Cipher Techniques

ECS 153 Spring Quarter 2021

Module 16

Problems

- Using cipher requires knowledge of environment, and threats in the environment, in which cipher will be used
  - Is the set of possible messages small?
  - Can an active wiretapper rearrange or change parts of the message?
  - Do the messages exhibit regularities that remain after encipherment?
  - Can the components of the message be misinterpreted?

Attack #1: Precomputation

- Set of possible messages M small
- Public key cipher f used
- Idea: precompute set of possible ciphertexts $f(M)$, build table $(m, f(m))$
- When ciphertext $f(m)$ appears, use table to find m
- Also called *forward searches*

Example

- Cathy knows Alice will send Bob one of two messages: enciphered BUY, or enciphered SELL
- Using public key e_{Bob}, Cathy precomputes

$$m_1 = \{ \text{BUY} \}\, e_{Bob},\ m_2 = \{ \text{SELL} \}\, e_{Bob}$$

- Cathy sees Alice send Bob m_2
- Cathy knows Alice sent SELL

May Not Be Obvious

- Digitized sound
  - Seems like far too many possible plaintexts, aa initial calculations suggest 2^{32} such plaintexts
  - Analysis of redundancy in human speech reduced this to about 100,000 ($\approx 2^{17}$), small enough for precomputation attacks

Misordered Blocks

- Alice sends Bob message
  - n_{Bob} = 262631, e_{Bob} = 45539, d_{Bob} = 235457
- Message is TOMNOTANN (191412 131419 001313)
- Enciphered message is 193459 029062 081227
- Eve intercepts it, rearranges blocks
  - Now enciphered message is 081227 029062 193459
- Bob gets enciphered message, deciphers it
  - He sees ANNNOTTOM, opposite of what Alice sent

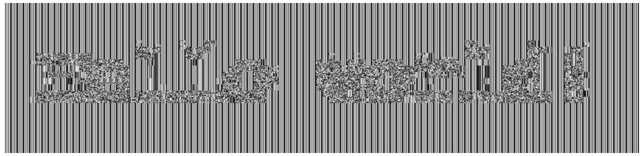
Solution

- Digitally signing each block won’t stop this attack
- Two approaches:
  - Cryptographically hash the *entire* message and sign it
  - Place sequence numbers in each block of message, so recipient can tell intended order; then sign each block

Statistical Regularities

- If plaintext repeats, ciphertext may too
- Example using AES-128:
  - Input image: 

  - corresponding output image:



  - Note you can still make out the words
- Fix: cascade blocks together (chaining) More details later

Type Flaw Attacks

- Assume components of messages in protocol have particular meaning
- Example: Otway-Rees:

$$\text{Alice} \xrightarrow{n \,||\, \text{Alice} \,||\, \text{Bob} \,||\, \{ r_1 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \} k_A} \text{Bob}$$

$$\text{Cathy} \xleftarrow[\{ r_2 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \} k_B]{n \,||\, \text{Alice} \,||\, \text{Bob} \,||\, \{ r_1 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \} k_A \,||} \text{Bob}$$

$$\text{Cathy} \xrightarrow{n \,||\, \{ r_1 \,||\, k_s \} k_A \,||\, \{ r_2 \,||\, k_s \} k_B} \text{Bob}$$

$$\text{Alice} \xleftarrow{n \,||\, \{ r_1 \,||\, k_s \} k_A} \text{Bob}$$

The Attack

- Ichabod intercepts message from Bob to Cathy in step 2
- Ichabod *replays* this message, sending it to Bob
    - Slight modification: he deletes the cleartext names
- Bob *expects* $n \,||\, \{ r_1 \,||\, k_s \}\, k_A \,||\, \{ r_2 \,||\, k_s \}\, k_B$
- Bob *gets* $n \,||\, \{ r_1 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \}\, k_A \,||\, \{ r_2 \,||\, n \,||\, \text{Alice} \,||\, \text{Bob} \}\, k_B$
- So Bob sees $n \,||\, \text{Alice} \,||\, \text{Bob}$ as the session key — and Ichabod knows this
- When Alice gets her part, she makes the same assumption
- Now Ichabod can read their encrypted traffic

Solution

- Tag components of cryptographic messages with information about what the component is
  - But the tags themselves may be confused with data …

What These Mean

- Use of strong cryptosystems, well-chosen (or random) keys not enough to be secure
- Other factors:
  - Protocols directing use of cryptosystems
  - Ancillary information added by protocols
  - Implementation (not discussed here)
  - Maintenance and operation (not discussed here)

Stream, Block Ciphers

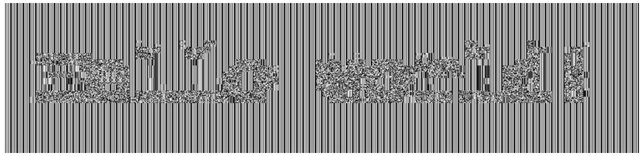
- *E* encipherment function
  - $E_k(b)$ encipherment of message *b* with key *k*
  - In what follows, $m = b_1 b_2 \ldots$, each b_i of fixed length
- Block cipher
  - $E_k(m) = E_k(b_1)E_k(b_2) \ldots$
- Stream cipher
  - $k = k_1 k_2 \ldots$
  - $E_k(m) = E_{k1}(b_1)E_{k2}(b_2) \ldots$
  - If $k_1 k_2 \ldots$ repeats itself, cipher is *periodic* and the kength of its period is one cycle of $k_1 k_2 \ldots$

Example

- AES-128
  - b_i = 128 bits, k = 128 bits
  - Each b_i enciphered separately using k
  - Block cipher

Block Ciphers

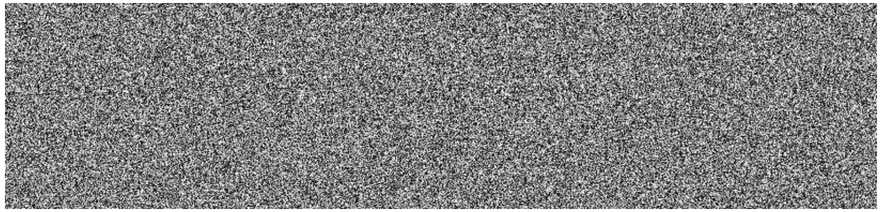
- Encipher, decipher multiple bits at once
- Each block enciphered independently
- Problem: identical plaintext blocks produce identical ciphertext blocks
- Plaintext image: 

- Ciphertext image: 

Solutions

- Insert information about block's position into the plaintext block, then encipher
- *Cipher block chaining*:
  - Exclusive-or current plaintext block with previous ciphertext block:
    - $c_0 = E_k(m_0 \oplus I)$
    - $c_i = E_k(m_i \oplus c_{i-1})$ for i > 0

    where *I* is the initialization vector
- Example encipherment of image on previous slide:



Authenticated Encryption

- Transforms message providing confidentiality, integrity, authentication simultaneously
- May be associated data that is not to be encrypted
  - Called Authenticated Encryption with Associated Data (AEAD)
- An examples:
  - Galois Counter Mode (GCM)
- *message* is part to be encrypted; *associated data* is part not to be encrypted
  - Both are authenticated and integrity-checked; if omitted, treat as having length 0

Galois Counter Mode (GCM)

- Can be implemented efficiently in hardware
- If encrypted, authenticated message is changed, new authentication value can be computed with cost proportional to number of changed bits
- Allows nonce (initialization vector) of any length
- Parameters
  - nonce *IV* up to 2^{64} bits; 96 bits recommended for efficiency reasons
  - message *M* up to $2^{39} - 2^{8}$ bits long; ciphertext C same length
  - associated data A up to 2^{64} bits long

GCM Notation

- Authentication value T is t bits long
- $M = M_0 \ldots M_n$, each block 128 bits long
  - M_n may not be complete block; call its length u bits
- $C = C_0 \ldots C_n$, each block 128 bits long; C is L_C bits long
  - Number of bits in C is the same as number of bits in M
- $A = A_0 \ldots A_m$, each block 128 bits long; A is L_A bits long
  - A_m may not be complete block; call its length v bits
- 0^x, 1^y mean x bits of 0 and y bits of 1, respectively

Multiplication in GF(2^{128})

```
/* multiply X and Y to produce Z in GF (2^128 ) */
function GFmultiply(X, Y: integer )
begin
        Z := 0
        V := X;
        for i := 0 to 127 do begin
                if  Yi  = 1 then Z := Z ⊕ V;
                V = rightshift(V, 1);
                if V127 = 1 then V := V ⊕ R;
        end
        return Z;
end
```

- This is written $Z = X \cdot Y$
- Y_i is ith leftmost bit of Y, so Y_{127} is the rightmost bit of Y
- rightshift(V, 1) means to shift V right 1 bit, and bring in 0 from the left
- R is bits 11100001 followed by 120 0 bits

GCM Hash Function

GHASH(H, A, C) computed as follows:

1. $X_0 = 0$
2. for $i = 1, \ldots, m-1$, $X_i = (X_{i-1} \oplus A_i) \cdot H$
3. $X_m = (X_{m-1} \oplus A_m) \cdot H$
    - A_m is right-padded with 0s if not a complete block
4. for $i = m+1, \ldots, m+n-1$, $X_i = (X_{i-1} \oplus C_i) \cdot H$
5. $X_{m+n} = (X_{m+n-1} \oplus C_n) \cdot H$
    - C_n is right-padded with 0s if not a complete block
6. $X_{m+n+1} = (X_{m+n} \oplus (L_A \mid\mid L_C)) \cdot H$
    - L_A, L_C left-padded with 0 bits to form 64 bits each

GCM Authenticated Encryption

This computes C and T:

1. $H = E_k(0^{128})$
2. If IV is 96 bits, $Y_0 = IV \,||\, 0^{31}1$; otherwise, $Y_0 = \mathrm{GHASH}(H, \nu, IV)$
   - ν empty string
3. for $i = 1, \ldots n$, $I_i = I_{i\text{-}1} + 1 \bmod 2^{32}$; set $Y_i = L_{i\text{-}1} \,||\, I_i$
   - $I_{i\text{-}1}$ right part of $Y_{i\text{-}1}$; treat it as unsigned 32 bit integer; $L_{i\text{-}1}$ left part of $Y_{i\text{-}1}$
4. for $i = 1, \ldots n\text{–}1$, $C_i = M_i + E_k(Y_i)$
5. $C_n = M_n + \mathrm{MSB}_u(E_k(Y_n))$
   - $\mathrm{MSB}_u(X)$ is u most significant (leftmost) bits of X
6. $T = \mathrm{MSB}_t(\mathrm{GHASH}(H, A, C) + E_k(Y_0))$

GCM Transmission and Decryption

- Send *C*, *T*
- To verify, perform steps 1, 2, 6, 3, 4, 5
- When authentication value is computed, compare to sent value
  - Note this is done *before* decrypting the message
  - If they do not match, return failure and discard messages

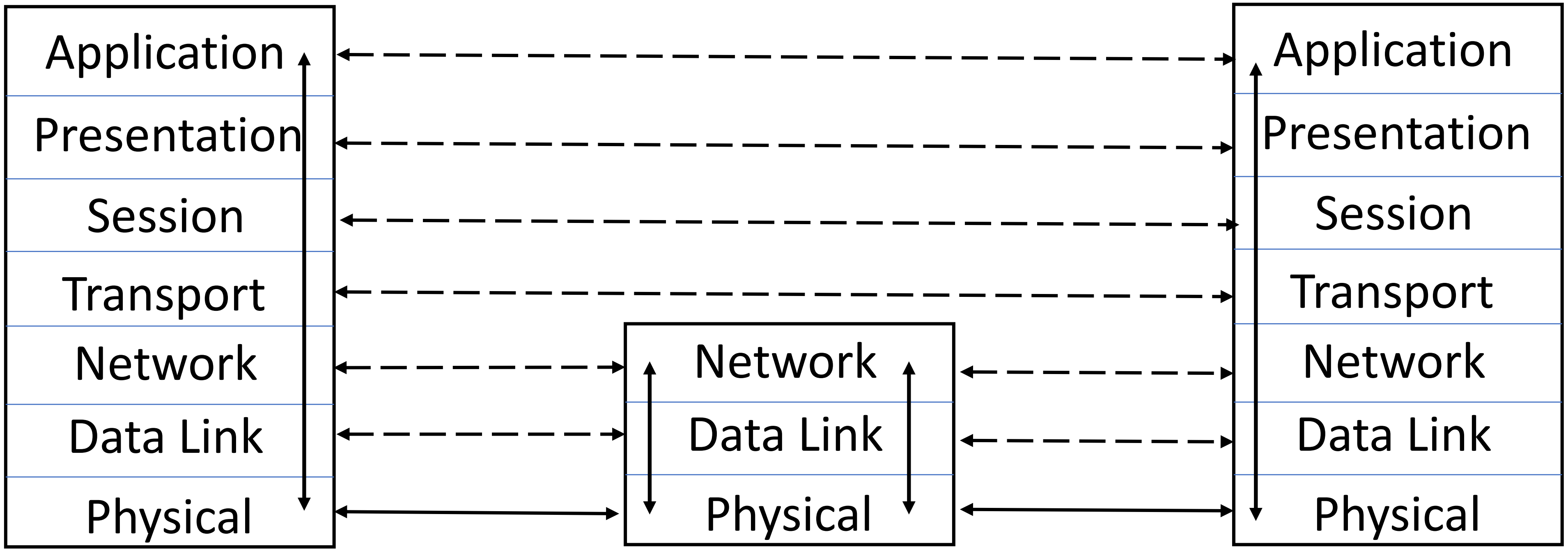
GCM Analysis

Strength depends on certain properties

- If *IV* (nonce) reused, part of *H* can be obtained
- If length of authentication value too short, forgeries can occur and from that, *H* can be determined (enabling undetectable forgeries)
- Under study is whether particular values of *H* make forging messages easier
- Restricting length of *IV* to 96 bits produces a stronger AEAD cipher than when the length is not restricted

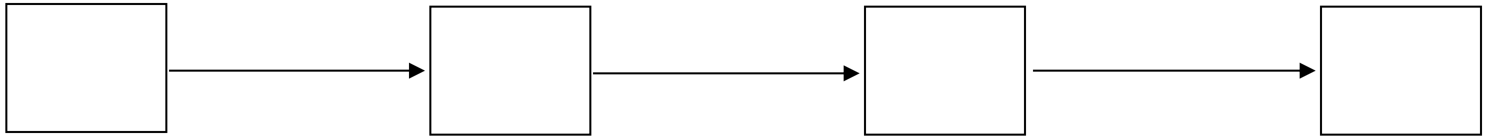
Networks and Cryptography

- ISO/OSI model
- Conceptually, each host communicates with peer at each layer

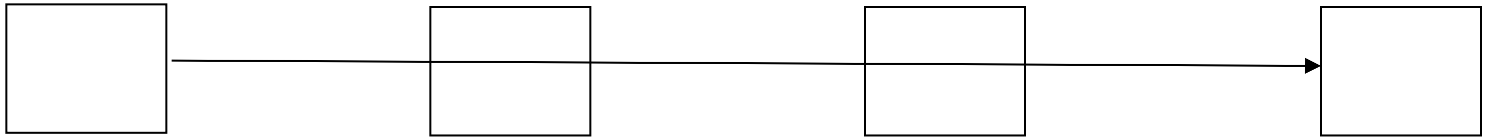


Link and End-to-End Protocols

Link Protocol



End-to-End (or E2E) Protocol



Encryption

- Link encryption
  - Each host enciphers message so host at “next hop” can read it
  - Message can be read at intermediate hosts
- End-to-end encryption
  - Host enciphers message so host at other end of communication can read it
  - Message cannot be read at intermediate hosts

Examples

- SSH protocol
  - Messages between client, server are enciphered, and encipherment, decipherment occur only at these hosts
  - End-to-end protocol
- PPP Encryption Control Protocol
  - Host gets message, deciphers it
    - Figures out where to forward it
    - Enciphers it in appropriate key and forwards it
  - Link protocol

Cryptographic Considerations

- Link encryption
  - Each host shares key with neighbor
  - Can be set on per-host or per-host-pair basis
    - Windsor, stripe, seaview each have own keys
    - One key for (windsor, stripe); one for (stripe, seaview); one for (windsor, seaview)
- End-to-end
  - Each host shares key with destination
  - Can be set on per-host or per-host-pair basis
  - Message cannot be read at intermediate nodes

Traffic Analysis

- Link encryption
  - Can protect headers of packets
  - Possible to hide source and destination
    - Note: may be able to deduce this from traffic flows
- End-to-end encryption
  - Cannot hide packet headers
    - Intermediate nodes need to route packet
  - Attacker can read source, destination

Example Protocols

- Securing Electronic Mail (OpenPGP, PEM)
  - Applications layer protocol
  - Start with PEM as goals, design described in detail; then lool at OpenPGP
- Securing Instant Messaging (Signal)
  - Applications layer protocol
- Secure Socket Layer (TLS)
  - Transport layer protocol
- IP Security (IPSec)
  - Network layer protocol

Transport Layer Security

- Internet protocol: TLS
  - Provides confidentiality, integrity, authentication of endpoints
  - Focus on version 1.2
- Old Internet protocol: SSL
  - Developed by Netscape for WWW browsers and servers
  - Use is deprecated

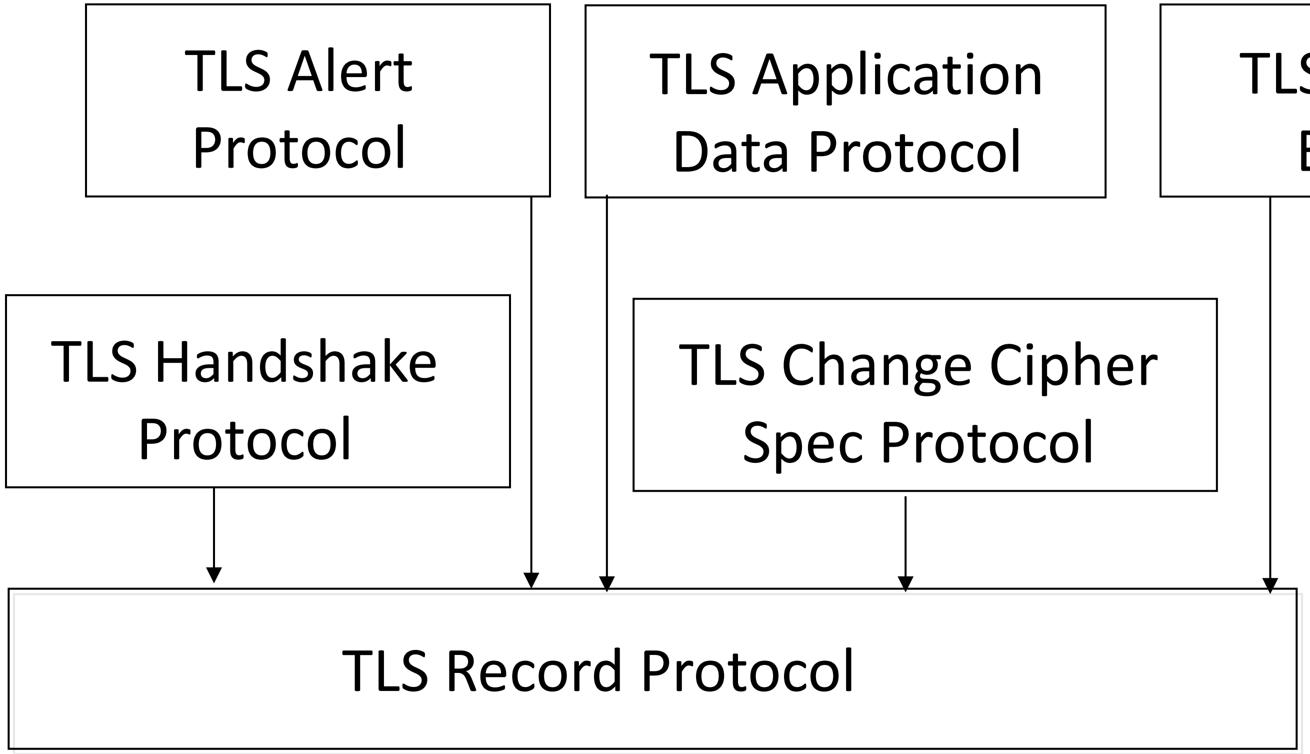
TLS Session

- Association between two peers
  - May have many associated connections
  - Information related to session for each peer:
    - Unique session identifier
    - Peer’s X.509v3 certificate, if needed
    - Compression method
    - Cipher spec for cipher and MAC
    - “Master secret” of 48 bits shared with peer
    - Flag indicating whether this session can be used to start new connection

TLS Connection

- Describes how data exchanged with peer
- Information for each connection
  - Whether a server or client
  - Random data for server and client
  - Write keys (used to encipher data)
  - Write MAC key (used to compute MAC)
  - Initialization vectors for ciphers, if needed
  - Sequence numbers for server, client

Structure of TLS



Supporting Cryptogrphy

- All parts of TLS use them
- Initial phase: public key system exchanges keys
  - Messages enciphered using classical ciphers, checksummed using cryptographic checksums
  - Only certain combinations allowed
    - Depends on algorithm for interchange cipher
  - Interchange algorithms: RSA, Diffie-Hellman

Diffie-Hellman: Types

- Diffie-Hellman: certificate contains D-H parameters, signed by a CA
  - DSS or RSA algorithms used to sign
- Ephemeral Diffie-Hellman: DSS or RSA certificate used to sign D-H parameters
  - Parameters not reused, so not in certificate
- Anonymous Diffie-Hellman: D-H with neither party authenticated
  - Use is "strongly discouraged" as it is vulnerable to attacks
- Elliptic curve Diffie-Hellman supports Diffie-Hellman and ephemeral Diffie-Hellman
    - But not anonymous Diffie-Hellman

Derivation of Master Secret

- *master_secret* = *PRF*(*premaster*, "master secret", $r_1 \,||\, r_2$)
  - *premaster* set by client, ˚sent to server during setup
  - r_1, r_2 random numbers from client, server respectively
- *PRF*(*secret*, *label*, *seed*) = P_*hash*(*secret*, *label* || *seed*)
- P_*hash*(*secret*, *seed*) = HMAC_*hash*(*secret* || *A*(1) || *seed*) ||
  HMAC_*hash*(*secret* || *A*(2) || *seed*) ||
  HMAC_*hash*(*secret* || *A*(3) || *seed*) || …
  - Use first 48 bits of output to set *PRF*
- *A*(0) = *seed*, *A*(*i*) = HMAC_*hash*(*secret*, *A*(*i*-1)) for $i > 0$

Derivation of Keys

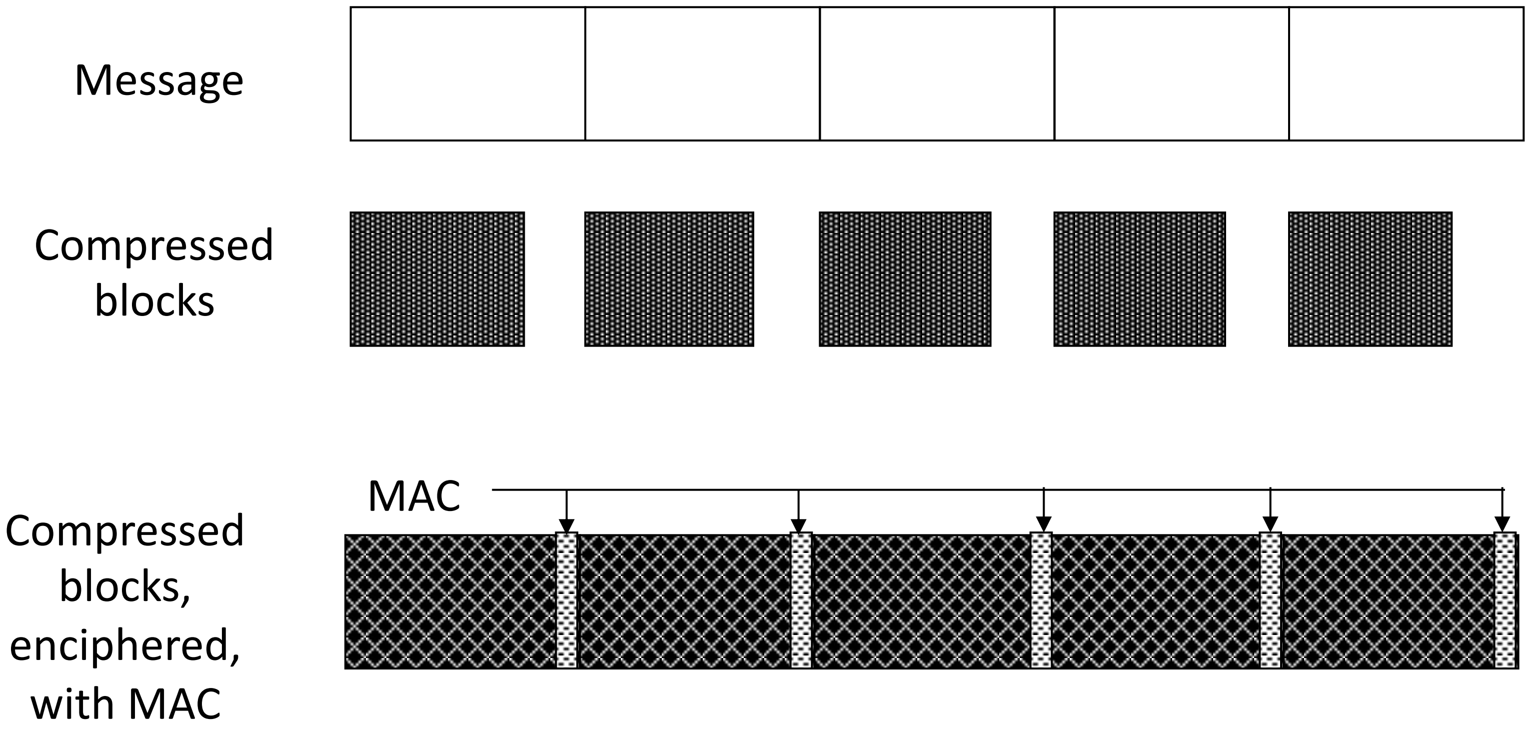
- *key_block* = PRF(*master*, “key expansion”, $r_1 \| r_2$)
  - r_1, r_2 as before
- Break it into blocks of 48 bits
  - First two are client, server keys for computing MACs
  - Next two are client, server keys used to encipher messages
  - Next two are client, server initialization vectors
    - Omitted if cipher does not use initialization vector

MAC for Block

hash(*MAC_ws*, *seq* || *TLS_comp* || *TLS_vers* || *TLS_len* || *block*)

- *MAC_ws*: MAC write key
- *seq*: sequence number of *block*
- *TLS_comp*: message type
- *TLS_vers*: TLS version
- *TLS_len*: length of *block*
- *block*: block being sent

TLS Record Layer



Record Protocol Overview

- Lowest layer, taking messages from higher
  - Max block size 2^{14} = 16,384 bytes
  - Bigger messages split into multiple blocks
- Construction
  - Block b compressed; call it b_c
  - MAC computed for b_c
    - If MAC key not selected, no MAC computed
  - b_c, MAC enciphered
    - If enciphering key not selected, no enciphering done
  - TLS record header prepended

TLS Handshake Protocol

- Used to initiate connection
  - Sets up parameters for record protocol
  - 4 rounds
- Upper layer protocol
  - Invokes Record Protocol
- Note: what follows assumes client, server using RSA as interchange cryptosystem

Overview of Rounds

1. Create TLS connection between client, server
2. Server authenticates itself
3. Client validates server, begins key exchange
4. Acknowledgments all around

Handshake Round 1

1. Client $\xrightarrow{\{ v_C \mid\mid r_1 \mid\mid s_1 \mid\mid ciphers \mid\mid comps \mid\mid ext_C \}}$ Server

2. Client $\xleftarrow{\{ v \mid\mid r_2 \mid\mid s_2 \mid\mid cipher \mid\mid comp \mid\mid ext \}}$ Server

v_C	Client's version of TLS
v	Highest version of TLS that client, server both understand
r_1, r_2	nonces (timestamp and 28 random bytes)
s_1	Current session id (empty if new session)
s_2	Current session id (if s_1 empty, new session id)
ciphers	Ciphers that client understands
comps	Compression algorithms that client understand
cipher	Cipher to be used
comp	Compression algorithm to be used
ext_C	List of extensions client supports
ext	List of extensions server supports (subset of ext_C)

Handshake Round 2

3. Client $\leftarrow$ { *certificate chain* } — Server

4. Client $\leftarrow$ $\{ p \mid\mid g \mid\mid K_S \mid\mid \{ h(r_1 \mid\mid r_2 \mid\mid p \mid\mid g \mid\mid K_S) \} k_S \}$ — Server

5. Client $\leftarrow$ {*ctype* || *sigalgs* || *gca* } — Server

6. Client $\leftarrow$ { *server_hello_done* } — Server

If server not going to authenticate itself, only last message sent

Second step is for Diffie-Hellman with RSA certificate

Third step omitted if server does not need client certificate

K_S, k_S	Server's Diffie-Hellman public, private keys
ctype	Certificate type accepted (by cryptosystem)
sigalgs	List of hash, signature algorithm pairs server can use
gca	Acceptable certification authorities

Handshake Round 3

7. Client $\xrightarrow{\{\ client\_certificate\ \}}$ Server

8. Client $\xrightarrow{\{\ pre\ \}\ K_S}$ Server

9. Client $\xrightarrow{\{\ hash(all\ previous\ messages)\ \}\ k_C}$ Server

pre	Premaster secret
K_S	Server's public key
k_C	Client's private key

Handshake Round 4

10. Client —— *change_cipher_spec* ——→ Server

11. Client —— { *PRF*(*master* || “client finished” || *hash*(*all previous messages*) } ——→ Server

12. Client ←—— *change_cipher_spec* —— Server

13. Client ←—— { *PRF*(*master* || “server finished” || *hash*(*all previous messages*) } —— Server

change_cipher_spec	Begin using cipher specified

TLS Change Cipher Spec Protocol

- Send single byte
- In handshake, new parameters considered "pending" until this byte received
  - Old parameters in use, so cannot just switch to new ones

TLS Alert Protocol

- Closure alert
  - Sender will send no more messages
  - Pending data delivered; new messages ignored
- Error alerts
  - Warning: connection remains open
  - Fatal error: connection torn down as soon as sent or received

TLS Heartbeat Extension

- Message has 4 fields
  - Value indicating message is request
  - Length of data in message
  - Data of given length
  - Random data
- Message sent to peer; peer replies with similar message
  - If second field is too large (> 214 bytes), ignore message
  - Reply message has same data peer sent, new random data
- When peer sends this for the first time, it sends nothing more until a response is received

TLS Application Data Protocol

- Passes data from application to TLS Record Protocol layer

Differences Between TLSv2 and SSLv3

- SSLv3 master secret computed differently

  master = MD5(*premaster* || SHA('A' || *premaster* || r_1 || r_2) ||
  MD5(*premaster* || SHA('BB' || *premaster* || r_1 || r_2) ||
  MD5(*premaster* || SHA('CCC' || *premaster* || r_1 || r_2)

- SSLv3 key block also computed differently

  key_block = MD5(*master* || SHA('A' || *master* || r_1 || r_2) ||
  MD5(*master* || SHA('BB' || *master* || r_1 || r_2) ||
  MD5(*master* || SHA('CCC' || *master* || r_1 || r_2) || . . .

Differences Between TLSv2 and SSLv3

SSLv3 MAC for each block computed differently:

hash(*MAC_ws* || *opad* ||

hash(*MAC_ws* || *ipad* || *seq* || *SSL_comp* || *SSL_len* || *block*))

- *hash*: hash function used
- *MAC__ws*, *seq*, *SSL_comp*, *SSL_len*, *block*: as for TLS (with obvious changes)
- *ipad, opad*: as for HMAC

Differences Between TLSv2 and SSLv3

- Verification message (9, above) is different:

9′. Client $\xrightarrow{\{\ \mathit{hash}(\mathit{master} \,||\, \mathit{opad} \,||\, \mathit{hash}(\mathit{all\ previous\ messages} \,||\, \mathit{master} \,||\, \mathit{ipad}))\ \}}$ Server

- Messages after change cipher spec (11, 13 above) are also different:

{ *hash*(*master* || *opad* ||
hash(*all previous messages* || 0x434C4E54 || *master* || *ipad*)) }

11′. Client ⟶ Server

{ *hash*(*master* || *opad* ||
hash(*all previous messages* || 0x53525652 || *master* || *ipad*)) }

13′. Client ⟶ Server

Differences Between TLSv2 and SSLv3

- Different sets of ciphers
  - SSL allows use of RC4, but its use is deprecated
  - SSL allows set of ciphers for the Fortezza cryptographic token used by the U.S. Department of Defense

Problems with SSL

- POODLE attack focuses on padding of messages
  - In SSL, all but the last byte of the padding are random and so cannot be checked
- How padding works (assume block size of *b*):
  - Message ends in a full block: add additional block of padding, and last byte is the number of bytes of random padding ($b - 1$)
  - Message ends in part of a block: add random bytes out to last byte, set that to number of random bytes (so if block is $b - 1$ bytes, one padding byte added and it is 0)

The POODLE Attack

- Peer receives incoming ciphertext message $c_1, \ldots, c_n$
- Peer decrypts it to $m_1, \ldots, m_n$: $m_i = D_k(c_i) \oplus c_{i-1}$, where c_0 is initialization vector
  - Validates by removing padding, computes and checks MAC over remaining bytes
- Attacker replaces c_n with some earlier block, say c_j, $j \neq n$
  - If last byte of c_j is same as c_n, message accepted as valid; otherwise, rejected
- So attacker arranges for HTTP messages to end with known number of padding bytes
  - Then server should accept changed message in at least 1 out of 256 tries

Example POODLE Attack

- Here's HTTP request (somewhat simplified):

  GET / HT TP/1.1\r\n Cookie: abcdefgh \r\n\r\nxxxx MAC •••••••7

- Attacker cannot see plaintext
- Run Javascript in browser that duplicates cookie block and overwrites last block
  - It's enciphered using (for example) 3DES-CBC
- You see enciphered block
  - If it is accepted, then plaintext block xor'ed with previous ciphertext block ends in 7

SSL, TLS, and POODLE

- POODLE serious enough that SSL is being discarded in favor of TLS
- TLS not vulnerable, as all padding bytes set to length of padding
    - And TLS implementations must check this padding (all of it) for validity before accepting messages