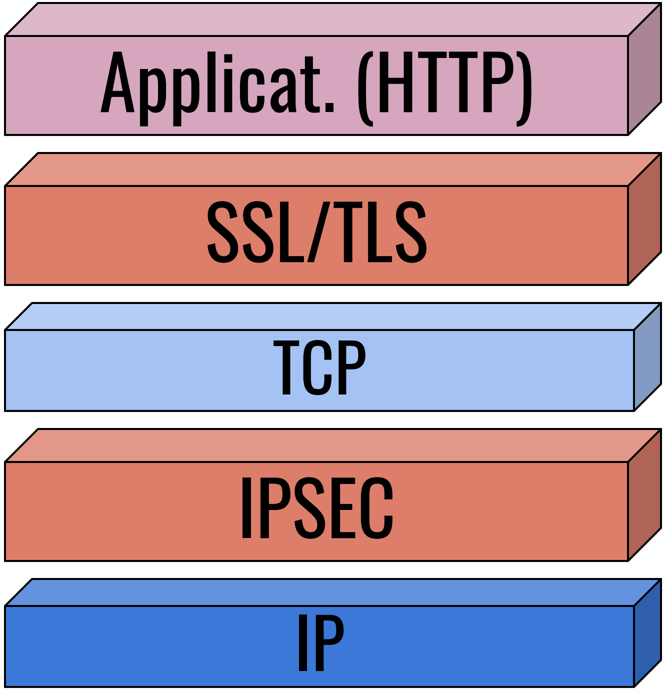
SSL/TLS

Computer and Network Security

Emilio Coppa

Security architecture and protocol stack



Approaches:

- Security in the applications: PGP, HTTPS, S-HTTP, SFTP.
- Security in the protocol stack:
  - SSL/TLS between TCP and application layer
  - IPSEC between IP and TCP

SSL/TLS intro

- Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols that provide security for communications over networks

- TLS and SSL encrypt the segments of network connections at the Transport Layer end-to-end

SSL/TLS intro (2)

- Several versions of the protocols are in widespread use in applications like web browsing, electronic mail, Internet faxing, instant messaging and VoIP
- TLS is an IETF standards track protocol, last updated in RFC 8446 (2018)
- TLS derives from the earlier SSL specifications developed by Netscape Corporation (Taher El-Gamal)

SSL/TLS intro (3)

- SSL/TLS allow client/server applications to communicate across a network in a way designed to prevent eavesdropping, tampering and message forgery
- provide endpoint authentication and communications confidentiality over the Internet using cryptography
  - e.g., RSA security with 1024 and 2048 bit strengths, AES, etc.
- current version TLS 1.3 (RFC 8446 by Mozilla)
  - SSL 1, 2, 3 broken
  - TLS 1.0 having several weakness
  - TLS 1.1 fair
  - TLS 1.2 still good

Main threats addressed

- eavesdropping: the act of secretly listening to private conversation
- tampering: the act of altering something secretly or improperly
- message forgery: sending of a message to deceive the recipient as to whom the real sender is

SSL/TLS intro (4)

- In typical end-user/browser usage, TLS authentication is unilateral: only server is authenticated, but not vice versa
- TLS also supports mutual authentication
  - provided that partners diligently scrutinize identity information

SSL/TLS intro (5)

- Mutual authentication requires that the TLS client-side also holds a certificate (which is not usually the case in the end-user/browser scenario)
  - Unless TLS-PSK (TLS with pre-shared keys), the Secure Remote Password (SRP) protocol, or some other protocol is used that can provide strong mutual authentication in the absence of certificates

SSL/TLS intro (6)

SSL/TLS involves three basic phases:

1. Peer negotiation for algorithm support
2. Key exchange and authentication
3. Symmetric cipher encryption and message authentication

SSL/TLS intro (7)

- During first phase, client and server negotiate cipher suites, which determine ciphers to be used, key exchange and authentication algorithms, as well as message authentication codes (MACs)
  - key exchange and authentication algorithms are typically public key algorithms, or, as in TLS-PSK, preshared keys (PSKs) could be used
  - message authentication codes are made up from cryptographic hash functions using the HMAC construction for TLS, and a non-standard pseudorandom function for SSL.

SSL/TLS: typical algorithms

- For key exchange: RSA, Diffie-Hellman, ECDH (Elliptic Curve Diffie–Hellman), SRP (Secure Remote Password protocol), PSK
- For authentication: RSA, DSA, ECDSA (Elliptic Curve Digital Signature Algorithm)
- Symmetric ciphers: RC4, Triple DES, AES, IDEA, DES, or Camellia. In older versions of SSL, RC2 was also used.
- For cryptographic hash function: HMAC-MD5 or HMAC-SHA are used for TLS, MD5 and SHA for SSL, while older versions of SSL also used MD2 and MD4.

SSL/TLS and digital certificates

The key information and certificates necessary for TLS are handled in the form of X.509 certificates, which define required fields and data formats.

How SSL/TLS works: basic idea

- Client and server negotiate a stateful connection by using a handshaking procedure. During handshake, client and server agree on various parameters used to establish connection's security and exchange some random numbers
- Handshake begins when client connects to TLS-enabled server requesting a secure connection and presents a list of supported ciphers and hash functions
- From this list, server picks the strongest cipher and hash function that it also supports and notifies client of the decision

How SSL/TLS works: basic idea (2)

- Server sends back its identification in the form of a digital certificate X.509
- Client may contact the CA and confirm that the certificate is authentic and not revoked before proceeding
  - modern browsers support Extended Validation certificates and can use the Online Certificate Status Protocol (OCSP)

How SSL/TLS works: basic idea (3)

- For generating session keys used for secure connection, for instance when using RSA for key exchange the client may encrypt a secret number (S) with server's public key (PbK), and sends result to server.
  - Only server is able to decrypt it (with its private key (PvK)): this is the one fact that makes the keys hidden from third parties, since only the server and the client have access to this data.

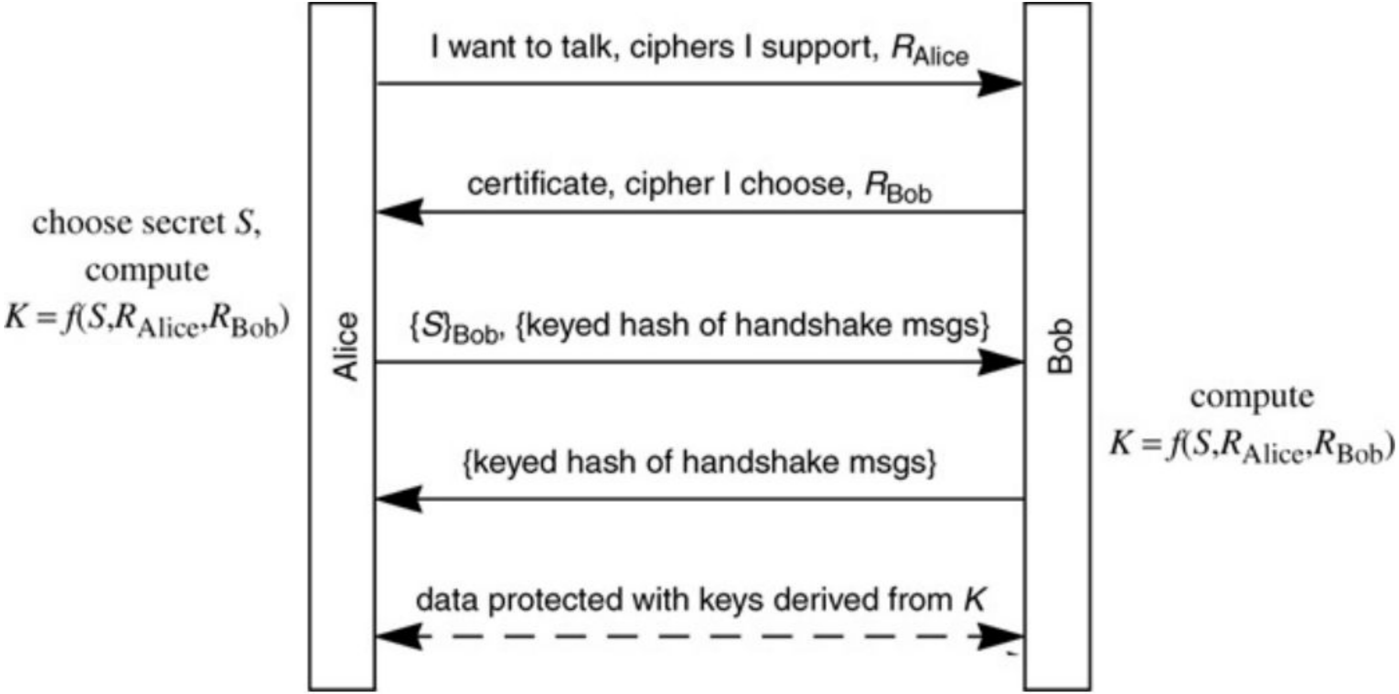
How SSL/TLS works: basic idea (4)

- Client knows PbK and RN, and server knows PvK and (after decryption of the client's message) S. A third party may only know PbK, unless PvK has been compromised.
- From the secret number, both parties generate key material for encryption and decryption.

How SSL/TLS works: basic idea (5)

- This concludes the handshake and begins the secured connection, which is encrypted and decrypted with the key material until the connection closes.
- If any one of the above steps fails, the TLS handshake fails, and the connection is not created.

How SSL/TLS works: basic idea (6)



Remark. This is just a basic idea. TLS can generate different messages based on specific cipher suite in use.

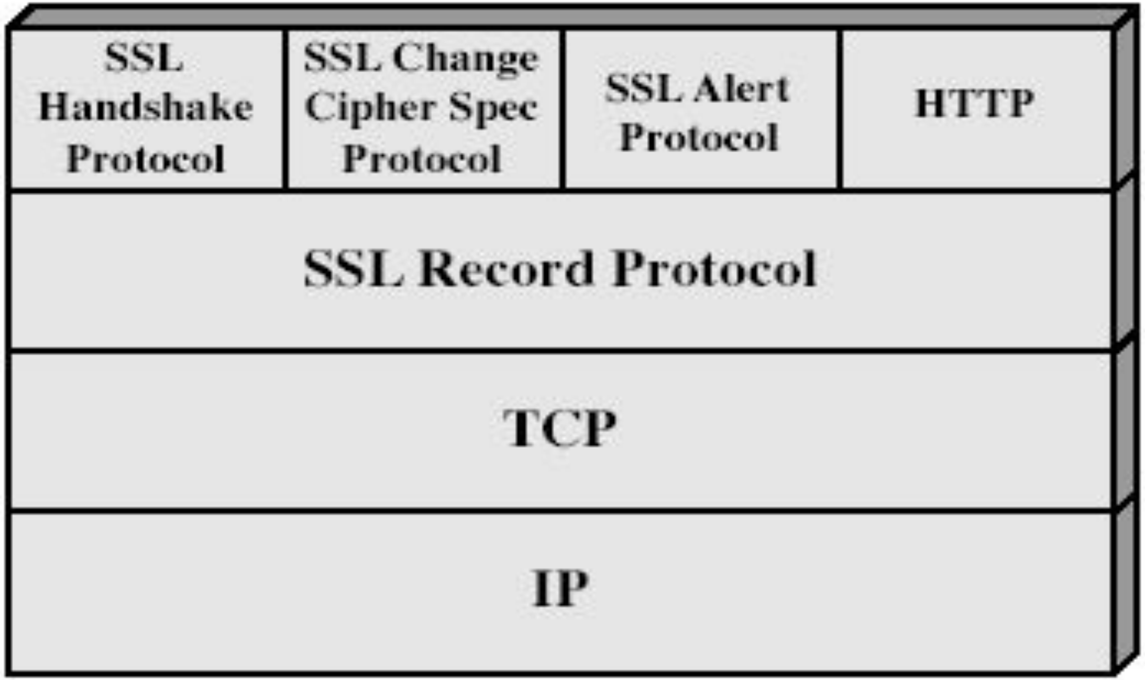
SSL (Secure Socket Layer)

- transport layer security service
- uses TCP to provide a reliable end-to-end service
  - originally developed by Netscape
  - version 3 designed with public input
  - subsequently became Internet standard known as TLS (Transport Layer Security)

- SSL has two layers of protocols

SSL Architecture

2 layers of
protocol



SSL Architecture (2)

- SSL session
  - an association between client & server
  - created by the Handshake Protocol
  - defines a set of cryptographic parameters
  - maybe shared by multiple SSL connections (re-negotiating can be onerous hence a “resume” procedure is allowed)
  - stateful
- SSL connection
  - a transient, peer-to-peer, communications link
  - associated with 1 SSL session, still require a minimal handshake procedure
  - stateful

Sessions and Connections

- between any pair of parties there may be multiple secure connections
  - there may also be multiple simultaneous sessions between parties, but this feature is not used in practice
- several states associated with each session
  - once a session is established, there is a current operating state for both read and write (i.e., receive and send)
  - during Handshake Protocol, pending read and write states are created
  - after conclusion of Handshake Protocol, the pending states become the current states

Parameters defining session state

- Session identifier: arbitrary byte sequence chosen by the server to identify an active or resumable session state
- Peer certificate: X509.v3 certificate of the peer. This element of the state may be null
- Compression method: algorithm used to compress data prior to encryption.

Parameters defining session state (2)

- Cipher spec: specifies the bulk data encryption algorithm (such as null, DES, etc.) and hash algorithm (such as MD5 or SHA-l) used for MAC calculation. It also defines cryptographic attributes such as the hash_size.
- Master secret: 48-byte secret shared between the client and server.
- Is resumable: flag indicating whether the session can be used to initiate new connections.

Parameters defining session state (3)

- Server and client random: byte sequences that are chosen by the server and client for each connection.
- Server write MAC secret: secret key used in MAC operations on data sent by the server
- Client write MAC secret: secret key used in MAC operations on data sent by the client.
- Server write key: conventional encryption key for data encrypted by the server and decrypted by the client

Parameters defining session state (4)

- Client write key: The conventional encryption key for data encrypted by the client and decrypted by the server.

- Initialization Vectors: when a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL Handshake Protocol. Thereafter the final ciphertext block from each record is preserved for use as the IV with the following record

Parameters defining session state (5)

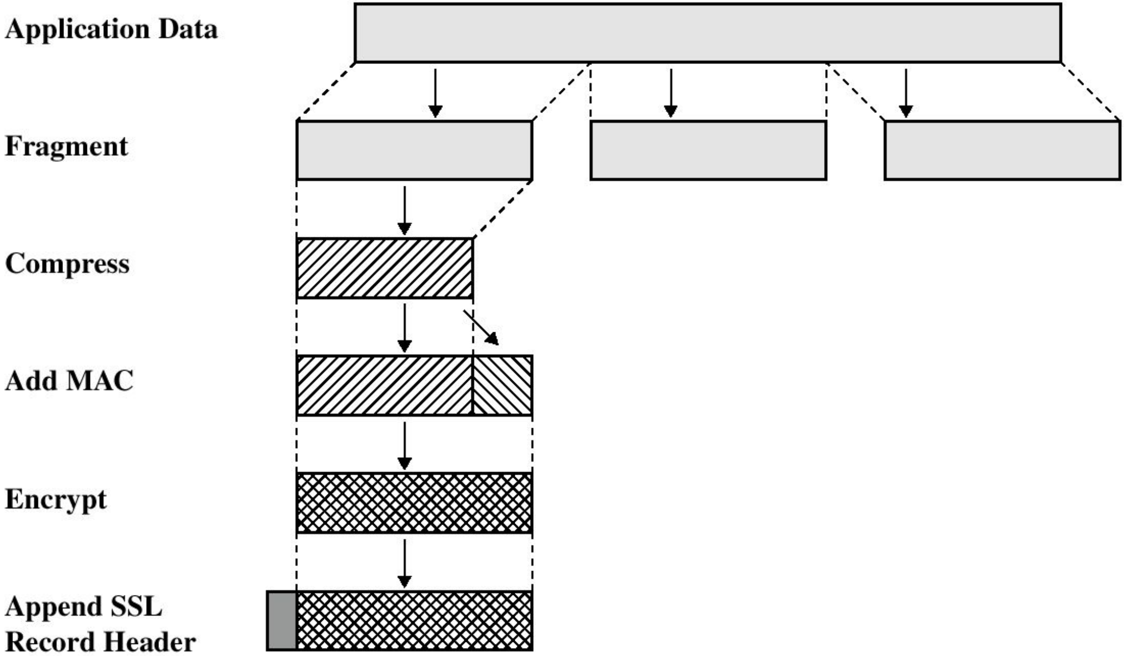
- Sequence numbers: each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a change cipher spec message, the appropriate sequence number is set to zero. Sequence numbers may not exceed $2^{64} - 1$.

SSL Record Protocol

Two main services

- confidentiality
  - using symmetric encryption with a shared secret key defined by Handshake Protocol
  - IDEA, RC2-40, DES-40, DES, 3DES, Fortezza, RC4-40, RC4-128
  - message is compressed before encryption
- message integrity
  - using a MAC with shared secret key
  - similar to HMAC but with different padding

SSL Record Protocol (2)



Authentication: MAC

In SSL, similar to HMAC (uses concatenation instead of XOR)

Hash(MAC_secret_key || pad2
|| hash(MAC_secret_key || pad1 || seqNum ||
SSLcompressed.type ||
SSLcompressed.length ||
SSLcompressed.fragment))

pad1: 0x36 repeated 48 times (MD5); 40 times (SHA-1)

pad2: 0x5C repeated 48 times (MD5); 40 times (SHA-1)

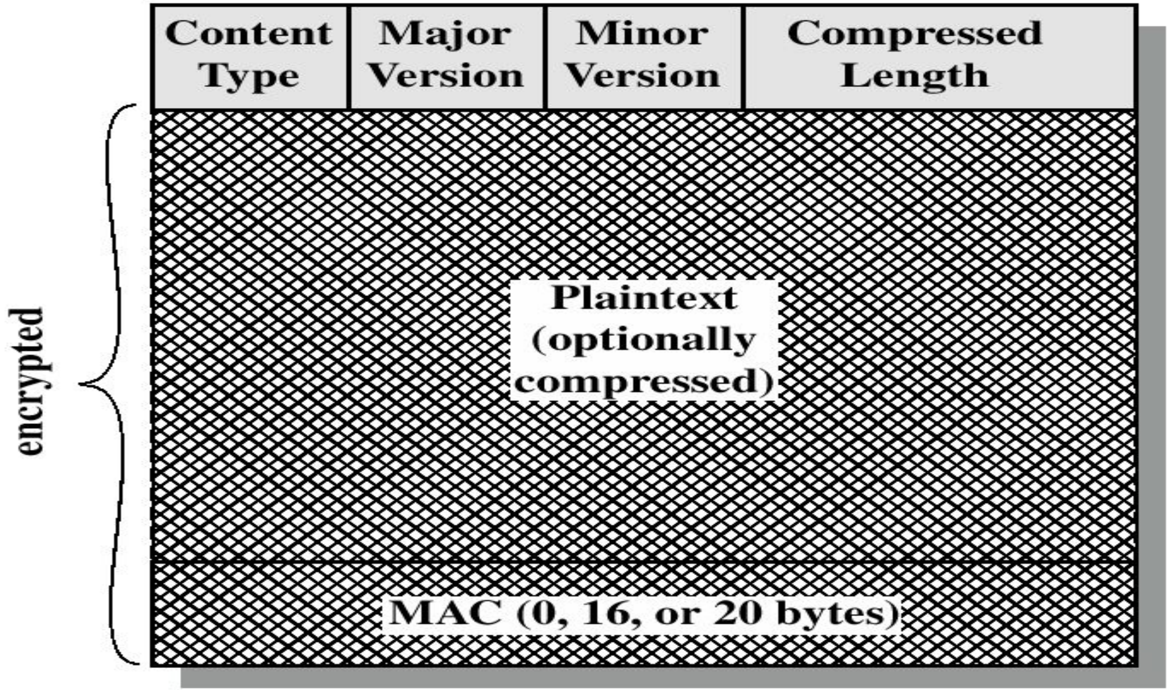
SSLcompressed.type: high level protocol used to process segment

Encoding methods

- segment into blocks of 2^{14} = 16384 bytes
- compression (optional):
  - must be no lossy and must guarantee to reduce pack size
  - default in SSLv3 : no compression
- MAC computation (see previous slide)
  - on compressed data
- several (symmetric) encryption methods:
  - block ciphers: IDEA (128) RC2-40, DES-40, DES (56), 3DES (168),
  - Stream Cipher: RC4-40, RC4-128
  - Smart card: Fortezza

SSL record

fields of the header:



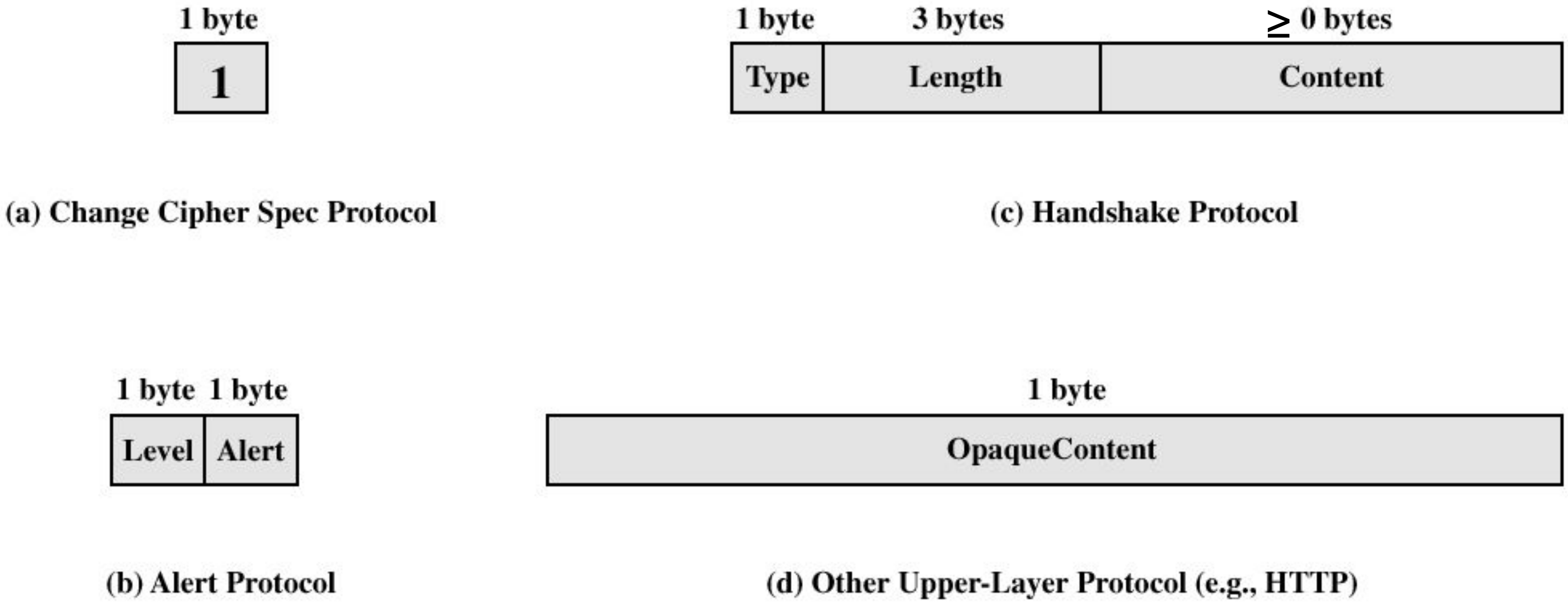
Fields

- Content Type (8 bits)
  - The higher layer protocol used to process the enclosed fragment (change_cipher_spec, alert, handshake, and application_data). The first three are the SSL-specific protocols; no distinction is made among the various applications (e.g., HTTP) that might use SSL)
- Major Version (8 bits)
  - Indicates major version of SSL in use. For SSLv3 and TLS1.2, the value is 3
- Minor Version (8 bits)
  - Indicates minor version in use. For SSLv3, the value is 0; for TLS1.2 it is 3
- Compressed Length (16 bits)
  - The length in bytes of the plaintext fragment (or compressed fragment if compression is used).

Versions

Major version	Minor version	Version type
3	0	SSL 3.0
3	1	TLS 1.0
3	2	TLS 1.1
3	3	TLS 1.2

SSL Payload


1 byte
1
(a) Change Cipher Spec Protocol
1 byte
3 bytes
≥ 0 bytes
Type
Length
Content
(c) Handshake Protocol
1 byte 1 byte
Level
Alert
(b) Alert Protocol
1 byte
OpaqueContent
(d) Other Upper-Layer Protocol (e.g., HTTP)

SSL Change Cipher Spec Protocol

- one of 3 SSL specific protocols which use the SSL Record protocol
- a single message
- to cause the pending state to be copied into the current state, which updates the cipher suite to be used on this connection
- usually sent just after handshaking

SSL Alert Protocol

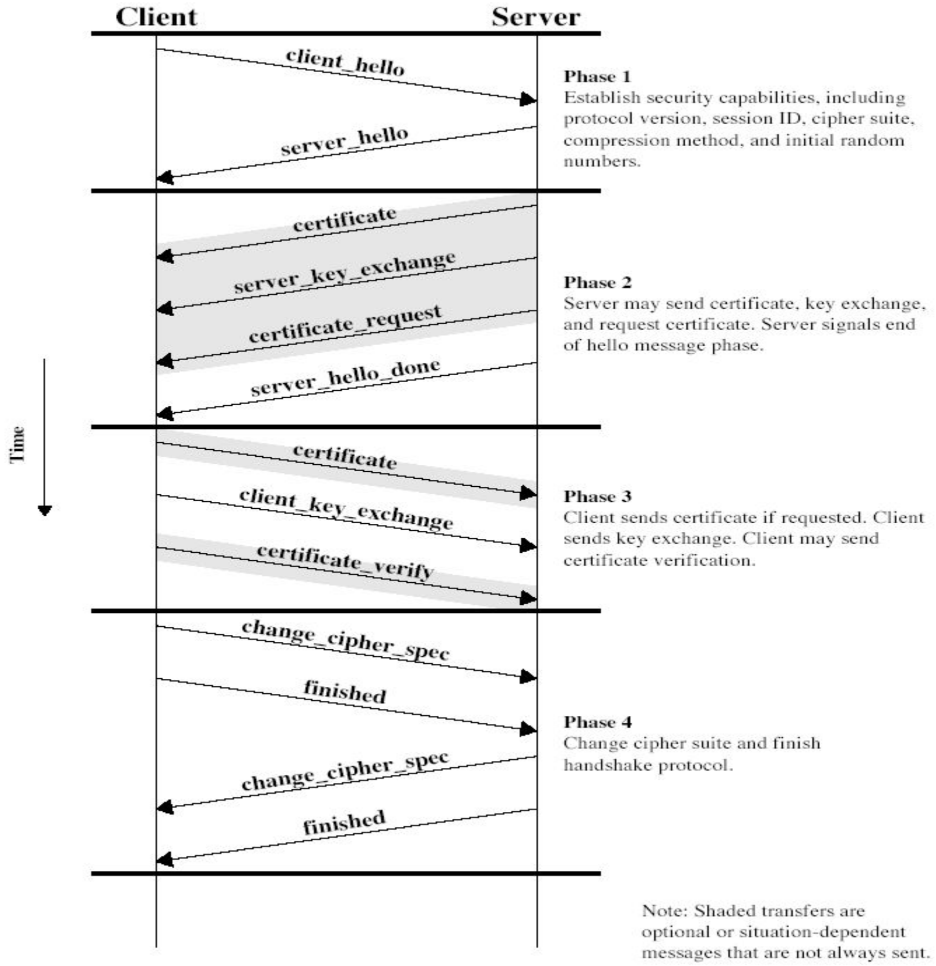
- conveys SSL-related alerts to peer entity
- severity
  - two possibilities: *warning* or *fatal* (close connection)
- specific alert
  - fatal: unexpected message, bad record mac, decompression failure, handshake failure, illegal parameter
  - warning: close notify, no certificate, bad certificate, unsupported certificate, certificate revoked, certificate expired, certificate unknown
- compressed & encrypted like all SSL data

SSL Handshake Protocol

Most complex part of SSL

- allows server & client to:
  - authenticate each other
  - to negotiate encryption & MAC algorithms
  - to negotiate cryptographic keys to be used
- comprises a series of messages in phases
  - Establish Security Capabilities
  - Server Authentication and Key Exchange
  - Client Authentication and Key Exchange
  - Finish

SSL Handshake Protocol



Handshake protocol

4 phases:

1. Hello: determine security capabilities
2. Server sends certificate, asks for certificate and starts exchange session keys
3. Client sends certificate and continues exchanges of keys
4. End of handshake protocol: encoded methods changes

Remark: some requests are optional and there is clear separation between handshake and the rest (to avoid attacks)

Handshake: parameters

message type (1^{st} byte of payload)	parameters
Hello-request	null [may be sent by server at any time: notification that client should begin negotiation process anew by sending a client hello message when convenient; this message is ignored by client in some cases]
Client-hello	version, 32-bit timestamp + 28 random bytes (nonce), sessionID, cipher suite and compression method
Server_hello	<same as Client_hello>
Certificate	X.509v3 chain of certificates
Server_key_exchange	info, signature of mess.
Certificate_request	type of cert., authority
Server_done	null
Certificate_verify	signature of certificate
Client_key_exchange	info, signature of mess.
Finished	hash of all exchanged messages (integrity of handshake protocol)

Handshake Protocol: phase 1

Initialization

- : Client_hello: client to server
  - Version = highest SSL version used by client
  - 32-bit timestamp + 28 bytes random (a pseudo number generator is required)
  - sessionID: = 0 new connection in new session; ≠0 update previous connection
  - Cipher suite: list that contains the combinations of cryptographic algorithms supported by the client, in decreasing order of preference. Each element of the list (each cipher suite) defines both a key exchange algorithm and a CipherSpec.
  - Compression algorithms: ordered sequence of acceptable algorithms
- : Server_hello: server to client
  - same as all above (if sessionID of client = 0 generates new sessionID)

Cipher suite

Algorithms for key exchange

- RSA : session key is encoded with server public key
- Diffie-Hellman (several versions)
  - Fixed
  - Ephemeral
  - Anonymous
- Fortezza

CipherSpec

- Crypto algorithm (either a stream algo or a block algo)
- MAC algorithm
- Hash (in byte): 0, 16 (for MD5), 20 (for SHA-1)
- Key material – info used to generate session keys
- Info for initializing CBC (initial vector)

Fixed Diffie-Hellman (or static DH)

- Diffie-Hellman key exchange in which server's certificate contains Diffie-Hellman public parameters signed by the certificate authority (CA)
- Client provides its Diffie-Hellman public key parameters either in a certificate, if client authentication is required, or in a key exchange message. The client can perform static (called static-static DH) or ephemeral DH (more common, called static-ephemeral DH).
- When both sides perform static DH, i.e., static-static DH, then this method results in a fixed secret key between two peers, based on the Diffie-Hellman calculation using the fixed public keys and there is no forward secrecy.

Ephemeral Diffie-Hellman

- Used to create ephemeral (temporary, one-time) secret keys. In this case, the Diffie-Hellman public keys are exchanged, signed using the sender's private RSA or DSS key.
- The receiver can use the corresponding public key to verify the signature. Certificates are used to authenticate the public keys.
- This would appear to be the most secure of the three Diffie-Hellman options because it results in a temporary, authenticated key.

Anonymous Diffie-Hellman

- The base Diffie-Hellman algorithm is used, with no authentication.
- Each side sends its public Diffie-Hellman parameters to the other, with no authentication. This approach is vulnerable to man-in-the-middle attacks, in which the attacker conducts anonymous Diffie-Hellman with both parties.

Handshake Protocol: phase 2

Server authentication and key exchange

Server to client

- Certificate: X.509 certificate chain (optional)
- Server_key_exchange (optional)
  - a signature is created by taking the hash of a message and encrypting it with the sender's private key. In this case the hash is defined as
    - hash(ClientHello.random || ServerHello.random || ServerParams)
  - So the hash covers also the two nonces from the initial hello messages. ServerParams will be specified in the next slides
- Certificate_request: (optional)
- Server_hello_done: I am done and I wait for answers

Certificate message

- The server begins this phase by sending its certificate, if it needs to be authenticated; the message contains one (or a chain of) X.509 certificates.
- The certificate message is required for any agreed-on key exchange method except anonymous Diffie-Hellman.
  - If fixed Diffie-Hellman is used, this certificate message functions as the server's key exchange message because it contains the server's public Diffie- Hellman parameters.

server_key_exchange not needed

A server_key_exchange message is not required in two instances:

(1) The server has sent a certificate with fixed Diffie-Hellman parameters, or

(2) RSA key exchange is to be used.

server_key_exchange needed

- Anonymous Diffie-Hellman. Message content consists of the two global D.H. values (a prime number and a primitive root of that number) plus the server's public D.H. key
- Ephemeral Diffie-Hellman. Message content includes the three D.H. parameters provided for anonymous D.H. plus a signature of those parameters.
- RSA key exchange, in which the server is using RSA but has a signature-only RSA key. Server creates a temporary RSA public/private key pair and use server_key_exchange message to send public key. Message content includes the two parameters of the temporary RSA public key (exponent and modulus) plus a signature of those parameters.

Handshake Protocol: phase 3

Client authentication

- Client verifies server certificates and parameters
- Client to server

    Client Certificate and info to verify it: (if asked)

    Message for key exchange (Client_key_exchange)

Deriving secrets

- client and server exchange random nonces during {client, server}_hello
- they obtain/compute the pre_master_key:
  - [RSA] client chooses a pre_master_key and sends it to server encrypted with public key of the server during a client_key_exchange
  - [DHKE] client and server independently compute the premaster_key (they still have to exchange DH params during {client, server}_key exchange)

- they derive a master_key using the pre_master_key:

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

This step is independent from the key exchange suite and this why we need a also a pre_master_key. The pre_master_key can be thrown away after computing the master_key.

Deriving secrets (2)

- they derive several secrets from the master_key:

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

- Then the key block is divided to provide six keys used for different operations:
  - 2 encryption keys
  - 2 MAC keys
  - 2 IV (when needed by encryption primitive)

Handshake Protocol: phase 4

Client to server

Message: Change_cipher_spec

Finished message under new algorithms and keys (new cipher_spec)

Server sends back

Message: Change_cipher_spec

Finished message under new algorithms and keys (new cipher_spec)

Change_cipher_spec

This command indicates that the contents of subsequent SSL record data sent by the client (server) during the SSL session will be encrypted. The 5-byte SSL record headers are never encrypted.

SSL Session Resumption

To avoid the cost of SSL key exchange, when establishing a new connection the server chooses a session id. Client can resume a connection by indicating into client_hello message the session id. The server can accept (same session id in the server_hello) or reject (session id different from the one from the client in the server_hello) the resumption.

If resumption is performed, then same master_key will be used by client and server. After hello, they still have to exchange change_cipher_spec and finished messages.

SSL vs TLS

TLS is a IETF standard RFC 2246 similar to SSLv3 with minor differences:

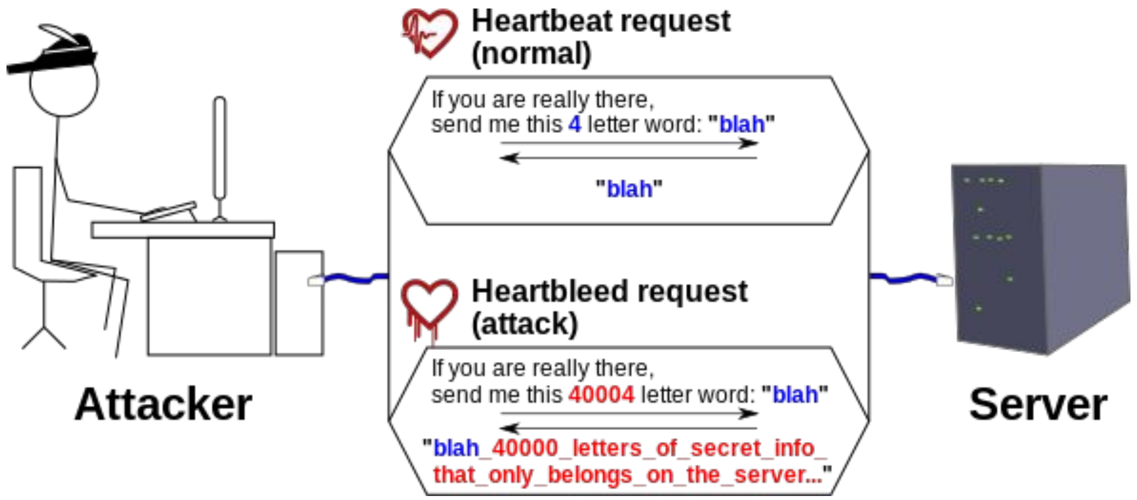
- in record format version number
- uses HMAC for MAC
- a pseudo-random function expands secrets
- has additional alert codes
- some changes in supported ciphers
- changes in certificate negotiations
- changes in use of padding
- session tickets

Paying in the Web: SSL

- SSL and credit card are used for paying
  - simple
  - no need of specialized software
  - compliant with credit card mechanisms
  - most used method for paying in the web
- Problems
  - malicious sellers have info on clients
  - clients can in principle refuse to pay (there is no signature)
  - many disputes (20%- 60%)
  - expensive method for the shop

SSL/TLS: heartbeat & heartbleed

- severe vulnerability in OpenSSL allowing attackers obtaining huge amounts of data from the "secure" server, by directly accessing to its memory
- according to some sources it was known since 2012
- fixed in April 2014



POODLE attack

- SSL 3.0 vulnerable to POODLE attack, based on MITM
  - POODLE: Padding Oracle On Downgraded Legacy Encryption
- attackers need (on average) to make 256 SSL 3.0 requests to reveal one byte of encrypted messages
- the Google Security Team discovered this vulnerability in Sept. 2014

TLS 1.3 vs 1.2

Main news in TLS 1.3

- The list of supported symmetric algorithms has been pruned of all legacy algorithms. The remaining algorithms all use Authenticated Encryption with Associated Data (AEAD) algorithms
- A zero-RTT (0-RTT) mode was added, saving a round-trip at connection setup for some application data at the cost of certain security properties.
- All handshake messages after the ServerHello are now encrypted.
- Key derivation functions have been re-designed, with the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) being used as a primitive.
- The handshake state machine has been restructured to be more consistent and remove superfluous messages.
- ECC is now in the base spec and includes new signature algorithms. Point format negotiation has been removed in favour of single point format for each curve.
- Compression, custom DHE groups, and DSA have been removed, RSA padding now uses PSS.
- TLS 1.2 version negotiation verification mechanism was deprecated in favour of a version list in an extension.
- Session resumption with and without server-side state and the PSK-based ciphersuites of earlier versions of TLS have been replaced by a single new PSK exchange.

TLS and SSL protocols

Protocol	Published	Status
SSL 1.0	Unpublished	Unpublished
SSL 2.0	1995	Deprecated in 2011 (RFC 6176)
SSL 3.0	1996	Deprecated in 2015 (RFC 7568)
TLS 1.0	1999	Deprecated in 2020[11][12][13]
TLS 1.1	2006	Deprecated in 2020[11][12][13]
TLS 1.2	2008	
TLS 1.3	2018	

TLS and SSL key exchange 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	
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[51]	
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)[52]	No	No	Yes	Yes	Yes	Yes	Defined for TLS 1.2 in RFCs
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	
SRP	No	No	Yes	Yes	Yes		
SRP-DSS	No	No	Yes	Yes	Yes		
SRP-RSA	No	No	Yes	Yes	Yes		
Kerberos	No	No	Yes	Yes	Yes		
DH-ANON (insecure)	No	Yes	Yes	Yes	Yes		
ECDH-ANON (insecure)	No	No	Yes	Yes	Yes		
GOST R 34.10-94 / 34.10-2001[53]	No	No	Yes	Yes	Yes		Proposed in RFC drafts

TLS and SSL cipher suite

Cipher			Protocol version						Status
Type	Algorithm	Nominal strength (bits)	SSL 2.0	SSL 3.0 [n 1][n 2][n 3][n 4]	TLS 1.0 [n 1][n 3]	TLS 1.1 [n 1]	TLS 1.2 [n 1]	TLS 1.3	
Block cipher with mode of operation	AES GCM[54][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	Secure	Defined for TLS 1.2 in RFCs
	AES CCM[55][n 5]		N/A	N/A	N/A	N/A	Secure	Secure	
	AES CBC[n 6]		N/A	Insecure	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	Camellia GCM[56][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	N/A	
	Camellia CBC[57][n 6]		N/A	Insecure	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	ARIA GCM[58][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	N/A	
	ARIA CBC[58][n 6]		N/A	N/A	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	SEED CBC[59][n 6]	128	N/A	Insecure	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	3DES EDE CBC[n 6][n 7]	112[n 8]	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	
	GOST 28147-89 CNT[53][n 7]	256	N/A	N/A	Insecure	Insecure	Insecure	N/A	Defined in RFC 4357
	IDEA CBC[n 6][n 7][n 9]	128	Insecure	Insecure	Insecure	Insecure	N/A	N/A	Removed from TLS 1.2
	DES CBC[n 6][n 7][n 9]	56	Insecure	Insecure	Insecure	Insecure	N/A	N/A	
		40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	Forbidden in TLS 1.1 and later
	RC2 CBC[n 6][n 7]	40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	
Stream cipher	ChaCha20-Poly1305[64][n 5]	256	N/A	N/A	N/A	N/A	Secure	Secure	Defined for TLS 1.2 in RFCs
	RC4[n 11]	128	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	Prohibited in all versions of TLS by RFC 7465
		40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	
None	Null[n 12]	–	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	Defined for TLS 1.2 in RFCs

TLS and SSL MAC algorithms

Algorithm	SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1	TLS 1.2	TLS 1.3	Status
HMAC-MD5	Yes	Yes	Yes	Yes	Yes	No	Defined for TLS 1.2 in RFCs
HMAC-SHA1	No	Yes	Yes	Yes	Yes	No	
HMAC-SHA256/384	No	No	No	No	Yes	No	
AEAD	No	No	No	No	Yes	Yes	
GOST 28147-89 IMIT[53]	No	No	Yes	Yes	Yes		Proposed in RFC drafts
GOST R 34.11-94[53]	No	No	Yes	Yes	Yes		

https://www.howsmyssl.com/

How's My SSL? Home About API

Your SSL client is

Probably Okay.

Check out the sections below for information about the SSL/TLS client you used to render this page.

Yeah, we really mean "TLS", not "SSL".

How's My SSL? Home About API

Version

Good Your client is using TLS 1.3, the most modern version of the encryption protocol. It gives you access to the fastest, most secure encryption possible on the web.

Learn More

Ephemeral Key Support

Good Ephemeral keys are used in some of the cipher suites your client supports. This means your client may be used to provide forward secrecy if the server supports it. This greatly increases your protection against snoopers, including global passive adversaries who scoop up large amounts of encrypted traffic and store them until their attacks (or their computers) improve.

Learn More

Session Ticket Support

Good Session tickets are supported in your client. Services you use will be able to scale out their TLS connections more easily with this feature.

Learn More

How's My SSL? Home About API

TLS Compression

Good Your TLS client does not attempt to compress the settings that encrypt your connection, avoiding information leaks from the CRIME attack.

Learn More

BEAST Vulnerability

Good Your client is not vulnerable to the BEAST attack because it's using a TLS protocol newer than TLS 1.0. The BEAST attack is only possible against clients using TLS 1.0 or earlier using Cipher-Block Chaining cipher suites that do not implement the 1/n-1 record splitting mitigation.

Learn More

Insecure Cipher Suites

Good Your client doesn't use any cipher suites that are known to be insecure.

Learn More

Given Cipher Suites

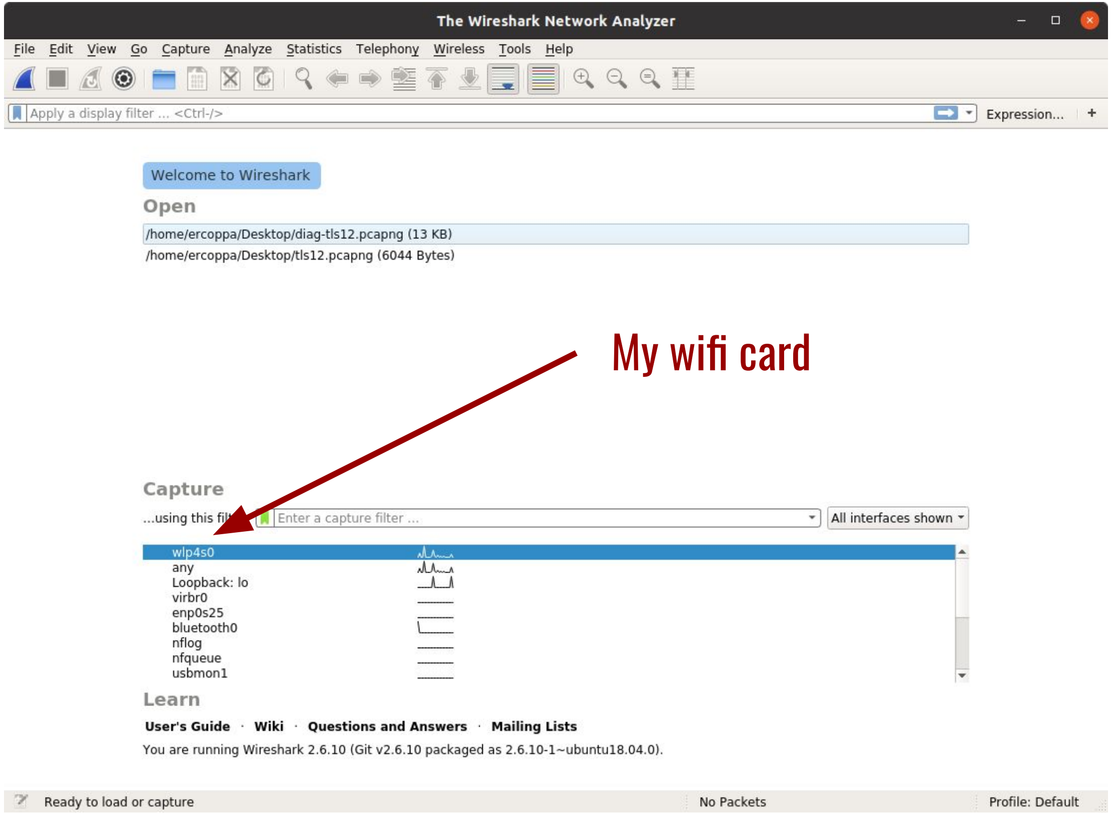
The cipher suites your client said it supports, in the order it sent them, are:

- TLS_GREASE_IS_THE_WORD_9A
- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA

Demo: TLSv1.2 connection

Let us analyze a TLSv1.2 connection with WireShark:

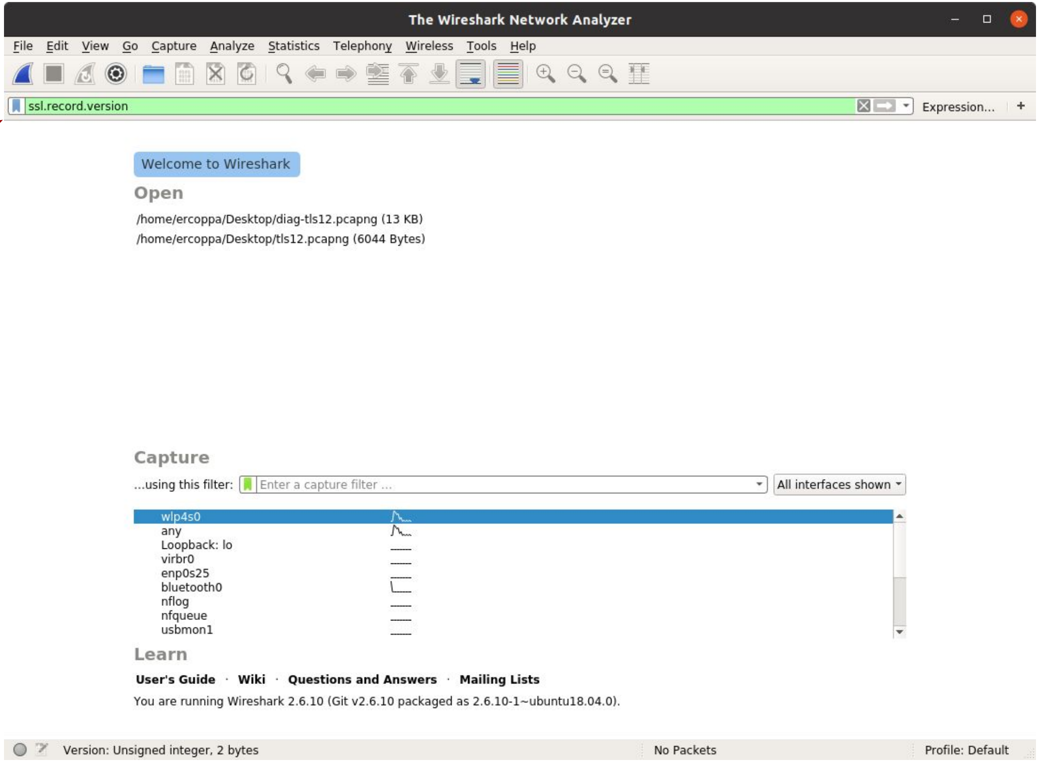
- start wireshark (you may need root privileges to capture packets)



Demo: TLSv1.2 connection (2)

Let us analyze a TLSv1.2 connection with WireShark:

- apply filter “ssl.record.version” to keep only TLS/SSL packets

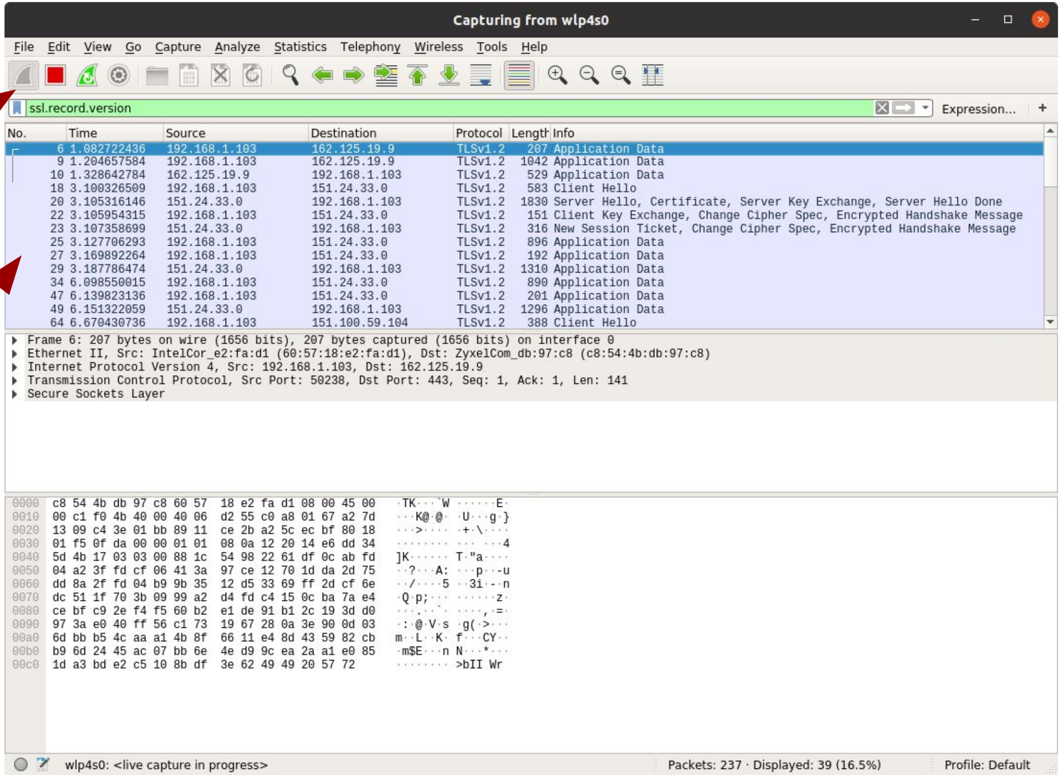


Demo: TLSv1.2 connection

Let us analyze a TLSv1.2 connection with WireShark:

- Push button (shark) start capturing packets

  You will likely immediately see packets as your PC is constantly communicating over the internet.



Demo: TLSv1.2 connection (4)

Let us analyze a TLSv1.2 connection with WireShark:

- Generate a new SSL/TLS connection, e.g., using your browser or wget

```
ercoppa@thinkpad:~/Downloads$ wget https://www.diag.uniroma1.it/
--2020-12-08 13:20:46--  https://www.diag.uniroma1.it/
Resolving www.diag.uniroma1.it (www.diag.uniroma1.it)... 151.100.59.104
Connecting to www.diag.uniroma1.it (www.diag.uniroma1.it)|151.100.59.104|:443...
 connected.
HTTP request sent, awaiting response... 200 OK
Length: 30395 (30K) [text/html]
Saving to: ‘index.html’

index.html          100%[===================>]  29,68K  --.-KB/s    in 0,01s

2020-12-08 13:20:46 (2,72 MB/s) - ‘index.html’ saved [30395/30395]
```

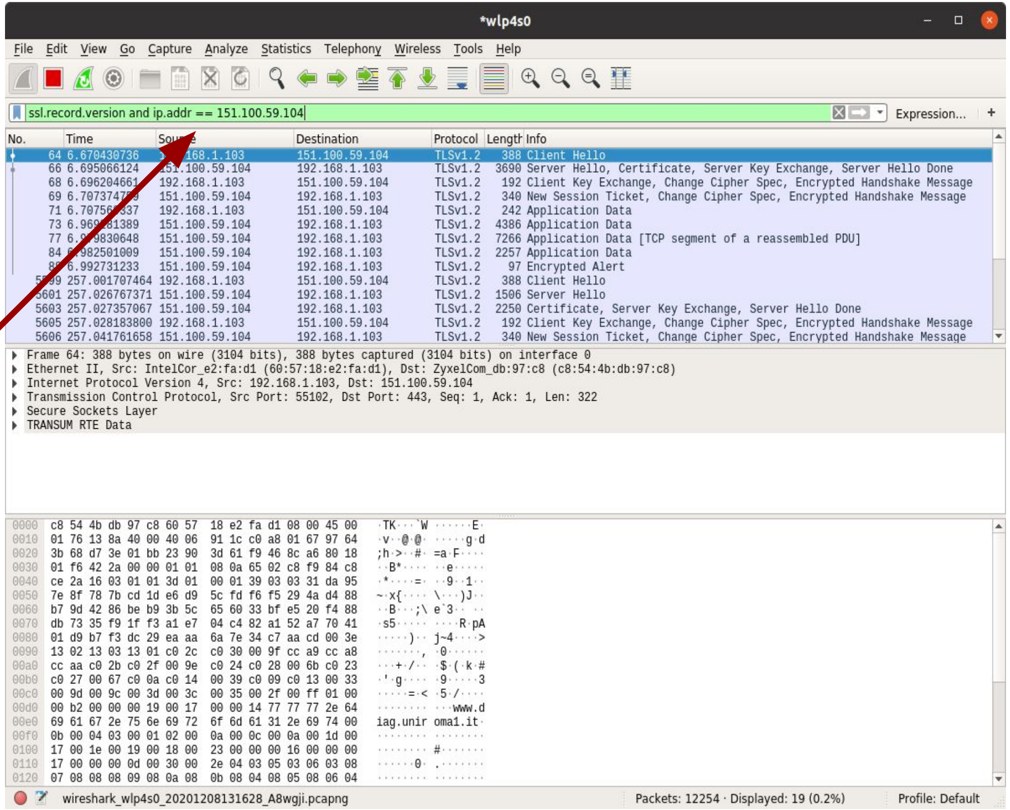
In this example, we start a connection with Sapienza DIAG at 151.100.59.104

Demo: TLSv1.2 connection (5)

Let us analyze a TLSv1.2 connection with WireShark:

- We can now analyze packets exchanged with ip 151.100.59.104

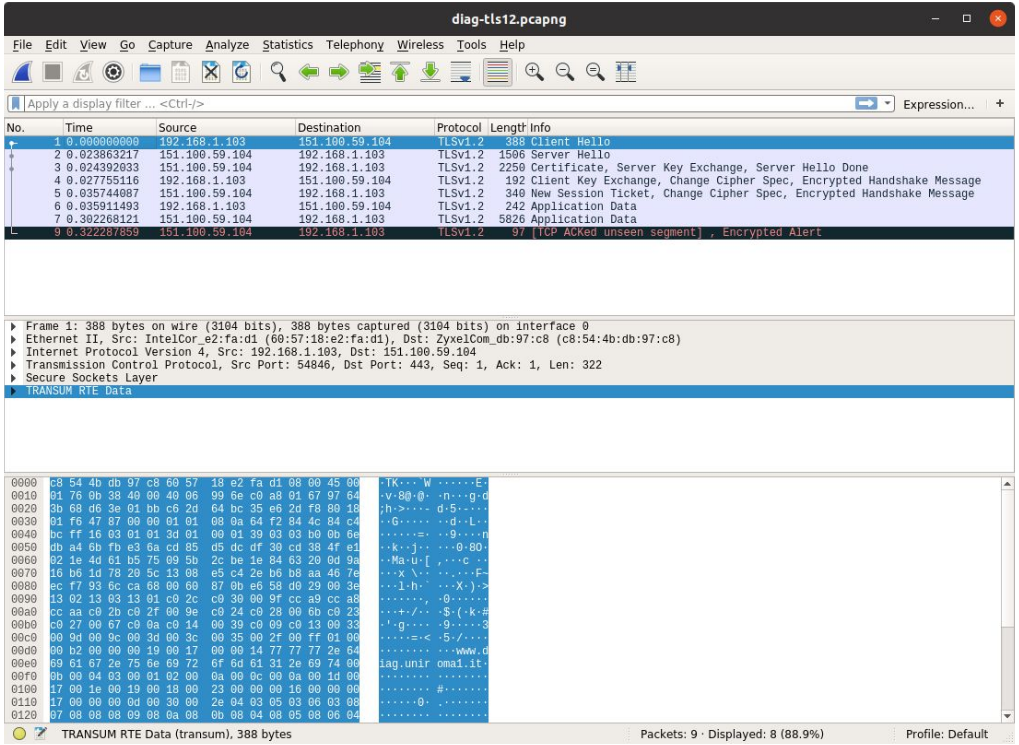
- To make it easier we can add the condition "and ip.addr == 151.100.59.104" in the filter



Demo: TLSv1.2 connection (6)

Let us analyze a TLSv1.2 connection with WireShark:

- Packets can be marked and exported into a pcap.
- We now consider packets “diag-tls12.pcap” (available on Piazza)
- WireShark > File > Open then select the .pcap file

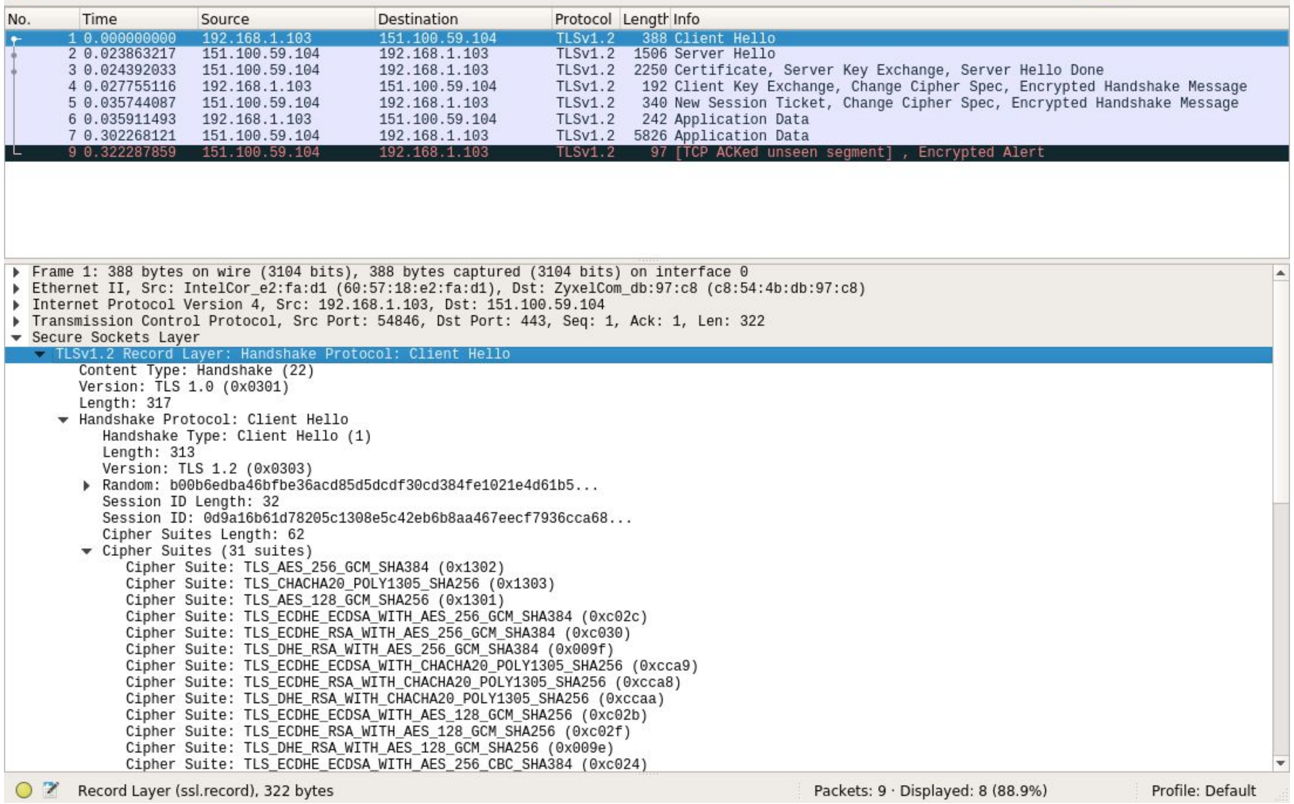


Demo: TLSv1.2 connection (7)

Client to server:

Handshake Protocol:

- Client Hello
    - Random
    - Cipher Suites
    - Compression Method is null
    - Session ID of a previous connection

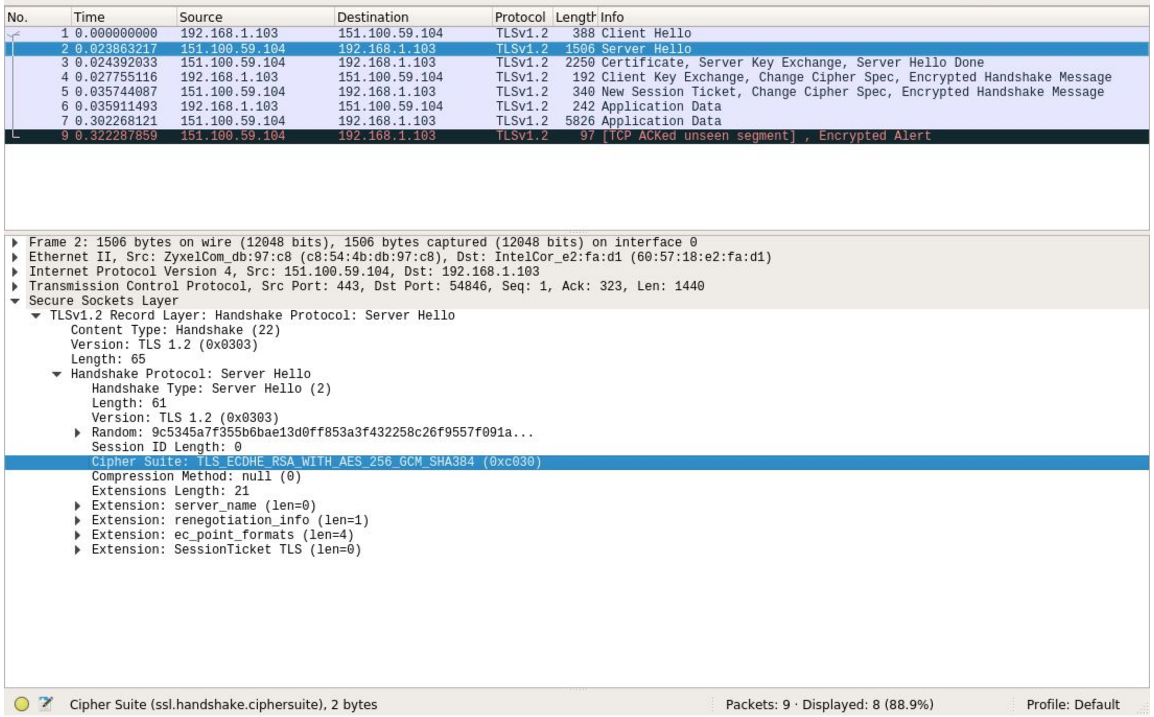


Demo: TLSv1.2 connection (8)

Server to client:

Handshake Protocol:

- Server Hello
    - Random
    - Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030)
    - Compression Method is null
    - Session ID sent by the client is rejected by server: hence a new connection must be established



Demo: TLSv1.2 connection (9)

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384:

IANA name:

TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

OpenSSL name:

ECDHE-RSA-AES256-GCM-SHA384

GnuTLS name:

TLS_ECDHE_RSA_AES_256_GCM_SHA384

Hex code:

0xC0, 0x30

TLS Version(s):

TLS1.2

Protocol:

Transport Layer Security (TLS)

Key Exchange:

Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)

Authentication:

Rivest Shamir Adleman algorithm (RSA)

Encryption:

Advanced Encryption Standard with 256bit key in Galois/Counter mode (AES 256 GCM)

Hash:

Secure Hash Algorithm 384 (SHA384)

Demo: TLSv1.2 connection (10)

Server to client:

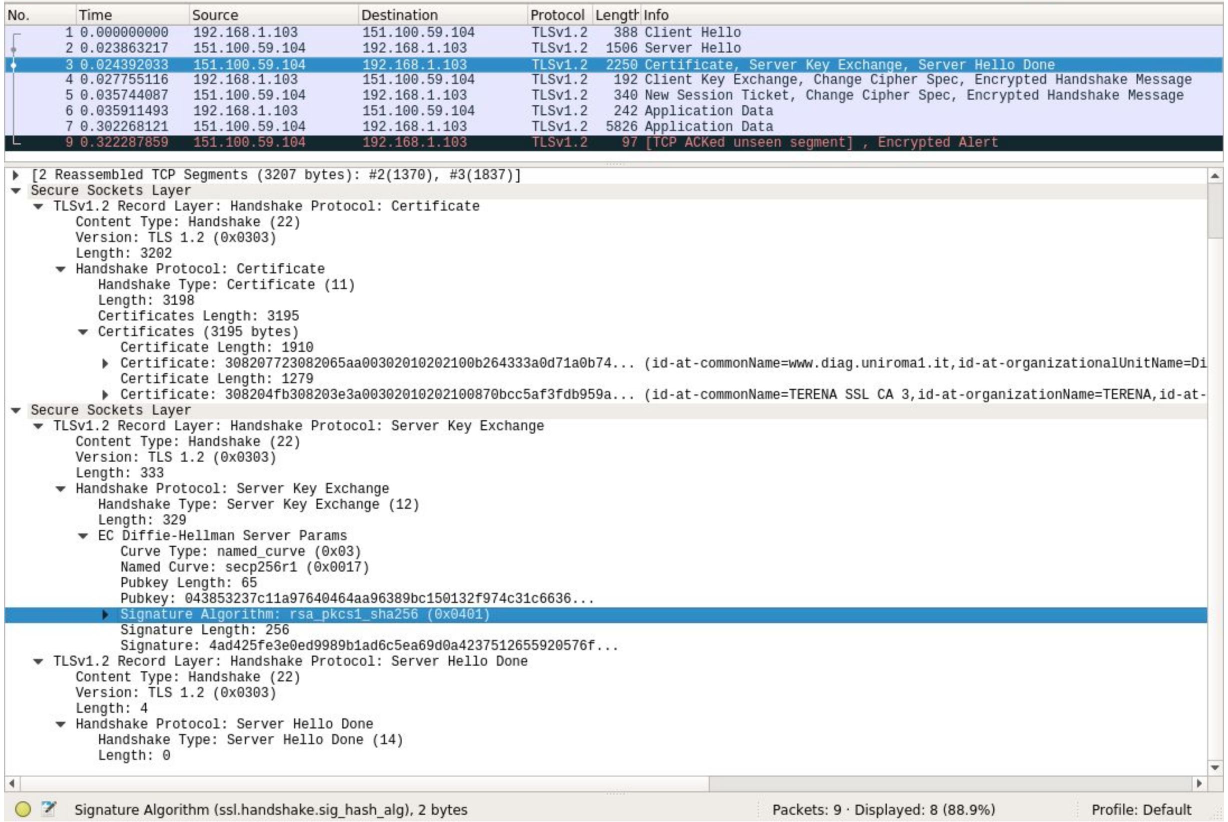
Handshake Protocol #1:

- Certificate
    - DIAG certificate
    - certificate of CA

Handshake Protocol #2:

- Server Key Exchange
    - ECDH params

- Server Hello Done



Demo: TLSv1.2 connection (11)

Client to server:

Handshake Protocol #1:

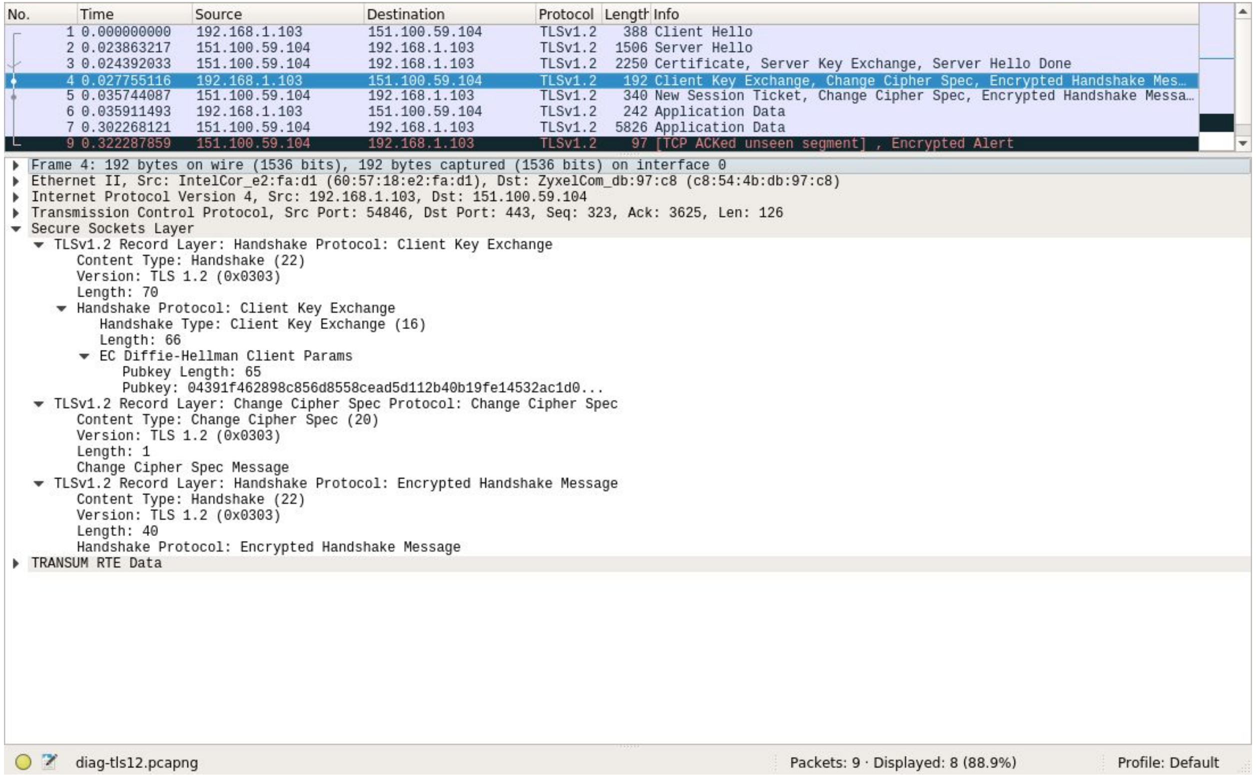
- Client Key Exchange
    - ECDH params

Change Cipher Spec Protocol:

- Change Cipher Spec

Handshake Protocol #2:

- Encrypted Handshake Message
  a "finished" encrypted message, containing a hash and MAC over the previous handshake messages



Demo: TLSv1.2 connection (11)

Server to client:

Handshake Protocol #1:

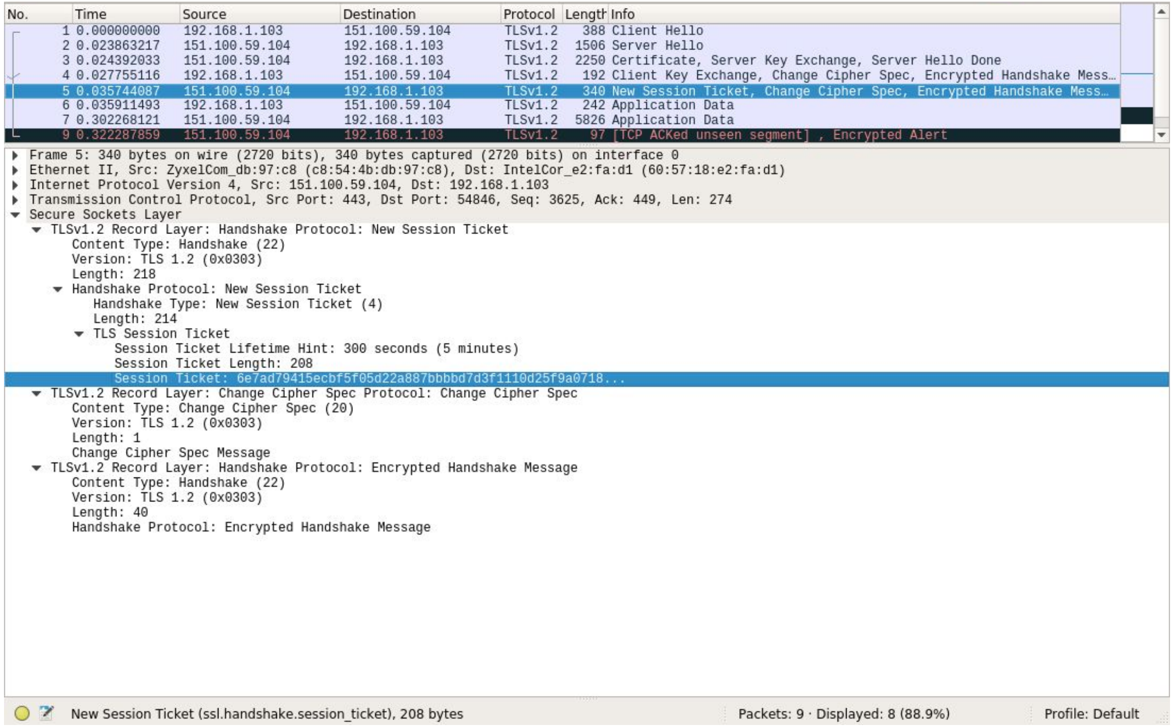
- New Session Ticket
    - [TLS Session Ticket](): TLS instead of using session IDs can work with tickets to resume a connection. This allows the server to not keep track of past connection.

Change Cipher Spec Protocol:

- Change Cipher Spec

Handshake Protocol #2:

- Encrypted Handshake Message



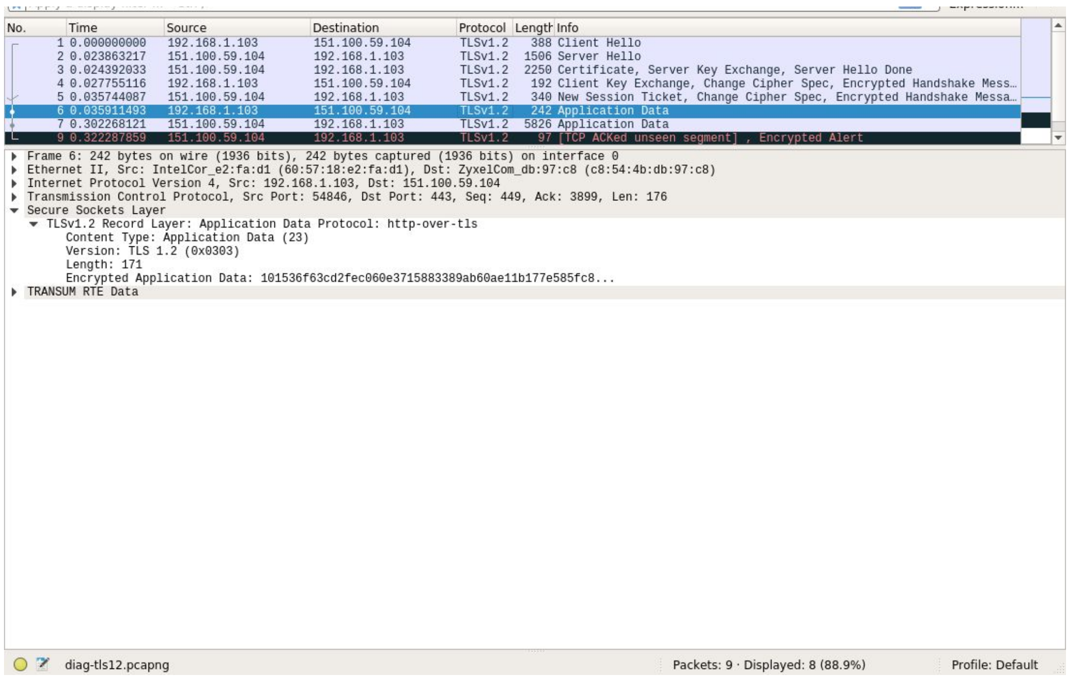
Demo: TLSv1.2 connection (12)

Client to server:

Application Data Protocol:
- http-over-tls
  - Encrypted App Data

Server to client:

Application Data Protocol:
- http-over-tls
  - Encrypted App Data

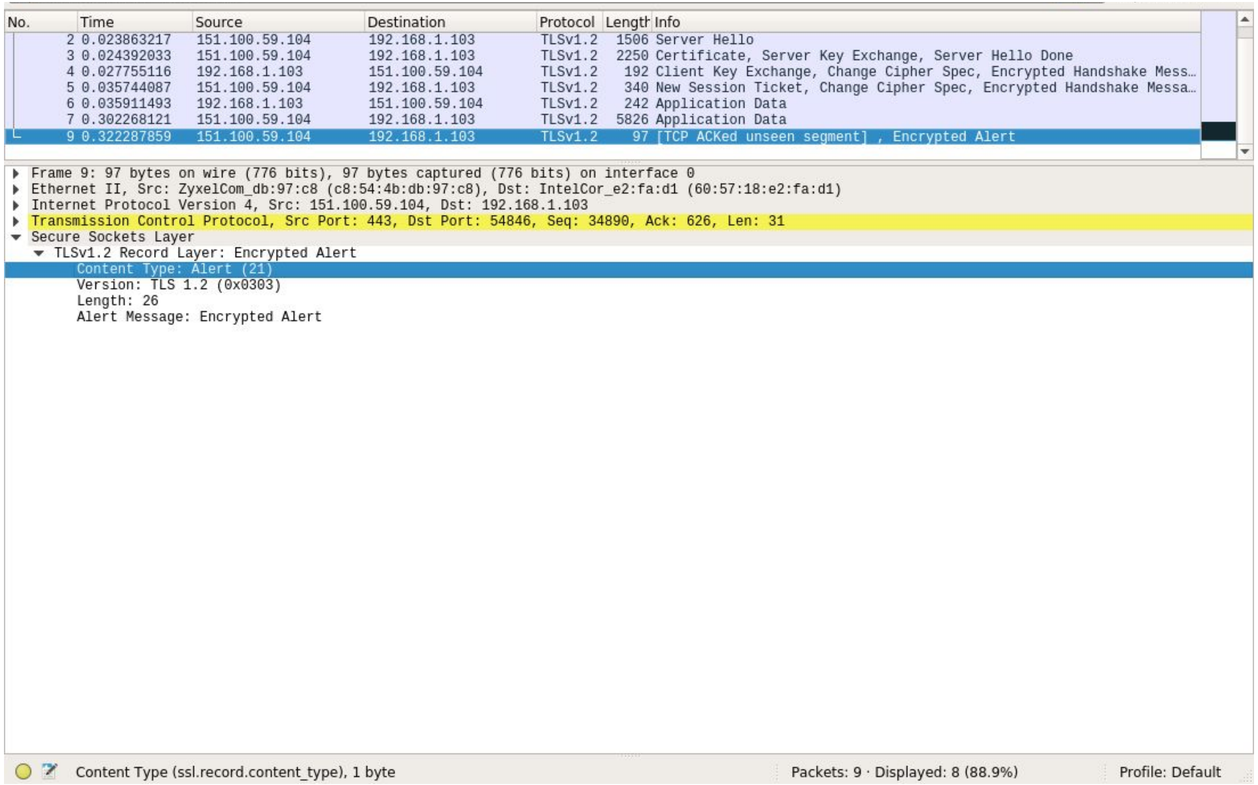


Demo: TLSv1.2 connection (13)

Server to client:

Encrypted Alter:

- Alert
  - Alert Message



Demo: TLSv1.2 connection (14)

Alerts:

Alert description types

Code	Description	Level types	Note
0	Close notify	warning/fatal	
10	Unexpected message	fatal	
20	Bad record MAC	fatal	Possibly a bad SSL implementation, or payload has been tampered with e.g. FTP firewall rule on FTPS server.
21	Decryption failed	fatal	TLS only, reserved
22	Record overflow	fatal	TLS only
30	Decompression failure	fatal	
40	Handshake failure	fatal	
41	No certificate	warning/fatal	SSL 3.0 only, reserved
42	Bad certificate	warning/fatal	
43	Unsupported certificate	warning/fatal	e.g. certificate has only server authentication usage enabled and is presented as a client certificate
44	Certificate revoked	warning/fatal	
45	Certificate expired	warning/fatal	Check server certificate expire also check no certificate in the chain presented has expired
46	Certificate unknown	warning/fatal	
47	Illegal parameter	fatal	
48	Unknown CA (Certificate authority)	fatal	TLS only
49	Access denied	fatal	TLS only – e.g. no client certificate has been presented (TLS: Blank certificate message or SSLv3: No Certificate alert), but server is configured to require one.

Not an actual error

from wikipedia

Credits

These slides are based on material from:

- Slides of Prof. D'Amore from CNS 2019-2020

- Christof Paar and Jan Pelzl. Understanding Cryptography: A Textbook for Students and Practitioners. Springer. http://www.crypto-textbook.com/

- Wikipedia (english version)