CT437
COMPUTER SECURITY AND FORENSIC COMPUTING

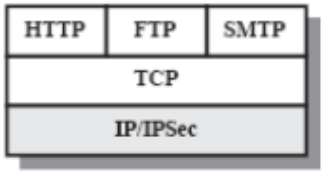
TRANSPORT LAYER SECURITY

Dr. Michael Schukat

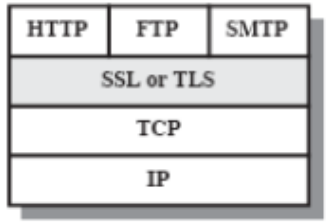
Ollscoil na Gaillimhe
University of Galway

Background

- The exponential growth of the internet in the 1990s resulted in a need for better security, thereby considering support of
  - ad-hoc and short-lived client/server connections
  - casual and untrained users
    - No awareness of risks and key concepts (confidentiality, integrity, authentication)
  - web browsers as the main vehicle for client / server communication
- The first attempt was Secure Socket Layer (SSL)
  - Introduced by Netscape in the 1990s
  - Embedded in web browsers / servers
  - Later became Internet standard known as TLS (Transport Layer Security)



(a) Network Level



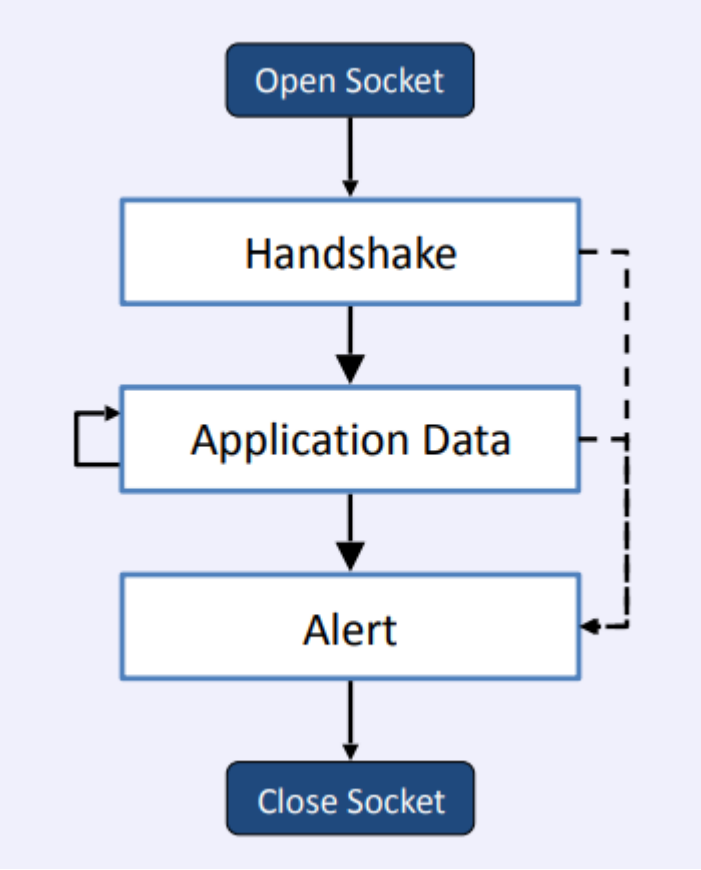
(b) Transport Level

TLS (Transport Layer Security)

- This application layer protocol is widely used for applications such as email, instant messaging and VoIP
  - Mainly known for securing HTTP (i.e. HTTPS)
- TLS provides
  - privacy (confidentiality) of exchanged data
  - integrity of exchanged data
  - authentication of server (and optionally client) through the use of digital certificates
- Composed of two layers:
  - TLS handshake protocol (main focus)
  - TLS record protocol
- It operates on top of TCP, which in turn is gradually replaced by the QUIC (also called TCP/2) protocol

Sequence of a TLS Session

- Handshake Protocol
  - Agree a cipher suite
  - Agree a master secret
  - Authentication using certificate(s)
- Record Protocol
  - Secure data communication
    - Symmetric key encryption
    - Data authentication
    - Often in combination with HTTP
  - Alerts
    - Graceful closure, or
    - Problem detected



Website Protocol Support (Wikipedia)

Protocol version	Website support[72]	Security[72][73]
SSL 2.0	0.4%	Insecure
SSL 3.0	3.0%	Insecure[74]
TLS 1.0	43.8%	Deprecated[9][10][11]
TLS 1.1	47.8%	Deprecated[9][10][11]
TLS 1.2	99.6%	Depends on cipher[n 1] and client mitigations[n 2]
TLS 1.3	49.7%	Secure

- SSL 2.0 / 3.0 contain a number of security flaws
- Support for TLS versions 1.0 and 1.1 was widely deprecated by web sites around 2020
- TLS 1.3 was released as RFC 8446 in August 2018. It is a streamlined version of the earlier TLS 1.2 specification with someone notable changes:
  - Streamlined handshake
  - Focus on elliptic curve cryptography using a reduced list of curves, RSA is not supported any more
  - Removing support for the MD5 and SHA-224 cryptographic hash functions
  - No more backwards compatibility beyond TLS 1.2
- As we'll see later, TLS 1.3 presents itself as 1.2 (well almost), this is apparently for compatibility reasons
- **Today, only TLS 1.2 and TLS 1.3 are in use, that's the focus of this lecture!**

Issues with Legacy TLS Versions: The Heartbleed Vulnerability in TLS 1.0 (2014)

The Heartbleed Bug

The Heartbleed Bug is a serious vulnerability in the popular OpenSSL cryptographic software library. This weakness allows stealing the information protected, under normal conditions, by the SSL/TLS encryption used to secure the Internet. SSL/TLS provides communication security and privacy over the Internet for applications such as web, email, instant messaging (IM) and some virtual private networks (VPNs).

The Heartbleed bug allows anyone on the Internet to read the memory of the systems protected by the vulnerable versions of the OpenSSL software. This compromises the secret keys used to identify the service providers and to encrypt the traffic, the names and passwords of the users and the actual content. This allows attackers to eavesdrop on communications, steal data directly from the services and users and to impersonate services and users.

What leaks in practice?

We have tested some of our own services from attacker's perspective. We attacked ourselves from outside, without leaving a trace. Without using any privileged information or credentials we were able steal from ourselves the secret keys used for our X.509 certificates, user names and passwords, instant messages, emails and business critical documents and communication.

How to stop the leak?

As long as the vulnerable version of OpenSSL is in use it can be abused. Fixed OpenSSL has been released and now it has to be deployed. Operating system vendors and distribution, appliance vendors, independent software vendors have to adopt the fix and notify their users. Service providers and users have to install the fix as it becomes available for the operating systems, networked appliances and software they use.

Issues with Legacy TLS Versions: Apple 'goto fail;' Vulnerability in TLS 1.0 and TLS 1.1 (2014)

- Affected iOS and Mac OS X operation systems
- This vulnerability enabled MitM attacks on TLS connections

```
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,                    /* plaintext */
                       dataToSignLen,                 /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
            sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
                "returned %d\n", (int)err);
            goto fail;
    }

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
```

TLS Record Protocol Characteristics

- The connection is private because a symmetric-key algorithm (i.e., AES) is used to encrypt the data transmitted
- The identity of the communicating parties is authenticated via digital certificates that are exchanged and validated during the initial handshake
  - This (server-side) authentication is required for the server and optional for the client (i.e. client-side authentication)
    - We focus on server-side authentications for now
- The connection is reliable, because each message transmitted includes a message integrity check using a message authentication code to prevent undetected loss or alteration of the data during transmission

TLS Handshake Protocol Overview

- Secure (TLS) connection is initiated by client
  - Typically, via dedicated port, e.g. HTTP port 80 versus HTTPS port 443
- It uses public key cryptography to establish cipher settings and session-specific shared private keys with which further communication is encrypted using a symmetric cipher
  - Client and server agree on a **cipher suite** (a cipher and a hash function)
- The server also presents its digital certificate to the client for authentication
- To initiate the generation of session keys used for a secure connection, the client either:
  1. Encrypts a random number (PreMasterSecret) with the server's (RSA or EC) public key and sends the result to the server (only up to TLS 1.2)
     - Forward secrecy is not provided!
  2. Uses (Elliptic Curve) Diffie–Hellman key exchange (in TLS 1.2 and TLS 1.3)
     - This key may have the property of forward secrecy, but MitM attacks need to be mitigated

Recall Forward Secrecy

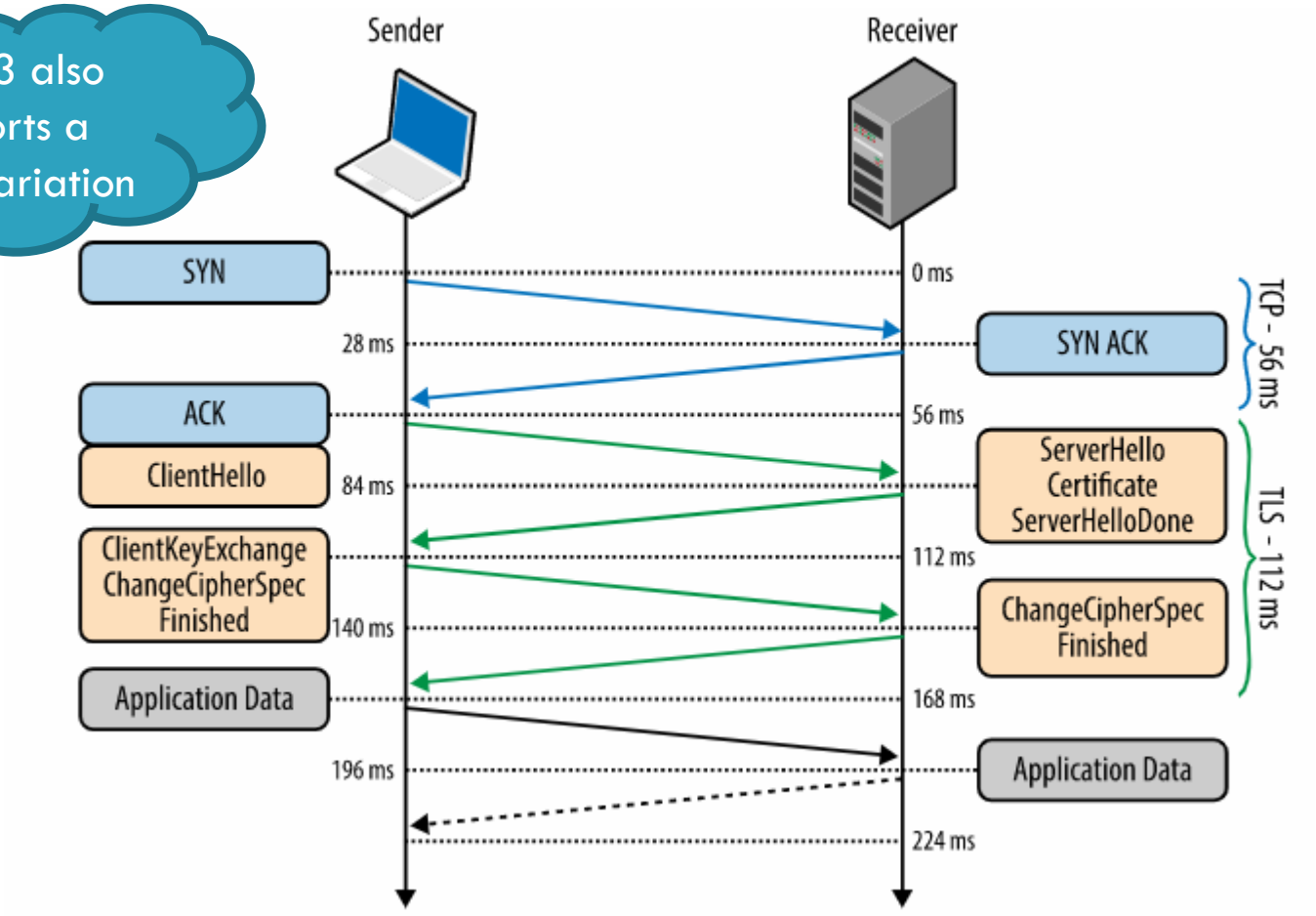
- Consider an attacker who
  - intercepts and records all client / server messages, including the handshake
  - recovers the server's private key sometime in the future, using the public key in the server's digital certificate as a starting point
- In option 1 the *PreMasterSecret* can now be retrospectively recovered, session keys can be calculated, and all subsequent messages can be decrypted by the attacker
- However, the DH key negotiation in option 2 is based on other secret token not linked to the server's private key
  - Nonetheless the key exchange has to be protected to prevent a MitM attack as seen before

Ephemeral Diffie-Hellman vs static Diffie-Hellman

- Static Diffie-Hellman key exchange (in TLS 1.2 only)
  - Always use the same Diffie-Hellman private keys (this saves CPU cycles)
  - Each time the same parties do a DH key exchange, they end up with the same shared secret → only partial forward secrecy
- Ephemeral Diffie-Hellman key exchange (compulsory in TLS 1.3)
  - A temporary DH key is generated for every connection and thus the same key is never used twice
  - This enables forward secrecy, which means that if the long-term private key of the server gets leaked, past communication is still secure
- This distinction also holds for the Elliptic Curve DH variants
  - ECDHE (ephemeral, provides Forward Secrecy) and
  - ECDH (static)

TLS Handshake Overview

TLS 1.3 also supports a faster variation



In-Class Activity: Analysis of TLS Handshake

- Option 1:
  - Open Wireshark and start packet recording
  - In your browser open a HTTPS secured website you never visited before (e.g. fussball.de)
  - Stop packet recording and filter all TLS-related packets (Filter option 'tls')
- Option 2:
  - Load pcap file "revenue tls" (Blackboard file name "Example Wireshark TLS Handshake")
- Wireshark does a great job analysing the content of the packets

TLS Handshake

- TCP connection establishment
  - SYN – SYN/ACK – ACK
- The *ClientHello* message
  - The client initiates the handshake by sending a (plaintext) “hello" message to the server
  - The message includes
    - the highest TLS version the client supports (1.2 or 1.3)
    - the cipher suites supported (i.e. what algorithms are available to client, see next slide),
    - a session identifier
      - Note that the session id is kept empty if the clients starts an entirely new session
    - a string of random bytes known as the "client random“

Cipher Suite Naming Scheme

- Examples:
  - TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
  - TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- Here:
  - TLS defines the protocol that this cipher suite is for
  - ECDHE indicates the key exchange algorithm being used (Elliptic Curve Diffie-Hellman Ephemeral)
  - RSA or ECDSA (Elliptic Curve Digital Signature Algorithm) authentication mechanism during the handshake
    - Remember the *ServerHello* message contains the server's public DH parameter signed with its private (RSA) key or signed via ECDSA
  - AES cipher for symmetric data encryption
  - 128-bit or 256-bit AES key size
  - GCM type of encryption (Galois/Counter Mode, covered before)
  - SHA256 / SHA384 hash function (HMAC) indicates the message authentication algorithm which is used to authenticate a message
    - 256-bit or 384-bit digest size

Cipher Suite (Wireshark Screenshot)


```
Cipher Suites (16 suites)
    Cipher Suite: Reserved (GREASE) (0x7a7a)
    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)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
```

- Two bytes specify a cipher suite
- Suits have different levels of robustness
- See also for details https://ciphersuite.info/cs/

Cipher Suite (Wireshark Screenshot)


```
Cipher Suites (16 suites)
    Cipher Suite: Reserved (GREASE) (0x7a7a)
    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)
    Cipher Suite: TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca8)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
    Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
    Cipher Suite: TLS_RSA_WITH_AES_128_GCM_SHA256 (0x009c)
    Cipher Suite: TLS_RSA_WITH_AES_256_GCM_SHA384 (0x009d)
    Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
    Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
```

- Two bytes specify a cipher suite
  - Suits have different levels of robustness
  - See also for details https://ciphersuite.info/cs/
- Cipher suits 0x1301, 0x1302 and 0x1303 do not describe the
  - server authentication mechanism (e.g., RSA)
  - key exchange algorithm (e.g., ECDHE)
- This is a simplification introduced with TLS 1.3 (more later!)

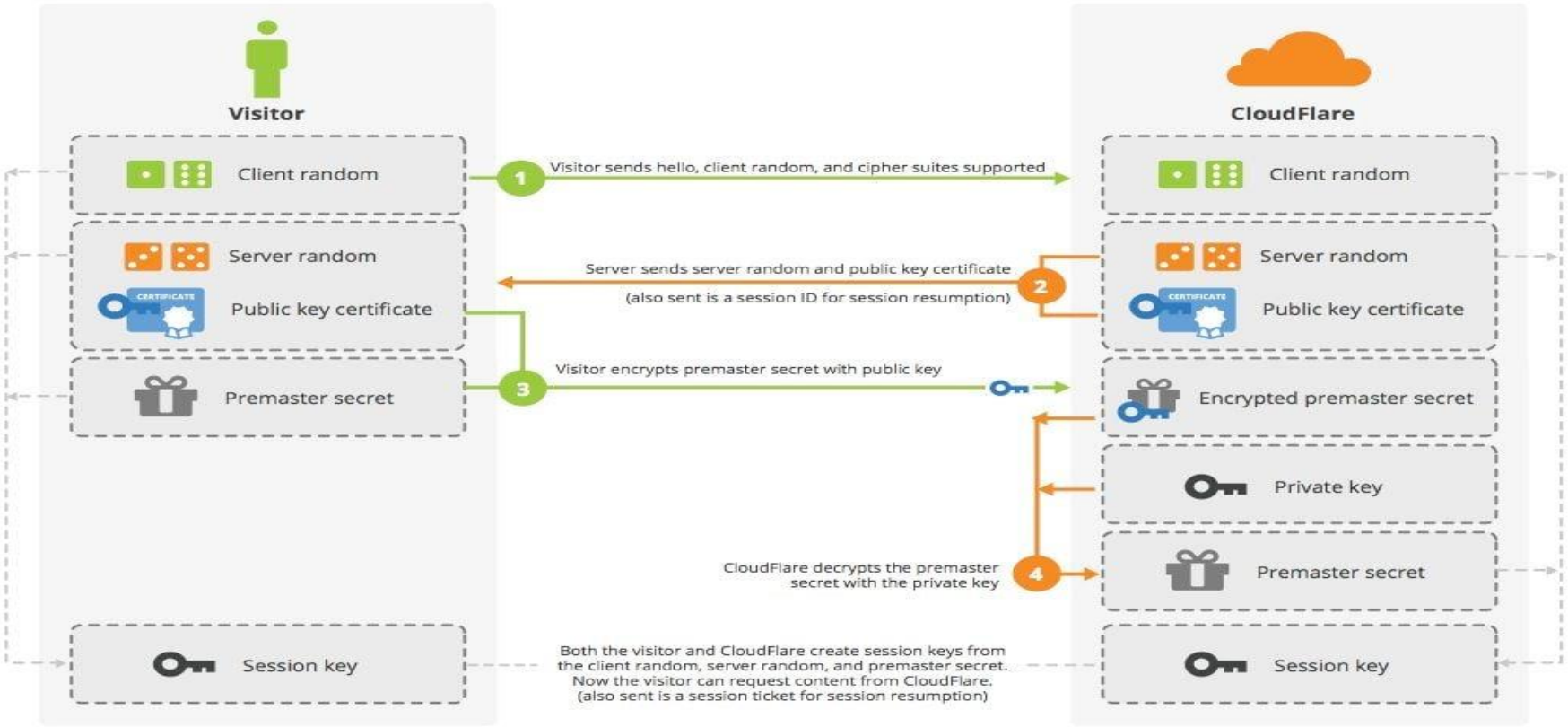
TLS Handshake

- The *ServerHello* message
  - In reply to the *ClientHello* message, the server sends a (plaintext) message containing
    - the server's digital certificate
    - a certificate chain that includes all intermediate certificates up to the root CA along the certification path
    - the server's chosen cipher suite,
    - its chosen session id (session resumption → later), and
    - the "server random," another random string of bytes that's generated by the server

TLS Handshake

- Authentication
  - The client verifies the server's digital certificate with the certificate authority that issued it using the intermediate certificates
  - This confirms that the public key is linked to the certificate owner, but does not confirm the authenticity of the server yet (as any threat actor could use the server's certificate in a spoofing attack)
- Key negotiation (next slides)
  - Option 1: RSA handshake (not supported anymore with TLS 1.3)
  - Option 2: DH handshake (ECDH to be exact)
- Change Cipher Spec (not shown in the diagrams in the following slides)
  - In due course both parties will send a ChangeCipherSpec message which is used to indicate that their subsequent messages will be sent encrypted using the negotiated key and algorithm
- Finished (not shown in the diagrams on the following slides)
  - This is an encrypted message (more later)

Option 1 Overview: RSA Handshake



Option 1: RSA Handshake

- Premaster secret generation
  - The client generates a random string of bytes, the "premaster secret"
  - The premaster secret is encrypted with the server's public key
- Premaster secret distribution
  - The client sends the encrypted secret to the server
  - The server decrypts the premaster secret
- Master secret creation
  - Both client and server generate a master secret (which is not the encryption key used), using
    - the client random,
    - the server random,
    - and the premaster secret

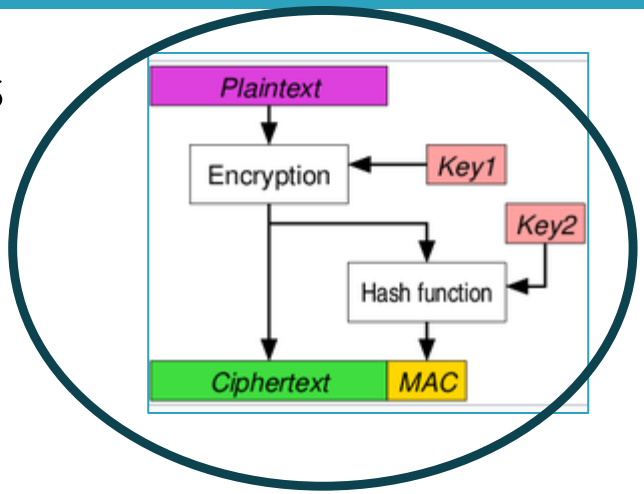
3 4

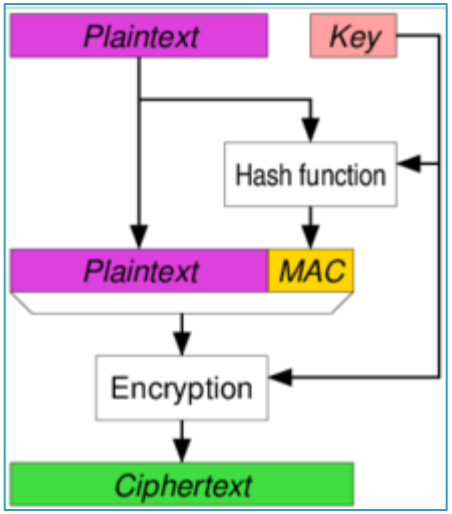
Option 1: RSA Handshake

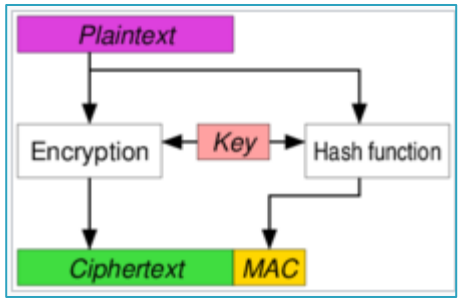
- Session keys generation
  - Using the master secret both client and server generate 4 session keys (see next slide):
    - Client-write symmetric encryption key
    - Server-write symmetric encryption key
    - Client-write MAC key (for client message authentication)
    - Server-write MAC key (for server message authentication)
- Client is ready
  - The client sends a *finished* message that is encrypted with the session key
- Server is ready
  - The server sends a *finished* message encrypted with the session key
    - This validates the authenticity of the server, i.e. the client has proof that the server is in possession of the private key linked to the server certificate
- Secure symmetric encryption can be provided
  - The handshake is completed, and communication continues using the session keys

Recall: Authenticated Encryption with Additional Data

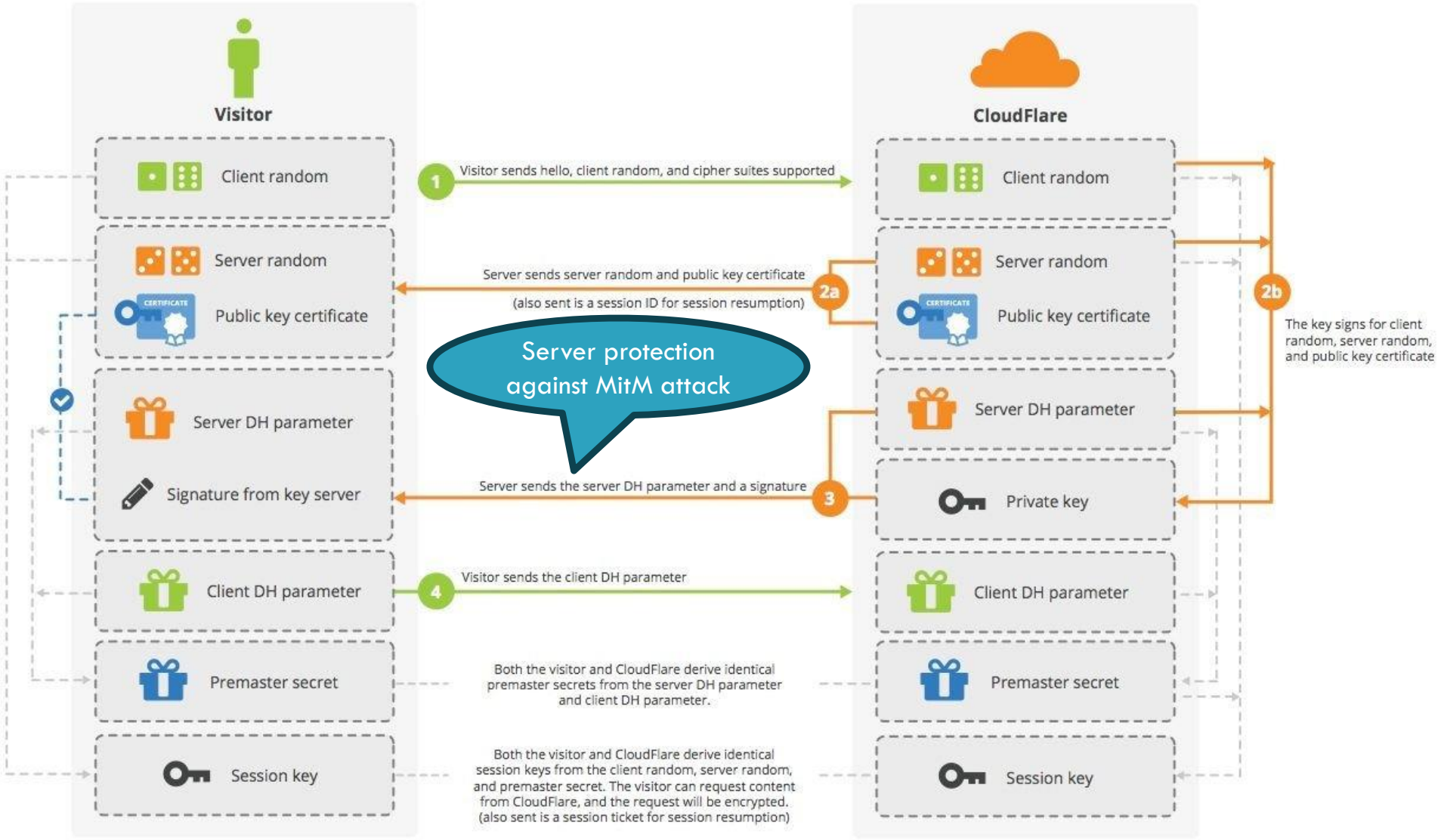
- Links back to the use of hash functions (→ previous lecture):
  - Encrypt-then-MAC (EtM) → top right
  - Encrypt-and-MAC (E&M) → bottom right
  - MAC-then-Encrypt (MtE) → bottom left







Option 2 Overview: DH Handshake



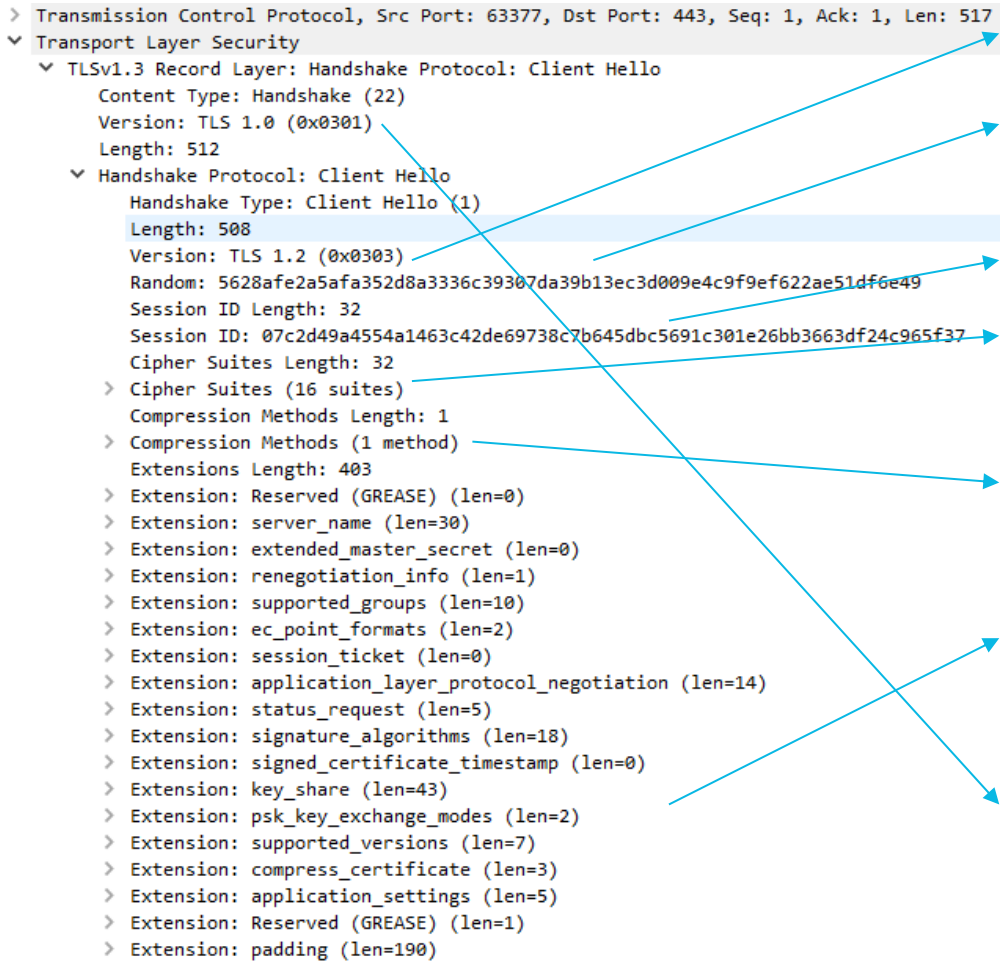
Option 2: DH Handshake

- Server Key Exchange 3
  - This message contains either ECDH parameters (elliptic curve + primitive root + public ECDH parameter) or DH parameters (modulus, primitive root, public DH value) to be used by the client
  - The values are signed by using the private (RSA or EC) key of the server so that the client can verify (using corresponding public key in the certificate) that the parameter indeed came from the server it is talking to and not an attacker that impersonates the server
    - Note that in
      - TLS 1.2: DH, ephemeral DH (DHE), ECDH, or ECDHE can be used
      - TLS 1.3: only ECDHE is allowed
- Client Key Exchange 4
  - Contains the client's public parameters for the DH algorithm
  - Client parameters are **not signed** (as the client does not have a certificate)

Option 2 Overview: DH Handshake

- Client and server calculate the **premaster secret**
  - Instead of the client generating the premaster secret and sending it to the server, as seen before, the client and server use the DH parameters they exchanged to calculate a matching premaster secret separately
- Master secret creation
  - The client and server calculate the master secret using the premaster secret, client random, and server random
- Session keys generation
  - Same as before
- Client is ready
  - Same as before
- Server is ready
  - Secure symmetric encryption achieved

ClientHello (Wireshark Screenshot)



- Highest TLS version supported
- 32-byte random structure (contains a 4-byte timestamp and a 28-byte random → next slide)
- Random 32-byte session id
- List of supported cryptographic algorithms
- List of supported data compression methods, obsolete with TLS 1.3
- List of extensions
- **Note that all is plaintext!**
- Version of the record protocol (still 1.0)

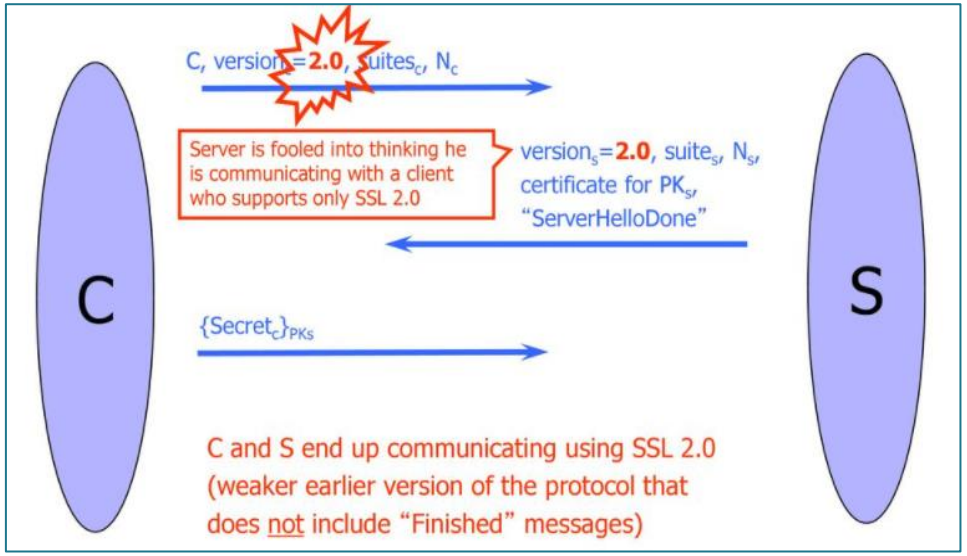
Client Hello: 32-Byte Random Structure

- From RFC 5246 Section 7.4.1.2:

```
The ClientHello message includes a random structure, which is used
later in the protocol.

   struct {
       uint32 gmt_unix_time;
       opaque random_bytes[28];
   } Random;

gmt_unix_time
   The current time and date in standard UNIX 32-bit format
   (seconds since the midnight starting Jan 1, 1970, UTC, ignoring
   leap seconds) according to the sender's internal clock.  Clocks
   are not required to be set correctly by the basic TLS protocol;
   higher-level or application protocols may define additional
   requirements.  Note that, for historical reasons, the data
   element is named using GMT, the predecessor of the current
   worldwide time base, UTC.

random_bytes
   28 bytes generated by a secure random number generator.
```

The *Version Rollback* Attack

- This MitM attack targets SSL 3.0
- Here the attacker intercepts the plaintext *ClientHello* message, that includes the highest TLS version the client supports (i.e. SSL 3.0)

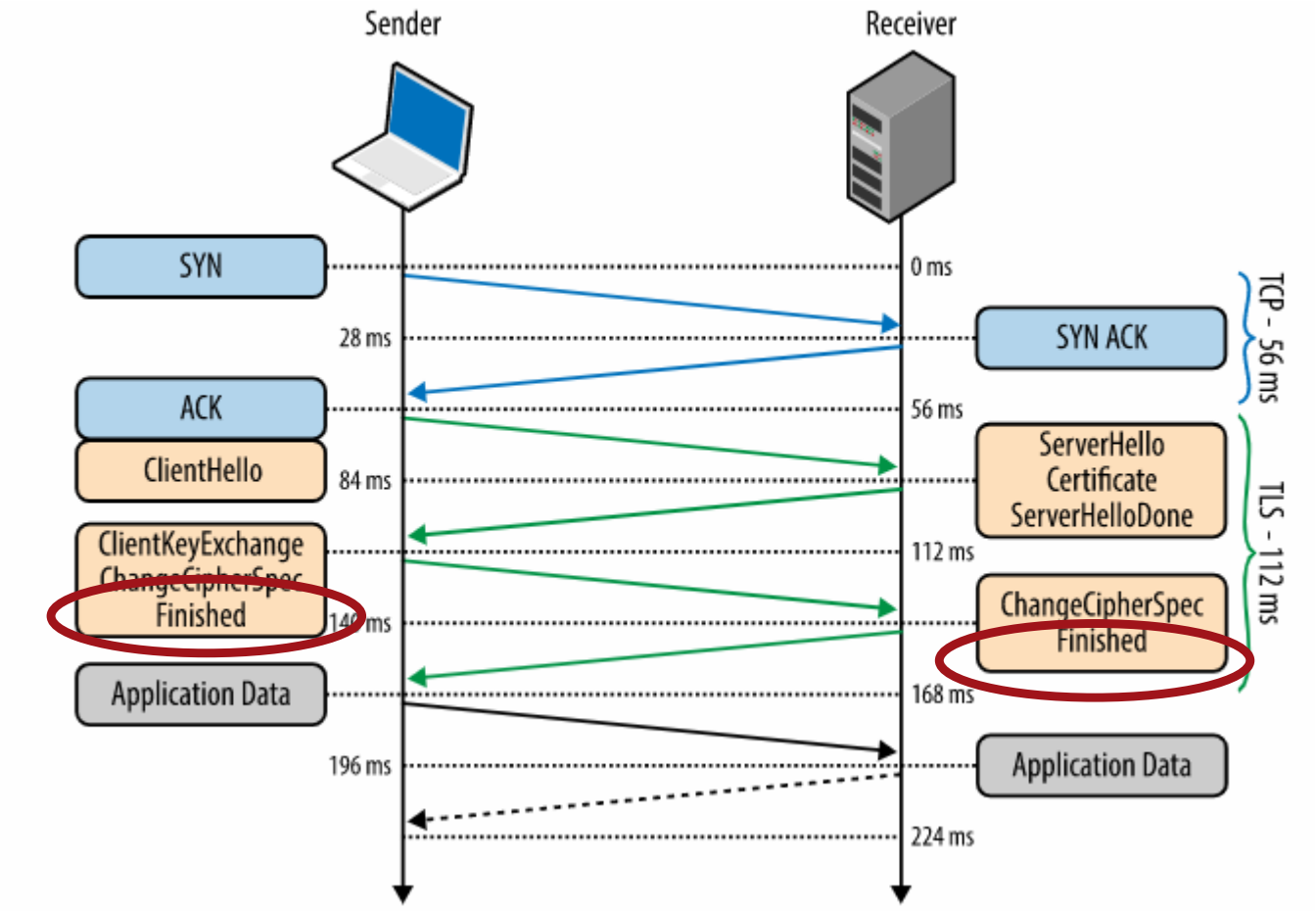


- The attacker changes the message content to "SSL 2.0", thereby tricking both server and client to accept a weaker (i.e. flawed) protocol
  - The server assumes the client only understands SSL 2.0
  - The client assumes the server only understands SSL 2.0

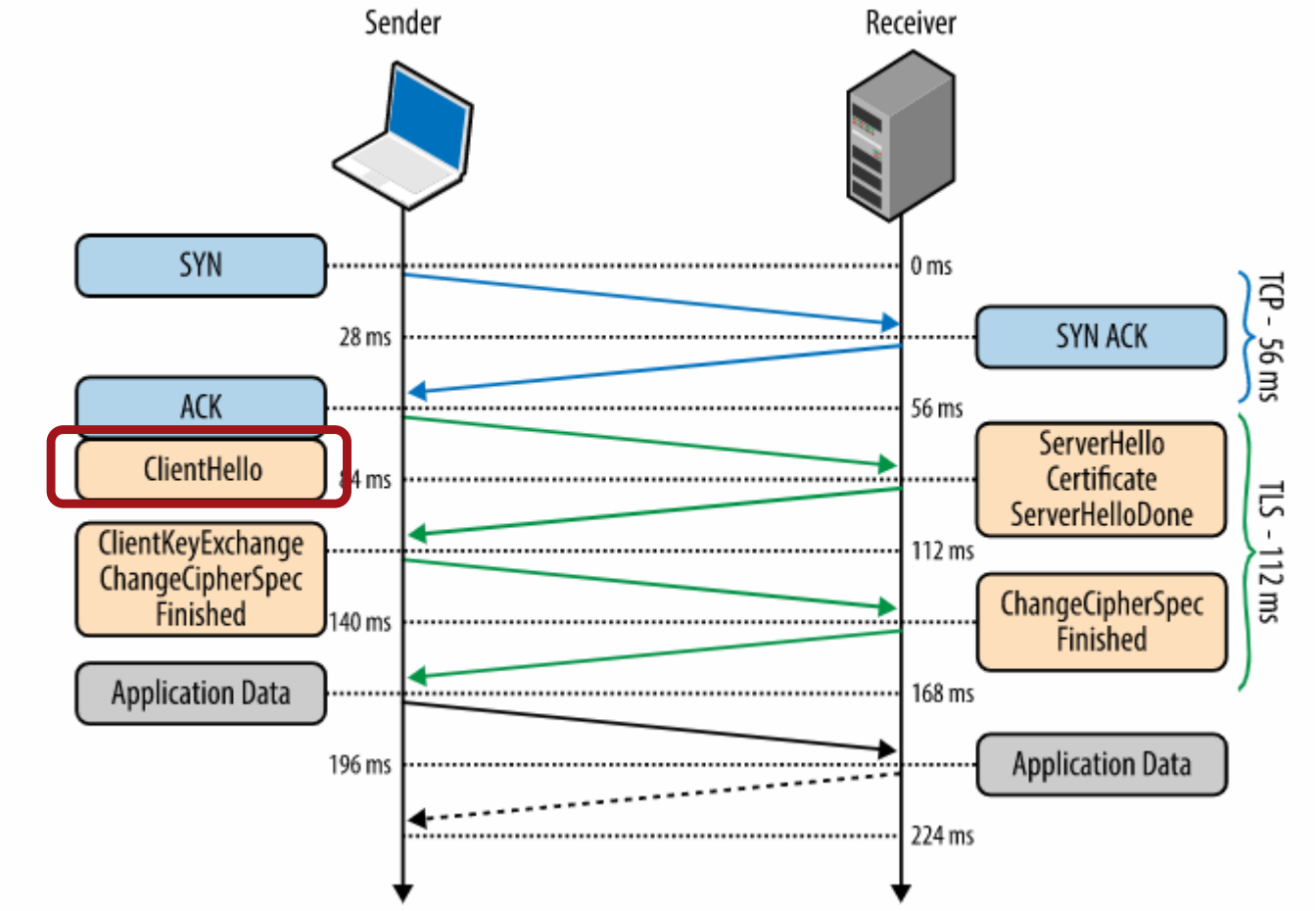
TLS Protection against MitM Attacks

- MitM attacks cannot be mitigated, as *Client Hello* and *Server Hello* messages, as well as the client key exchange messages for DH key negotiation are sent as plaintext
- Instead, the *Finished* messages of both client and server contain the result of the HMAC of the negotiated cyphersuite, truncated to 12 bytes (therefore called a pseudo-random function (PRF)), of:
  - The master secret
  - A hash of all the previous handshake messages (from *ClientHello* up to but excluding the *Finished* message)
  - The finished-label string ("client finished" for client message and "server finished" for server message)
- Therefore, both sides can retrospectively validate the integrity of the handshake protocol
  - This includes all MitM attacks during the key exchange protocol (remember only the server value was signed)

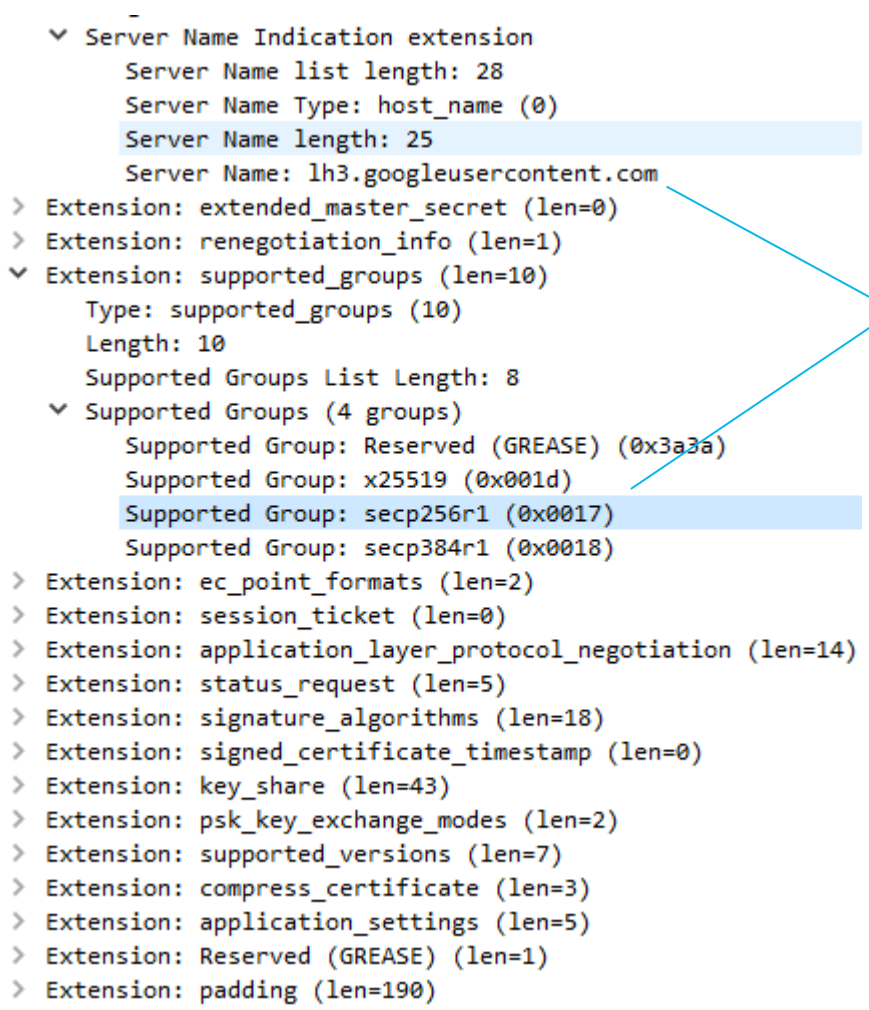
TLS Protection against MitM Attacks



TLS Handshake Extensions



TLS Handshake Extensions in the Client Hello Message



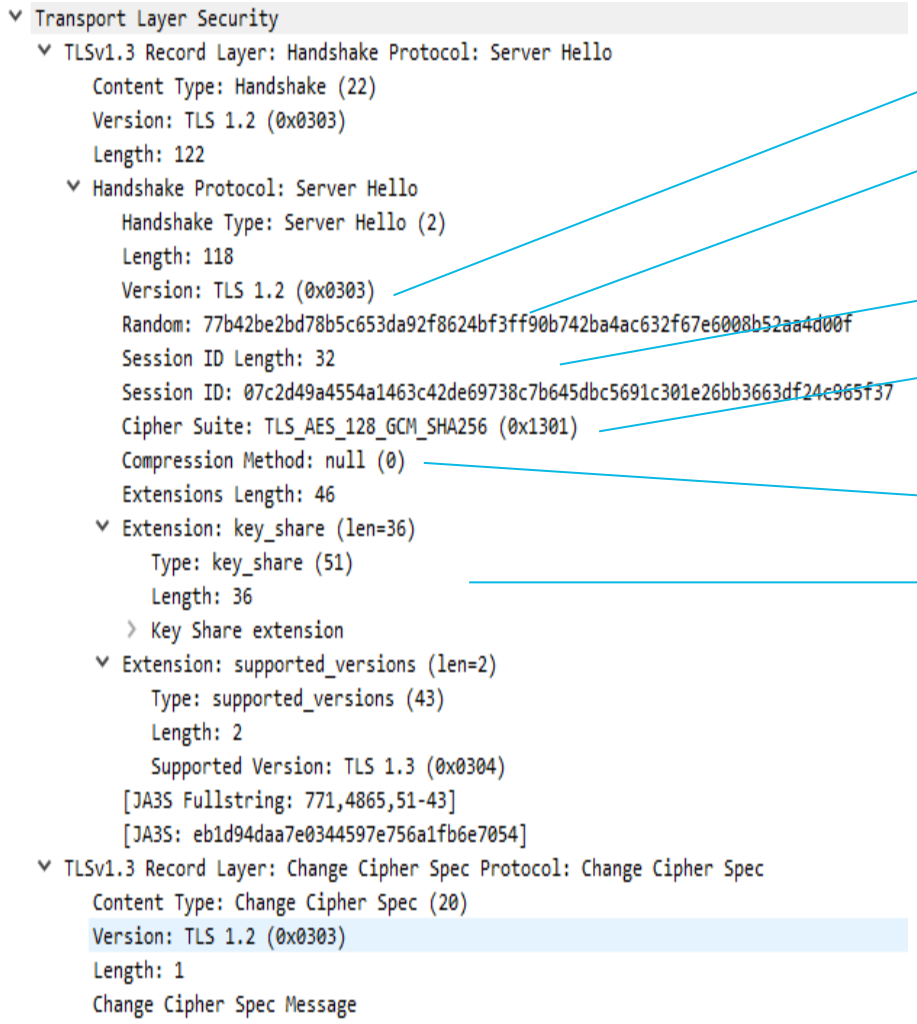
- These provide additional info to the server
- A few notable examples:
  - Supported elliptic curves
  - Server Name Indication
    - A client indicates which hostname it is attempting to connect to at the start of the handshake process
    - This allows a server to present one of multiple possible certificates on the same IP address and TCP port number and hence allows multiple secure (HTTPS) websites to be served by the same IP address without requiring all those sites to use the same certificate

TLS Handshake Extensions in the Client Hello Message


```
Extensions Length: 403
> Extension: Reserved (GREASE) (len=0)
v Extension: server_name (len=30)
    Type: server_name (0)
    Length: 30
  > Server Name Indication extension
> Extension: extended_master_secret (len=0)
> Extension: renegotiation_info (len=1)
> Extension: supported_groups (len=10)
> Extension: ec_point_formats (len=2)
> Extension: session_ticket (len=0)
> Extension: application_layer_protocol_negotiation (len=14)
> Extension: status_request (len=5)
v Extension: signature_algorithms (len=18)
    Type: signature_algorithms (13)
    Length: 18
    Signature Hash Algorithms Length: 16
  v Signature Hash Algorithms (8 algorithms)
    > Signature Algorithm: ecdsa_secp256r1_sha256 (0x0403)
    > Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
    > Signature Algorithm: rsa_pkcs1_sha256 (0x0401)
    > Signature Algorithm: ecdsa_secp384r1_sha384 (0x0503)
    > Signature Algorithm: rsa_pss_rsae_sha384 (0x0805)
    > Signature Algorithm: rsa_pkcs1_sha384 (0x0501)
    > Signature Algorithm: rsa_pss_rsae_sha512 (0x0806)
    > Signature Algorithm: rsa_pkcs1_sha512 (0x0601)
> Extension: signed_certificate_timestamp (len=0)
v Extension: key_share (len=43)
    Type: key_share (51)
    Length: 43
  v Key Share extension
      Client Key Share Length: 41
    > Key Share Entry: Group: Reserved (GREASE), Key Exchange length: 1
    > Key Share Entry: Group: x25519, Key Exchange length: 32
```

- Signature hash algorithms
  - In TLS1.2 only, the client MAY include the *signatureAlgorithms* extension indicating what types of signatures it supports verifying
  - This includes the signatures on the certificates in the server's chain
  - This feature is dropped again in TLS 1.3

ServerHello (Wireshark Screenshot)

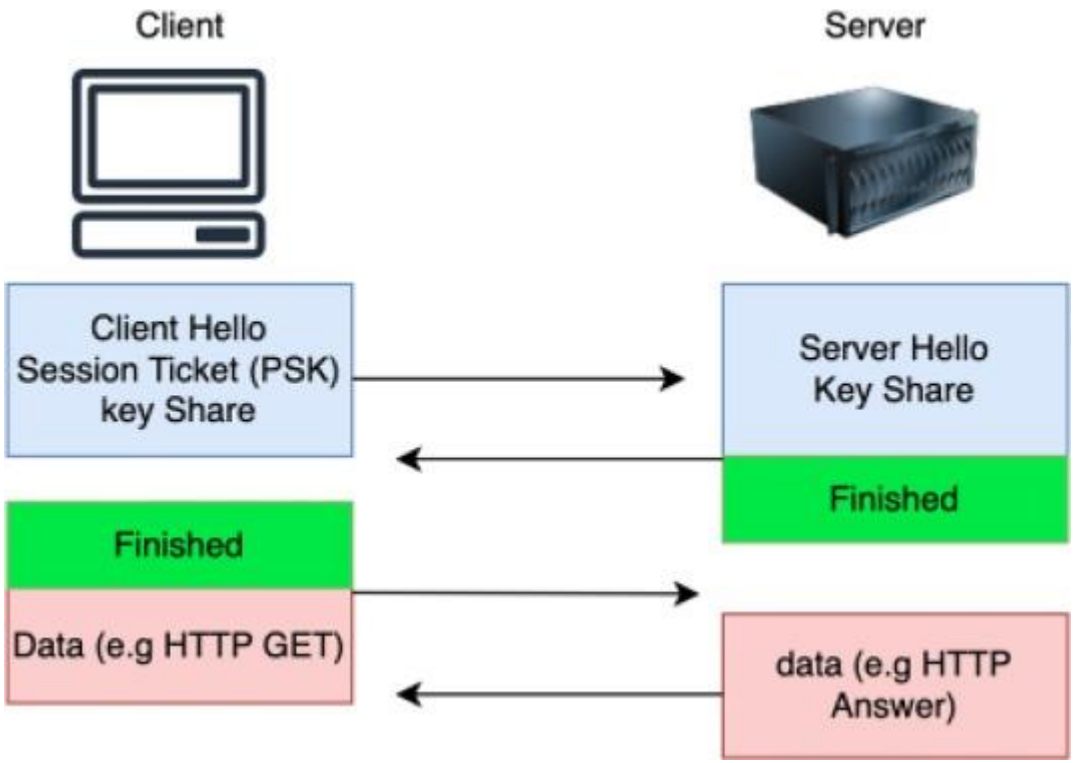


- Highest TLS version supported
- Random 32 byte nonce (contains a timestamp)
- Client session id
- Chosen cipher suite
- List of supported data compression methods, obsolete with TLS 1.3
- List of extensions
- Again, all plaintext!

TLS 1.2 Session Resumption

- Assume a client wants to reconnect to a server it has previously communicated with
- If the client still has the negotiated cipher suite and keys from the previous handshake cached, it can send the server the previously used session id in the *ClientHello* message
- If the server has cached all this data too, it can shorten the handshake
- However, it still requires a round trip to verify the session, which can introduce some latency
  - Otherwise, a full new session negotiation is required, which will generate a new session ID, and which will take longer
- If a browser requires multiple connections to the same host (e.g., when HTTP/1.x is used), it will often wait for the first TLS negotiation to complete before opening additional connections to the same server, such that they can be "resumed" and reuse the same session parameters
- On the other hand, caching the parameters of many client sessions over long periods of time does not scale and it rarely used

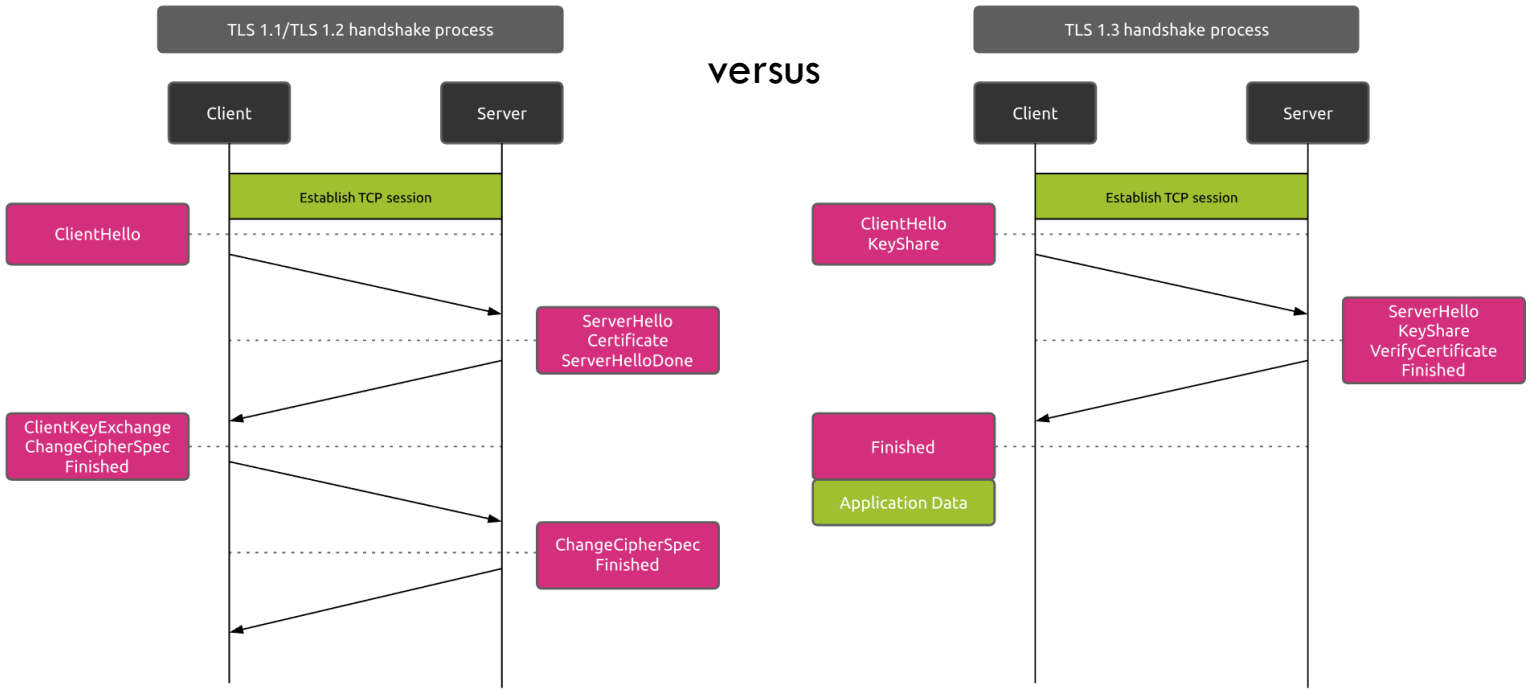
TLS 1.2 Session Resumption



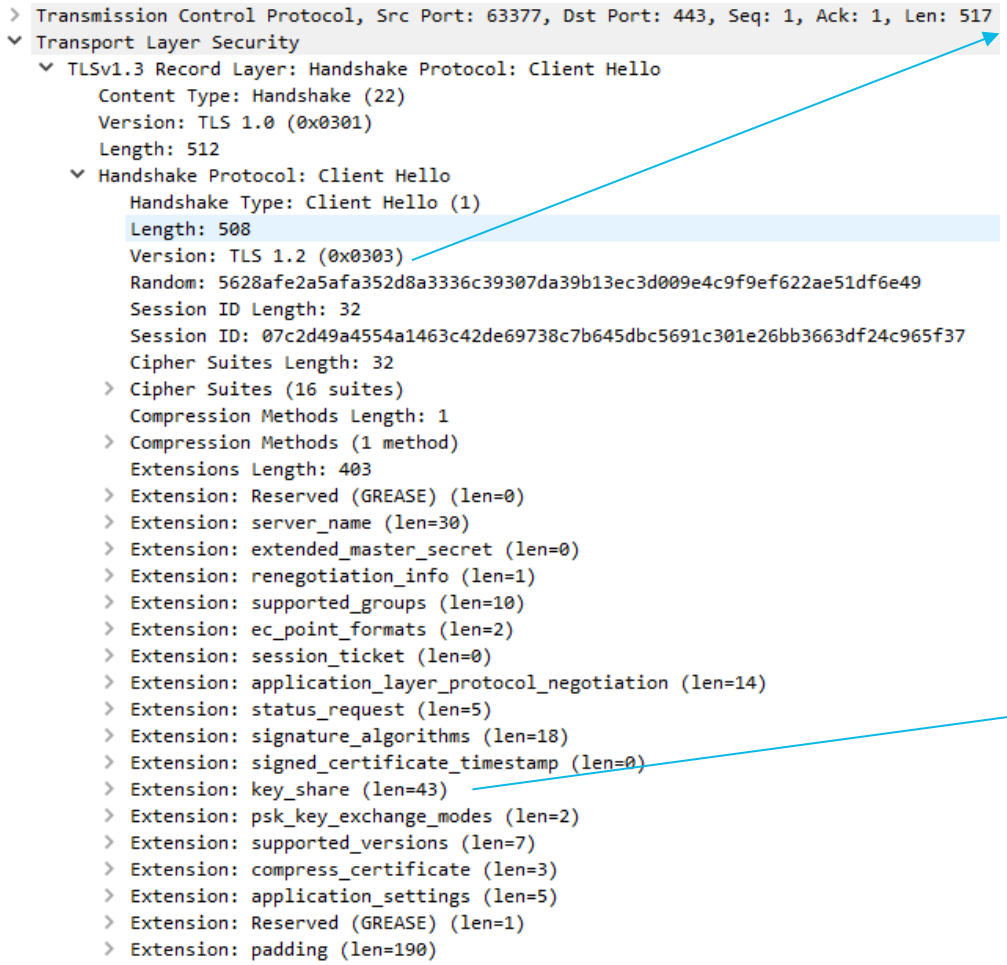
- Here all key negotiation steps are excluded

TLS 1.3 and 1-Round Trip Time (1-RTT)

- Beside only supporting a streamlined ciphersuite for key negotiation (ECDHE only), TLS 1.3 also supports a new accelerated handshake process called 1-RTT

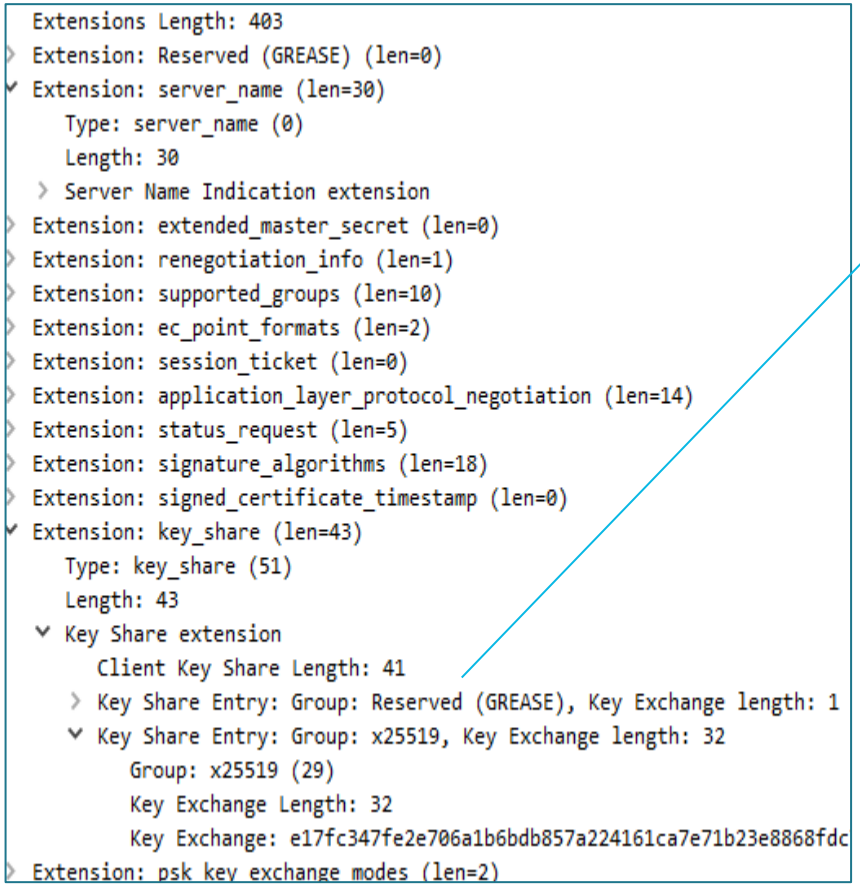


The key_share Extension



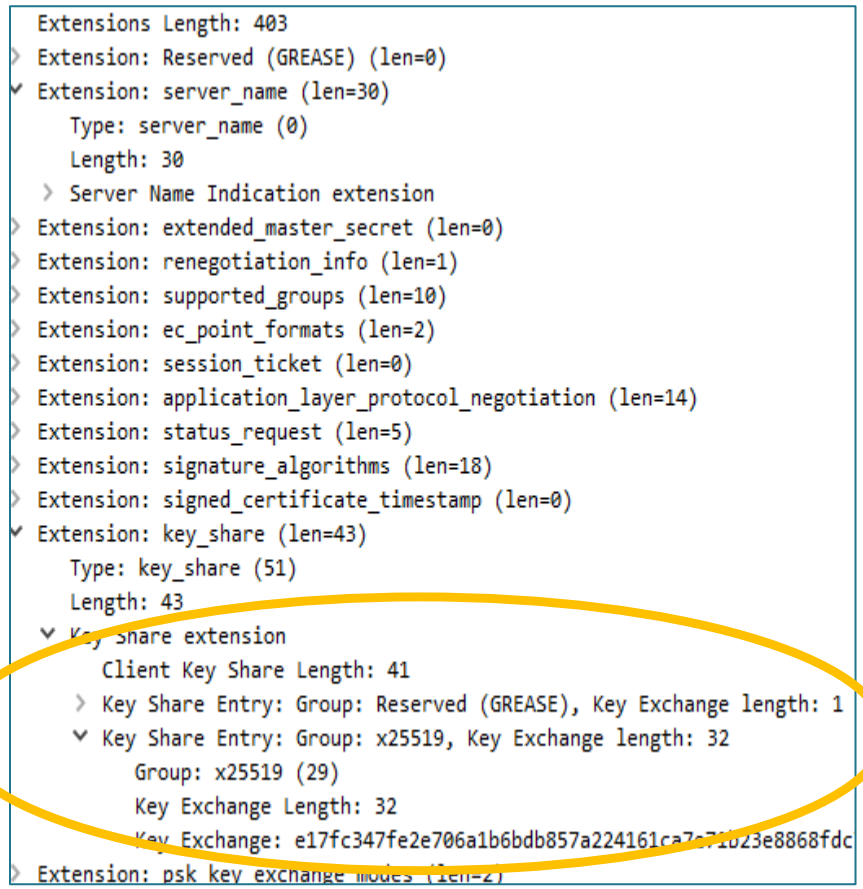
- This seems to suggest that the client is requesting a TLS 1.2 handshake
- A TLS 1.3 client hello looks superficially exactly like a TLS 1.2 handshake, right down to the version number
- If the server only understands TLS 1.2, it will just negotiate a TLS 1.2 handshake as before
- However, the new *ClientHello* extension *key_share* indicates that the client understands version 1.3

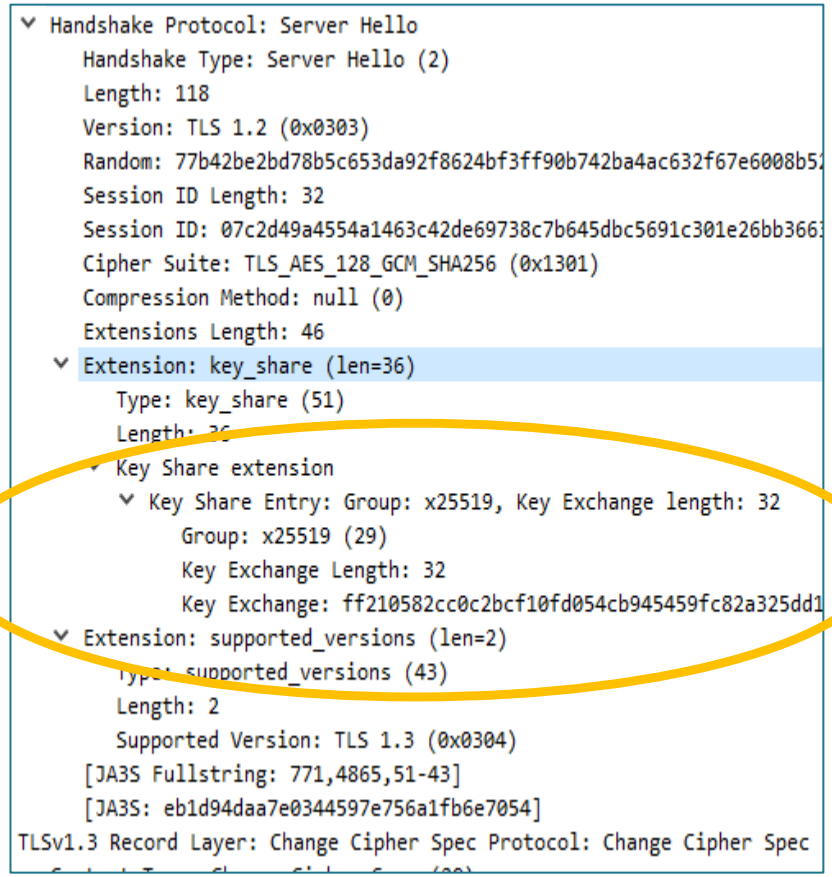
The key_share Extension



- In TLS 1.2, the *ClientKeyExchange* message is used *to kick-off the key exchange*
- This is now complemented by a method where the client presents the server with a ECDHE key exchange right at the start
- The idea is that the client just goes ahead and assumes that the server will select its preferred key exchange method and returns its ECDHE parameter
- If the server selects a different key exchange method, it will respond with a *RetryHelloRequest* message (not shown here) which restarts the handshake; this can be the result of either:
  - An ECDHE group that is not supported by the server
  - A server (security) policy that necessitate the use of different ECDH parameters than those proposed by the client
- In most cases the server will support the preferred key exchange method, so the handshake is shorter

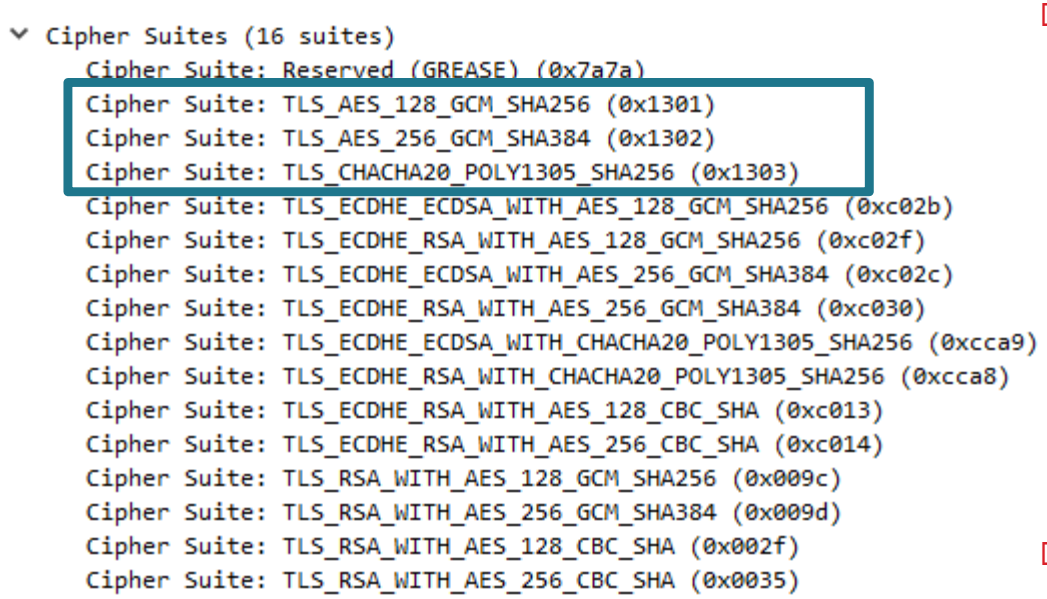
The *key_share* Extension in both *ClientHello* (Left) and *ServerHello* (Right)





- Note that in the server response all messages after the *ServerHello* message are already encrypted

Cipher Suite (Wireshark Screenshot)



- The highlighted cipher suits are used in TLS 1.3
- The negotiable bits are:
  - 128- or 256-bit AES in GCM mode, or
  - 256-bit ChaCha20 combined with POLY1305
    - ChaCha20 is a stream cipher
    - POLY1305 is a hash function, here used for authenticated encryption
  - SHA256 or SHA384 hashing
- The fixed elements are:
  - ECDHE using Curve25519
  - Message authentication using RSA or ECDSA
    - Depending if the server certificate contains a public RSA or EC key

Review a TLS Handshake with OpenSSL

1. Init TLS connection:

   openssl s_client -connect universityofgalway.ie:443

2. Review the output on screen

3. You may decode the server certificate via https://www.sslshopper.com/certificate-decoder.html

Review a TLS Handshake with OpenSSL

- Connection Status:
  - CONNECTED(00000003): Indicates that the connection to the server was successful.
- Certificate Verification:
  - depth=2, depth=1, depth=0: These lines show the verification process of the certificate chain. Each depth level represents a certificate in the chain, starting from the root CA (depth=2) to the server's certificate (depth=0).
  - verify return:1: Indicates that the certificate at each depth level was successfully verified.
- Certificate Chain:
  - Lists the certificates in the chain, including the subject (s:) and issuer (i:) details. The chain starts from the server's certificate and goes up to the root CA.
- Server Certificate:
  - The server's certificate is displayed in PEM format, including details like the subject and issuer.
- Peer Information:
  - No client certificate CA names sent: Indicates that no client certificate authority names were sent.
  - Peer signing digest: SHA256: Specifies the digest algorithm used for signing.
  - Peer signature type: RSA-PSS: Specifies the signature algorithm used.
  - Server Temp Key: X25519, 253 bits: Indicates the temporary key used for key exchange.

The HTTPS Protocol

- HTTPS (Hypertext Transfer Protocol Secure) is syntactically identical to the HTTP protocol, but operates on top of TLS (rather than TCP)
  - TLS on the other hand operates on top of TCP
- It provides secure client / (web) server HTTP data communication, while also allowing a client (i.e. web browser) to authenticate the (web) server, as part of the TLS handshake
- The default HTTPS port is 443

The Importance of Server-Side Authentication: Pharming Scams

What is it?

- Pharming scams use domain spoofing (in which the domain appears authentic) to redirect users to copies of popular websites where personal data like usernames, passwords and financial information can be 'farmed' and collected for fraudulent use

How can it be achieved – Simple Pharming!

- Copy a website 1:1 and present it to the victim using a slightly different domain name

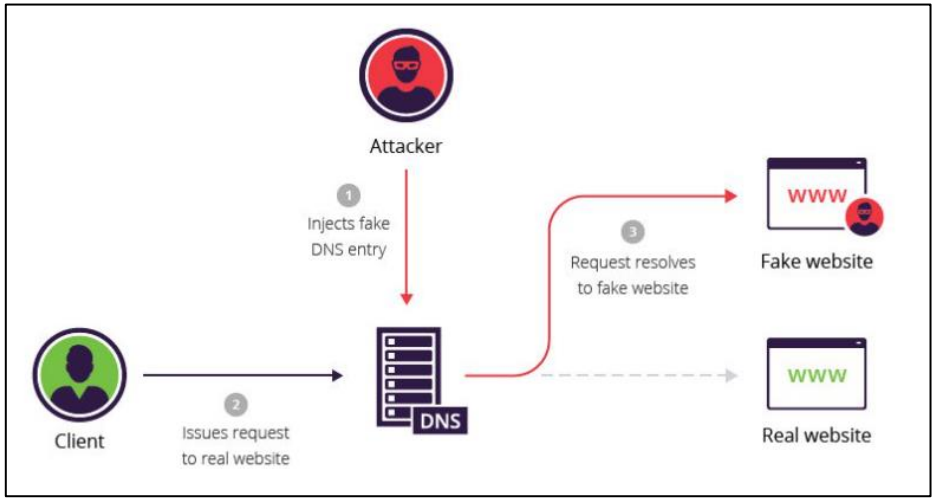
The Importance of Server-Side Authentication: Pharming Scams

What is it?

- Pharming scams use domain spoofing (in which the domain appears authentic) to redirect users to copies of popular websites where personal data like usernames, passwords and financial information can be 'farmed' and collected for fraudulent use

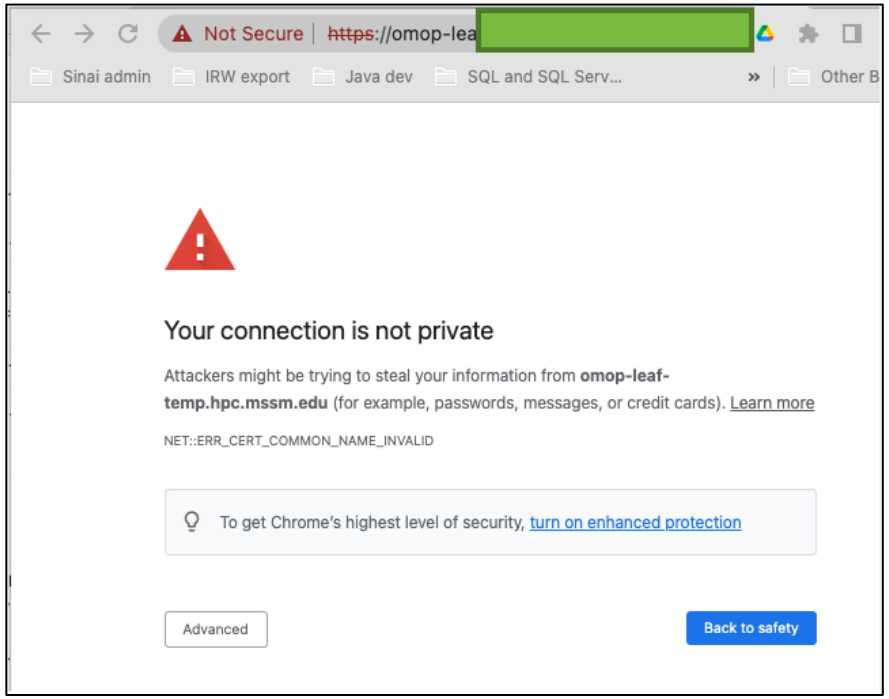
How can it be achieved - DNS Spoofing!

- Similar to simple pharming, but also manipulate the DNS server to redirect DNS queries to the attacker's website, i.e. the same domain name is used
- Known as DNS poisoning, DNS cache poisoning or DNS spoofing



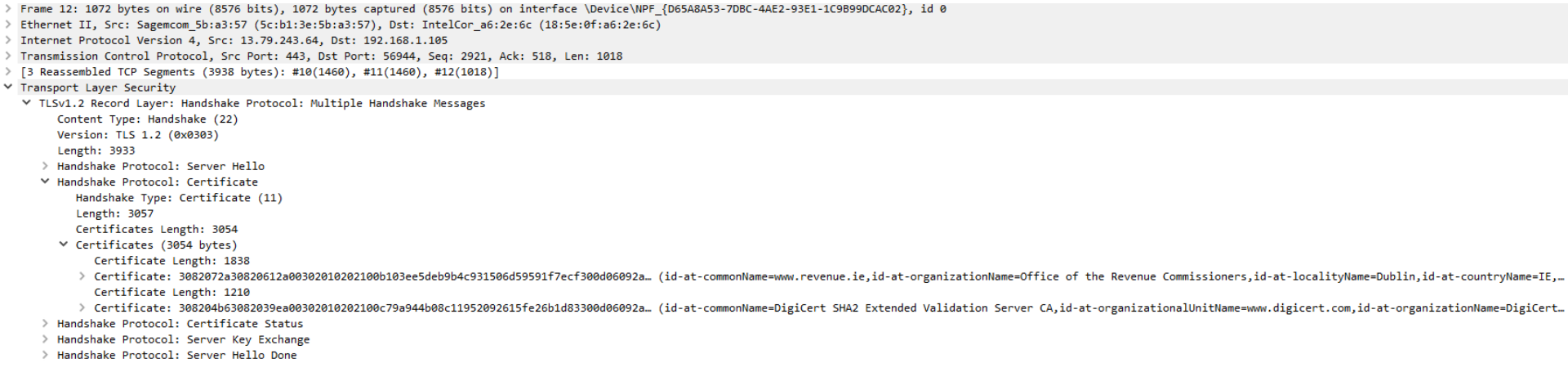
Anti-Pharming Support in your Browser

- In DNS spoofing, the malicious server cannot support HTTPS or TLS, as its doesn't have the spoofed server's private key
  - It has its certificate though, but that's not enough to complete the TLS handshake
- All modern browsers pick up on this and abort the connection
- Also, users are warned if TCP rather than TLS is used (see image)

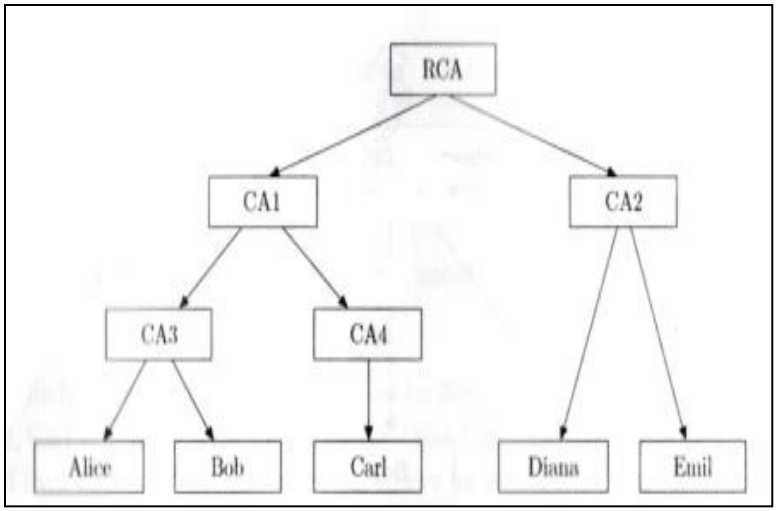


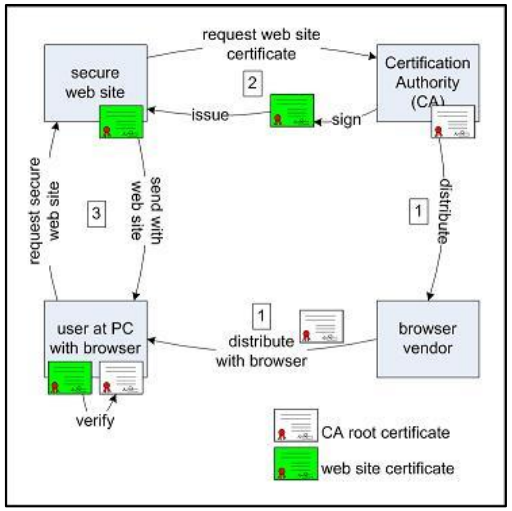
Certificate Stapling

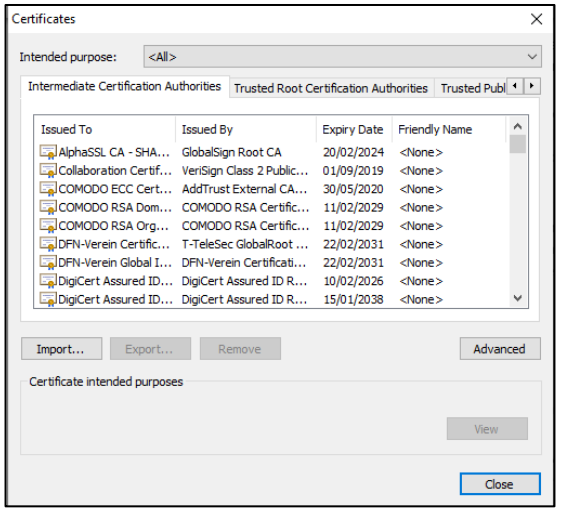
- In certificate stapling, the server appends all certificates in the path up to the root CA / RCA in a “Certificate” message, which is sent together with its *ServerHello* message to the client
- These stapled certificates are sent as
  - plaintext in TLS 1.2 (see Wireshark screenshot below)
  - ciphertext in TLS 1.3 (as all messages after the “Server Hello” message are already encrypted



Example Certificate Path Validation





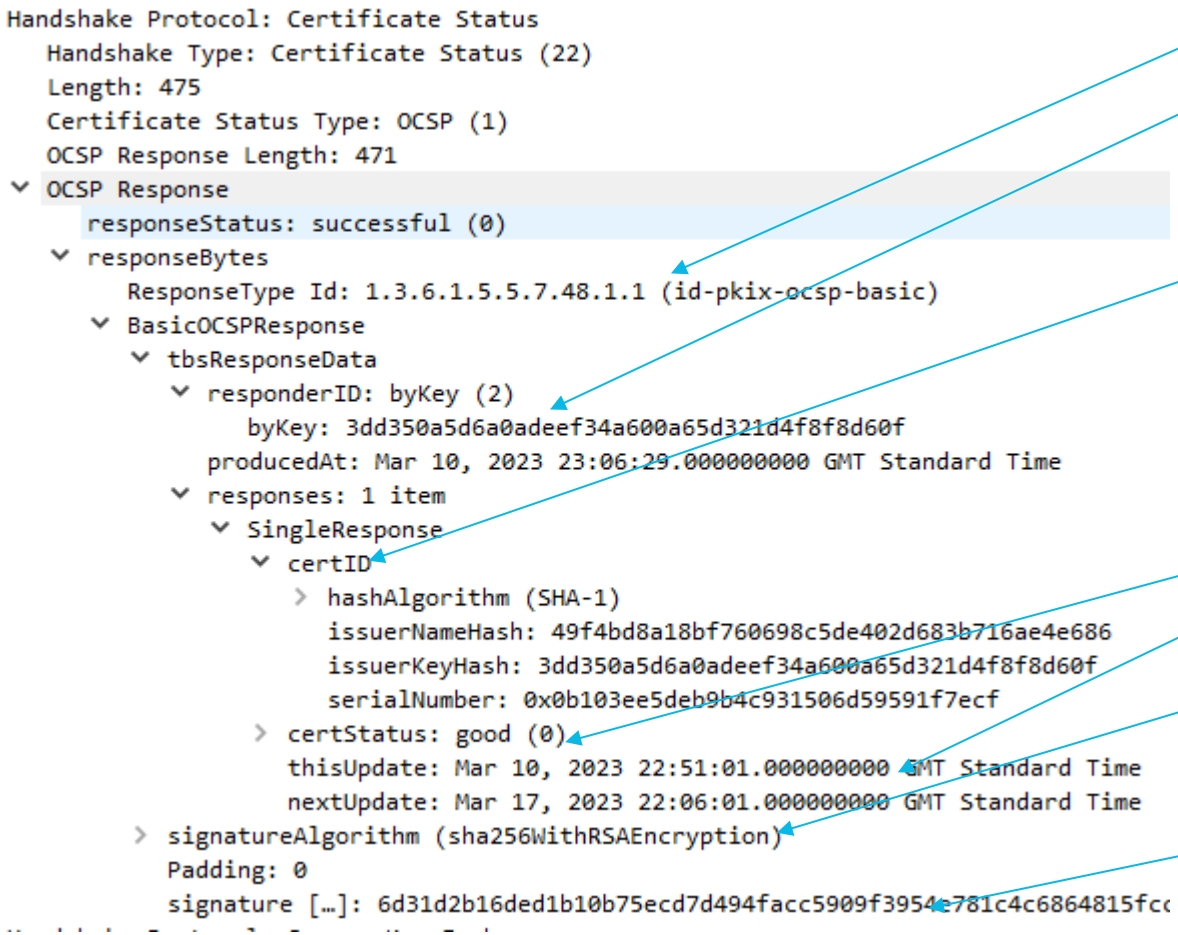


- For Alice (Client PC with web browser) to authenticate Diana (Server that hosts secure website), she requires CA2's (Certification Authority) certificate
- This may be already installed in Alice's browser (right image) together with RCA's certificate
- However, there's no guarantee that a browser contains the certificates of all intermediate CAs
- On the other hand, the handshake process should not be delayed by the client collating all the certificates belonging the Diana's certificate path
- Therefore, the server (Diana) provides Alice with the chain of certificates up to RCA level via certificate stapling

OCSP Stapling

- Recall: The Online Certificate Status Protocol (OCSP) is a standard for checking the revocation status of X.509 digital certificates
  - An OCSP response is digitally signed and time-stamped by the CA (OCSP server) that confirmed the revocation status of a certificate
- In OCSP stapling
  - The client includes a "status_request" extension in its *ClientHello* message
  - The server includes the OSCP response “Certificate Status” message in the *ServerHello* response
- This eliminates the need for a client to contact the CA, thereby improving overall performance
- However, the status of intermediate and root certificates is typically managed by separate OCSP checks

The ServerHello OCSP Response

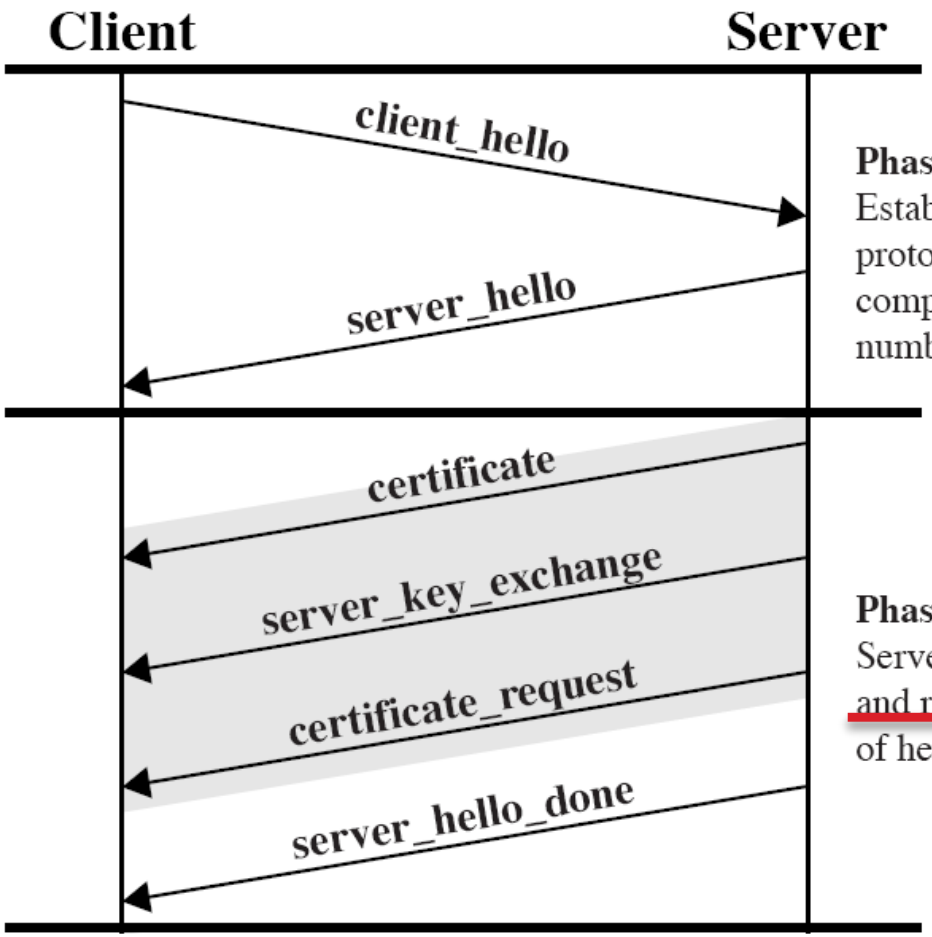


- Type (BasicResponse)
- ResponderID identifies the OCSP server via its DN issuer information, or its hashed public key (as shown here)
- CertId determines the cert that is being validated; using a hash algorithm (SHA-1) a hash of the issuer's DN, a hash of it's public key, and the certificates serial number are provided
- certStatus (good)
- Validity period of OCSP response
- The entire message us digitally signed by the OCSP responder
- That's the signature

Mutual Authentication (Server-Side and Client-Side Authentication)

- Consider a scenario where both the server and the client need to mutually authenticate, e.g.
  - Server-to-server data communication
  - IoT sensor network communication
  - Online Revenue services where client (browser) needs to be authenticated too
- Mutual authentication is just an extension of the process as seen before with the difference that the client sends it certificate (chain) to the server too for verification

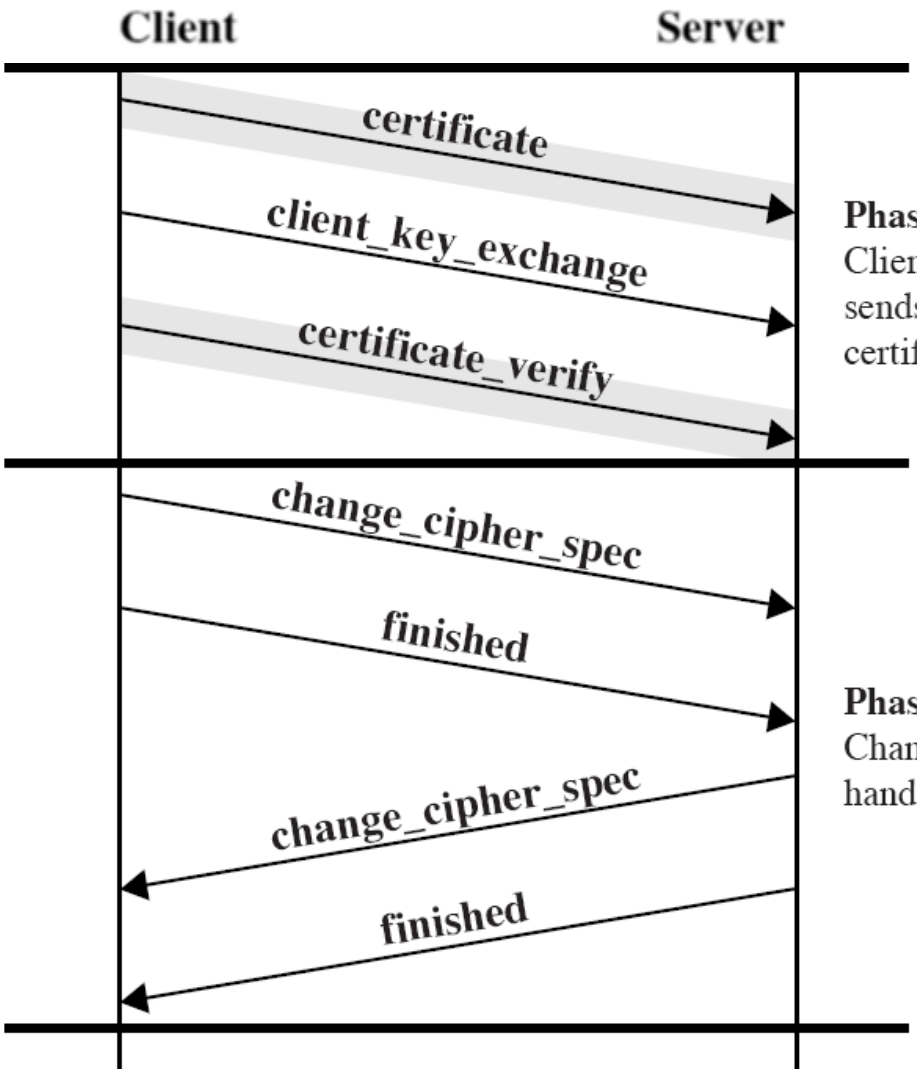
Mutual Authentication I



Phase 1
Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers.

Phase 2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.

Mutual Authentication II



Phase 3
Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.

Phase 4
Change cipher suite and finish handshake protocol.

In Summary

- TLS is the de-facto security protocol used in Internet data communication
- It went through a series of versions, and today only TLS 1.2 and TLS 1.3 are used
- TLS combines a lot of the foundation topics we've discussed in recent weeks
- Practically it is very hard to break TLS security, as the protocol went through various improvements over the years
- Therefore, from an attacker perspective, it is more promising to compromise a system by attacking either the client, the server, or the end user directly