FIT5037: Network Security
Security at Transport layer

Faculty of Information Technology
Monash University

April 17, 2019

Copyright Notice

Lecture 7: Security at Transport layer

Lecture Topics:

- Symmetric key cryptography
- Asymmetric key cryptography
- Pseudorandom Number Generators and hash functions
- Authentication Methods and AAA protocols
- Security at Network layer (IPsec)
- Security at Network layer (firewalls and wireless security)
- **Security at Transport layer**
- Security at Application layer
- Computer system security and malicious code
- Computer system vulnerabilities and penetration testing
- Intrusion detection
- Denial of Service Attacks and Countermeasures / Revision

Outline

- SSL/TLS Overview
- TLS 1.3 Protocol
- Attacks on SSL 3.0 and TLS 1.0
- DTLS Protocol Overview

SSL/TLS Overview

- A new layer inserted between transport layer and application layer therefore capable of protecting communication from any application protocol above TCP
- Originally developed by Netscape
- Version 3 was designed with public input
- Subsequently became Internet standard known as TLS (Transport Layer Security)
- The first version of TLS is essentially SSLv3.1
  - it evolved into TLS v1.0 specified in RFC 2246
  - TLS v1.2 is widely in use and is defined in RFC 5246
  - TLS v1.3, is the latest version which obsoletes previous versions, and is defined in RFC 8446 provides backward compatibility with previous versions

<table>
<tr><td>HTTP</td><td>FTP</td><td>SMTP</td></tr>
<tr><td colspan="3">SSL / TLS</td></tr>
<tr><td colspan="3">TCP</td></tr>
<tr><td colspan="3">IP</td></tr>
<tr><td colspan="3">Data Link</td></tr>
<tr><td colspan="3">Physical</td></tr>
</table>

SSL/TLS Services

The following services are provided by TLS:

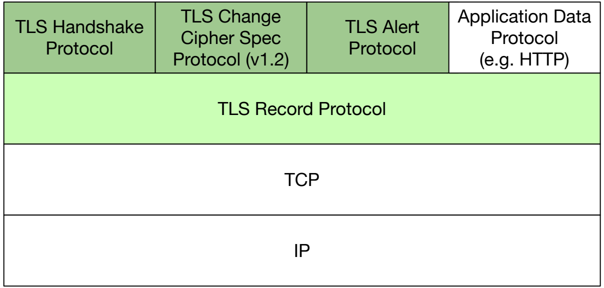
- Authentication:
  - the server side is always authenticated
  - the client side is optionally authenticated
  - can use the following methods
    - asymmetric cryptography (e.g. RSA, ECDSA, EdDSA)
    - symmetric cryptography (pre-shared key)
- Confidentiality:
  - data items transferred in the session are encrypted to protect against eavesdropping
    - symmetric key algorithms
- Integrity:
  - messages are protected with MAC

SSL/TLS Architecture

The TLS protocol is composed of two primary components:

- TLS Handshake Protocol
  - authentication of parties (server and client)
  - negotiation of cryptographic parameters
  - establishing shared cryptographic keys
    - negotiation is integrity-protected
- TLS Record Protocol
  - layered on top of a reliable transport protocol (TCP layer, TCP connection)
  - encapsulates higher-level protocols (including Handshake Protocol messages)
  - provides connection security:
    - confidentiality (symmetric cryptography)
    - integrity (MAC)
    - secret keys are negotiated and established by TLS Handshake Protocol

SSL/TLS Protocol Stack



- TLS Record protocol encapsulates higher layer protocol messages
- four protocol use the record protocol:
  - TLS Handshake Protocol
  - TLS Change Cipher Spec Protocol (TLS 1.2 and below)
  - TLS Alert Protocol
  - Application Data Protocol

TLS 1.3 Components: Handshake Protocol

allows peers to

- negotiate a protocol version
- select cryptographic algorithms
- optionally authenticate each other
- establish shared secret keying material

Handshake message structure:

- Type: 1 byte specifying the message type
- Length: 3 bytes indicating the message length
- Content: handshake message

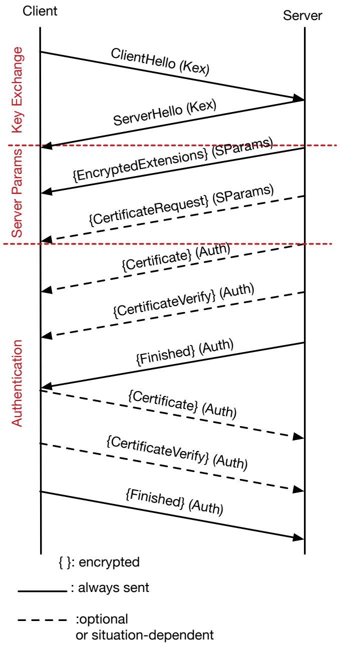
1 byte	3 bytes	
Type	Length	Content

TLS 1.3 Handshake Protocol: Overview

TLS 1.3 handshake exchange three sets of messages

1. Key Exchange
   - establish shared keying material
   - select cryptographic parameters
2. Server Parameters
   - establish other handshake parameters
     - e.g. whether client is authenticated, application-layer protocol support etc.
3. Authentication
   - authenticate the server
   - optionally authenticate client
   - key confirmation and handshake integrity

- Handshake messages must be sent in order
  - a peer that receives an out of order handshake message must abort the handshake



TLS 1.3 Handshake Protocol: Key Exchange

Negotiation starts with `ClientHello` message which includes

- a 32-byte random value
- a list of cipher suites indicating AEAD/HKDF hash pairs supported by client in descending order of preference
- a `supported_groups` extension indicating DHE or ECDHE groups supported by the client
  - optionally a `key_share` extension containing (EC)DHE shares for some or all of the supported groups
- a `signature_algorithms` extension indicating the supported signature algorithms
- a `pre_shared_key` extension containing a list of symmetric key identities known to the client
  - and a `psk_key_exchange_modes` extension indicating the key exchange modes that may be used with PSKs

Server responds with `ServerHello` message containing

- a 32-byte random value
- a selected cipher suite from the proposed list by the client
- an (EC)DHE group and server public key in a `key_share` extension
  - if client has not offered a `key_share` extension for selected group server will send `HelloRetryRequest`

TLS 1.3 Handshake Protocol: Server Parameters and Server Authentication

- Server continues with Server Parameters messages encrypted with keys derived from `server_handshake_traffic_secret`
  - an `EncryptedExtensions` containing the extensions
    - must not contain any extension associated with individual certificates
    - if client finds such extensions it must abort the handshake
  - an optional `CertificateRequest` to authenticate client
- the Server Authentication messages encrypted with keys derived from `server_handshake_traffic_secret`
  - a signature algorithm/certificate pair to authenticate itself to the client in `Certificate` message
    - if a PSK method is used then the server will select a key establishment mode from the list proposed by client
  - a `CertificateVerify` message containing a signature over the `Handshake Context` and the `Certificate`
    - `Handshake Context` consists of previous handshake messages
  - a `Finished` message containing a MAC over `Handshake Context`, `Certificate`, and `CertificateVerify`

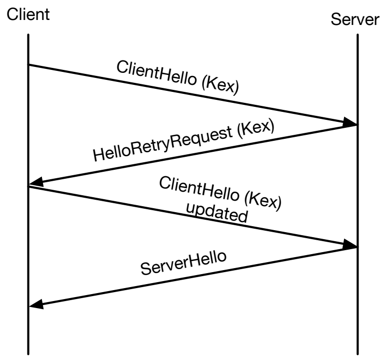
TLS 1.3 Handshake Protocol: Client Authentication

Client responds with authentication messages encrypted with keys derived from `client_handshake_traffic_secret`

- an optional `Certificate`
- an optional `CertificateVerify` message containing a signature over the `Handshake Context` and the `Certificate`
- a `Finished` message containing a MAC over `Handshake Context`
  - including `Certificate` and `CertificateVerify` if present
- the `Finished` messages provide
  - key confirmation
  - binds the endpoint's identity to the exchanged keys
  - in case of PSK mode also authenticates the handshake

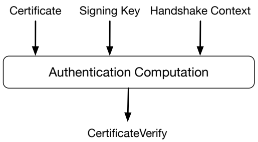
TLS 1.3 Handshake Protocol: `HelloRetryRequest` and updated `ClientHello`

- server will send `HelloRetryRequest`
  - if `ClientHello` contains acceptable set of parameters
  - but client has not offered a `key_share` extension for selected group by server
- `HelloRetryRequest` has the same format as the `ServerHello` message
- client processes the message
  - will abort the handshake if the `HelloRetryRequest` results in no changes in `ClientHello`
  - if accepted cipher suite was not offered by the client it will abort the handshake
- client sends an updated `ClientHello`
  - other fields will be the same
- the negotiation continues with `ServerHello`

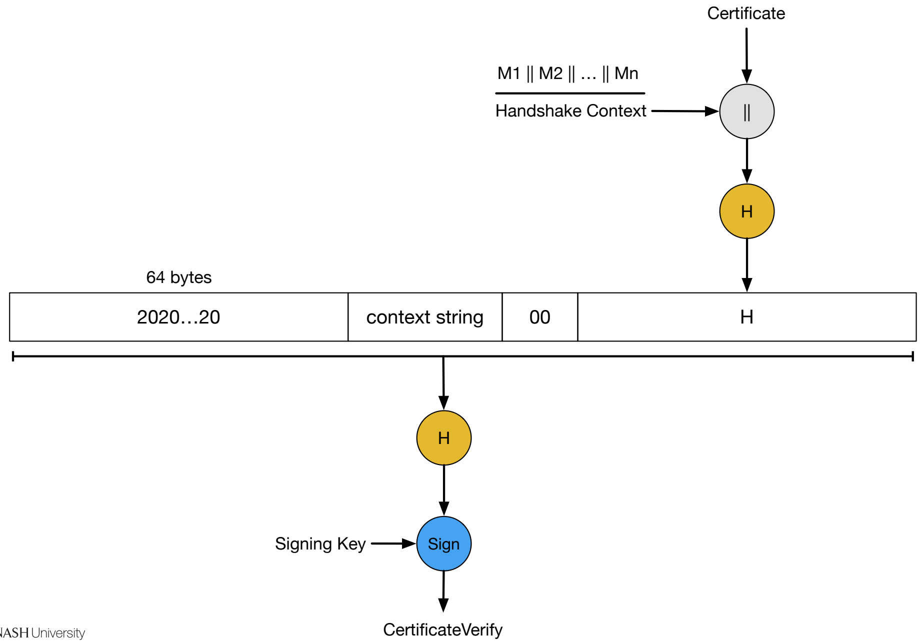


TLS 1.3 Handshake Protocol Authentication Messages: `CertificateVerify`

- inputs to authentication computation
  - the certificate and signing key
  - a `Handshake Context` consisting of the set of messages to be included in the *transcript hash*
- `Certificate`: the certificate to be used for authentication and any supporting certificates in the chain
  - the `server_name` and `certificate_authorities` extensions are used to guide certificate selection
  - self-signed and certificates expected to be trust anchors are not validated as part of the chain
- `CertificateVerify`: a signature over the value
  - $\overbrace{2020\ldots20}^{\text{64 bytes}}$ `|| context string || 00 || Transcript-Hash(Handshake Context, Certificate)`



TLS 1.3 Handshake Protocol: `CertificateVerify`

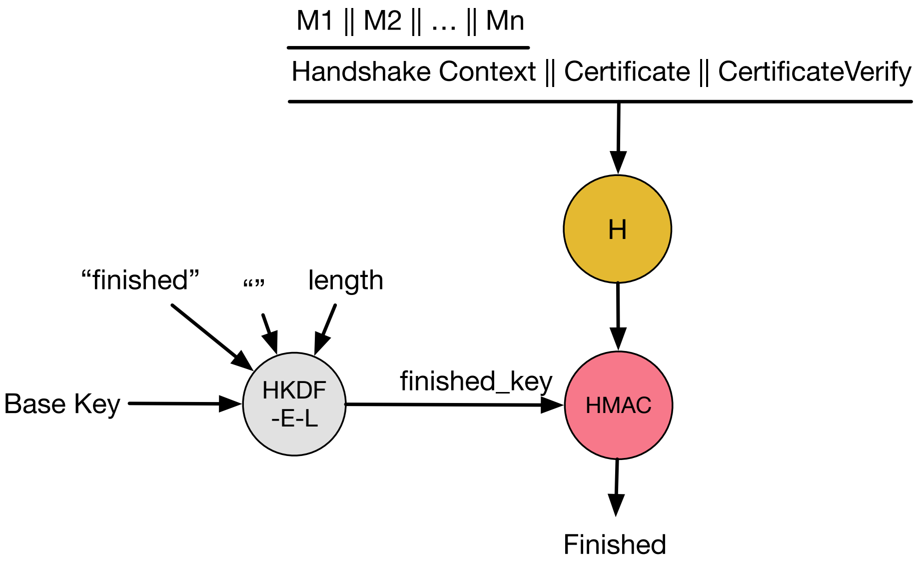


TLS 1.3 Handshake Protocol Authentication Messages: `Finished`

- `Finished`: a MAC over the value `Transcript-Hash(Handshake Context, Certificate, CertifiacteVerify)` using a MAC key derived from the `Base Key`

Mode	Handshake Context	Base Key
Server	ClientHello … later of EncryptedExtensions/CertificateRequest	`server_handshake_traffic_secret`
Client	ClientHello … later of later of server Finished/EndOfEarlyData	`client_handshake_traffic_secret`

TLS 1.3 Handshake Protocol Authentication Messages: `Finished` (diagram)



TLS 1.3 Handshake Protocol: `Transcript-Hash`

- hash the concatenation of
  - each included handshake message
  - including the handshake message header (type+length)
  - but excluding Record Layer headers
- for instance:

```
Transcript-Hash(M1, M2, ... Mn) = Hash(M1 || M2 || ... || Mn)
```

- the transcript hash starts with the first `ClientHello` and only includes messages that were sent
- the exception to the general rule is `HelloRetryRequest` where
  - `ClientHello1` is replaced with `Hash(ClientHello1)` and the message type is set to a special type: `message_hash`
  - this allows for a stateless `HelloRetryRequest` on the server side

Key Derivation Functions

- uses `HKDF-Extract` and `HKDF-Expand` functions defined in RFC 5869: HMAC-based Extract-and-Expand Key Derivation Function
- defines the data structure `HkdfLabel` $= \overbrace{\text{Length}}^{\text{2 bytes}} \;||\; \overbrace{\text{"tls13 "} + \text{Label}}^{\text{7 to 255 bytes}} \;||\; \overbrace{\text{Context}}^{\text{0 to 255 bytes}}$
- defines two functions:

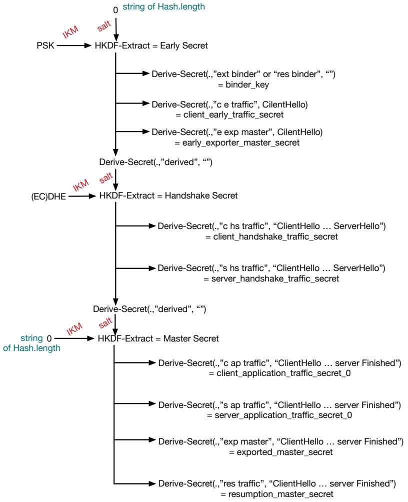
```
HKDF-Expand-Label(Secret, Label, Context, Length):
    HkdfLabel = Length || "tls13 " || Label || Context
    return HKDF-Expand(Secret, HkdfLabel, Length)

Derive-Secret(Secret, Label, Messages):
    return HKDF-Expand-Label(Secret, Label, Transcript-Hash(Messages), Hash.length)
```

- the hash function used in both HKDF and Transcript-Hash is the one accepted in the cipher suite
- the `Messages` is the concatenation of the indicated handshake messages

Deriving Keying Materials

```
HKDF-Extract(salt, IKM, Hash):
    PRK = HMAC-Hash(salt, IKM)
    return PRK
```

- `HKDF-Extract` function
  - accepts two input values
    - `salt`: a non-secret random value
    - `IKM`: input keying material
  - select a hash function to be used in HMAC
  - returns `PRK`: pseudo-random key
- `HKDF-Extract` inputs in TLS 1.3
  - `salt`: current secret state
  - `IKM`:
    - in PSK: the pre-shared key or the `resumption_master_secret` from a previous connection
    - in (EC)DHE: the calculated shared secret
- `Derive-Secret` is then used
  - with the key derived from `HKDF-Extract` as the `Secret`
  - proper label for client or server handshake
  - handshake messages



Traffic Key Calculation

- traffic key material is generated from
  - a secret value
  - a label describing the purpose
  - the length of the key

```
[sender]_write_key = HKDF-Expand-Label(Secret, "key", "", key_length)
[sender]_write_iv = HKDF-Expand-Label(Secret, "iv", "", iv_length)
```

Mode	**Secret**
0-RTT Application	client_early_traffic_secret
Handshake	[sender]_handshake_traffic_secret
Application Data	[sender]_application_traffic_secret_N

Key Usage Limits and Updating Traffic Secrets

- the limits are specified under the assumption that underlying primitive (AES or CHACHA20) has no weakness
- `AES-GCM`: $2^{24.5}$ full-size records (about 24 million)
- `ChaCha20/Poly1305`: the record sequence number would wrap before safety limit is reached
- after handshake completion each side can change its sending traffic keys with `KeyUpdate` handshake message
- the next generation of traffic keys is computed as

```
application-traffic_secret_N+1 =
    HKDF-Expand-Label([sender]_application_traffic_secret_N,
                                   "traffic upd", "", Hash.length)
```

TLS 1.3 Record Protocol: Overview

takes messages to be transmitted:

- fragments data into manageable blocks
  - 2^{14} bytes or less
- protects the records
- transmits the result

TLS records are typed

- allows multiple higher-level protocols to be multiplexed over the same record layer
  - `handshake`
  - `application_data`
  - `alert`
  - `change_cipher_spec`: only for compatibility

received data

- is verified
- decrypted
- reassembled and then delivered to higher-level clients

1 byte	2 bytes	2 bytes	
Type	Version (legacy)	Length	Content

TLS 1.3 Record Protocol: Record Payload Protection

- the record protection functions translate a `TLSPlaintext` structure into a `TLSCiphertext` structure
  - the de-protection function reverses the process
- in TLS 1.3 (as opposed to previous versions) all ciphers are modeled as AEAD
  - each encrypted record consists of:
    - a plaintext header
    - encrypted body consisting of a type and optional padding
  - the additional data (for AEAD) is the record header
  - nonce is derived from the sequence number and `[sender]_write_iv`

```
AEADEncrypted = AEAD-Encrypt([sender]_write_key, nonce, additional_data, plaintext)
```

at receiver

```
plaintext = AEAD-Decrypt(peer_write_key, nonce, additional_data, AEADEncrypted)
```

- if decryption fails the receiver must terminate the connection with a `bad_record-mac`

TLS 1.3 Record Protocol: Per-Record Nonce

- the sequence number is a 64-bit number maintained separately for reading and writing records
- is set to zero at beginning of a connection and incremented by 1 for each record
- sequence number must not wrap
  - either rekey if it wraps
  - or terminate the connection
- each AEAD algorithm specifies a range of possible values for per-record nonce: `N_MIN` bytes to `N_MAX` bytes
  - algorithms with `N_MAX` less than 8 bytes must not be used
  - the length of per-record nonce is set to the larger of 8 bytes and `N_MIN`
- per-record nonce:
  - the 64-bit sequence number is padded with zero to the left to `iv_length`
  - the result is then XORed with `[sender]_write_iv`

TLS 1.3 Alert Protocol: Overview

- Alert messages convey
  - a description of the alert
  - a (legacy) security level
- two classes:
  - Closure Alerts: indicate orderly closure of one direction of the connection
    - TLS implementation should indicate `end-of-data` to the application
  - Error Alerts: indicate abortive closure of the connection
    - TLS implementation should indicate an error to the application
    - must not allow any further data to be sent or received on the connection
    - server and client must forget the secret values and keys established
- examples of each class:
  - Closure Alerts: `close_notify` and `user_canceled`
  - Error Alerts: `unexpected_message`, `bad_record_mac`, `record_overflow`, `handshake_failure`, `bad_certificate`, etc.

Summary of Major Differences of TLS 1.3 with 1.2

- updated list of supported encryption algorithms
  - only supports AEAD
  - removes legacy algorithms
  - cipher suite representation is changed
    - to separate authentication and key exchange from the record protection algorithm
    - hash is added to be used in key derivation and handshake MAC
- static RSA and (static) DH cipher suites are removed from key exchange
  - all public key based key exchange methods provide forward secrecy
- all handshake messages after `ServerHello` are encrypted (see Appendix B for TLS 1.2 handshake)
- handshake state machine is restructured
  - `ChangeCipherSpec` is deprecated
- new key derivation design
- EC methods are added to the base specification and RSA padding method is updated to RSASSA-PSS
  - point representation negotiation is deprecated and single point representation is used per curve
- version negotiation mechanism is deprecated and replaced with version list in an extension
- a new PSK exchange replaces session resumption and PSK-based cipher suites

Case Study: Chosen Ciphertext and Padding Oracle Timing-Side Channel Attacks

Case Study: Attack on SSL 3.0[1]

- Consider a MAC-then-Encrypt method with CBC mode of operation
- to encrypt:
  - first generate the tag t then pad $m||t$ and encrypt $m||t||padding$
  - SSL 3.0 padding of CBC: when a message require p bytes of padding add $p-1$ arbitrary bytes plus one last byte containing $p-1$
- to decrypt:
  - decrypt the ciphertext
  - remove padding: read the last byte and remove as many bytes from the end
  - verify the tag t

[1] This POODLE bites: exploiting the SSL 3.0 fallback, Moller, Bodo and Duong, Thai and Kotowicz, Krzysztof, Security Advisory, 2014

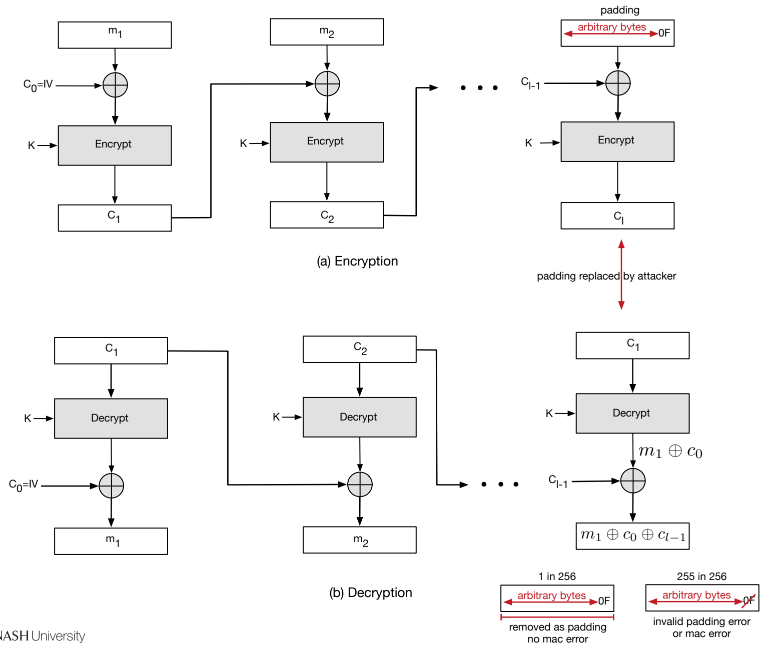
Chosen Ciphertext Attack on SSL 3.0[2]

the attacker learns something about last byte of m_1

- consider the ciphertext where length of $m||t$ is a multiple of block size (say 16 bytes for AES):
  - $c = \underbrace{c_0}_{IV}, \underbrace{c_1, c_2, \ldots}_{\text{encryption of m}} \underbrace{\ldots, c_{l-1}}_{\text{encrypted tag}}, \underbrace{c_l}_{\text{encrypted pad}}$
- the attacker prepares a chosen cipher text $\hat{c}$ as follows:
  - the attacker removes the last block which is the encrypted pad and replaces that with c_1
  - $\hat{c} = c_0, c_1, \ldots, c_{l-1}, c_1$
- by definition of CBC, decryption of the last block of $\hat{c}$ results in
  - $v = D(k, c_1) \oplus c_{l-1} = m_1 \oplus c_0 \oplus c_{l-1}$
- one of the following cases will occur:
  - the last byte of v is 15 hence the added block will be removed and $\hat{c}$ will be accepted
    - the tag and message is unchanged and is a valid pair
    - in this case the last byte of m_1 is $0F \oplus c_0[15] \oplus c_{l-1}[15]$
  - the last byte of v is not 15 and the attacker will receive an error

2 Section 9.4.2 of "A Graduate Course in Applied Cryptography" by Dan Boneh and Victor Shoup

Chosen Ciphertext Attack on SSL 3.0 (diagram)



A Complete Break of SSL 3.0

consider a Web browser and a target Web server `bank.com` where the browser and server share a secret: `cookie` which is included in every request sent from browser to the server

- the abstract requests would look like: `GET path  cookie: cookie`
  - the `path` identifies the name of a requested resource
- the attacker's goal is to recover the secret `cookie`

1. the attacker makes the browser visit `attacker.com` where it sends a Javascript to the browser (e.g. an ad on a web site the victim visits)
2. the script makes the browser request `/AA` from `bank.com` (to ensure the length of the message and tag is a multiple of block size)
3. the request `GET /AA  cookie: cookie` is encrypted using SSL 3.0
4. the attacker intercepts the ciphertext c and mounts a CCA sending $\hat{c}$ and observe if error happens
   - 1 in 256 chance of success
   - attacker can cause browser to repeatedly issue request giving attacker fresh encryptions to learn one byte of `cookie`

A Complete Break of SSL 3.0 (continued)

5. after learning one byte of the `cookie`, shift the `cookie` one byte to the right by requesting `GET /AAA  cookie: cookie`
   - the attack is repeated to learn the second to last byte of `cookie`
6. repeat the attack to learn all bytes of the `cookie`

Padding Oracle Timing Side-Channel Attack on TLS 1.0[3]: Overview

- exploits a naive implementation of MAC-then-Encrypt decryption
  - CBC decryption recovers $m||t||pad$
  - checks whether pad is valid
    - rejects the cipher text if invalid
  - checks the integrity tag and if valid returns m
- problem: only checks tag if pad is valid
  - takes less time to reject invalid pad
  - takes more time to reject invalid tag
- attacker can measure time to learn if pad or tag was invalid

[3] Section 9.4.3 of "A Graduate Course in Applied Cryptography" by Dan Boneh and Victor Shoup

Padding Oracle Timing Side-Channel Attack on TLS 1.0[5]: Further Detail

- suppose an attacker intercepts an encrypted TLS 1.0 record c
- attacker wishes to check if last byte of m_2 is equal to byte value b

1. attacker chooses an arbitrary 16-byte value B where the last byte is equal to b
2. creates $\hat{c}_1 = c_1 \oplus B$
3. sends $\hat{c} = c_0, \hat{c}_1, c_2$ to the server
    - after CBC decryption of $\hat{c}$, $\hat{m}_2 = \hat{c}_1 \oplus D(K, c_2) = m_2 \oplus B$
    - if the last byte of m_2 is equal to b then the last block of $\hat{m}_2$ is zero which is a valid pad
        - the server will attempt to verify tag which will take longer
    - if the last byte of m_2 is **not** equal to b then the pad will likely be invalid
        - takes less time to receive an error

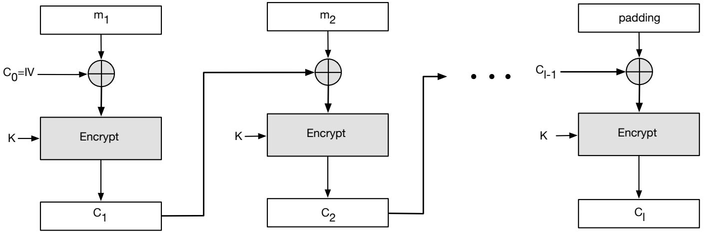
- Another example of oracle timing side-channel attack is published as Lucky13[4]

---

[4] Lucky Thirteen: Breaking the TLS and DTLS Record Protocols," 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, 2013, pp. 526-540

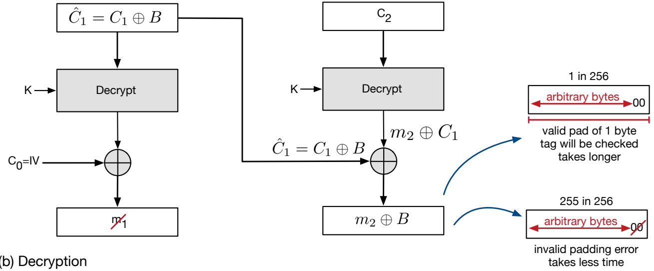
[5] Section 9.4.3 of "A Graduate Course in Applied Cryptography" by Dan Boneh and Victor Shoup

Padding Oracle Timing Side-Channel Attack on TLS 1.0



(a) Encryption

Sent by attacker: $C_0, \hat{C}_1, C_2$



(b) Decryption

Datagram Transport Layer Security

DTLS: Datagram Transport Layer Security

Datagram Transport Layer Security (DTLS): Overview

- developed to secure unreliable datagram traffic (over UDP)
  - for instance: Session Initiation Protocol (SIP) for VoIP, electronic gaming
- current version: 1.2 published in RFC 6347
  - version 1.3 is in draft
- main issue of running TLS over UDP is that packets may be lost or reordered
  - TLS cannot deal with this kind of unreliability
  - DTLS provides minimal changes to deal with this problem
    - designed with "TLS over datagram transport" philosophy
- unreliability creates problems at two levels
  - independent decryption of individual records are not allowed in TLS
    - if record N is not received integrity check of record N+1 will fail
  - TLS handshake depends on reliable delivery of handshake messages
    - handshake will fail if messages are lost

DTLS Loss-Insensitive Messaging

the encryption in TLS record protocol has inter-record dependency

1. the cryptographic context i.e. stream cipher key stream is retained between records
2. the anti-replay and message reordering protection depends on an implicit sequence number

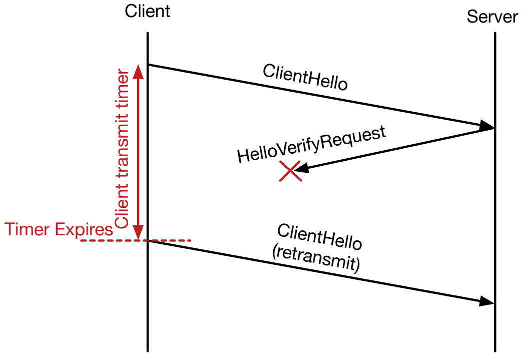
DTLS addresses these problems (for running TLS over unreliable transport):

1. stream ciphers that do not allow random access are not used in DTLS
2. explicit sequence numbers are added

To see the complete list of TLS Parameters including cipher suites and whether compatible with DTLS refer to Transport Layer Security (TLS) Parameters

DTLS Reliable Handshake

- TLS handshake is a *lockstep* cryptographic handshake
  - the order of transmitted and received messages is important
  - handshake will fail if messages are lost or are out of order
- to handle packet loss during handshake, retransmission timers are used
- to handle reordering specific sequence numbers are used
  - out of order handshake messages are stored until prior messages are received
- DTLS handshake messages are fragmented over several DTLS records
  - each record would fit in a single IP packet
  - each DTL handshake also has a fragment offset and fragment length
- DTLS uses a bitmap window of received records to protect against replay
  - similar to IPsec AH/ESP method
  - records outside the window and the ones already received are discarded



References

- Additional resources for this week: RFCs 5246, 8446, 4251-4254
- This POODLE bites: exploiting the SSL 3.0 fallback, Moller, Bodo and Duong, Thai and Kotowicz, Krzysztof, Security Advisory, 2014
- A Graduate Course in Applied Cryptography, Dan Boneh and Victor Shoup

Appendix A: TLS 1.3 Extensions

- CH: `ClientHello`
- SH: `ServerHello`
- EE: `EncryptedExtension`
- CT: `Certificate`
- CR: `CertificateRequest`
- NST: `NewSessionTicket`
- HRR: `HelloRetryRequest`

Extension	TLS 1.3 Message Type						
server_name [RFC6066]	CH		EE				
max_fragment_length [RFC6066]	CH		EE				
status_request [RFC6066]	CH			CT	CR		
supported_groups [RFC7919]	CH		EE				
signature_algorithms (RFC 8446)	CH				CR		
use_srtp [RFC5764]	CH		EE				
heartbeat [RFC6520]	CH		EE				
application_layer_protocol_negotiation [RFC7301]	CH		EE				
signed_certificate_timestamp [RFC6962]	CH			CT	CR		
client_certificate_type [RFC7250]	CH		EE				
server_certificate_type [RFC7250]	CH		EE				
padding [RFC7685]	CH						
key_share (RFC 8446)	CH	SH				HRR	
pre_shared_key (RFC 8446)	CH	SH					
psk_key_exchange_modes (RFC 8446)	CH						
early_data (RFC 8446)	CH		EE				NST
cookie (RFC 8446)	CH					HRR	
supported_versions (RFC 8446)	CH	SH				HRR	
certificate_authorities (RFC 8446)	CH				CR		
oid_filters (RFC 8446)					CR		
post_handshake_auth (RFC 8446)	CH						
signature_algorithms_cert (RFC 8446)	CH				CR		

Appendix B: TLS 1.2 Handshake

- published in RFC 5246
- TLS 1.2 handshake protocol exchange three sets of messages
  - Hello messages: `ClientHello` and `ServerHello` establish
    - Protocol Version
    - Session ID
    - Cipher Suite
    - Compression Method
    - two random values: `ClientHello.random` and `ServerHello.random`
  - Key exchange and authentication messages
    - the server `Certificate` and `ServerKeyExchange`
    - the client `Certificate` and `ClientKeyExchange`
    - client authenticates server and server (optionally) authenticates client
  - Signal transition to protected payload
    - the `ChangeCipherSpec` message is by itself a separate protocol and is not part of handshake
    - the `Finished` messages are encrypted under negotiated cryptographic algorithm and established shared secrets