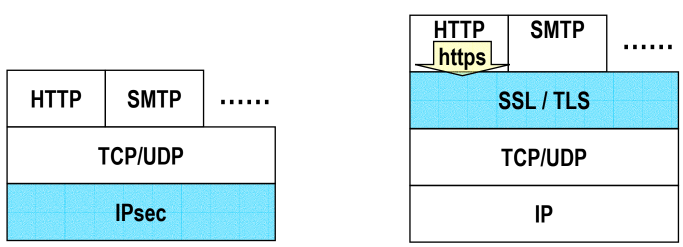
Lecture 4:

Transport Layer Security (secure Socket Layer)

Recommended reading: Thomas, SSS and TLS essentials (old but very well written)

SSL/TLS: layered view



Network layer security

Transport layer security

SSL/TLS: operates on top of TCP, but below application layer (can be considered as top sublayer for L4)

SSL/TLS: it is NOT a security enhancement of TCP!

History of SSL/TLS

SSL v1	SSL v2	SSL v3	TLS v1
by Netscape never released	Integrated in netscape 1.1 Badly broken!	Redesigned from scratch by Netscape	IETF SSL design (versus Netscape) SSL v3.1
1994	1995	1996	1996-1999

Public Domain implementation available @ www.openssl.org

- **TLS 1.0 specification in RFC 2246**
  - More recent RFC specification: TLS 1.1, RFC 4346, April 2006
  - Even more recent: TLS 1.2, Internet Draft, March 2007
- **Basically SSL with minor modification**
  - Also referred to as SSL v3.1
  - However NOT backward compatible with SSL 3.0
- **Not necessarily limited to Internet transport!**
  - Devised for point-to-point relationships in general
  - E.g. EAP-TLS (RFC 2716)
    - TLS mechanisms employed for authentication and integrity protection over L2 EAP

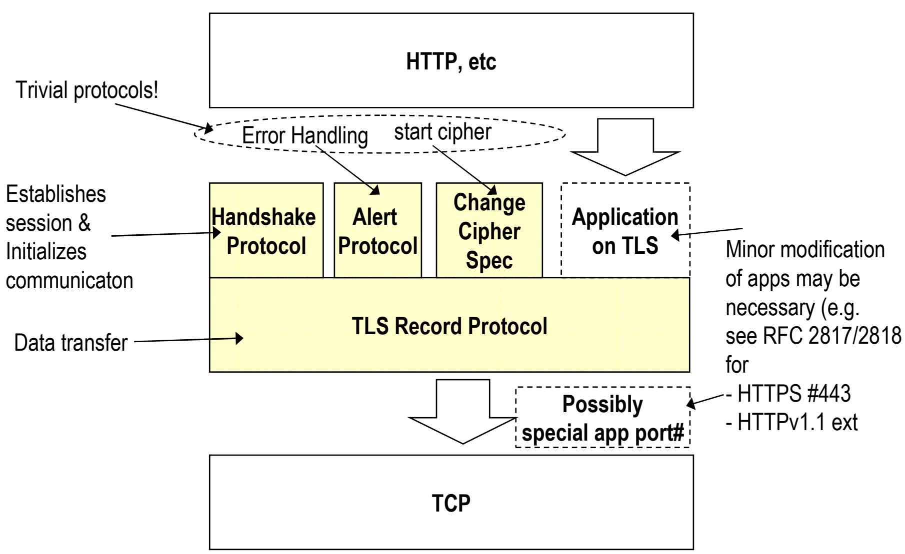
Goals

- **Establish a session**
  - Agree on algorithms
  - Share secrets
  - Perform authentication
- **Transfer application data**
  - Communication privacy
    - Symmetric encryption
  - Data integrity
    - Keyed Message Authentication Code (HMAC)

Session vs Connection

- ➔**Connection:**
  - ⇨Transport service
  - ⇨TLS provides encryption and integrity
  - ⇨*TLS Record Protocol*
- ➔**Session:**
  - ⇨Association between Client and Server
    - →Authentication and exchange of security parameters
  - ⇨May include several connections
    - →Heavy work: done once at the beginning
    - →Designed for HTTPv1.0;
  - ⇨*TLS Handshake Protocol*

TLS protocol stack

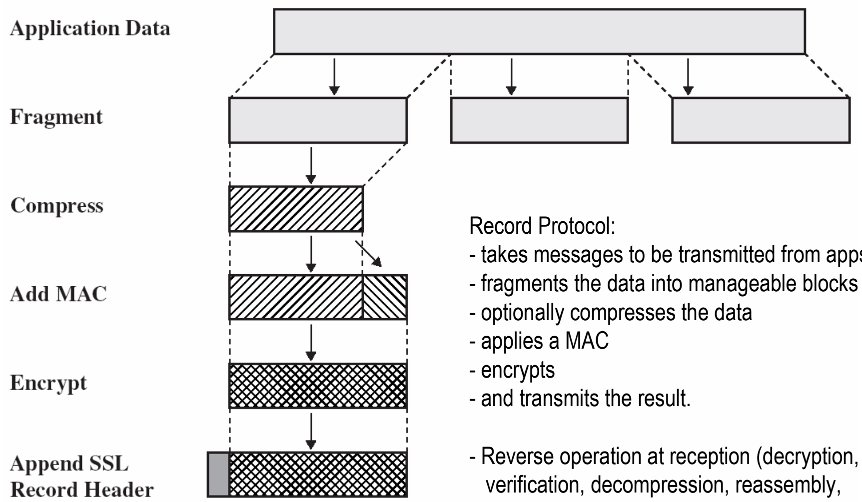


Support of applications

- **Typical approach: reserve a special port number for SSL/TLS mediated application**
  - Example:
    - →port 80 = HTTP over TCP
    - →Port 443 = HTTP over SSL/TLS (HTTPS)
- **SSL/TLS common application port numbers**
  - ssmtp 465
  - spop3 995
  - imaps 991
  - telnets 992
  - …
- **Alternative approach: slightly modify application to reuse traditional port number**
  - E.g. HTTPv1.1:
  - upgrade: TLS/1.0 new command (see RFC 2817)

TLS Record Protocol

Record Protocol operation



Record Protocol:
- takes messages to be transmitted from apps
- fragments the data into manageable blocks
- optionally compresses the data
- applies a MAC
- encrypts
- and transmits the result.

- Reverse operation at reception (decryption, verification, decompression, reassembly, delivery to apps)

Secret key dependent operation

- ➔**Message Authentication Code computation**
  - ⇨Uses HMAC
    - →see Radius Message Authenticator lesson
  - ⇨Secret (symmetric) negotiated during handshake
- ➔**Fragment Encryption**
  - ⇨Symmetric encryption
  - ⇨Algorithm (RC4, DES, 3DES, etc) negotiated during handshake, too
  - ⇨Secret key derived from security parameters negotiated during handshake
    - →Differs from key used in MAC

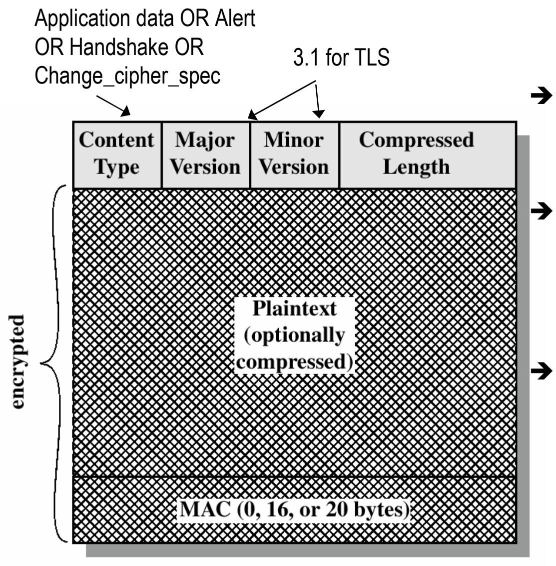
Fragmentation

- ➔**At application ←→TLS interface**
  - ⇨DON'T get confused with TCP segmentation!!
- ➔**Input: block message of arbitrary size**
  - ⇨possibly multiple aggregated messages of SAME protocol
- ➔**Fragment size: 2^{14}=16384 bytes**

Compression

- ➔**Based on compression state negotiated…**
  - ⇨Lossless compression, if size increases (may happen in special cases) increase must NOT exceed 1024 bytes
- ➔**… but no compression by default**
  - ⇨Until recently, no compression was employed in TLS & SSLv3!
    - →Feature formerly used only in SSLv2
  - ⇨But some specifications recently emerged
    - » RFC 3749 – support for DEFLATE (RFC 1951)
    - » RFC 3943 – support for Lempel-Ziv-Stac
    - →Reason: widespread diffusion of "verbose" languages – e.g. XML

SSL Record Protocol Data Unit



- ➔ **5 bytes header**
  - ⇨ Content type = 1
  - ⇨ Version = 1+1
  - ⇨ Length = 2
- ➔ **Content Type**
  - ⇨ higher layer protocol
    - → 20 = 0x14 = Change Cipher Spec
    - → 21 = 0x15 = Alert
    - → 22 = 0x16 = Handshake
    - → 23 = 0x17 = Application Data
- ➔ **Sequence number**
  - ⇨ Not transmitted, but kept at both connection extremes
    - → Remember: reliable transport, hence no "holes"

MAC generation

- ➔ **Computed through HMAC**
  - ⇨ HMAC(seq.num | ctype | version | len | data)
    - → Minor differences with SSLv3
      - » No version in SSLv3
      - » Slightly different HMAC construction
        - – (RFC final specification not yet finalized at the time ☺)
  - ⇨ Hash used in HMAC:
    - → MD5 → 16 bytes hash
    - → SHA-1 → 20 bytes hash
- ➔ **Negotiation may decide not to use MAC**
  - ⇨ In practice, always present
- ➔ **Sequence number:**
  - ⇨ Not transmitted but included in the MAC
    - → to detect missing/extra data
    - → and to prevent replay/reordering attacks

Encryption

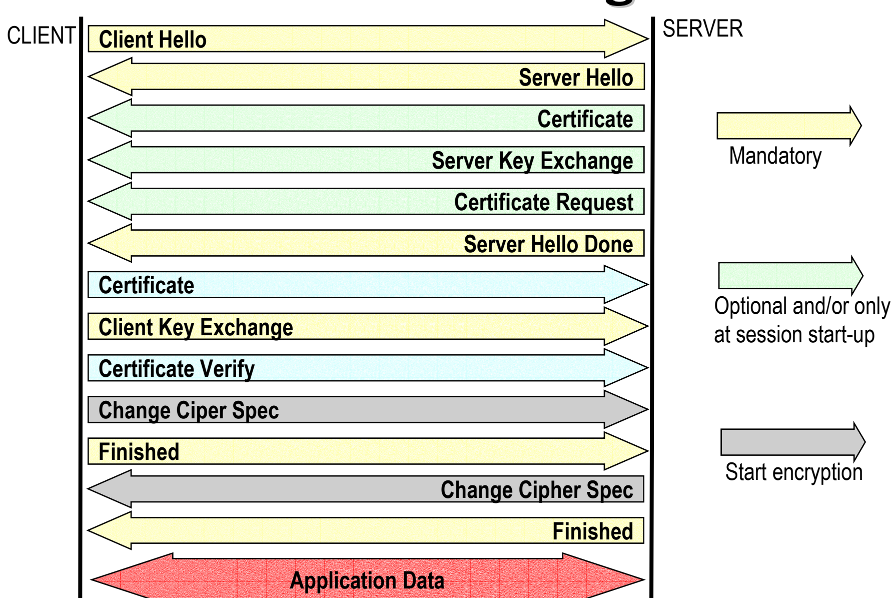
- ➔ **Applies to both (compressed) fragment and MAC**
- ➔ **Symmetric encryption algorithm**
  - ⇨ Block or stream cipher
  - ⇨ Key generated from security secrets exchanged during handshake
- ➔ **Algorithm negotiated during handshake**
  - ⇨ Clear text possible
    - → If no encryption negotiated
    - → Or in early handshake phases
- ➔ **Encryption algorithm CANNOT increase size of more than 1024 bytes**
- ➔ **If block cipher, padding necessary to achieve block size**

TLS Handshake Protocol

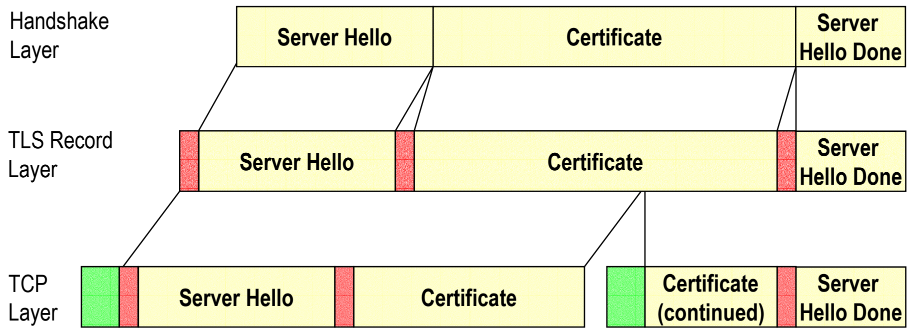
Handshake goals

- **Negotiation and exchange of parameters**
  - to agree on algorithms, exchange random values, check for session resumption.
- **Secure negotiation of a shared secret**
  - Never transmitted in clear text; derived from crypto parameters exchanged
  - Robust to MITM attacks if connection authenticated
- **Optional authentication**
  - for both client and/or server
    - in practice always required for server
  - Through asymmetric, or public key, cryptography
    - Exchange certificates and/ore crypto information
- **Reliable Negotiation:**
  - Allow C&S to verify that no tampering by an attacker occurred
  - Attacker cannot tamper communication without being detected by the involved C&S parties

Handshake Messages



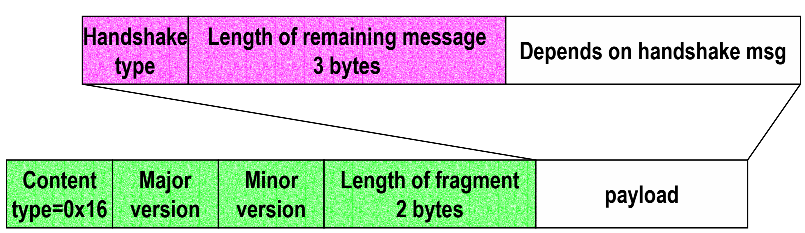
TCP segmentation (example)



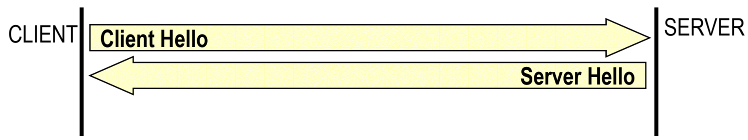
- **Simplified example: server responds to client hello with certificate only**
- **Arbitrary correspondence between TLS Records and TCP segments (TLS records fragmented into TCP segments)**
  - Obvious! Do you remember TCP? ☺

Handshake message format

- **Incapsulated in TLS Record**
- **Handshake types:**
  - hello_request(0), client_hello(1), server_hello(2),
  - certificate(11), server_key_exchange (12),
  - certificate_request(13), server_hello_done(14),
  - certificate_verify(15), client_key_exchange(16),
  - finished(20)



Handshake phase 1



- ➔ **General goal:**
  - ⇨ Create TLS/SSL connection between client and server
- ➔ **Detailed goals**
  - ⇨ Agree on TLS/SSL version
  - ⇨ Define session ID
  - ⇨ Exchange nonces
    - → timestamp + random values used to, e.g., generate keys
  - ⇨ Agree on cipher algorithms
  - ⇨ Agree on compression algorithm

Client Hello

- ➔ **First message, sent in plain text**
- ➔ **Incapsulated in 5 bytes TLS Record**
- ➔ **Content:**
  - ⇨ Handshake Type (1 byte), Length (3 bytes), Version (2 bytes)
    - → Type 01 for Client Hello
    - → Version: 0300 for SSLv3, 0301 for TLS
  - ⇨ 32 bytes Random (4 bytes Timestamp + 28 bytes random)
    - → First 4 bytes in standard UNIX 32-bit format
      - » Seconds since the midnight starting Jan 1, 1970, GMT
  - ⇨ (Session ID length, session ID)
    - → 1+32 bytes – either empty or previous session ID
      - » Previous session ID = resumed session state
  - ⇨ (Cipher Suites length, Cipher suites)
    - → 2+Nx2 bytes, in order of client preference
  - ⇨ (Compression length, compression algorithm)
    - → 1+1 byte (in all cases, today: compression algo = 00 = null)

Cipher suites

PUBLIC-KEY ALGORITHM | SYMMETRIC ALGORITHM | HASH ALGORITHM

TLS_AAAAAAA_WITH_BBBBBBBB_CCCCCCCC

TLS_NULL_WITH_NULL_NULL ← INITIAL (NULL) CIPHER SUITE

TLS_RSA_WITH_NULL_MD5
TLS_RSA_WITH_NULL_SHA

TLS_RSA_WITH_RC4_128_MD5
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
TLS_RSA_WITH_3DES_EDE_CBC_SHA

TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA
TLS_DH_RSA_WITH_DES_CBC_SHA

TLS_DHE_DSS_WITH_DES_CBC_SHA
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA

TLS_DH_anon_WITH_RC4_128_MD5

JUST A FEW EXAMPLES (FULL LIST IN RFC)

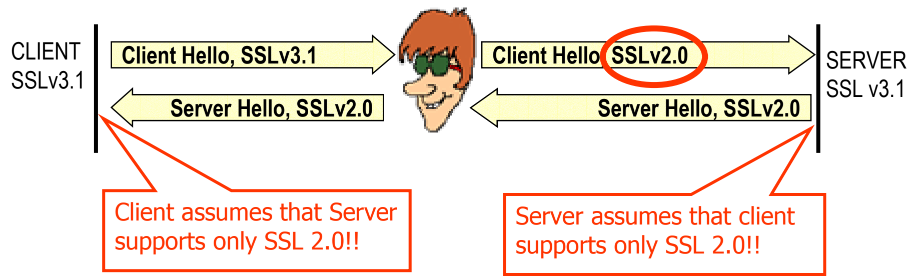
Note: arbitrary combination not possible: must choose from a (small) list of combinations

Server Hello

- **In reply to Client Hello, sent in plain text**
- **Content:**
  - Handshake Type (1 byte), Length (3 bytes)
    - Type = 02 for Server Hello)
  - Version
    - Highest supported by both Client and Server
  - 32 bytes Random (4 bytes Timestamp + 28 bytes random)
    - Different random value than Client Hello
  - (Session ID length, session ID)
    - If Client session ID=0, then generate session ID
    - Otherwise, if resumed session ID OK also for Server, use it;
    - Otherwise generate new one
  - Cipher Suite, 2 bytes
    - Selected from client list (usually best one, but no obligation)
  - Compression algo, 1 byte
    - Selected from Client list

Downgrade attack

MITM – doesn't break SSL3.x, but knows very well
how to break SSL2.0 (40 bit encryption only)

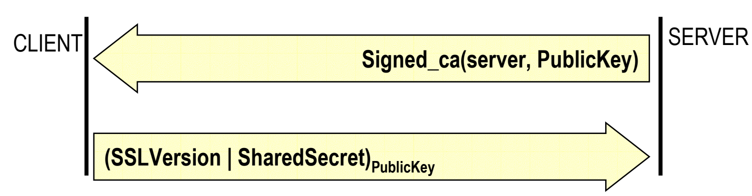


➔ **Downgrade attacks on cipher suites, too**
- ⇨ MITM: remove "strong" cipher suites, and leave in Client Hello only the ones he knows he can break!

Since hello message authentication not viable at the moment (not yet a shared secret available), verification must be necessarily delayed to a subsequent phase…

Giuseppe Bianchi

Handshake phase 2 & 3 (schematic, RSA case)

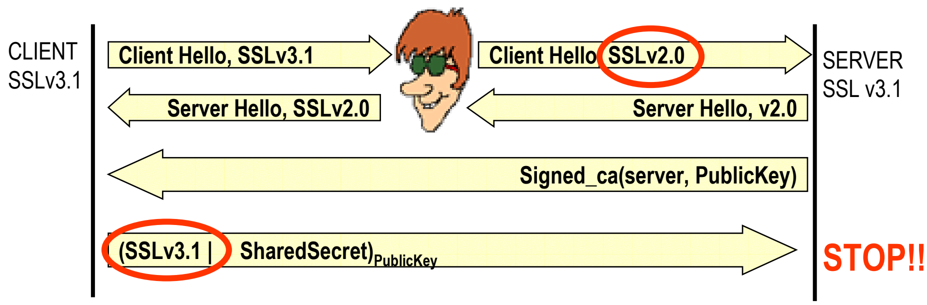


➔ **Phase 2:**
- ⇨ Server sends authentication information (certificate)
- ⇨ Along with Public Key (in certificate or in extra msg)

➔ **Phase 3:**
- ⇨ Client generates a Shared Secret
  - →Length depends on agreed cipher suite (RSA = 46 bytes)
- ⇨ And transmits it to Server, encrypted with Public Key
  - →Native SSL version included to early combat downgrade attacks

Giuseppe Bianchi

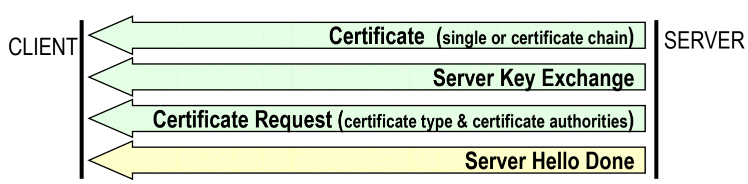
Against downgrade attack



- ➔ **Since (signed) certificate cannot be tampered…**
- ➔ **… and since MITM cannot decrypt SharedSecret**
  - ⇨ and must act as pass-through
- ➔ **Downgrade attack on SSL version easily discovered!**
  - ⇨ Encrypted SSL version is the one initially proposed, NOT the one negotiated in server hello, of course

Giuseppe Bianchi

Handshake Phase 2 details



- ➔ **Certificate**
  - ⇨ typically a certificate chain
- ➔ **Server Key exchange**
  - ⇨ When certificate not issued (no server authentication required – UNSAFE!!)
  - ⇨ When certificate key is for signature only (not for encryption)
  - ⇨ When certificate key cannot be used for legal reasons
    - → E.g. RSA key larger than 512 bits may not be used <u>for encryption</u> outside US, but only for signature
    - → Larger RSA key encoded in certificates may be used to sign shorter, temporary, RSA key
  - ⇨ When Ephemeral Diffie-Hellman used
- ➔ **Certificate request**
  - ⇨ Requires client authentication (asks for an acceptable certificate)
- ➔ **Server Hello Done**
  - ⇨ empty message

Giuseppe Bianchi

Example: Ephemeral Diffie-Hellman /1

- **Review of DH**
  - Server transmits public value g^X mod p
  - Client replies with public value g^Y mod p
  - Both compute shared key as
    - $K = (g^Y)^X \bmod p = (g^X)^Y \bmod p$
- **Fixed versus Ephemeral**
  - Depends on whether X and Y values are pre-assigned or dynamically generated
- **Support in TLS:**
  - Certificate contains key used for signature
  - KeyExchange message contains signed DH parameters
    - If not signed (Anonymous DH), MITM would be trivial!

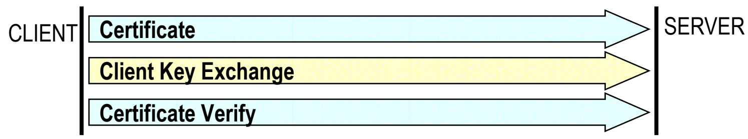
Example: Ephemeral Diffie-Hellman /2

ServerKeyExchange message format:

Content type=0x14	Major version=3	Minor version=1	Length of fragment 2 bytes = nn	TLS Record Header (5 bytes) 0x14 = Handshake protocol
Handshake type=0x0c	Length 3 bytes = nn-4			Handshake Header (4 bytes); 0x0c=ServerKeyEx
DH p length 2 bytes	DH p value ***			DH parameters (variable length)
DH g length 2 bytes	DH g value ***			
DH Ys length 2 bytes	DH Ys value ***			
Signature RSA → 16 MD5 + 20 SHA-1 DSA → 20 SHA-1 only				Signature size and hash used depends on signature type (RSA or DSA). Note: RSA uses BOTH MD5 and SHA-1

Client replies with DH public value Ys only (p and g are already known by the Server ☺)

Handshake Phase 3 details

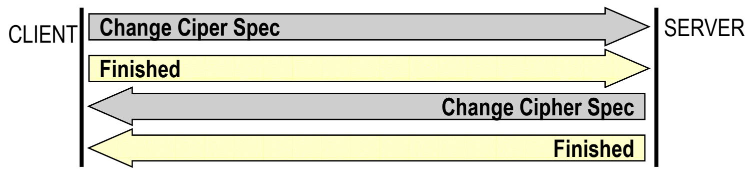


- ➔ **Certificate:**
  - ⇨ Client certificate if requested
- ➔ **Client Key exchange**
  - ⇨ Transmit encrypted symmetric (premaster) key or information to generate secret key at server side (e.g. Diffie-Hellman Ys)
- ➔ **Certificate Verify**
  - ⇨ Signature of "something" known at both client and server
    - →ALL the messages exchanged up to now
      - » Which is not only known, but also useful! Allows to detect at an early stage (more later on this) tampering attacks (e.g. cipher suite downgrade)
  - ⇨ To prove Client KNOWS the private key behind the certificate
    - →Otherwise I could authenticate by simply copying a certificate ☺

A (smart) detail on certificate verify

- ➔**Q: Why Certificate Verify does not immediately follows certificate?**
- ➔**A: to include connection specific crypto parameters into signature!**
  - →master secret, client & server random values
    - » Since master secret can be computed only after the ClientKeyExchange message, Certificate Verify follows ClientKeyExchange
  - ⇨Otherwise attacker may sniff both certificate and certificate verify and replay it later on to impersonify the user!

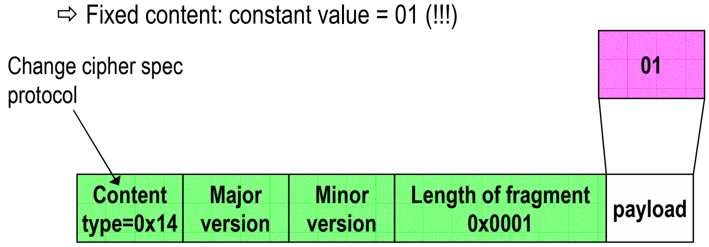
Phase 4



- ➔ **Phase 4 has two fundamental goals**
  - ⇨ Switch to the new security connection state
    - →Hence authentication and encryption based on keys computed with the exchanged security parameters
    - →Immediately applied to finished message
      - » First check that everything went OK! If Client and(orserver cannot decrypt finished message, something has gone wrong!
  - ⇨ Authenticate all the previous handshake messages
    - →Finished = digital signature of all the previous handshake messages <u>as transmitted and received by the peer up to now</u>
    - →To avoid MITM tampering

Change Cipher Spec

- ➔ **Defined as a separate protocol (!)**
- ➔ **Only one message:**
  - ⇨ 1 single byte
  - ⇨ Fixed content: constant value = 01 (!!!)



TLS Record Header

Why a separate protocol, and not part of the Handshake?
Wise choice! TLS specification allows aggregation of multiple messages of a SAME upper layer protocol into single TLS Record. How this possible with a Change Cipher Spec? (TLS Record is either ALL encrypted, or NOT encrypted)

TLS Key Computation

Secret hierarchy

- ➔ **Pre Master Secret**
  - ⇨ Exchanged during handshake (RSA)
  - ⇨ Derived from D-H parameters
- ➔ **Master Secret**
  - ⇨ Per-connection (explicitly includes timestamp & random)
  - ⇨ Derived from:
    - → Pre Master Secret
    - → Timestamp+Random (server)
    - → Timestamp+Random (client)
- ➔ **Connection state: up to 6 keys**
  - ⇨ Encryption keys
  - ⇨ Initialization Vectors
  - ⇨ MAC secrets
  - ⇨ x 2 (1 set per client, 1 set per server)

PRF

- ➔ **Pseudo Random Function**
- ➔ **A fundamental (new) function in TLS**
  - ⇨ SSLv3 and below used different, less robust, approaches
- ➔ **Allows to generate an arbitrary amount of crypto material from limited initial material**
  - ⇨ Essential, as the amount of key material depends on cipher suites and is not known or bounded a priori!
- ➔ **Used also to generate master key**

Giuseppe Bianchi

Expansion function P_{hash}

- ➔ A_0 = seed
- ➔ $A_1 = HMAC_{hash}(A_0)$
- ➔ $A_2 = HMAC_{hash}(A_1)$
- ➔ $A_3 = HMAC_{hash}(A_2)$
- ➔ …

Same secred used in all HMACs
Hash = chosen hash function

P_{hash} =	$HMAC_{hash}(A_1 \mid seed)$	$HMAC_{hash}(A_2 \mid seed)$	$HMAC_{hash}(A_3 \mid seed)$	……

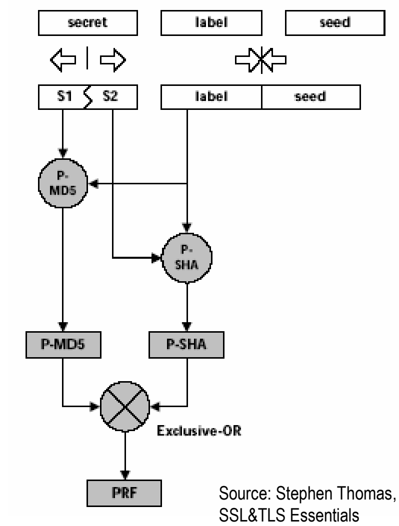
- ➔ **P_{hash} function of:**
  - ⇨ Chosen hash function
  - ⇨ Secret
  - ⇨ seed
- ➔ **P_{hash} size: any (unlike HMAC size)**

Giuseppe Bianchi

PRF (until TLS 1.1)

(TLS1.2 moved to SHA-256)

- **Employs two hash algorithms**
  - MD5 and SHA-1
  - Greater robustness! Must break both…
- **Input:**
  - Desired length
  - Secret
    - Assume even size, split it in two parts (S_1,S_2)
  - Seed
  - Label (an ascii string)
- **PRF(Secret, Label, Seed) = P_{MD5}(Label | Seed) XOR P_{SHA-1}(Label | Seed)**
  - MD5 uses S1 as secret
  - SHA-1 uses S2 as secret



Source: Stephen Thomas, SSL&TLS Essentials

Key generation

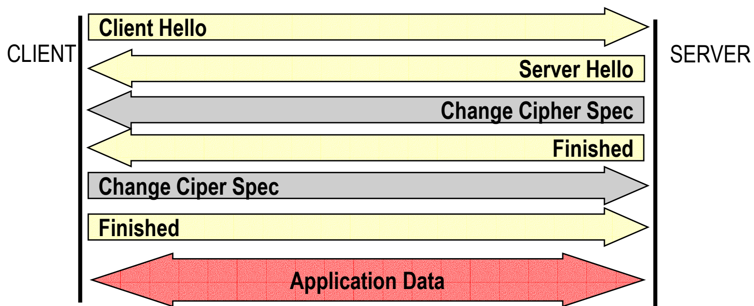
- **Master secret [48 bytes]:**
  - PRF(pre-master-secret, “master secret”, Client-Random | Server-Random)
- **Key Block [size depends on cipher suites]:**
  - PRF(master-secret, “key expansion”, Server-Random | Client-Random)
- **Individual keys:**
  - Partition key block into up to 6 fields in the following order:
  - Client MAC, Server MAC, Client Key, Server Key, Client IV, Server IV

PRF used also in computation of finished message instead of “normal” MD5 or SHA-1 Hash. E.g. For client finished message:

PRF(master-secret, “client finished”, MD5(all handshake msg) | SHA-1(all handshake msg)) [12 bytes]

Abbreviated handshake (3-way) (session resumption)

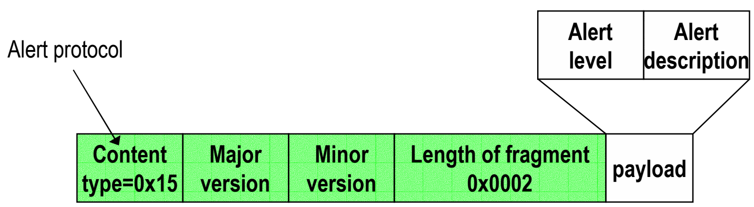
- ➔ **Used to re-generate key material**
  - ⇨ For new connection
- ➔ **Avoids to reauthenticate peers**
  - ⇨ Done at start of session only
- ➔ **Avoid to re-exchange pre-master-secret**
- ➔ **Exchange new TS+random values**



TLS connection management & application support

Alert Protocol

- **TLS defines special messages to convey "alert" information between the involved fields**
- **Alert Protocol messages encapsulated into TLS Records**
  - ⇨ And accordingly encrypted/authenticated
- **Alert Protocol format (2 bytes):**
  - ⇨ First byte: Alert Level
    - →warning(1), fatal(02)
  - ⇨ Second Byte: Alert Description
    - →23 possible alerts



TLS Record Header

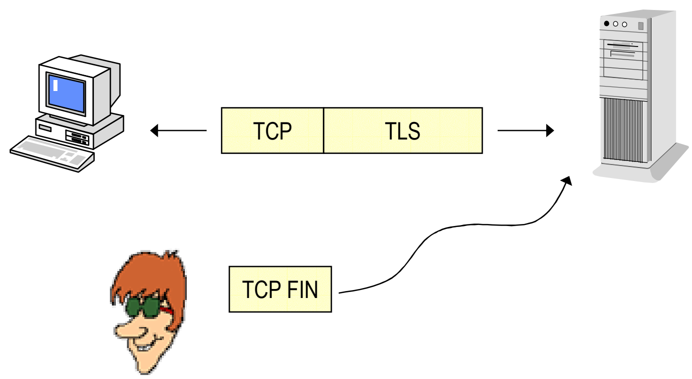
Sample alerts

See RFC2246 for all alerts and detailed description

- **unexpected_message**
  - ⇨ Inappropriate message received
    - → Fatal - should never be observed in communication between proper implementations
- **bad_record_mac**
  - ⇨ Record is received with an incorrect MAC
    - → Fatal
- **record_overflow**
  - ⇨ Record length exceeds 2^{14}+2048 bytes
    - → Fatal
- **handshake_failure**
  - ⇨ Unable to negotiate an acceptable set of security parameters given the options available
    - → Fatal
- **bad_certificate, unsupported_certificate, certificate_revoked, certificate_expired, certificate_unknown**
  - ⇨ Various problems with a certificate (corrupted, signatures did not verify, unsupported, revoked, expired, other unspecified issues which render it unacceptable
    - → Warning or Fatal, depends on the implementation

If fatal, terminate and do not allow resumption with same security parameters (clear all!)

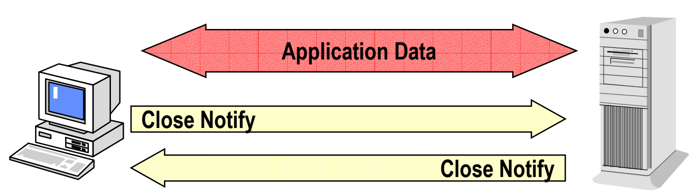
Truncation attack



An attacker may end connection at any time by sending a TCP FIN
Part of the intended information exchange is lost

How can Server and Client distinguish this from a transaction that "normally" completes?

Solution: Close Notify



- ➔ **Issued by any party**
  - ⇨ Close Notify = Alert (warning level)
- ➔ **Informs that no more data will be transmitted**
- ➔ **A connection that ends abruptly without a Close Notify may be a truncation attack**

A remark: note the weakness of TLS against TCP DOS attacks in general!

Conclusive remarks

- **Performance drawbacks:**
  - Increased overhead and latency!
    - Mostly for encryption overhead and handshake overhead
  - Computational overhead may kill server performance
    - Up to two orders of magnitude
      - ref: Transport Layer Security: how much does it really costs? Infocom 99
- **TLS does NOT protect TCP**
  - How to protect non SSL/TLS-aware applications?
  - Tools available to protect a generic TCP connection
    - stunnel = TCP over TLS over TCP
      - www.stunnel.org
      - Crazy (tcp tunneled over tcp), but as last resort…
  - DOS attacks to TCP remains a significant issue (no protection at all)

DTLS

- **Recently specified**
  - RFC 4347 – Datagram TLS – April 2006
    - TLS over UDP
  - DTLS design goal:
    - Be as most as possible similar to TLS!
  - DTLS vs TLS
    - TLS assumes orderly delivery
      - DTLS: Sequence number explicitly added in record header
    - TLS assumes reliable delivery
      - Timeouts added to manage datagram loss
    - TLS may generate large fragments up to 16384B
      - DTLS includes fragmentation capabilities to fit into single UDP datagram, and recommends Path MTU discovery
    - TLS assumes connection oriented protocol
      - DTLS connection = (TLS handshake – TLS closure Alert)