClientHello.random} + \text{ServerHello.random})) + \text{MD5}(\text{pre\_master\_secret} + \text{SHA}('CCC' + \text{pre\_master\_secret} + \text{ClientHello.random} + \text{ServerHello.random}));$

# DERIVING SECRETS

- `key_block =`  
`MD5(master_secret + SHA('A' + master_secret +`  
`ServerHello.random + ClientHello.random))`  
`+ MD5(master_secret + SHA('BB' + master_secret +`  
`ServerHello.random + ClientHello.random))`  
`+ MD5(master_secret + SHA('CCC' + master_secret +`  
`ServerHello.random ClientHello.random))`  
`+ [...];`

# KEYS FROM KEY BLOCK

- `client_write_MAC_secret[CipherSpec.hash_size]`
- `server_write_MAC_secret[CipherSpec.hash_size]`
- `client_write_key[CipherSpec.key_material]`
- `server_write_key[CipherSpec.key_material]`
- `client_write_IV[CipherSpec.IV_size]`
- `server_write_IV[CipherSpec.IV_size]`

# **TLS 1.0**

- Very similar to SSLv3
- Different key derivation and MAC
- Not compatible

# TLS 1.0 HMAC

- Standardized HMAC, RFC 2104
- $H(K \text{ XOR } opad, H(K \text{ XOR } ipad, \text{text}))$
- Ipad: the byte 0x36 repeated B times
- Opad: the byte 0x5C repeated B times
- B: block size of H (64 bytes for SHA-1)
- K: Key

# TLS 1.0 MAC

- HMAC\_hash(MAC\_write\_secret, seq\_num + TLSCompressed.type + TLSCompressed.version + TLSCompressed.length + TLSCompressed.fragment));

# TLS 1.0 DATA EXPANSION

- $P_{\text{hash}}(\text{secret}, \text{seed}) = \text{HMAC\_hash}(\text{secret}, A(1) + \text{seed}) + \text{HMAC\_hash}(\text{secret}, A(2) + \text{seed}) + \text{HMAC\_hash}(\text{secret}, A(3) + \text{seed}) + \dots$
- $A()$  is defined as:
  - $A(0) = \text{seed}$
  - $A(i) = \text{HMAC\_hash}(\text{secret}, A(i-1))$

# **TLS 1.0 PRF**

- Pseudo Random Function
- $\text{PRF}(\text{secret}, \text{label}, \text{seed}) = \text{P\_MD5}(\text{S1}, \text{label} + \text{seed}) \text{ XOR}$   
 $\text{P\_SHA-1}(\text{S2}, \text{label} + \text{seed});$
- **S1 = first half of secret**
- **S2 = second half of secret**

# TLS 1.0 KEY BLOCK

- `key_block = PRF(SecurityParameters.master_secret, "key expansion", SecurityParameters.server_random + SecurityParameters.client_random);`
- Session keys derived from `key_block` similar to SSLv3

# TLS 1.0 FINISHED MSG

- `verify_data = PRF(master_secret, finished_label,  
MD5(handshake_messages) + SHA-1(handshake_messages))  
[0..11];`

# TLS 1.0 BULK DATA TRANSFER

- The record layer encapsulates a bulk data fragment
- ```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    opaque fragment[TLSPrintext.length];
} TLSPrintext;
```
- (Opaque just means raw bytes)

# FRAAGMENT TYPES

```
select (CipherSpec.cipher_type) {  
    case stream: GenericStreamCipher;  
    case block: GenericBlockCipher;  
} fragment;
```

# STREAM TYPE

```
stream-ciphered struct {  
    opaque content[TLSCompressed.length];  
    opaque MAC[CipherSpec.hash_size];  
} GenericStreamCipher;
```

# CBC BLOCK CIPHER

```
block-ciphered struct {  
    opaque content[TLSCompressed.length];  
    opaque MAC[CipherSpec.hash_size];  
    uint8 padding[GenericBlockCipher.padding_length];  
    uint8 padding_length;  
} GenericBlockCipher;
```

# CHANGE FROM SSLV3

- The contents of “padding” not specified in SSLV3
- In TLS 1.0, specified as:

“Each uint8 in the padding data vector must be filled with the padding length value.”

# TO CLARIFY

- Sample 10 legal padding bytes in SSLV3:
  - 0xAA 0xBB 0xCC 0xDD 0xEE 0xFF 0x00 0x11 0x22 0x33 0xA
  - (NOTE: 0xA = 10 = Length)
- Only 10 legal padding bytes in TLS 1.0:
  - 0xA 0xA 0xA 0xA 0xA 0xA 0xA 0xA 0xA 0xA
  - (NOTE: last 0xA is still length!)

# **NOTE: AUTH THEN ENCRYPT**

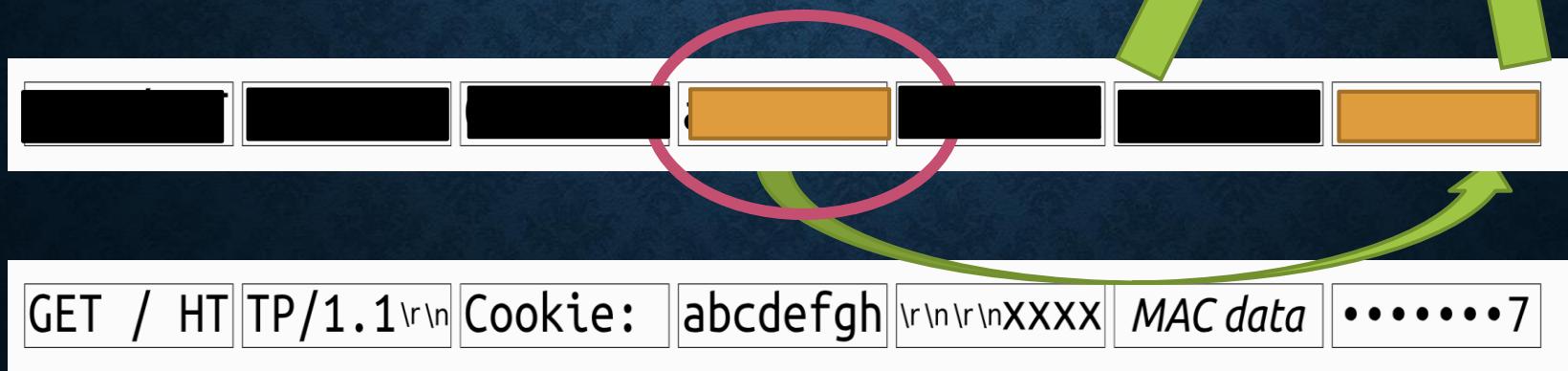
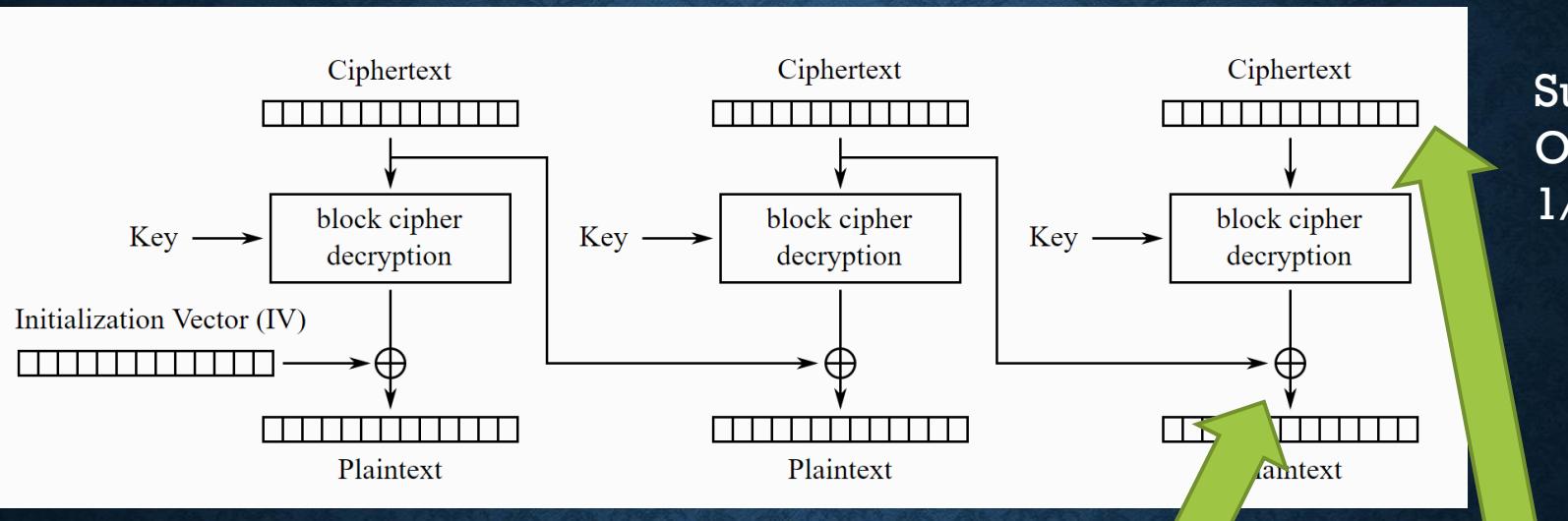
- Both SSLv3 and TLS 1.0 chose to auth then encrypt
- Look at the data structure. The MAC is included in the plaintext
- ALSO NOTE: DID NOT INCLUDE THE PADDING!!!!

# POODLE ATTACK

- SSLV3's lack of a padding specification resulted in an attack
- You could use a “padding oracle” attack to decrypt bytes
- Approximately, 256 messages would reveal one byte

# PADDING ORACLE ATTACK

Success  
Odds:  
1/256



# GET THE NEXT BYTE

```
GET /a H[TTT/1.1\r\nCookie: abcdefg|h\r\nr\r\nxxx] MAC data •••••7
```

# ATTACKING TLS 1.0

- POODLE would use an MITM to force a downgrade to SSLv3
- OR, some TLS1.0 versions DIDN'T CHECK the PADDING!

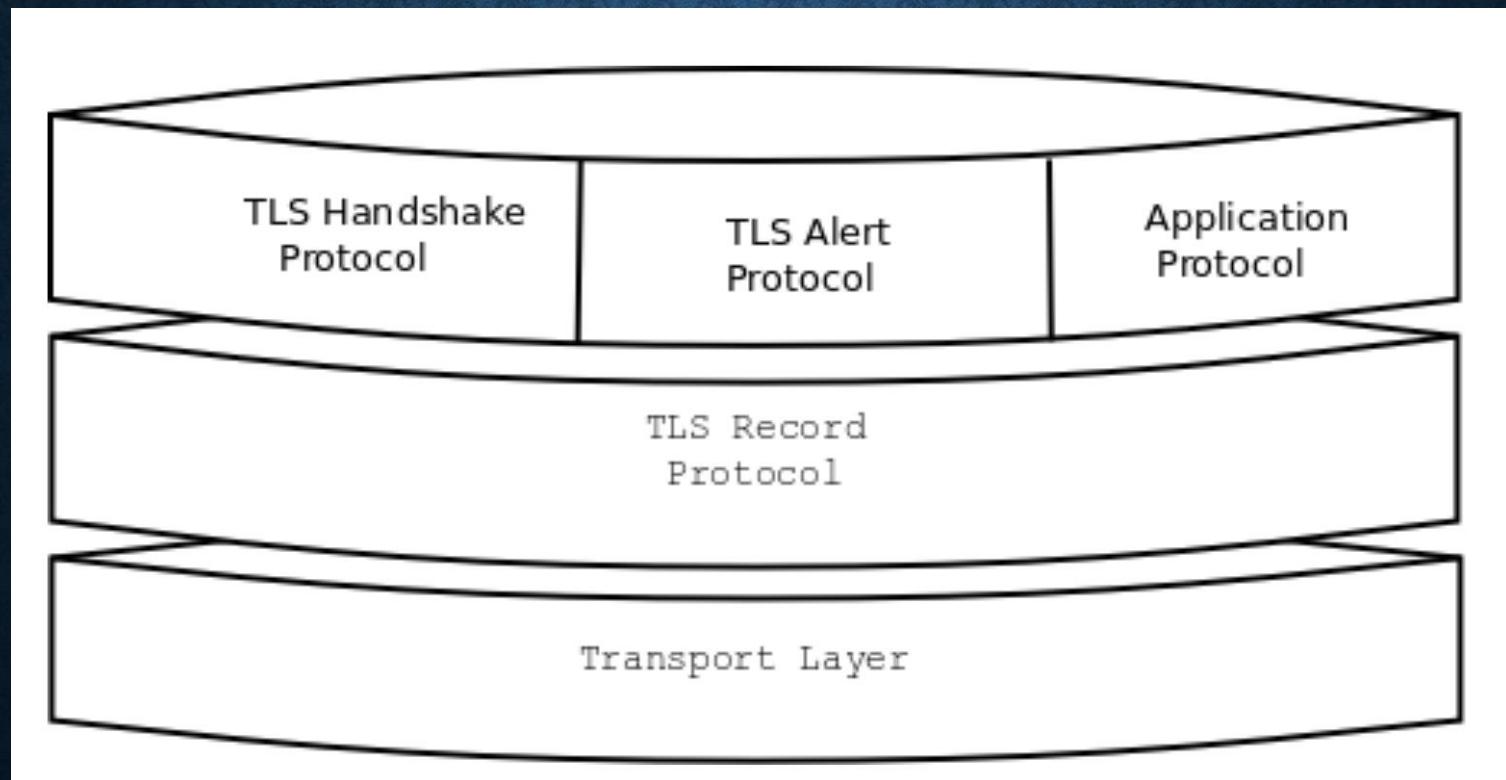
# **TLS 1.1**

- Attempted to fix various problems with TLS 1.0
- CBC IV's (explicit v implicit)
  - Now included in record
  - Prior to 1.1, IV was last ciphertext block of previous record
- Padding errors now just return error “bad\_record\_mac”
- Sometimes called the forgotten middle child

# **TLS 1.2**

- PRF replaced MD5/SHA1 with cipher suite specific function
- Signatures explicitly identify hashing algorithm
- Support for authenticated encryption
- Changes to resist known attacks against previous versions

# TO TALK ABOUT 1.2, LET'S REVIEW



# CONNECTION STATE

- Connection end – “client” or “server”
- PRF algorithm
- Bulk encryption algorithm
- MAC algorithm
- Compression algorithm
- Master secret
- Client random
- Server random

# CONNECTION STATE SECURITY PARAMETERS

```
struct {
    ConnectionEnd           entity;
    PRFAlgorithm            prf_algorithm;
    BulkCipherAlgorithm     bulk_cipher_algorithm;
    CipherType               cipher_type;
    uint8                   enc_key_length;
    uint8                   block_length;
    uint8                   fixed_iv_length;
    uint8                   record_iv_length;
    MACAlgorithm             mac_algorithm;
    uint8                   mac_length;
    uint8                   mac_key_length;
    CompressionMethod        compression_algorithm;
    opaque                  master_secret[48];
    opaque                  client_random[32];
    opaque                  server_random[32];
} SecurityParameters;
```

# DERIVED VALUES

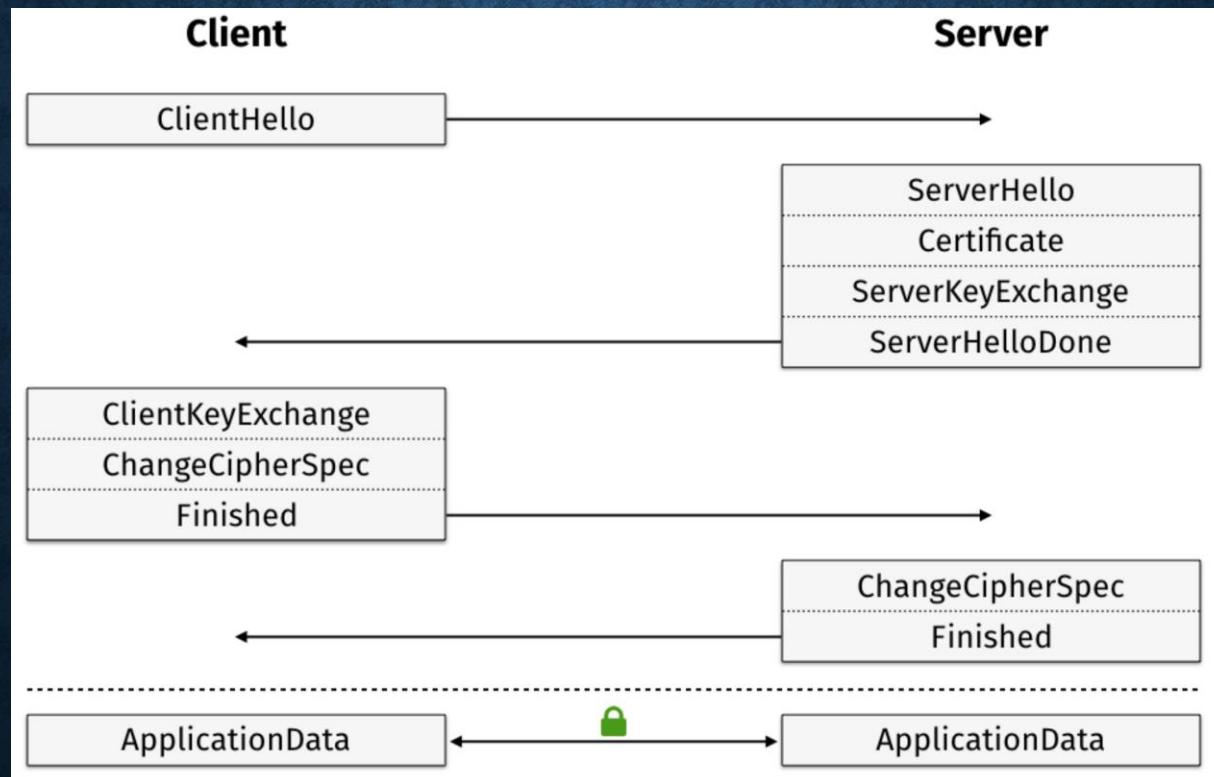
The record layer will use the security parameters to generate the following six items (some of which are not required by all ciphers, and are thus empty):

```
client write MAC key  
server write MAC key  
client write encryption key  
server write encryption key  
client write IV  
server write IV
```

# CONNECTION STATE

- Compression state
- Cipher state
- MAC key
- Sequence number

# TLS HANDSHAKE OVERVIEW



# CLIENT HELLO

```
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 HELLO EXAMPLE

```
✓ Handshake Protocol: Client Hello
  Handshake Type: Client Hello (1)
  Length: 199
  Version: TLS 1.2 (0x0303)
  ✓ Random
    GMT Unix Time: Jan 20, 2017 16:26:14.000000000 Eastern Standard Time
    Random Bytes: 2ccacc6176e4b62418b67e3a36cc5b87f8de3ef7696ff56c...
    Session ID Length: 0
    Cipher Suites Length: 56
  ✓ Cipher Suites (28 suites)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_GCM_SHA384 (0x009f)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_GCM_SHA256 (0x009e)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384 (0xc024)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256 (0xc023)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
    Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
    Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA256 (0x003d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
```

# TLS 1.2 ALSO PERMITS EXTENSIONS

```
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA256 (0x003c)
Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA256 (0x006a)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA256 (0x0040)
Cipher Suite: TLS_DHE_DSS_WITH_AES_256_CBC_SHA (0x0038)
Cipher Suite: TLS_DHE_DSS_WITH_AES_128_CBC_SHA (0x0032)
Cipher Suite: TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA (0x0013)

Compression Methods Length: 1
> Compression Methods (1 method)
  Extensions Length: 102
  ✓ Extension: server_name
    Type: server_name (0x0000)
    Length: 16
    ✓ Server Name Indication extension
      Server Name list length: 14
      Server Name Type: host_name (0)
      Server Name length: 11
      Server Name: example.com
    > Extension: status_request
    > Extension: elliptic_curves
    > Extension: ec_point_formats
    > Extension: signature_algorithms
    > Extension: SessionTicket TLS
```

# SERVER HELLO

```
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 CERT/KEY EXCHANGE

```
struct {
    ASN.1Cert certificate_list<0..2^24-1>;
} Certificate;
```

```
struct {
    select (KeyExchangeAlgorithm) {
        case dh_anon:
            ServerDHParams params;
        case dhe_dss:
        case dhe_rsa:
            ServerDHParams params;
            digitally-signed struct {
                opaque client_random[32];
                opaque server_random[32];
                ServerDHParams params;
            } signed_params;
        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 {
    opaque dh_p<1..2^16-1>;
    opaque dh_g<1..2^16-1>;
    opaque dh_Ys<1..2^16-1>;
} ServerDHParams;      /* Ephemeral DH parameters */
```

# CLIENT KEY EXCHANGE

```
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;
```

```
struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;
```

# FINISHED

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

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

verify\_data\_length can be specified by the cipher suite,  
otherwise, is 12.

# **PRF**

- In the base RFC, always uses SHA-256
- New cipher suites must define their PRF function
- SHOULD use SHA-256

# DATA EXPANSION

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                      HMAC_hash(secret, A(2) + seed) +
                      HMAC_hash(secret, A(3) + seed) + ...
```

A() is defined as:

$$A(0) = \text{seed}$$

$$A(i) = \text{HMAC\_hash}(\text{secret}, A(i-1))$$

# PRF DEFINITION

TLS's PRF is created by applying P\_hash to the secret as:

$$\text{PRF}(\text{secret}, \text{label}, \text{seed}) = \text{P}_{\text{<hash>}}(\text{secret}, \text{label} + \text{seed})$$

The label is an ASCII string. It should be included in the exact form it is given without a length byte or trailing null character. For example, the label "slithy toves" would be processed by hashing the following bytes:

73 6C 69 74 68 79 20 74 6F 76 65 73

# COMPUTING MASTER SECRET

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

# COMPUTING KEYS

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

```
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]
```

# CHANGE CIPHER SUITE

- Single byte. Indicates next messages use new ciphers

```
▼ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec  
Content Type: Change Cipher Spec (20)  
Version: TLS 1.2 (0x0303)  
Length: 1  
Change Cipher Spec Message
```

# RECORD LAYER TYPES

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
    select (SecurityParameters.cipher_type) {
        case stream: GenericStreamCipher;
        case block: GenericBlockCipher;
        case aead: GenericAEADCipher;
    } fragment;
} TLSCiphertext;
```

# STREAM/BLOCK DEF'S

```
stream-ciphered struct {  
    opaque content[TLSCompressed.length];  
    opaque MAC[SecurityParameters.mac_length];  
} GenericStreamCipher;
```

```
struct {  
    opaque IV[SecurityParameters.record_iv_length];  
    block-ciphered struct {  
        opaque content[TLSCompressed.length];  
        opaque MAC[SecurityParameters.mac_length];  
        uint8 padding[GenericBlockCipher.padding_length];  
        uint8 padding_length;  
    };  
} GenericBlockCipher;
```

# NEW: AEAD TYPE

```
struct {
    opaque nonce_explicit[SecurityParameters.record_iv_length];
    aead-ciphered struct {
        opaque content[TLSCompressed.length];
    };
} GenericAEADCipher;
```

# AEAD

- Authenticated Encryption with Additional Data
- Plaintext is simultaneously encrypted and integrity protected.
- Default TLS 1.2 algorithms: CCM and GCM
- No MAC key

```
AEADEncrypted = AEAD-Encrypt(write_key, nonce, plaintext,  
                                additional_data)
```

```
additional_data = seq_num + TLSCompressed.type +  
                  TLSCompressed.version + TLSCompressed.length;
```

# TLS 1.2 ATTACKS

- ROBOT
- Return Of Bleichenbacher's Oracle Threat
- Applies primarily to RSA encryption (not signatures)
- The RSA encryption in TLS 1.2 uses PKCS 1.5 padding
- Known padding oracle attack
- Countermeasures built in to TLS 1.2, rather than disabling
- Countermeasures are complicated, and many are vulnerable

# TLS 1.3

- Algorithms are ALL authenticated encryption
- Handshake messages after the ServerHello are now encrypted
- Key derivation based off Extract-and-Expand Key Derivation Function (HKDF)
- Compression, custom DHE groups, and DSA removed
- RSA padding now uses PSS

# TLS 1.3 HANDSHAKE

