Security Protocols

- Properties of a security protocol
- SSL/TLS

Goals with this lecture

The most important things you should learn are:

- What characterizes a secure protocol
- How TLS works and its main features
- How ciphers and crypto-keys are negotiated
- What security problems TLS addresses

- Most of these techniques come back in other protocols
  - SSH, IPsec, ...

Cryptographic Protocols

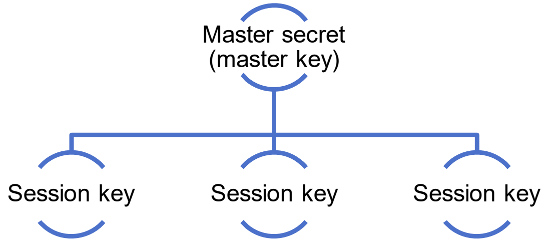
Layer	Protocols
Application layer	• SSL/TLS (IETF) • SSH (IETF)
Network layer	• IPSec (IETF)
Link layer	• WEP/WPA*n* (IEEE 802.11) • MACSec (IEEE 802.11ae)
Physical layer	

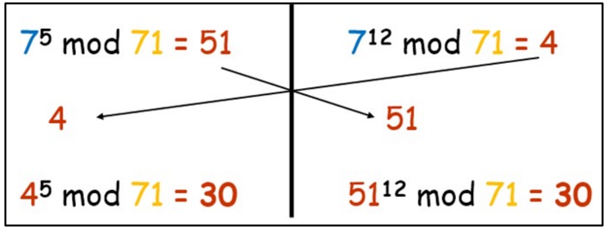
Properties for security protocols

- Data confidentiality
  - Encryption with symmetric keys (AES, …)
- Data integrity
  - Prevent modification of data (keyed hashes, HMAC)
- Data authenticity
  - Guarantees data can not be inserted, deleted, reordered or replayed (can be argued that this should be covered by data integrity)
  - Freshness guarantees (time-stamps, nonces)
- Mutual authentication of communicating parties
  - I requested, both parties know who they talk to (PSK, Certificates)
- Perfect forward secrecy, PFS
  - The transmission (i.e. session keys) should not be revealed if older or future session keys are revealed
  - Nor if user's pre-shared, public or private keys are compromised (Diffie-Hellman, …)

Perfect Forward Secrecy (PFS)

- *Forward Secrecy* = a compromised <u>session key</u> should not affect other sessions (in the past or in the future)
  - Therefore, all sessions must have unique session keys
- <u>*Perfect*</u> *Forward Secrecy* = the key will not be compromised even if other keys derived from the same long-term keying material are compromised
  - Not even if session keys are regularly changed and calculated from the same master secret
- Keys should not depend on a shared secret or material used for authentication
  - E.g. if based on RSA private/public keys, session keys will be linked
  - Use Diffie-Hellman or similar, to create unique session keys
- Use Diffie-Hellman RSA (DHE_RSA) or Elliptic Curve Diffie-Hellman (ECDHE)
  - E = Ephemeral = short lived (one negotiation per handshake) → forward secrecy
  - Together they offer both unique session keys (DHE) and authentication (RSA/EC)
- Sometimes PFS is sacrificed for lack of computational resources (IoT devices)
  - D-H can be used with static (fixed long-term) DH keys for one or both parties
  - Most common is "static-ephemeral-DH" = one party has a fixed key (value) and no calculation needed → The same $A = g^a \bmod p$ is reused, but if a is ever revealed…
  - Static-DH offers authentication – it may not even have to be sent





Typical Cryptographic Protocol

Negotiation of
security parameters

Client

Mutual authentication

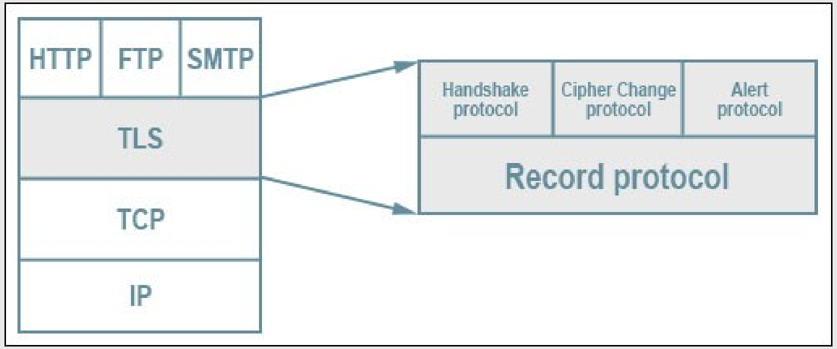
Server

Key exchange or
key agreement
(derive master secret)

Communication with message
confidentiality, integrity and
message authentication (MAC)

The SSL/TLS protocol

Chapter 17.1-3



SSL/TLS

SSL – Secure Sockets Layer

- Developed by Netscape
- Version 1 was for internal use (1994)
- Version 2 incorporated in their "Navigator" web browser (1995).
  Had some problems with MITM attacks
- Version 3 was created with public review from industry (1996)
- IETF standardized SSL version 3.1 and renamed it "TLS version 1.0" (1999)

TLS – Transport Layer Security (RFC 2246)

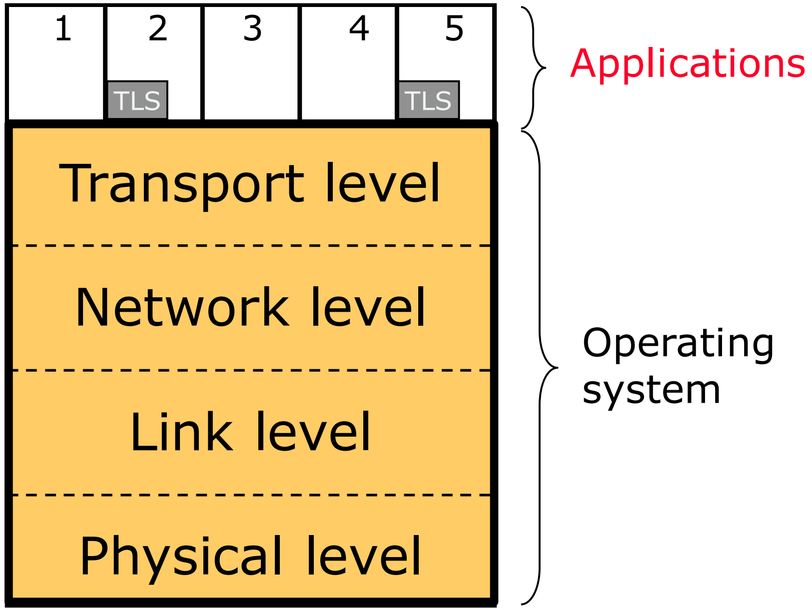
- TLS 1.0 (1999) and 1.1 (2006) – vulnerable and unsupported by web browsers (March 2020)
- TLS 1.2 (2008) with improved hash functions and ECC support
  - RFC 5246
  - Required for HTTP/2 (if TLS is used) (2015)
- TLS 1.3 (2018) redesigned from scratch
  - Took five years of development and testing
  - Faster, better privacy with encrypted handshake with fewer options, better ciphers and modes
  - Faster negotiation of ciphers (fewer RTT)
  - RFC 8446

SSL/TLS

- Designed to protect TCP traffic for applications
  - TLS can be used to secure all types of TCP connections
  - Applications must be TLS enabled by design
- Typical applications:
  - Secure http (https) in web browser uses TLS – port 443 instead of 80
  - SSL/TLS versions of known protocols: telnets, nntps, imaps, smtps, ldaps, https, …
- Clients authenticate servers when connecting (method negotiated)
  - Certificates commonly used and sent from server to client
  - Mutual authentication with client certificates supported – not used by https
- TLS extensions can be included in the first Hello message (similar to TCP options in the TCP three-way handshake)
  - Examples: padding, encrypt_then_mac, enable heartbeats [RFC 6520]
- This presentation is not a full description of all features in the protocol – the full protocol specification (in RFCs) is rather long

TLS is integrated in applications

- TLS is used by applications to secure their communications
- Each application runs in its own protected memory
  - They can not see each other
  - They need to do system calls and ask for a network service from the operating system
- Most TLS-enabled applications use a standard library performing all TLS functionality
  - The library becomes part of the application
  - Many free libraries available (OpenSSL, WolfSSL, mbed TLS, ...)
- The alternative would be to implement all crypto-functions yourself with bugs...
- Many libraries exist: https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations



OpenSSL
Cryptography and SSL/TLS Toolkit

Welcome to OpenSSL!

The OpenSSL Project develops and maintains the OpenSSL software - a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication. The project's technical decision making is managed by the OpenSSL Technical Committee (OTC) and the project governance is managed by the OpenSSL Management Committee (OMC). The project operates under formal Bylaws.

- **CVE-2024-3296** A timing-based side-channel flaw exists in the rust-openssl package, which could be sufficient to recover a plaintext across a network in a Bleichenbacher-style attack. To achieve successful decryption, an attacker would have to be able to send a large number of trial messages for decryption.

- **CVE-2024-2511** Some server configurations can cause unbounded memory growth when processing TLSv1.3 that would lead to a Denial of Service.

- **CVE-2024-0727** Processing a maliciously formatted PKCS12 file may lead OpenSSL to crash. A file in PKCS12 format can contain certificates and keys and may come from an untrusted source. The PKCS12 specification allows certain fields to be NULL, but OpenSSL does not correctly check for this case.

https://www.openssl.org/news/vulnerabilities.html

TLS Protocol Architecture [RFC 5246]

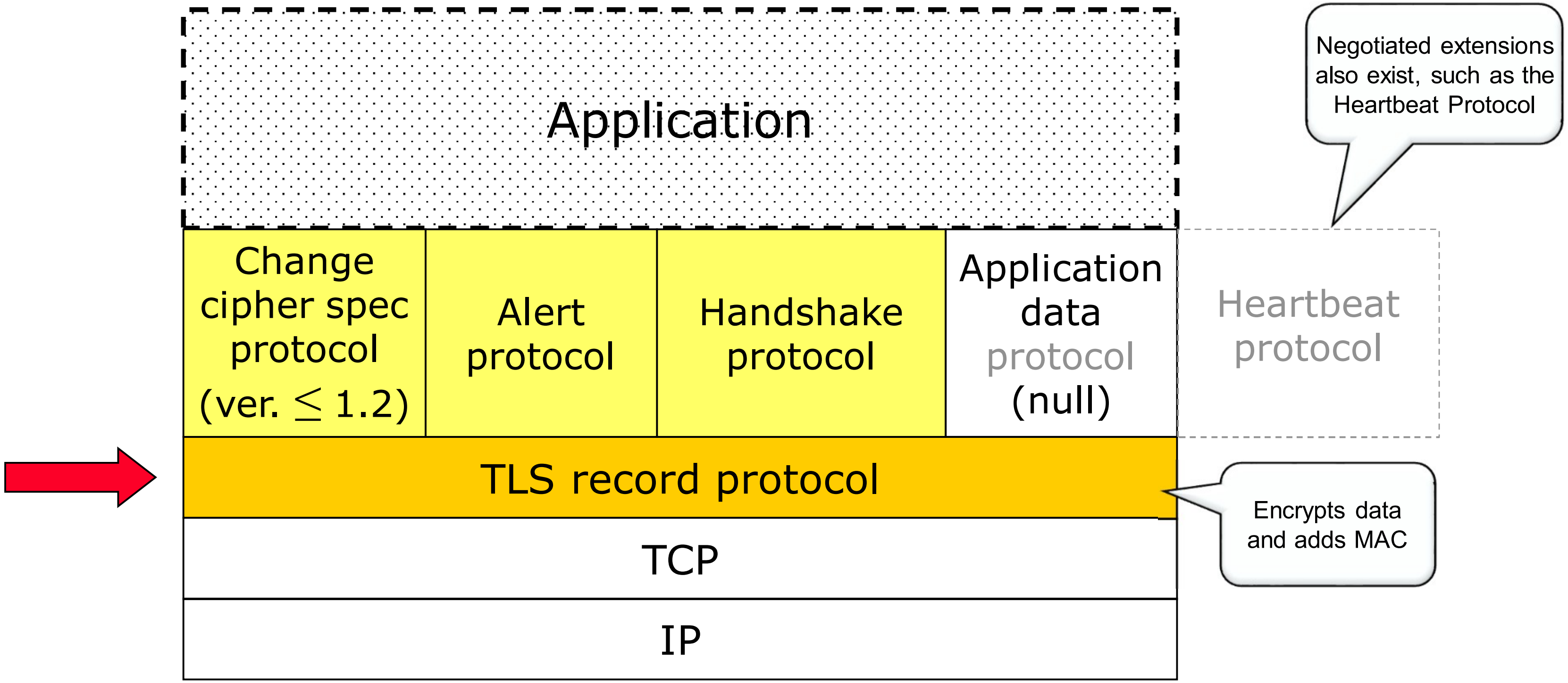
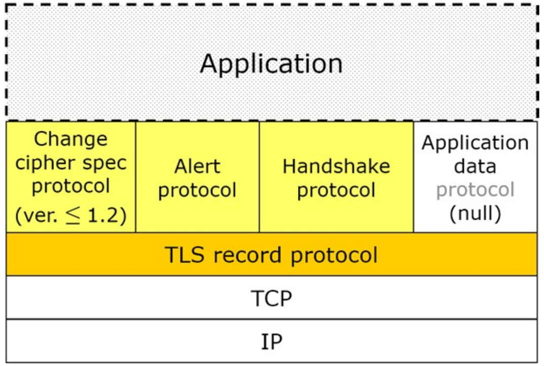


Fig. 17.2

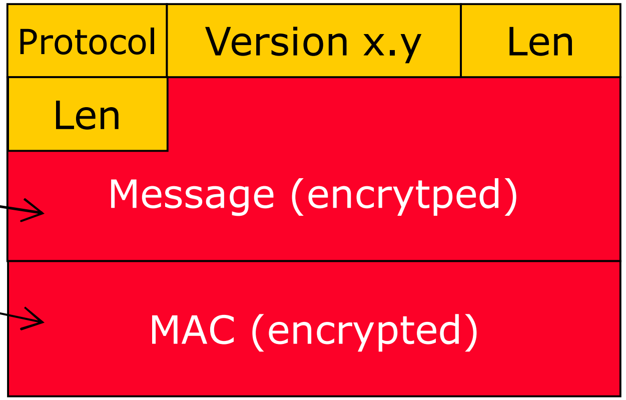
The TLS Record Protocol

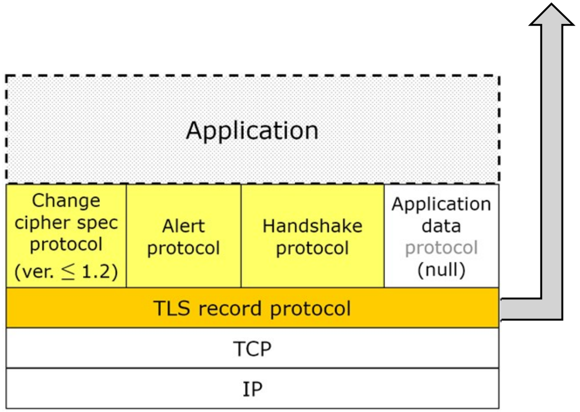
- Active after all initial handshaking done – when algorithms and keys are negotiated
- Relies on TCP to offer reliable communication (to do retransmissions)
- Provides confidentiality and message integrity
  - Encrypts data and adds MAC (HMAC, keyed hash)
- May also fragment and compress data



The TLS Record Protocol

- Format:
  - 5-byte header
  - Message (data), max 16,384 bytes optionally compressed
  - Message Authentication Code
- **Protocol** field tells what upper-layer TLS-protocol it encapsulates:
  - 20 = Change cipher protocol
  - 21 = Alert protocol
  - 22 = Handshake protocol
  - 23 = Application (protocol) data
- **Version** = SSL/TLS protocol version
  - 2 bytes: major and minor number
  - SSL = 2.0 or 3.0
  - TLS version 1.0 = 3.1, etc.





The TLS Record Protocol

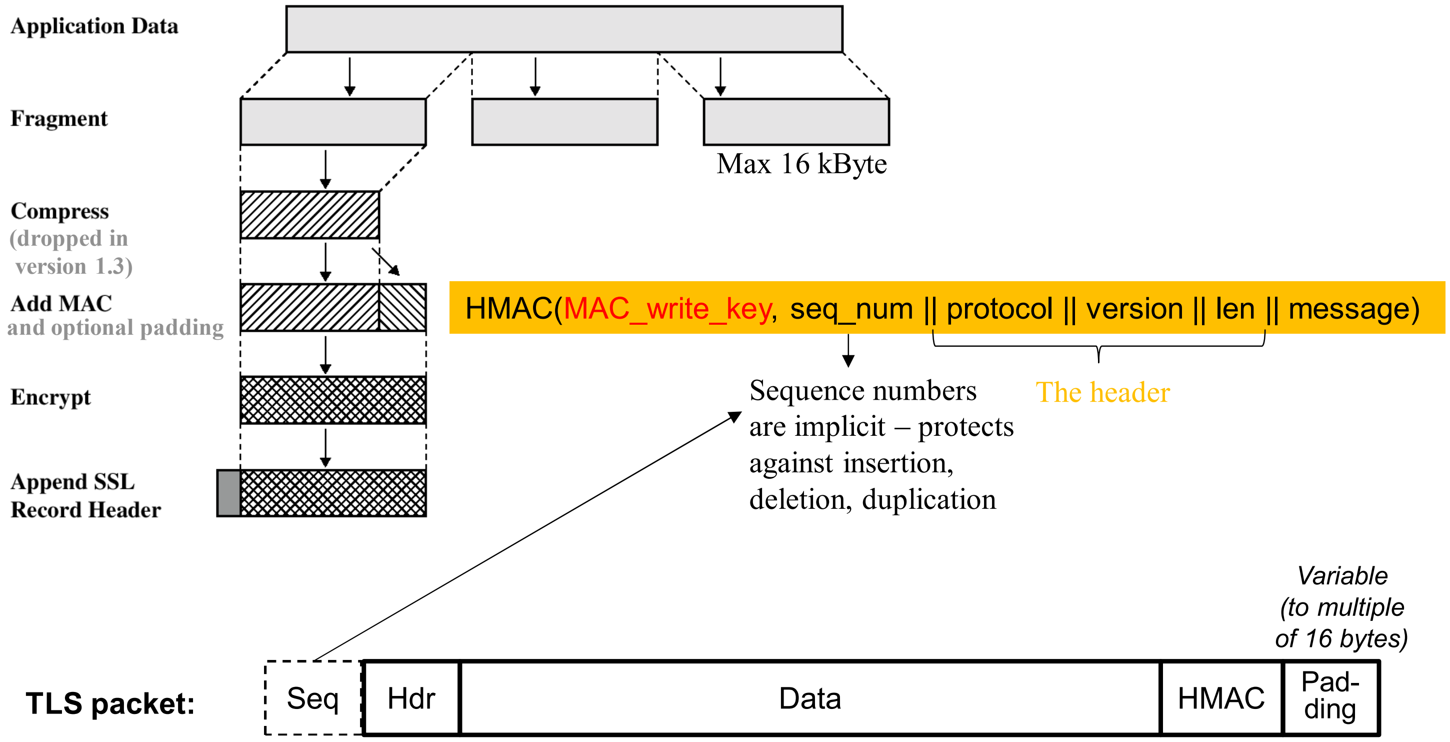
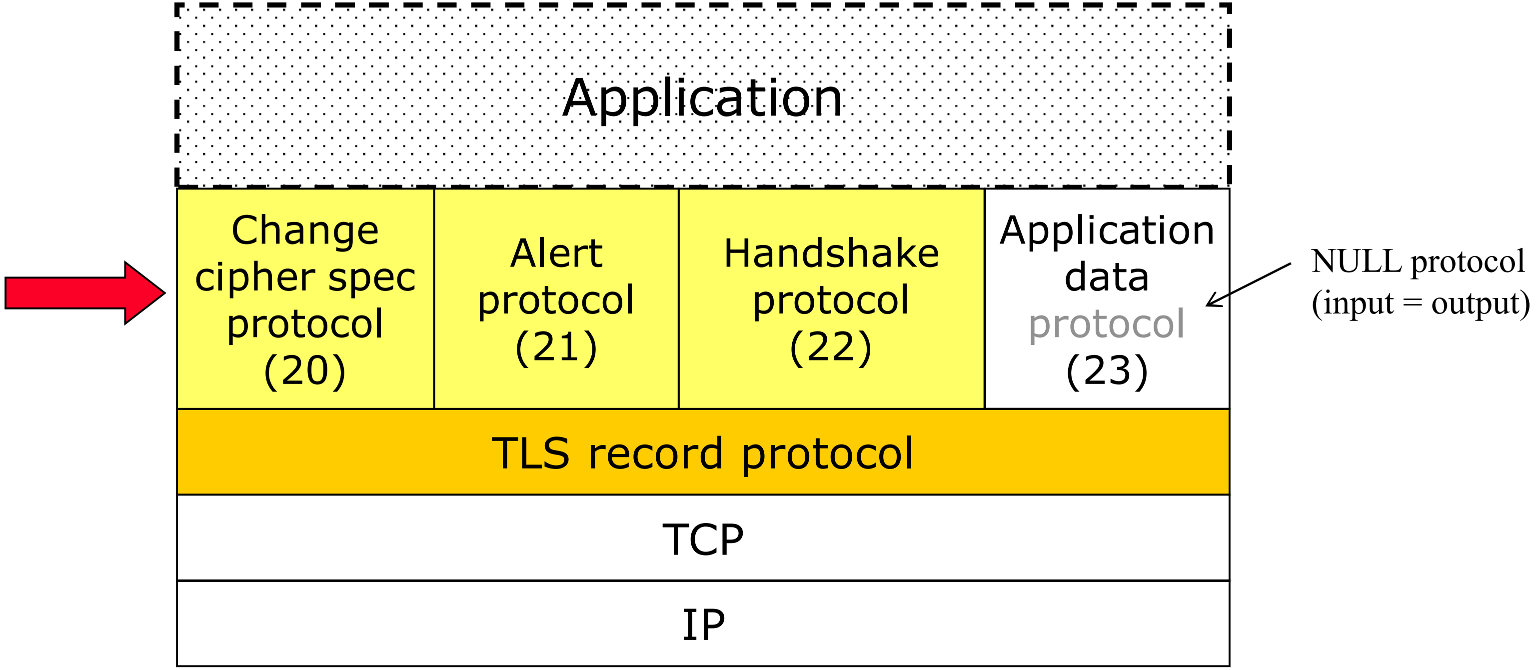
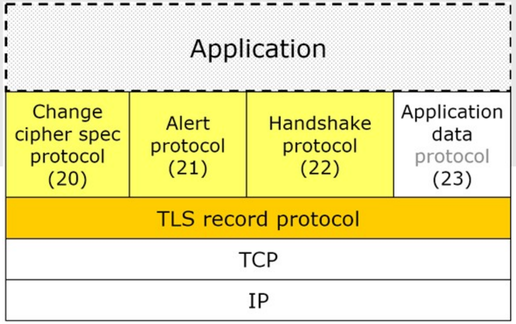


Fig. 17.3

TLS Protocol Architecture



TLS "Protocols"



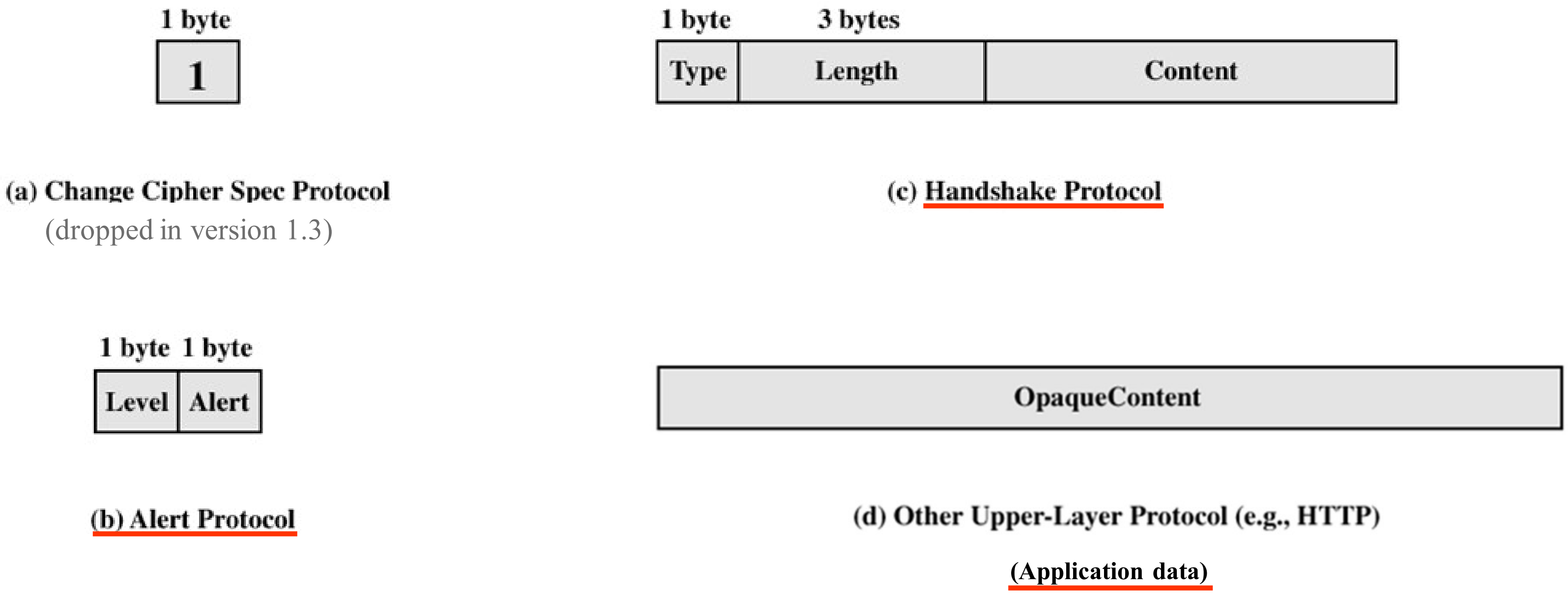
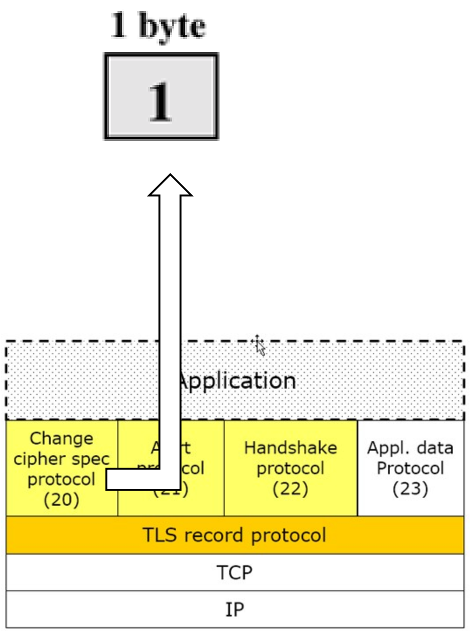


Fig. 17.5

The TLS Change Cipher Spec Protocol

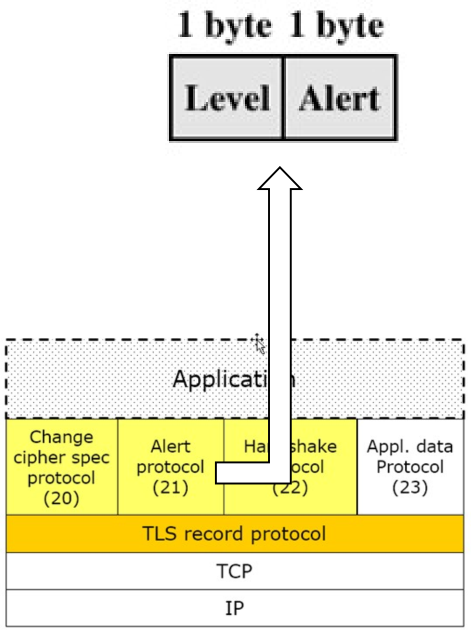
- Just a one-byte message: **1**
- Pending state becomes current state
  - Changes encryption to what has been negotiated earlier (algorithms, keys)
- Deprecated in TLS ver. 1.3
  (may be sent but is ignored)



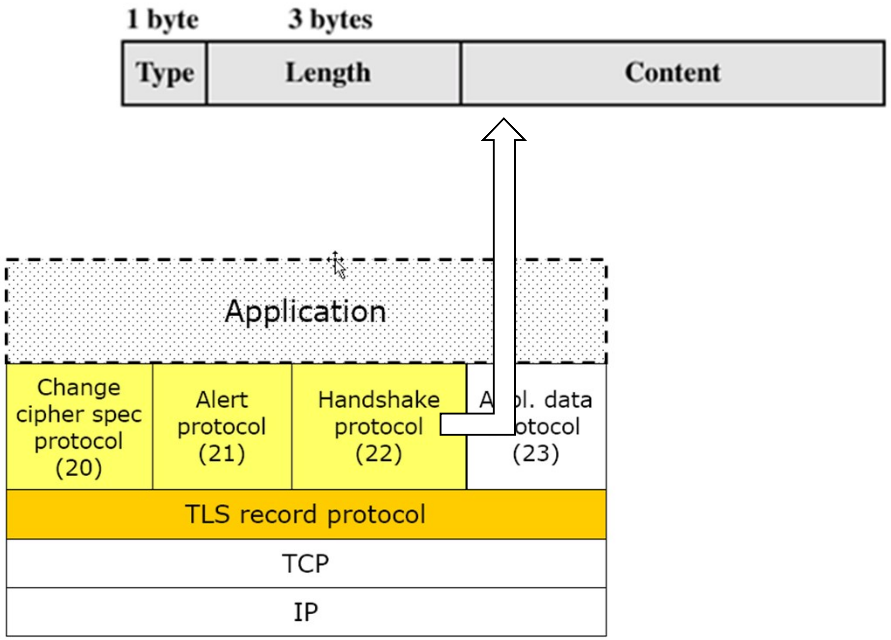
The TLS Alert Protocol

Signals error and alert conditions:

- Severity Level: warning or fatal
  - Must immediately disconnect if fatal
- Alert codes (SSL has 12, TLS twice as many):
  - CloseNotify
  - UnexpectedMessage (fatal)
  - BadRecordMAC (fatal)
  - Certificate Unsupported/Revoked/Expired/Unknown/Bad
  - Illegal Parameter
  - HandshakeFailure (not possible to agree on services, fatal)
  - …



SSL/TLS handshake protocol



Algorithm (cipher) negotiation

- Authentication method
- Data encryption algorithm
- Data protection algorithm
- Performs key exchange

Algorithms to negotiate

- Authentication + session key exchange
  - **PSK** (pre-shared keys) mixed with DH to get forward secrecy
  - **RSA** (public keys) – same here, DH must also be used (with short-lived keys)
  - Fixed Diffie-Hellman: certificate contains pre-calculated primes (groups)
  - (Ephemeral) Diffie-Hellman: each side generates one-time parameters
  - ECDHE = Elliptic curve Diffie-Hellman Ephemeral guarantees forward secrecy

- Cipher for data encryption
  - Weak ciphers: DES_40, DES_56, 3-DES, RC4
  - Weak cipher modes: AES_CBC
  - Ciphers: AES_128, AES_256, …

Weak ciphers only allowed in TLS ≤ 1.2

- MAC algorithm for data integrity (ver. 1.2):
  - Hash functions: MD5, SHA-1, SHA-256
  - $HMAC_K(msg) = hash(K \oplus opad \,||\, hash(K \oplus ipad \,||\, msg))$
  - Calculation:

opad = 0x5C5C… ipad = 0x3636…

$$HMAC_K(seq\_num \,||\, tls\_proto \,||\, tls\_vers \,||\, msg\_len \,||\, msg)$$

Examples of Cipher Suites (ver. 1.2)

Cipher Suite	Key Negotiation	Digital Signature Method	Symmetric Key Encryption Method	Hashing Method for HMAC	Strength	
NULL_WITH_NULL_NULL	None	None	None	None	None	0x0000
RSA_EXPORT_WITH_RC4_40_MD5	RSA export strength (40 bits)	RSA export strength (40 bits)	RC4 (40-bit key)	MD5	Very weak	0x0003
RSA_WITH_AES_256_CBC_SHA256	RSA	RSA	AES 256 bits	SHA-256	Weak, no PFS	0x0069
DHE_RSA_WITH_AES_128_GCM_SHA256	DH+RSA	RSA	AES 128 bits	SHA-256	Strong	0x009E

https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

Protocol support in Chrome, 124

https://www.ssllabs.com/ssltest/viewMyClient.html

Protocol Features

Protocols

TLS 1.3	Yes
TLS 1.2	Yes
TLS 1.1	No
TLS 1.0	No
SSL 3	No
SSL 2	No

Cipher Suites (in order of preference)

TLS_GREASE_7A (0x7a7a)	-
TLS_AES_128_GCM_SHA256 (0x1301) Forward Secrecy	128
TLS_AES_256_GCM_SHA384 (0x1302) Forward Secrecy	256
TLS_CHACHA20_POLY1305_SHA256 (0x1303) Forward Secrecy	256
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b) Forward Secrecy	128
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f) Forward Secrecy	128
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c) Forward Secrecy	256
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030) Forward Secrecy	256
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9) Forward Secrecy	256
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8) Forward Secrecy	256
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) WEAK	128
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) WEAK	256
TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c) WEAK	128

Qualys. SSL Labs

Home Projects Qualys Free Trial Contact

You are here: Home > Projects > SSL Server Test > williamstallings.com

SSL Report: williamstallings.com (209.237.150.20)

Assessed on: Mon, 06 Mar 2023 11:43:58 UTC | Hide | Clear cache

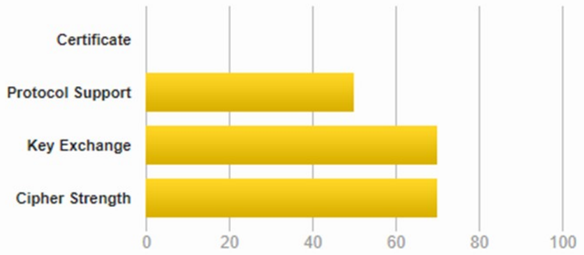
Scan Another »

Summary

Overall Rating

T

If trust issues are ignored: C



Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This server's certificate is not trusted, see below for details.

This server supports weak Diffie-Hellman (DH) key exchange parameters. Grade capped to B. MORE INFO »

Logjam attack

The server supports only older protocols, but not the current best TLS 1.2 or TLS 1.3. Grade capped to C. MORE INFO »

This server does not support Forward Secrecy with the reference browsers. Grade capped to B. MORE INFO »

RSA + DH but not short-lived keys

This server does not support Authenticated encryption (AEAD) cipher suites. Grade capped to B. MORE INFO »

TLS CBC timing attack

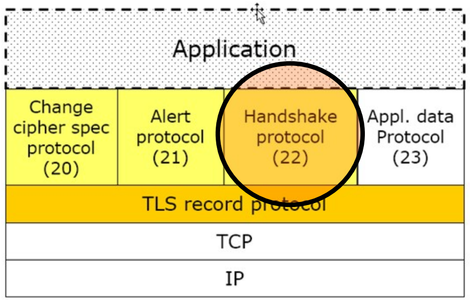
(AEAD = Authenticated encryption with associated data)

This server supports TLS 1.0. Grade capped to B. MORE INFO »

The Handshake Protocol

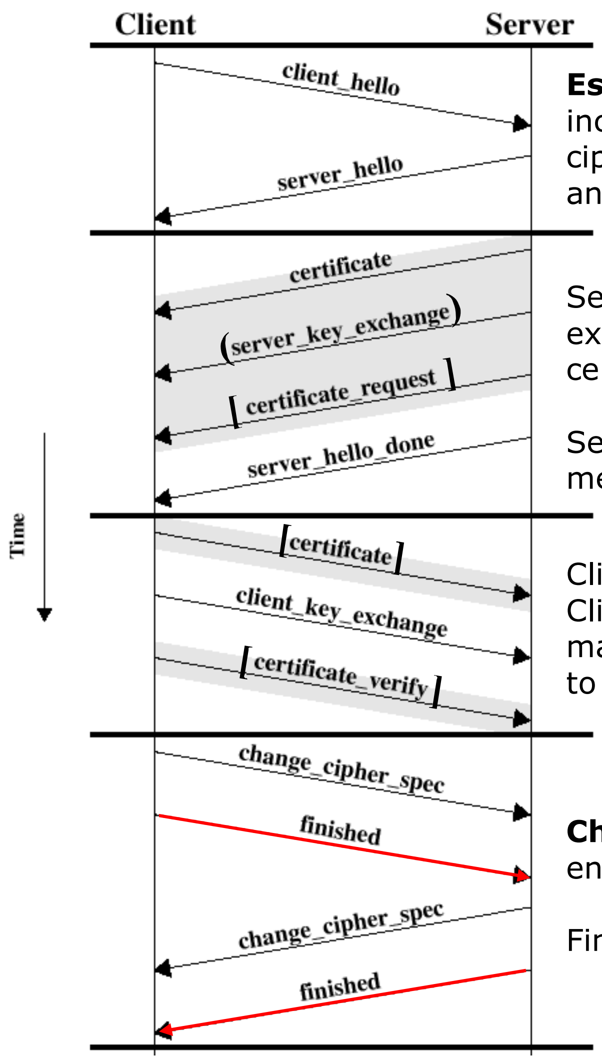
(version 1.2 and earlier)

SSL/TLS handshake protocol



Notes:
- Client certificates are not commonly used, thus messages within [ ] are normally not sent

- Only messages in red are encrypted



Establish security capabilities, incl. protocol version, session ID, cipher suite, compression method and initial random numbers

Server **may send certificate**, key exchange parameters and a certificate_request.

Server signals end of hello message phase.

Client sends certificate if requested. Client sends **key_exchange**. Client may send certificate_verification to show it knows the private key.

Change_cipher_spec turns on encryption.

Finish ends the handshake protocol

Note: Shaded transfers are optional or situation-dependent messages that are not always sent.

Fig. 17.6

Handshake Round 1

$$\text{Client hello} \quad \text{Client} \xrightarrow{\{\, ver_C \,\|\, r_c \,\|\, sid \,\|\, ciphers \,\|\, comps \,\}} \text{Server}$$

$$\text{Client} \xleftarrow{\{ver \,\|\, r_s \,\|\, sid \,\|\, cipher \,\|\, comp \,\}} \text{Server} \quad \text{Server hello}$$

ver_C	Highest version of protocol client supports
r_c, r_s	nonces (4 byte timestamp and 28 random bytes) – seed for encryption
sid	Current session id, 32 bytes long (0 if new session)
ciphers	List of ciphers client supports (preference order)
comps	List of compression algorithms client supports
ver	Version of protocol to be used (highest version both support)
cipher	Cipher to be used (selected from client's list)
comp	Compression algorithm to be used

Some notes about round 1

- The 32-byte nonces r_c and r_s contain two things:

Time stamp	Random number

  - Time stamp (4 bytes) to guarantee each nonce is unique
  - Random number (28 bytes)
  - Used when calculating keys
- Random numbers (general remark):
  - Must be created by a strong (i.e. cryptographic) random number generator
  - If not random, attackers can try to pre-compute values
  - Both server and client are responsible in creating random numbers (r_c and r_s)
- Session ID:
  - Always zero for a new session (or long random string which is ignored)
  - Server responds with a session_ID > 0 if it allows sessions to be resumed
  - Makes it possible to use keying material from a previously established session
  - Faster since public key operations are time consuming

The Client Hello message

```
⌄ Transport Layer Security
  ⌄ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 512
    ⌄ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 508
        Version: TLS 1.2 (0x0303)
      › Random: abf1fac49409cbaef37bf577e3b45fd61b3dec375c456ed8…
        Session ID Length: 32
        Session ID: d7e938ee24deb6fd30c827b4e4cc5f1205567afc125c601d…
        Cipher Suites Length: 34
      ⌄ Cipher Suites (17 suites)
          Cipher Suite: Reserved (GREASE) (0x1a1a)
          Cipher Suite: TLS_AES_128_GCM_SHA256 (0x1301)
          Cipher Suite: TLS_AES_256_GCM_SHA384 (0x1302)
          Cipher Suite: TLS_CHACHA20_POLY1305_SHA256 (0x1303)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384 (0xc02c)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)
           . . .
      ⌄ Compression Methods (1 method)
          Compression Method: null (0)
```

nonce r_c

Ciphers supported by the client. Listed in order of preference

← To be chosen by server…

The Server Hello message (from Google)

```
⌄ Transport Layer Security
  ⌄ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 89
    ⌄ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 85
        Version: TLS 1.2 (0x0303)
      › Random: f373e985c9a960e175a66ba6c0fc2a893fccff79e3518f44…
        Session ID Length: 32
        Session ID: 7fb1a61d097f4f2f8cb658e3e3a9a9a7071790a1e7ceefc8…
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
        Compression Method: null (0)
        Extensions Length: 13
      › Extension: renegotiation_info (len=1)
      › Extension: ec_point_formats (len=4)
```

nonce r_s

Selected cipher

Handshake Round 2

If TLS_DHE: Ephemeral D-H key exchange, server selects (p, g) and calcualtes $Y = g^b \bmod p$.

Client $\longleftarrow$ { X.509 *certificate (one or a chain)* } Server **certificate**

Client $\longleftarrow$ $\{ p \parallel g \parallel Y_s \parallel sign(k_S, r_c \parallel r_s \parallel p \parallel g \parallel Y_s) \}$ Server **Server key exchange**

Client $\longleftarrow$ { *ctype* || *gca* } Server **Certificate request**

Client $\longleftarrow$ { } Server **Hello done**

Contents of second message is algorithm dependent – **here ephemeral D-H**

p, g	Modulo and generator to use: $Y = g^b \bmod p$ where "b" is secret
k_S	Server key to sign hash – algorithm from negotiation (DSS, RSA, …)
ctype	Certificate type requested from client, if any
gca	List of names of acceptable certification authorities (helps client to chose)

Handshake Round 3

If requested by server. Contains client's public key

certificate Client $\xrightarrow{\{\ client\_cert\ \}}$ Server

If RSA: 48 random bytes encrypted with server's public key k_p

Client key exchange Client $\xrightarrow{\text{D-H: } \{\ Y_c\ \}\ ;\ \text{RSA: } \{\ pre\_master\_secret\ \}k_p}$ Server

If D-H: client parameters

Only if client has sent a certificate. Shows it is in possession of the private key k_c.

Certificate verify Client $\xrightarrow{\{\ sign(k_c,\ master\_secret,\ msgs)\ \}}$ Server

msgs	Concatenation of messages sent/received so far
ipad	0x3636… repeated to block length
opad	0x5c5c… repeated to block length
k_c	Client's private key
Master_secret	Calculated master secret (calculated from pre_master_secret)

The Pseudo-random function [RFC 5246]

- A pseudo-random function is defined and used in TLS:

```
PRF(k, label, x) =
    HMAC_k(HMAC_k(label||x) || label||x) ||
    HMAC_k(HMAC_k(HMAC_k(label||x)) || label||x) ||
    HMAC_k(HMAC_k(HMAC_k(HMAC_k(label||x))) || label||x) ||
    …
```

- SHA-256 gives 32 bytes per round/hash
- Used to expand a short secret to arbitrary length blocks

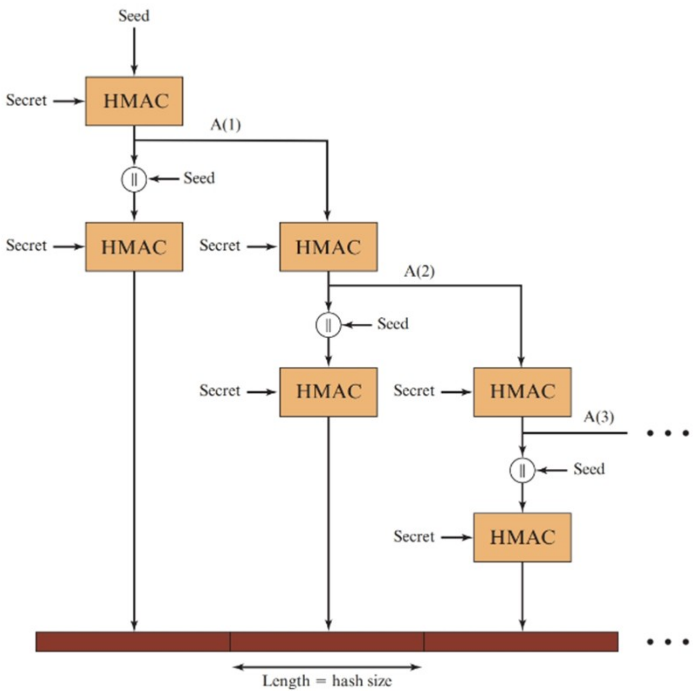


Fig 17.7

1 Pre-Master Secret → Master Secret

- If RSA server authentication (with certificates) then:
  - Client generates a ***pre_master_secret*** (48 random bytes), encrypts it with the server's public key and sends it to the server. Only the correct server can make use of it
  - Offers protection against MITM attacks but no PFS (problem if server's private key becomes known)
- Diffie-Hellman can (should) be used to generate the ***pre_master_secret***
  - MITM-attacks are possible so should be combined with RSA or ECC Signatures (ECDSA)
  - Offers Forward Secrecy
- The *master secret* can now be calculated:

$$\textit{master\_secret} = \textbf{PRF}(\underbrace{\textit{pre\_master\_secret}}_{\text{k}}, \underbrace{\text{“}\texttt{master secret}\text{”}}_{\text{label}}, \underbrace{r_c \,||\, r_s}_{\text{x}})$$

- Result: a 48 byte long master secret based on data that both the server and the client have generated

2 Master secret → Encryption keys

- An arbitrarily long stream of keys can now be created from the master secret:

  key_material = PRF(master_secret, "`key expansion`", r_c || r_s);

- The PRF is used whenever more key material is needed
- From the key material 6 different keys are created:
  - Write keys are used to encrypt data
  - MAC keys for integrity protection
  - IV for cipher if CBC mode used (only generated if needed)
- We have one key for each direction
- We never use the master secret directly
  - Used only to generate keys that are frequently changed

Server write key
Client write key
Server MAC key
Client MAC key
Server IV
Client IV

Handshake Round 4

Client | **Server**

Change cipher spec — { *change cipher spec message* } →

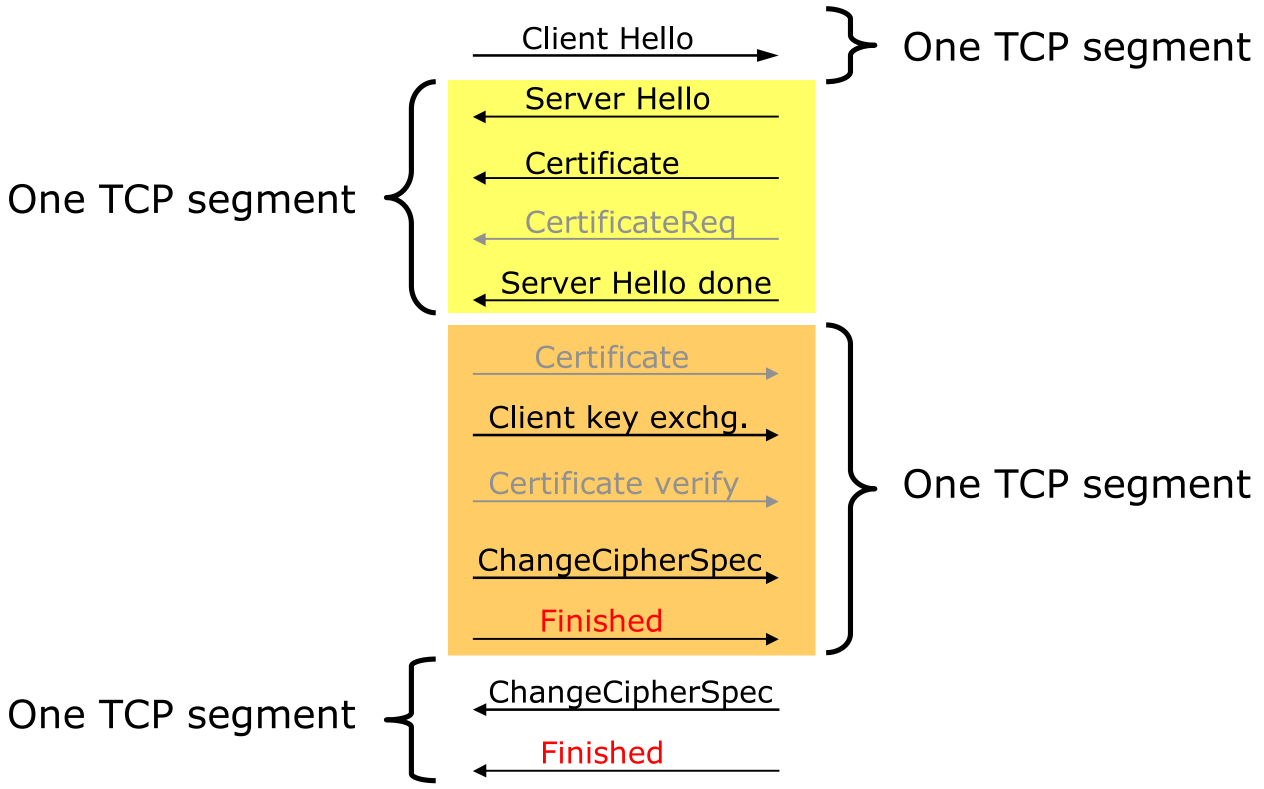
Finished Encrypted — { PRF(master_secret, "`client finished`", msgs) } →

← { *change cipher spec message* } — **Change cipher spec**

← { PRF(master_secret, "`server finished`", msgs) } — **Finished** Encrypted

msgs = hash(all_handshake_messages_sent)
The hash function to use was negotiated in the hello msgs

TLS rounds – from a TCP perspective



- Two round-trip delays
  - https will be slower than http ☹
  - IoT devices must use more power
- TLS 1.3 addresses this with a new design

Closing a TLS session

- TLS has a procedure to close a connection
  - Sends a “close_notify” message before TCP FIN
  - Protects against “truncation attacks” if an attacker manages to to close down the TCP connection prematurely
  - If a transaction consists of several messages, the last part may otherwise be lost
  - Rarely used for Web sessions

- Alternative: handle it in the application-level protocol (above TLS)
  - TCP can not be trusted

TLS Version 1.3

https://tools.ietf.org/html/rfc8446

160 pages

(Book only lists features of ver. 1.3 on the last page of chapter 17.2)

Problems
in earlier versions

Patching, patching
and patching…

Motivates a new redesign
of TLS, version 1.3

This has mitigated quite a few attacks..

OWASP
The Open Web Application Security Project

RC4
- Roos's Bias 1995
- Fluhrer, Martin & Shamir 2001
- Klein 2005
- Combinatorial Problem 2001
- Royal Holloway 2013
- Bar-mitzvah 2015
- NOMORE 2015

RSA-PKCS#1 v1.5 Encryption
- Bleichenbacher 1998
- Jager 2015
- DROWN 2016

Renegotiation
- Marsh Ray Attack 2009
- Renegotiation DoS 2011
- Triple Handshake 2014

3DES
- Sweet32

AES-CBC
- Vaudenay 2002
- Boneh/Brumley 2003
- BEAST 2011
- Lucky13 2013
- POODLE 2014
- Lucky Microseconds 2015

Compression
- CRIME 2012

MD5 & SHA1
- SLOTH 2016
- SHAttered 2017

15

https://www.owasp.org/images/9/91/OWASPLondon20180125_TLSv1.3_Andy_Brodie.pdf

TLS 1.3 released 2018 (RFC 8446)

- Took 5 years of work and testing, 28 drafts…
- Many smaller changes:
  - PRF -> HKDF (HMAC-based Extract-and-Expand Key Derivation Function)
  - New ciphers and signature algorithms added
- Unsafe and unused functions removed – fewer functions are more secure
  - Compression – may reveal information about payload [CRIME attack]
  - 3-DES, RC4, MD5, SHA-1
  - Cipher Block Chaining Mode (AES-CBC) timing leaks [Beast and Lucky13]
  - Export ciphers [Logjam, Freak, Drown, Beast]
  - Static D-H and weak D-H groups
  - …
- Perfect Forward Secrecy mandatory no longer optional
  - Most RSA methods dropped – offered no PFS + hard to implement correctly [Million-message and Robot]
- Faster handshake: **1-RTT** where client guesses ciphers to be used
  - Fewer cipher suites supported – easier to guess what to use
  - If server cannot support request, a new message exist: *HelloRetryRequest*
  - Everything after *Server Hello message* is encrypted
  - Change Cipher Spec protocol not needed
- **Also supports 0-RTT – session resumption with stored information**
  - Less secure – inspired by the QUIC protocol
  - If client and server has communicated before, sessions can be resumed with stored "session tickets"

Note that TLS 1.2 is still widely used and considered secure

https://www.thesslstore.com/blog/tls-1-3-everything-possibly-needed-know/
https://www.thesslstore.com/blog/tls-1-3-handshake-tls-1-2/

Key exchange algorithms in TLS 1.3

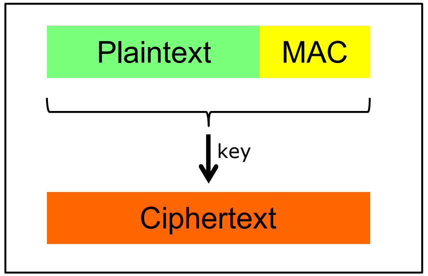
Key exchange/agreement and authentication

Algorithm	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3	Status
RSA	Yes	Yes	Yes	Yes	Yes	No	Defined for TLS 1.2 in RFCs
DH-RSA	No	Yes	Yes	Yes	Yes	No	
DHE-RSA (forward secrecy)	No	Yes	Yes	Yes	Yes	Yes	
ECDH-RSA	No	No	Yes	Yes	Yes	No	
ECDHE-RSA (forward secrecy)	No	No	Yes	Yes	Yes	Yes	
DH-DSS	No	Yes	Yes	Yes	Yes	No	
DHE-DSS (forward secrecy)	No	Yes	Yes	Yes	Yes	No[72]	
ECDH-ECDSA	No	No	Yes	Yes	Yes	No	
ECDHE-ECDSA (forward secrecy)	No	No	Yes	Yes	Yes	Yes	
ECDH-EdDSA	No	No	Yes	Yes	Yes	No	
ECDHE-EdDSA (forward secrecy)[73]	No	No	Yes	Yes	Yes	Yes	
PSK	No	No	Yes	Yes	Yes	?	
PSK-RSA	No	No	Yes	Yes	Yes	?	
DHE-PSK (forward secrecy)	No	No	Yes	Yes	Yes	Yes	
ECDHE-PSK (forward secrecy)	No	No	Yes	Yes	Yes	Yes	

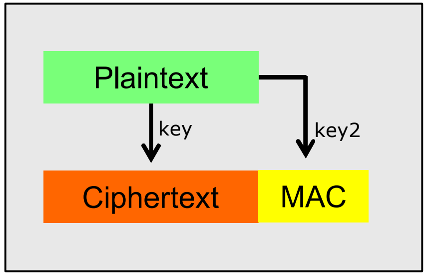
Wikipedia

Order of Encryption and MAC matters?

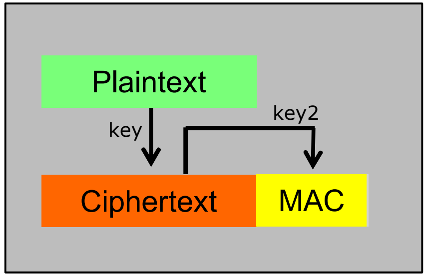
- **MAC-then-Encrypt**
  - Append a keyed MAC to the plaintext and encrypt
  - Only plaintext integrity
  - Receiver must decrypt message before MAC can be checked
  - Padding Oracle Attacks* possible: observe timing and behavior
  - **TLS version 1.2**



- **Encrypt-and-MAC**
  - Encrypt the plaintext, append a keyed MAC of the plaintext
  - Only plaintext integrity
  - Receiver must still decrypt message before MAC can be checked
  - Padding oracle attacks possible
  - **SSH**



- **Encrypt-then-MAC**
  - Encrypt plaintext, append a keyed MAC of the ciphertext
  - Ciphertext integrity but no plaintext integrity
  - If MAC ok, any text produces a cleartext (theoretical problem)
  - Receiver cannot be fed with invalid ciphertexts – good!
  - **IPsec, TLS 1.3****

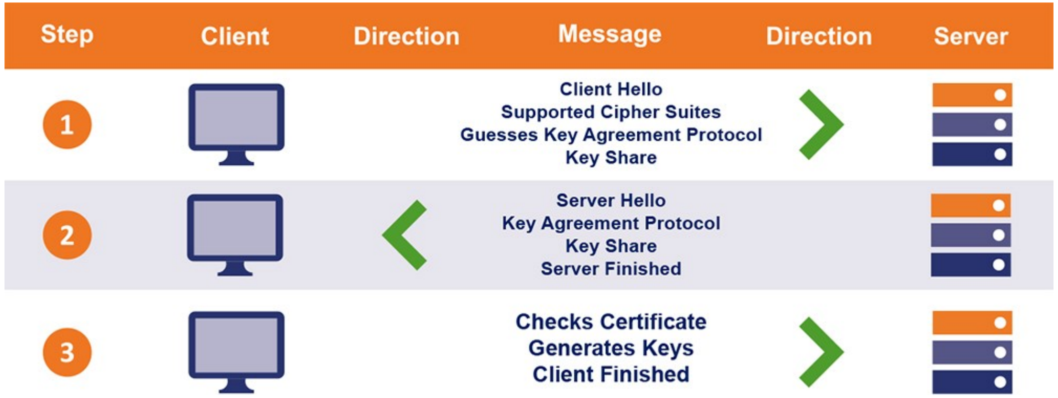


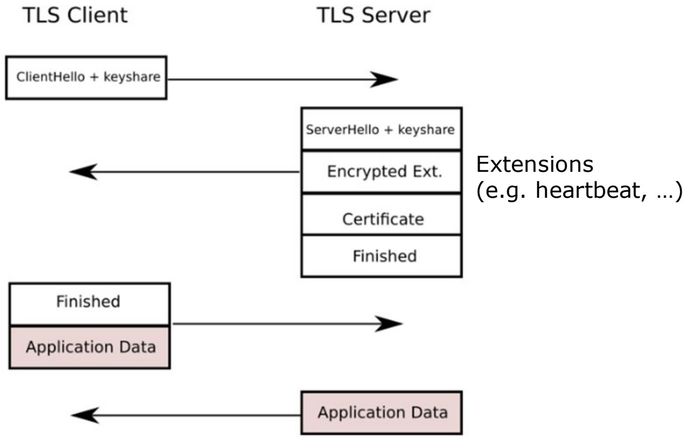
* Time to check MAC leaks information, even worse if packet does not make sense (invalid padding) where MAC is not checked at all

** To minimize implementation bugs, AEAD (Authenticated Encryption with Associated Data) is used which encrypts and creates the MAC in one step

TLS 1.3 with 1-RTT

- Client Hello message contain all necessary information:
  - List of supported cipher suites
  - A guess of key agreement protocol to use and necessary info to generate keys (Key Share)
- Server responds with key agreement protocol + Key Share/certificate





https://www.thesslstore.com/blog/tls-1-3-handshake-tls-1-2/

TLS and security

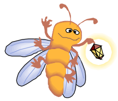
A Detailed Look at RFC 8446 (TLS 1.3)
Describes problems that motivated changes in TLS 1.3

A good summary of various problems in TLS!

https://blog.cloudflare.com/rfc-8446-aka-tls-1-3

TLS and Security – some notes

- Bugs, bugs, bugs – always the biggest security problem



  - Example: If the full certificate chain is not verified, then anyone can sign a certificate
- Poor random number generation
  - Very important with good seeds
- Timing-based crypto-analysis

  - Time to decrypt pre-master key may be measured:
    Time between ClientKeyExchg ←→ ChangeCipherSpec
  - May be hard to do timing in real life
  - Countermeasure: add random time or blinding technique (some of the data signed should be unknown to attacker)
  - Padding Oracle Attack can work (see crypto course lab)
- Traffic analysis (may) reveal size of encrypted user data
  - Size may tell what web page a user accessed
  - Tables can be created for popular web sites
  - Mitigated with random padding in TLS record protocol (wastes bandwidth)

The Heartbleed bug

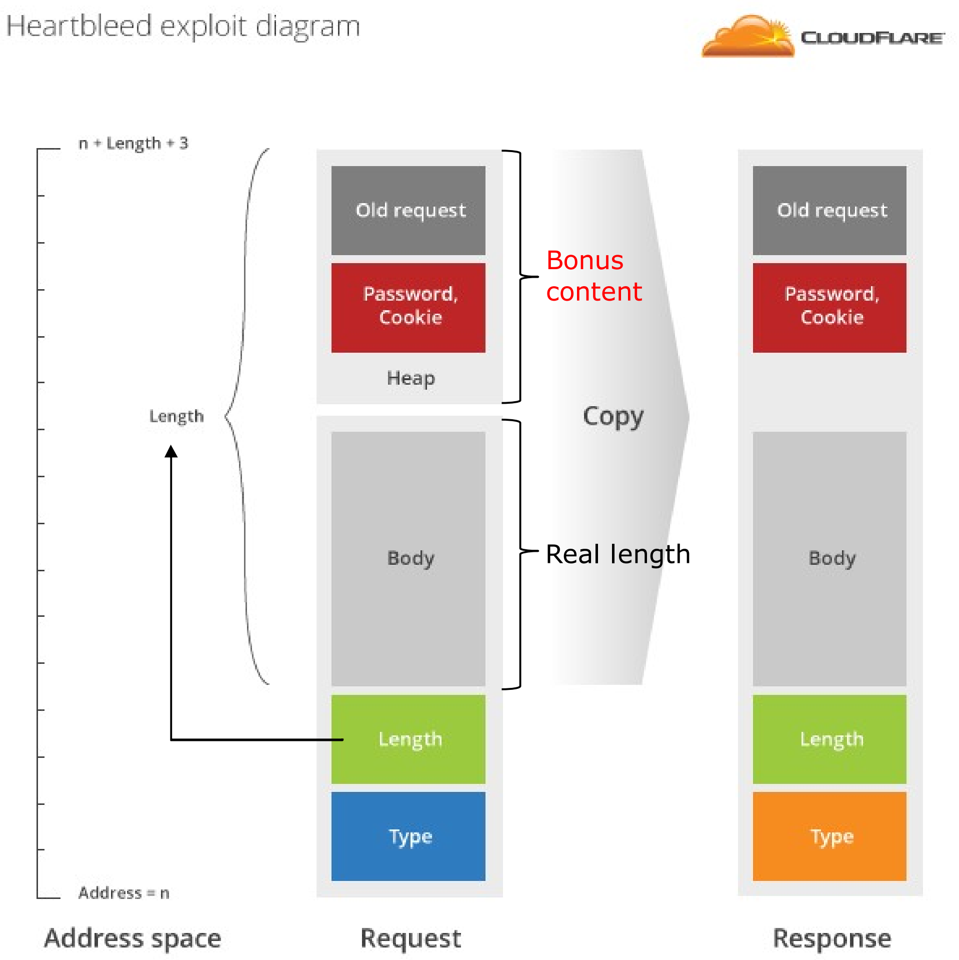
- The Heartbeat protocol is a protocol running on top of the Record Layer (RFC6520)
  - Idea is to check that connection is open
  - And to periodically send a message to make sure firewalls and other equipment does not consider the TCP session closed
- Has two message types:
  - HeartbeatRequest=1 and HeartbeatResponse=2
  - A HeartbeatRequest message can arrive at any time during the lifetime of a connection
  - The receiver echoes back the same message to the sender – trivial protocol!
- A bug is in the OpenSSL's implementation of the TLS heartbeat extension was introduced in OpenSSL March 2012
  - And fixed in OpenSSL 1.0.1g released April 2014
- Problem: there was no check that header length = packet length

From the OpenSSL code:

```
  Changes between 1.0.2 and 1.1.0  [xx XXX xxxx]

+ *) A missing bounds check in the handling of the TLS heartbeat extension
+    can be used to reveal up to 64k of memory to a connected client or
+    server.
+
+    Thanks for Neel Mehta of Google Security for discovering this bug and to
+    Adam Langley <agl@chromium.org> and Bodo Moeller <bmoeller@acm.org> for
+    preparing the fix (CVE-2014-0160)
+    [Adam Langley, Bodo Moeller]
+
```

Just send a minimal/short heartbeat packet but with length field = 64k.
There is no check that the length is the actual packet length.
The reply is a full 64k message!



Old contents on the stack and heap will be returned to the client. Length can be 64 kBytes!

If lucky (e.g. after reboot) the heap contains private keys

Without Perfect Forward Secrecy, all sessions (older and future) can be decrypted!

https://blog.cloudflare.com/answering-the-critical-question-can-you-get-private-ssl-keys-using-heartbleed

Heartbleed was severe – and still is

Massive Open Source Internet Security Flaw Affects Two Thirds of All Web Sites

Rod Trent — Apr 9, 2014

WindowsITPro

Anything that relies on OpenSSL for communication is vulnerable. Here's just a short list example (but it's growing): any Linux-based appliance, routers, Steam, iOS, Android, Mac OS, Smart TVs, DVD/Blu-Ray players, set-top boxes, OpenOffice, Apple Mobile Device Support, BartPE, Trillian, Plesk, ActivePerl, MailEnable, Gene6 FTP, Kindle for PC, IMAPSize, BIND DNS, wput, HP ProLiant System Management and HP Version Control Agent software.

It's being estimated that it may take ten years to clean this one up completely.

Despite best efforts to patch the vulnerability, Heartbleed remains a concern in 2024. The widespread adoption of OpenSSL means that many servers and devices may still be vulnerable, particularly those that have not been consistently patched or updated. Even large consumer sites, which may have conservative SSL/TLS termination equipment, are not immune to the threat posed by Heartbleed.

https://www.class-llc.com/blog/heartbreak-heartbleed-how-today-s-devices-remain-vulnerable

Summary

- TLS is a secure protocol, few changes over the years
  - SSL is now depreciated
  - Version 1.3 have been introduced due to various attacks
- Security protocols have some common properties:
  - Negotiate algorithms and ciphers to use
  - Always derive new key material
  - Only use private keys for authentication – enforce perfect forward secrecy
  - Change keys regularly
- Random number generation and use of strong ciphers essential
- Attackers will not break the ciphers
  - Heartbleed: send message with incorrect headers
  - Logjam: MITM degrades ciphers to make it crackable