# TLS (Transport Layer Security)


**Antonio Lioy**

**< antonio.lioy @ polito.it >**

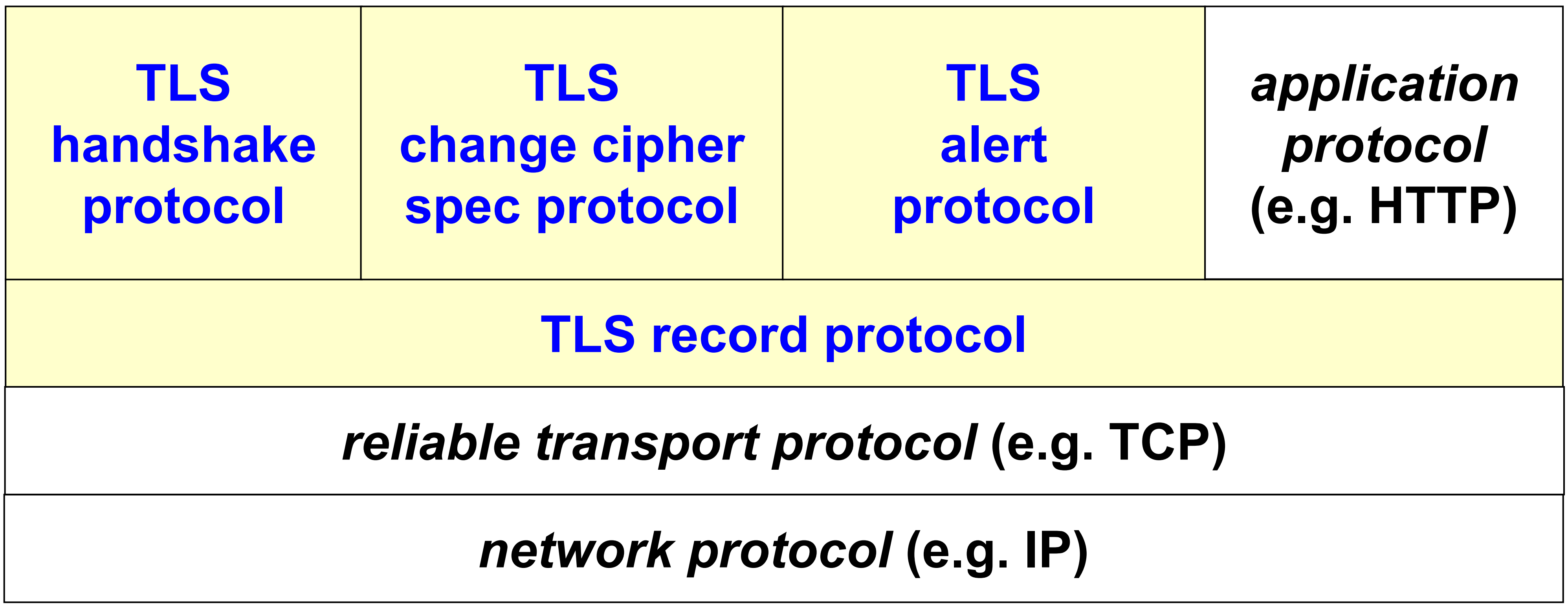
***Politecnico di Torino***

***Dip. Automatica e Informatica***

# In the origin it was SSL (Secure Socket Layer)

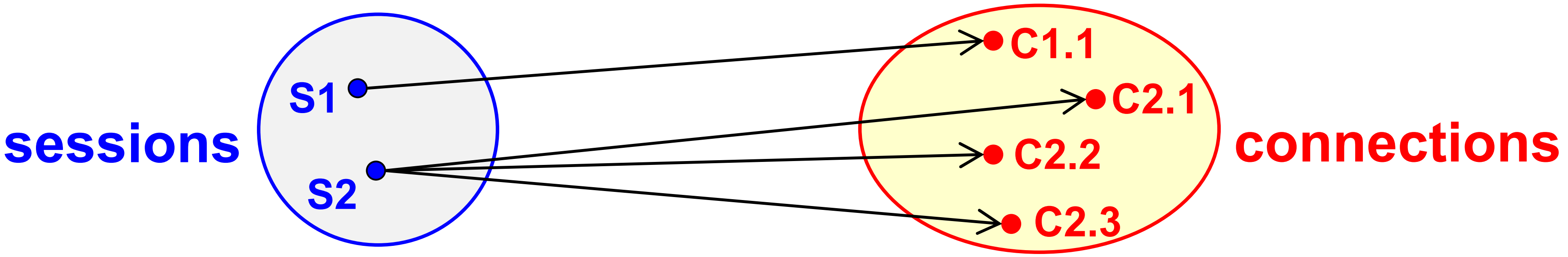
- **proposed by Netscape Communications in 1995**
- **secure transport channel (session level):**
  - peer authentication (server, server+client)
    - asymmetric challenge-response (implicit, explicit)
  - message confidentiality
    - symmetric encryption
  - message authentication and integrity
    - MAC computation
  - protection against replay, filtering, and reordering attacks
    - implicit record number (used in MAC computation!) plus layering on TCP

# TLS architecture



# TLS sessions and connections

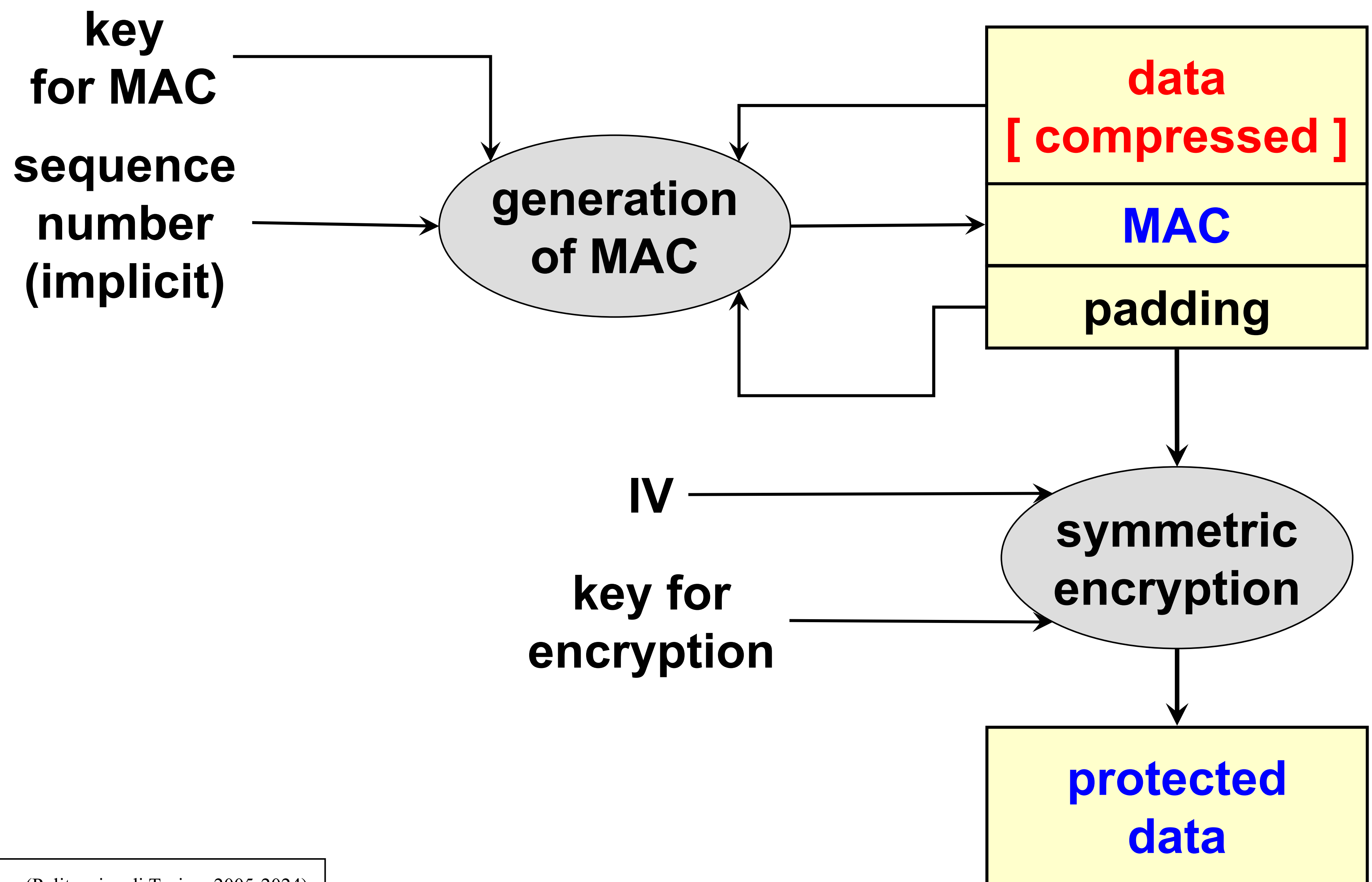
- **TLS session**
  - a logical association between client and server
  - created by the Handshake Protocol
  - defines a set of cryptographic parameters
  - is shared by one or more TLS connections (1:N)
- **TLS connection**
  - a transient TLS channel between client and server
  - associated to one specific TLS session (1:1)



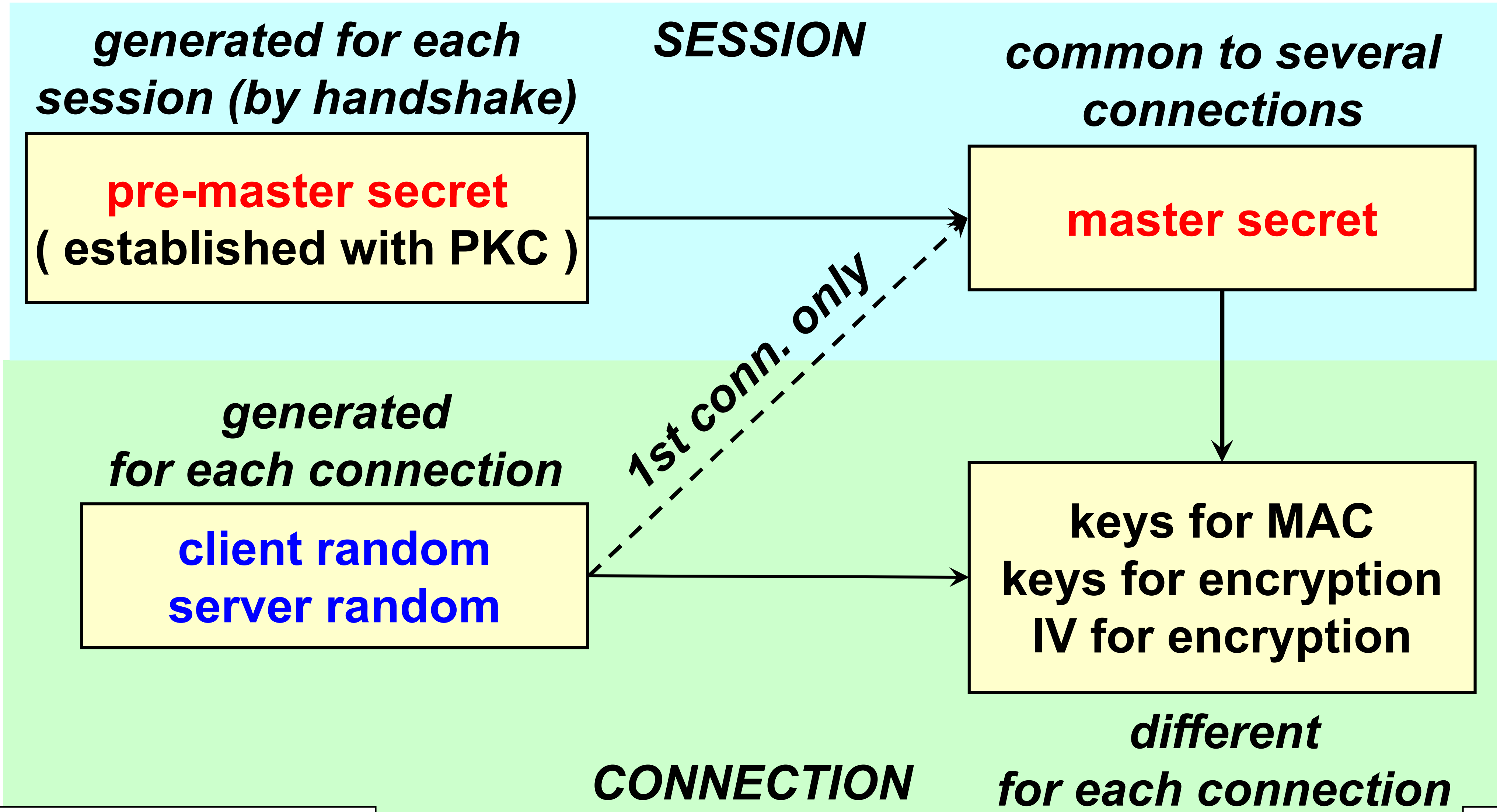
# TLS handshake protocol

- agree on a set of algorithms for confidentiality and integrity
- exchange random numbers between the client and the server to be used for the subsequent generation of the keys
- establish a symmetric key by means of public key operations (RSA, DH, ...)
- negotiate the session-id
- exchange the necessary certificates

# Data protection (authenticate-then-encrypt)



# Relationship among keys and sessions (between a server and the same client)

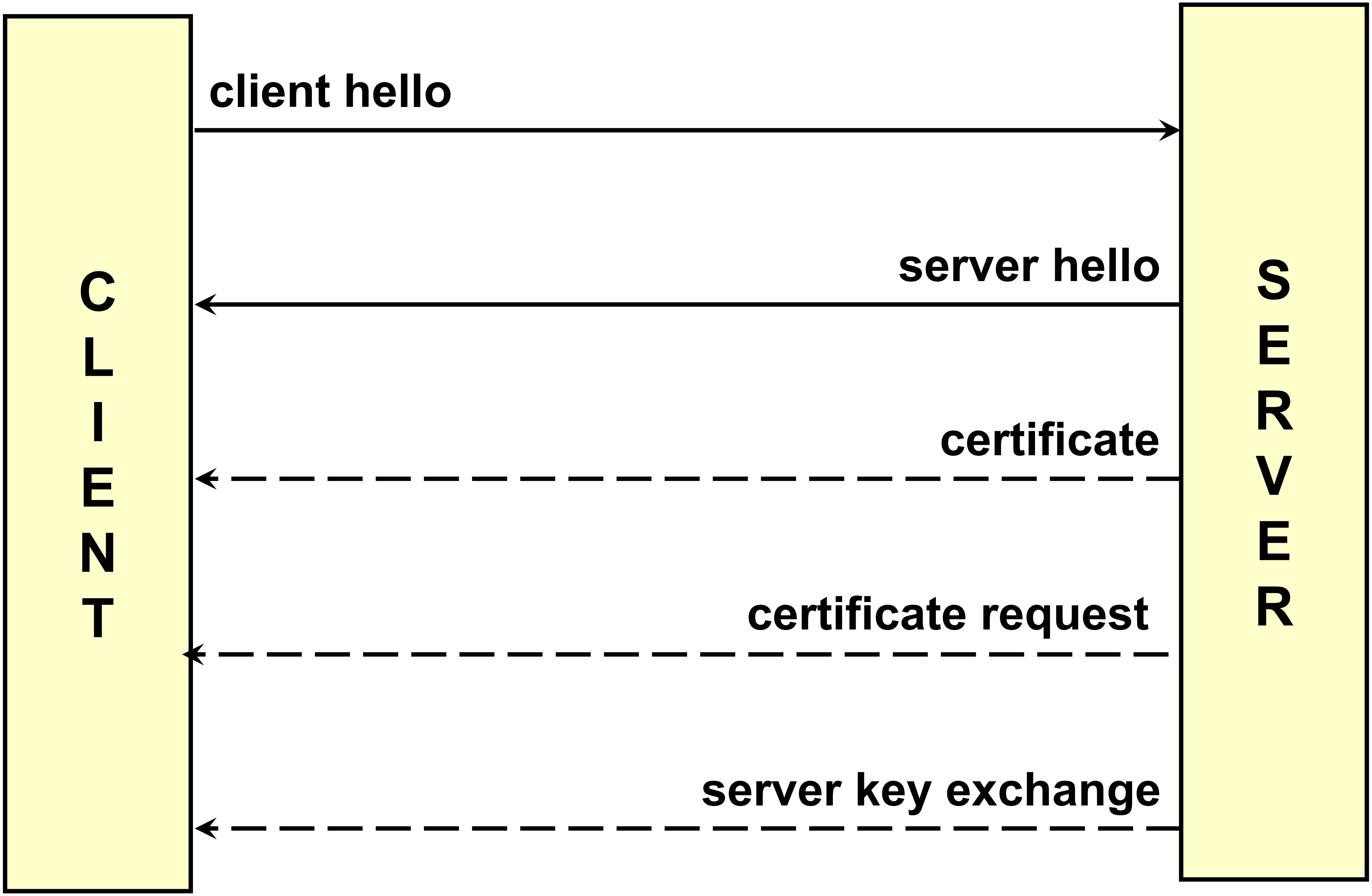


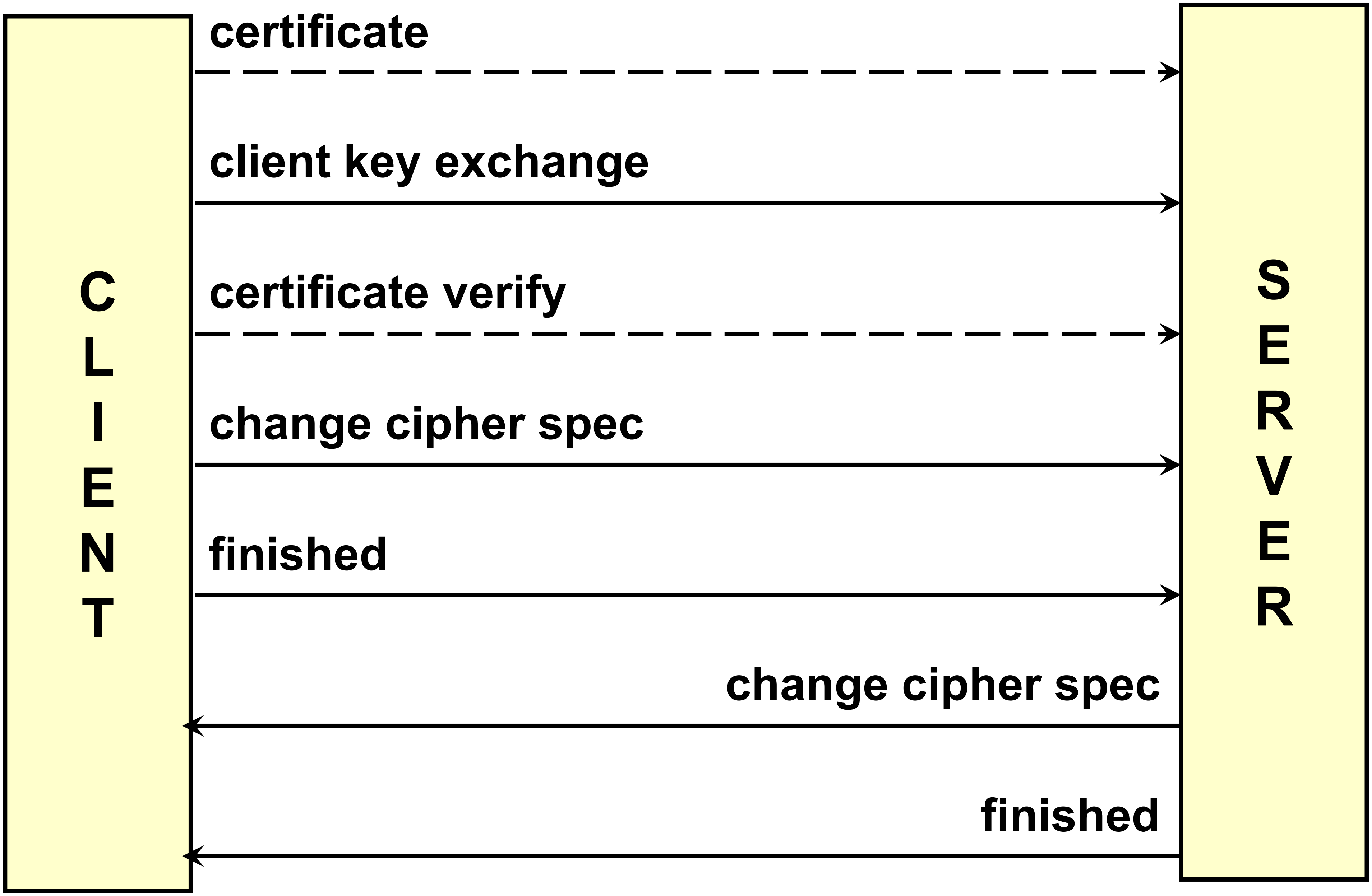
# Perfect forward secrecy (PFS)

- **if a server has a certificate valid for both signature and encryption**
- **... then it can be used both for authentication (via a signature) and key exchange (asymmetric encryption of the session key)**
- **... but if**
  - an attacker copies all the encrypted traffic
  - and later discovers the (long term) private key
- **... then the attacker can decrypt all the traffic, past, present, and future**
- **perfect forward secrecy:**
  - the compromise of the SK used for KE compromises only the current (and eventually future) traffic but not the past one

# “Ephemeral ” mechanisms

- **one-time asymmetric key (used for KE) generated on the fly:**
  - for authenticity it must be signed (but cannot have an associated X.509 certificate because the CA process is slow and often not on-line)
  - DH suitable, RSA slow
    - compromise for RSA = re-use N times
- **in this case the server’s private key is used only for signing**
- **... so we obtain perfect forward secrecy:**
  - if the (temporary or short-lived) private key is compromised then the attacker can decrypt only the related traffic
  - compromise of the long-term private key is an issue for authentication but not for confidentiality
- **examples: ECDHE**


CLIENT
client hello
server hello
certificate
certificate request
server key exchange
SERVER


CLIENT
certificate
client key exchange
certificate verify
change cipher spec
finished
change cipher spec
finished
SERVER

# Client hello

- **SSL version preferred by the client (highest supported)**
  - 2=SSL-2, 3.0=SSL-3, 3.1=TLS-1.0, ...
- **28 pseudo-random bytes (Client Random)**
- **a session identifier (session-id)**
  - 0 to start a new session
  - different from 0 to ask to resume a previous session
- **list of “cipher suite” (=algorithms for encryption + key exchange + integrity) supported by the client**
- **list of compression methods supported by the client**

# Server hello

- **SSL version chosen by the server**
- **28 pseudo-random bytes (Server Random)**
- **a session identifier (session-id)**
  - new session-id if session-id=0 in the client-hello or reject the session-id proposed by the client
  - session-id proposed by the client if the server accepts to resume the session
- **“cipher suite” chosen by the server**
  - should be the strongest one in common with the client
- **compression method chosen by the server**

# Cipher suite

- **key exchange algorithm**
- **symmetric encryption algorithm**
- **hash algorithm (for MAC)**
- **examples:**
  - SSL_NULL_WITH_NULL_NULL
  - SSL_RSA_WITH_NULL_SHA
  - SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
  - SSL_RSA_WITH_3DES_EDE_CBC_SHA
- **complete list maintained by IANA:**

**https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-parameters-4**

# Certificate (server)

- **certificate for server authentication**
  - the subject / subjectAltName must be the same as the identity of the server (DNS name, IP address, ...)
  - the whole chain (up to a trusted root) MUST be sent (exception: the chain MUST NOT include the root CA)
- **can be used only for signing or (in addition) also for encryption**
  - described in the field keyUsage
  - if it is only for signing then it is required also the phase for server-key exchange (i.e. the phase to exchange the ephemeral key)

# Certificate request

- **used for client authentication**
- **specifies also the list of CAs considered trusted by the server**
  - the browsers show to the users (for a connection) only the certificates issued by trusted CAs

# Server key exchange

- **carries the server public key for key exchange**
- **needed only in the following cases:**
  - the RSA server certificate is usable only for signature
  - anonymous or ephemeral DH is used to establish the pre-master secret
  - there are export problems that force the use of ephemeral RSA/DH keys
  - Fortezza ephemeral keys
- **explictly signed by the server**
- **important: this is the *only* message esplicitly signed by the server**

# Certificate (client)

- carries the certificate for client authentication
- the certificate must have been issued from one CA in the trusted CA list in the Certificate Request message

# Client key exchange

- **the client generates material for symmetric keys derivation and sends it to the server**
- **various ways**
  - pre-master secret encrypted with the server RSA public key (ephemeral or from its X.509 certificate)
  - client's public part of DH
  - client's Fortezza parameters

# Certificate verify

- explicit test signature done by the client
- hash computed over all the handshake messages before this one and encrypted with the client private key
- used only with client authentication (to identify and reject fake clients)

# Change cipher spec

- trigger the change of the algorithms to be used for message protection
- allows to pass from the previous unprotected messages to the protection of the next messages with algorithms and keys just negotiated
- theoretically is a protocol on its own and not part of the handshake
- some analysis suggest that it could be eliminated

# Finished

- **first message protected with the negotiated algorithms**
- **very important to authenticate the whole handshake sequence:**
  - contains a MAC computed over all the previous handshake messages (but change cipher spec) using as a key the master secret
  - prevents rollback man-in-the-middle attacks (version downgrade or ciphersuite downgrade)
  - different for client and server

# TLS, no ephemeral key, no client auth

*CLIENT* *SERVER*

1. client hello (ciphersuite list, client random) →

← 2. server hello (ciphersuite, server random)

← 3. certificate (keyEncipherment)

4. client key exchange (key encrypted for server) →

5. change cipher spec (activate protection on client side) →

6. finished (MAC of all previous messages) →

← 7. change cipher spec (activate protection on server side)

← 8. finished (MAC of all previous messages)

# TLS, no ephemeral key, client auth

*CLIENT* *SERVER*

1. client hello (ciphersuite list, client random) →

← 2. server hello (ciphersuite, server random)

← 3. certificate (keyEncipherment)

← 4*. certificate request (cert type, list of trusted CAs)

5*. certificate (client cert chain) →

6. client key exchange (encrypted key) →

7*. certificate verify (signed hash of previous messages) →

8+9. change cipher spec + finished→

← 10+11. change cipher spec + finished

# TLS, ephemeral key, no client auth

*CLIENT* *SERVER*

1. client hello (ciphersuite list, client random) →

← 2. server hello (ciphersuite, server random)

← 3*. certificate (digitalSignature)

← 4*. server key exchange (signed RSA key or DH exponent)

5. client key exchange (encrypted key or client exponent) →

6. change cipher spec (activate protection on client side) →

7. finished (MAC of all previous messages) →

← 8. change cipher spec (activate protection on server side)

← 9. finished (MAC of all previous messages)

# TLS, data exchange and link teardown

***CLIENT*** ***SERVER***

**(… *handshake* …)**

**N. client data (MAC, [ encryption ] ) →**

**← M. server data (MAC, [ encryption ] )**

**… … …**

**LAST-1. alert close notify (MAC)→**

**← LAST. alert close notify (MAC)**

# TLS, resumed session

*CLIENT* *SERVER*

1*. client hello (…, session-id X) →

← 2*. server hello (…, session-id X)

3. change cipher spec (activate protection on client side) →

4. finished (MAC of all previous messages) →

← 5. change cipher spec (activate protection on server side)

← 6. finished (MAC of all previous messages)

# TLS setup time

- **first TCP handshake**
- **then TLS handshake**
  - various messages can fit in a single TCP segment
- **typically, this requires 1-RTT for TCP and 2-RTT for TLS:**
  - (C>S) SYN
    - (S>C) SYN-ACK
  - (C>S) ACK + ClientHello
    - (S>C) ServerHello + Certificate
  - (C>S) ClientKeyExchange + ChangeCipherSpec + Finished
    - (S>C) ChangeCipherSpec + Finished
  - after 180 ms client and server are ready to send protected data (assuming 30 ms delay one-way)

# TLS 1.0 (SSL 3.1)

- **Transport Layer Security**
- **standard IETF:**
  - TLS-1.0 = RFC-2246 (jan 1999)
- **TLS-1.0 = SSL-3.1 (99% coincident with SSL-3)**
- **emphasis on standard (i.e. not proprietary) digest and asymmetric crypto algorithms; mandatory:**
  - DH + DSA + 3DES
  - HMAC-SHA1
  - ... that is the ciphersuite TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA

# TLS 1.1

- **RFC-4346 (April 2006)**
- **to protect against CBC attacks**
  - the implicit IV is replaced with an explicit IV
  - padding errors now use the bad_record_mac alert message (rather than the decryption_failed one)
- **IANA registries defined for protocol parameters**
- **premature closes no longer cause a session to be non-resumable**
- **additional notes added for various new attacks**

# TLS 1.2

- RFC-5246 (August 2008)
- ciphersuite specifies also the PRF (pseudo-random function)
- extensive use of SHA-256 (e.g. in Finished, HMAC)
- support for authenticated encryption (AES in GCM or CCM mode)
- incorporates the protocol extensions (RFC-4366) and the AES ciphersuite (RFC-3268)
- default ciphersuite TLS_RSA_WITH_AES_128_CBC_SHA
- IDEA and DES cipher suites deprecated

# TLS evolution (I)

- **ciphersuites / encryption:**
  - (RFC-3268) AES
  - (RFC-4492) ECC
  - (RFC-4132) Camellia
  - (RFC-4162) SEED
  - (RFC-6209) ARIA
- **ciphersuites / authentication:**
  - (RFC-2712) Kerberos
  - (RFC-4279) pre-shared key (secret, DH, RSA)
  - (RFC-5054) SRP (Secure Remote Password)
  - (RFC-6091) OpenPGP

# TLS evolution (II)

- **compression:**
  - (RFC-3749) compression methods + Deflate
  - (RFC-3943) protocol compression using LZS
- **other:**
  - (RFC-4366) extensions (specific and generic)
  - (RFC-4681) user mapping extensions
  - (RFC-5746) renegotiation indication extensions
  - (RFC-5878) authorization extensions
  - (RFC-6176) prohibiting SSL-2
  - (RFC-4507) session resumption w/o server state
  - (RFC-4680) handshake with supplemental data

# Heartbleed

- **RFC6520 = TLS/DTLS heartbeat extension**
  - to keep a connection alive without the need to constantly renegotiate the SSL session (DTLS!)
  - also useful in PMTU discovery
- **CVE-2014-0160 = openssl bug (buffer over-read)**
  - TLS server sends back more data (up to 64kB) than in the heartbeat request
  - see http://xkcd.com/1354/
- **attacker can get sensitive data stored in RAM, such as user+pwd and/or server private key (if not using HSM)**

# Bleichenbacher attack (and ROBOT)

- **(1998) Daniel Bleichenbacher's “million-message attack,”**
  - a vulnerability in the way RSA encryption was done
  - attacker can perform an RSA private key operation with a server’s private key by sending a million or so well-crafted messages and looking for differences in the error codes returned
  - attack refined over the years and in some cases only requires thousands of messages
    - feasible from a laptop (!)
- **(2017) ROBOT = variant of Bleichenbacher’s attack**
  - affected major websites (e.g. facebook.com)

# Other attacks against SSL/TLS (1)

- **CRIME (2012) an attacker able to**
  - (a) inject chosen plaintext in the user requests
    (b) measure the size of the encrypted traffic
  - may recover specific plaintext parts exploiting information leaked from the compression
- **BREACH (2013) deduces a secret within HTTP responses provided by a server that**
  - (a) uses HTTP compression (b) inserts user input into HTTP responses (c) contains a secret (e.g. a CSRF token) in the HTTP responses

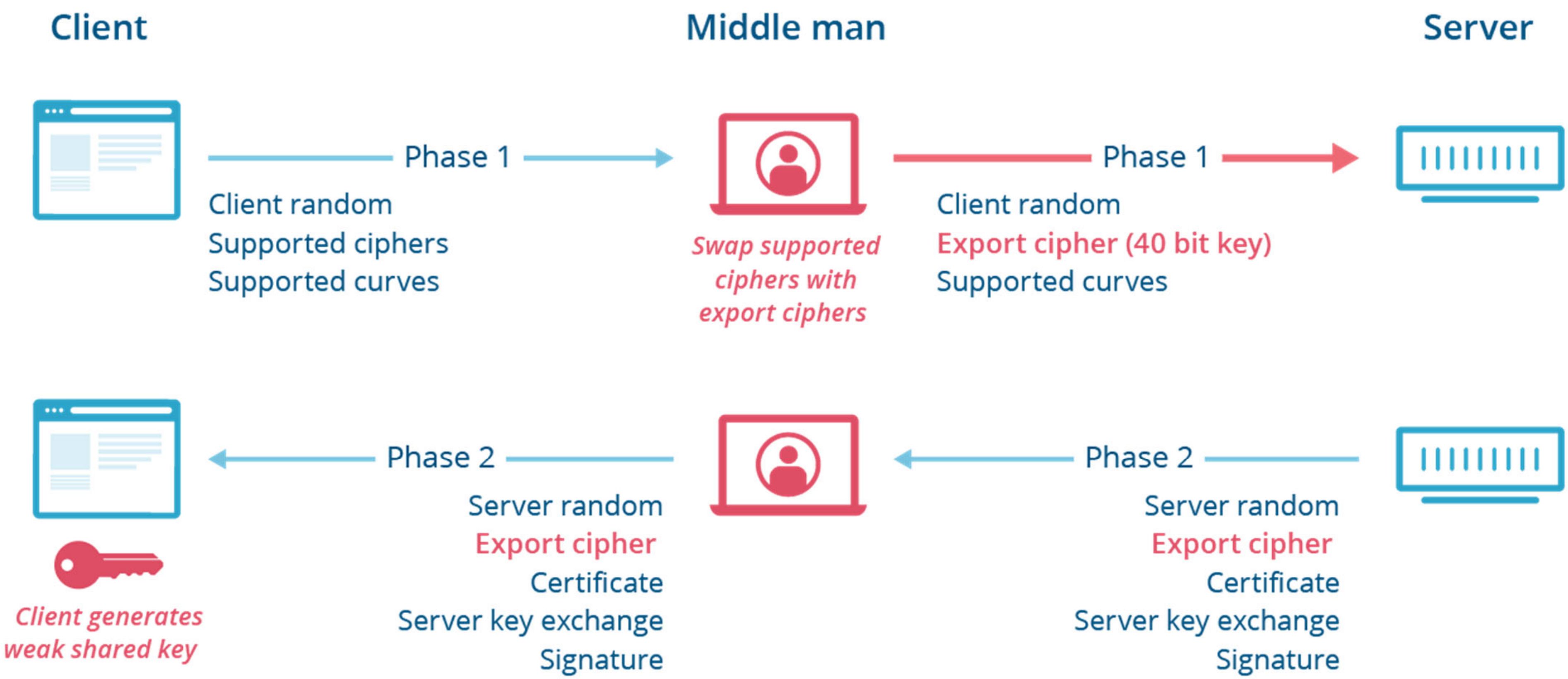
# Other attacks against SSL/TLS (2)

- **BEAST (Browser Exploit Against SSL/TLS) 2011**
  - SSL channel using CBC with IV concatenation
  - a MITM may decrypt HTTP headers with a blockwise-adaptive chosen-plaintext attack
  - the attacker may decrypt HTTPS requests and steal information such as session cookies
- **POODLE (Padding Oracle On Downgraded Legacy Encryption) is a MITM attack that exploits SSL-3 fallback to decrypt data**
  - POODLE, dec-2014 variant, exploits CBC errors in TLS-1.0-1.2 (so it works even if SSL-3 is disabled)

# Other attacks against SSL/TLS (3)

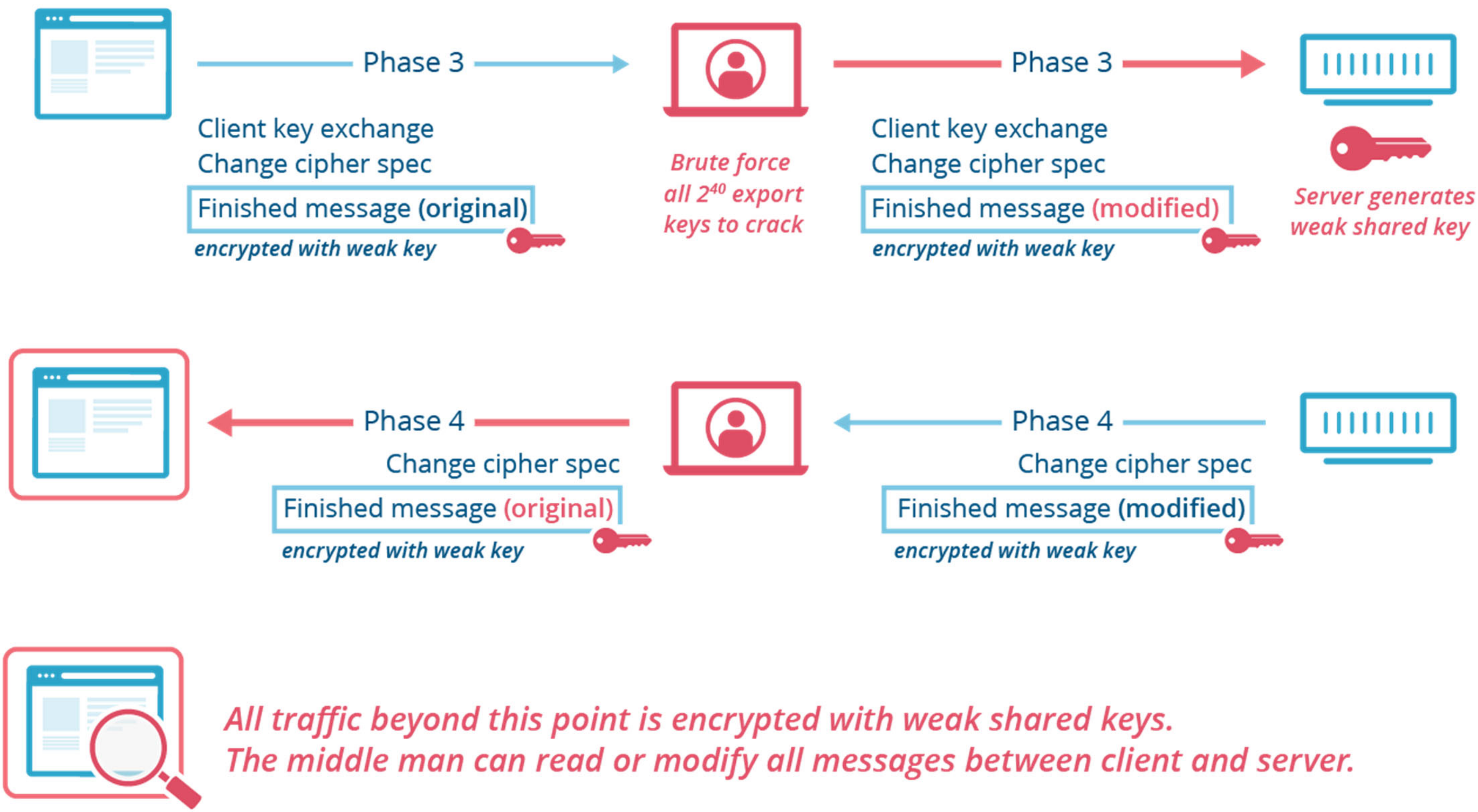
- **FREAK (Factoring RSA Export Keys) 2015**
  - downgrade to export-level RSA keys (512-bit)
    - then factorization and channel decryption
  - or downgrade to export-level symmetric key (40-bit)
    - then brute-force crack the key
- **SSL-3 has been disabled on most browsers (e.g. since FX34) or may be disabled**
  - FX: security.tls.version.min = 1 (i.e. TLS-1.0, alias SSL-3.1)
- **… but SSL-3 needed by IE6 (last version available for Windows-XP) and TLS cannot be disabled**
- **test browser/server with Qualys SSL labs tests**

# Freak attack (downgrade symmetric key)




**(picture source: Cloudflare)**

# Freak attack (downgrade symmetric key)




**(picture source: Cloudflare)**

# Other attacks against SSL/TLS (3)

- **implementation errors:**
  - Heartbleed, BERserk, goto fail;
  - Lucky13 (feb-13) timing side-channel attack, it's a variant of Vaudenay's attack that works even if that one was fixed
  - Lucky Microseconds (nov-2015) variant of Lucky13 to attack s2n (Google TLS library claiming to be more secure and resistant to Lucky13)
- **protocol design errors:**
  - (theoretical) SLOTH, CurveSwap
  - (require high resources) WeakDH, LogJam, FREAK, SWEET32
  - (practical and dangerous) POODLE, ROBOT

# ALPN extension (Application–Layer Protocol Negotiation)

- **RFC-7301**
- **application protocol negotiation (for TLS-then-proto) to speed up the connection creation, avoiding additional round-trips for application negotiation**
  - (ClientHello) ALPN=true + list of supported app. protocols
  - (ServerHello) ALPN=true + selected app. protocol
- **important to negotiate HTTP/2 and QUIC**
  - Chrome & Firefox support HTTP/2 only over TLS
- **useful also for those servers that use different certificates for the different application protocols**
- **some possible values:**
  - http/1.0, http/1.1, h2 (HTTP/2 over TLS), h2c (HTTP/2 over TCP)

# TLS False Start

- **RFC-7918**
- **the client can send application data together with the ChangeCipherSpec and Finished messages, in a single segment, without waiting for the corresponding server messages**
- **this reduces latency to 1-RTT**
- **it should work without changes but there are caveats:**
  - Chrome and FX require ALPN + forward secrecy
  - Safari requires forward secrecy
- **to enable TLS False Start for all browsers the server should:**
  - advertise supported protocols (via ALPN, e.g. "h2, http/1.1")
  - be configured to prefer cipher suites with forward secrecy.

# The TLS downgrade problem (I)

- **client sends (in ClientHello) the highest supported version**
- **server notifies (in ServerHello) the version to be used (highest in common with client)**
- **normal version negotiation:**
  - agreement on TLS-1.2
    - (C > S) 3,3
    - (S > C) 3,3
  - fallback to TLS-1.1 (e.g. no TLS-1.2 at server)
    - (C > S) 3,3
    - (S > C) 3,2

# The TLS downgrade problem (II)

- **(insecure) downgrade:**
  - some servers do not send the correct response, rather they close the connection …
  - then the client has no choice but to try again with a lower protocol version
- **downgrade attack:**
  - attacker sends fake server response, to force repeated downgrade until reaching a vulnerable version (e.g. SSL-3) …
  - then execute a suitable attack (e.g. Poodle)
- **not always an attack (e.g. connection with the server closed due to a network problem)**

# TLS Fallback
# Signalling Cipher Suite Value (SCSV)

- **RFC-7507**
- **to prevent protocol downgrade attacks**
- **new (dummy) ciphersuite TLS_FALLBACK_SCSV**
  - SHOULD be sent by the client when opening a downgraded connection (as last in ciphersuite list)
- **new fatal Alert value "inappropriate_fallback"**
  - MUST be sent by the server when receiving TLS_FALLBACK_SCSV and a version lower than the highest one supported
  - then the channel is closed and the client should retry with its highest protocol version

# SCSV - notes

- **many servers do not yet support SCSV**
- **… but most servers have fixed their bad behaviour when the client requests a version higher than the supported one**
- **… so browsers can now disable insecure downgrade**
  - Firefox (from 2015) and Chrome (from 2016)

# TLS session tickets

- **RFC-5077**
- **session resumption requires a Session-ID cache at server**
  - … which may become very large for high traffic servers
- **TLS session ticket is an extension allowing the server to send the session data to the client**
  - encrypted with a server secret key
  - returned by the client when resuming a session
  - in practice, it moves the session cache to the client
- **issues:**
  - needs support at the browser (it's an extension!)
  - in a load balancing environment, it requires key sharing among the various end-points (and periodic key update!)

# TLS and virtual servers: the problem

- **virtual server (frequent case with web hosting)**
  - different logical names associated to the same IP address
  - e.g. home.myweb.it=10.1.2.3, food.myweb.it=10.1.2.3
- **easy since HTTP/1.1**
  - the client uses the Host header to specify the server it wants to connect to
- **… but difficult in HTTPS**
  - because TLS is activated before HTTP
  - which certificate should be provided? (must contain the server's name)

# TLS and virtual servers: solutions

- **collective (wildcard) certificate**
  - e.g. CN=*.myweb.it
  - private key shared by all servers
  - different treatment by different browsers
- **certificate with a list of servers in subjectAltName**
  - private key shared by all servers
  - need to re-issue the certificate at any addition or cancellation of a server
- **use the SNI (Server Name Indication) extension**
  - in ClientHello (permitted by RFC-4366)
  - limited support by browsers and servers

# TLS-1.3

- **design targets:**
  - reducing handshake latency
  - encrypting more of the handshake (for security and privacy)
  - improving resiliency to cross-protocol attacks
  - removing legacy features
- **RFC-8446 (August 2018)**

# TLS-1.3: key exchange

- **remove static RSA and DH key exchange**
  - it's not forward secrecy
    - problem with Heartbleed attack
  - difficult to implement correctly
    - problem with the Bleichenbacher attack (and ROBOT)
- **use DHE … but do not permit arbitrary parameters**
  - (2015) LogJam and weakDH trick servers to use small numbers for DH (just 512-bit)
  - (2016) Sanso finds openSSL generates DH values without the required mathematical properties

- **TLS-1.3 uses only DHE with a few predefined groups**

# TLS-1.3: message protection

- **previous pitfalls:**
  - use CBC mode and authenticate-then-encrypt
    - culprit for Lucky13, Lucky Microseconds, POODLE
  - use RC4
    - (2013) plaintext can be recovered due to measurable biases
  - use of compression
    - culprit for CRIME attack
- **TLS-1.3 uses only safe cryptography:**
  - does not use CBC and authenticate-then-encrypt
    - only AEAD modes are permitted
  - dropped RC4, 3DES, Camellia, MD5, and SHA-1
    - only modern crypto algorithms and no compression at all

# TLS-1.3: digital signature

- **previous pitfalls:**
  - RSA signature of ephemeral keys
    - done wrongly with the PKCS#1v1.5 schema
  - handshake authenticated with a MAC, not a signature
    - makes possible attacks such as FREAK

- **TLS-1.3 uses:**
  - RSA signature with the modern secure RSA-PSS schema
  - the whole handshake is signed, not just the ephemeral keys
  - modern signature schemes

# TLS-1.3: ciphersuites

- **avoid the complexity of previous versions**
  - huge list, combinatorically increasing for every new algorithm
- **TLS-1.3 specifies only orthogonal elements:**
  - cipher (&mode) + HKDF hash
  - no certificate type (RSA, ECDSA, or EdDSA)
  - no key exchange (DHE/ECDHE, PSK, or PSK+DHE/ECDHE)
- **only 5 ciphersuites:**
  - TLS_AES_128_GCM_SHA256
  - TLS_AES_256_GCM_SHA384
  - TLS_CHACHA20_POLY1305_SHA256
  - TLS_AES_128_CCM_SHA256
  - TLS_AES_128_CCM_8_SHA256 (deprecated)

# TLS-1.3: EdDSA

- **Edwards-curve Digital Signature Algorithm**
  - DSA requires a PRNG that can leak the private key if the underlying generation algorithm is broken or made predictable
  - EdDSA does not need a PRNG
  - EdDSA picks a nonce based on a hash of the private key and the message, which means after the private key is generated there's no more need for random number generators
- **faster signature and verification wrt ECDSA**
  - simplified point addition and doubling
- **N-bit private and public keys, 2N-bit signatures**

# EdDSA implementations

- **Ed25519 uses SHA-512 (SHA-2) and Curve25519**
  - 256-bit key, 512-bit signature, 128-bit security
- **Ed448 uses SHAKE256 (SHA-3) and Curve448**
  - 456-bit key, 912-bit signature, 224-bit security
- **Curve25519 is the most widely used**
  - elliptic curve on the field $2^{255} - 19$
  - used also in X25519 (ECDH)
- **EdDSA has two standards, slightly different:**
  - RFC-8032 for general Internet applications, implementation details left to developers
  - FIPS 186-5 specifies stringent guidelines for secure key management, generation, and implementation practices

# TLS-1.3: other improvements

- **all handshake messages after the ServerHello are now encrypted**
- **the newly introduced EncryptedExtensions message allows various extensions previously sent in the clear in the ServerHello to also enjoy confidentiality protection**
- **the key derivation functions have been redesigned (to allow easier analysis by cryptographers thanks to their key separation properties) and HKDF is used as an underlying primitive**
- **the handshake state machine has been significantly restructured to be more consistent and to remove superfluous messages such as ChangeCipherSpec (except when needed for middlebox compatibility)**

# HKDF

- **HMAC-based extract-and-expand Key Derivation Function**
- **HKDF( salt, IKM, info, length ) =**
  **HKDF-Expand( HKDF-Extract ( salt, IKM ), info, length )**
- **the first stage takes the input keying material (IKM) and "extracts" from it a fixed-length pseudorandom key (PRK)**
- **… then the second stage "expands" this key into several additional pseudorandom keys (the output of the KDF)**
  - multiple outputs can be generated from a single IKM value by using different values "info" field
  - repeatedly call HMAC using the PRK as the key and the "info" as the message; the HMAC inputs are chained by prepending the previous hash block to the "info" field and appending an incrementing 8-bit counter

# HKDF usage in TLS 1.3 (I)

- **HKDF-Expand-Label( Secret, Label, Context, Length ) =**
  **HKDF-Expand( Secret, HkdfLabel, Length )**
- **where HkdfLabel is:**
  - struct {
    uint16 length = Length;
    opaque label<7..255> = "tls13 " + Label;
    opaque context<0..255> = Context;
    } HkdfLabel;

- **Derive-Secret( Secret, Label, Messages ) =**
  **HKDF-Expand-Label(**
  **Secret, Label, Transcript-Hash(Messages), Hash.length )**

# HKDF usage in TLS 1.3 (II)

- **finished_key = HKDF-Expand-Label( BaseKey, "finished", "", Hash.length )**
- **ticket_PSK = HKDF-Expand-Label(**
  **resumption_master_secret, "resumption",**
  **ticket_nonce, Hash.length )**

# TLS 1.3 key schedule (I)

- (for Extract) Salt from top, IKM from left
- (for Derive) Secret from left

```
         0
         |
         v
PSK -> HKDF-Extract = Early Secret
         |
         +---> Derive-Secret(., "ext binder" | "res binder", "")
         |                  = binder_key
         |
         +---> Derive-Secret(., "c e traffic", ClientHello)
         |                  = client_early_traffic_secret
         |
         +---> Derive-Secret(., "e exp master", ClientHello)
         |                  = early_exporter_master_secret
         v
      Derive-Secret(., "derived", "")
         |
```

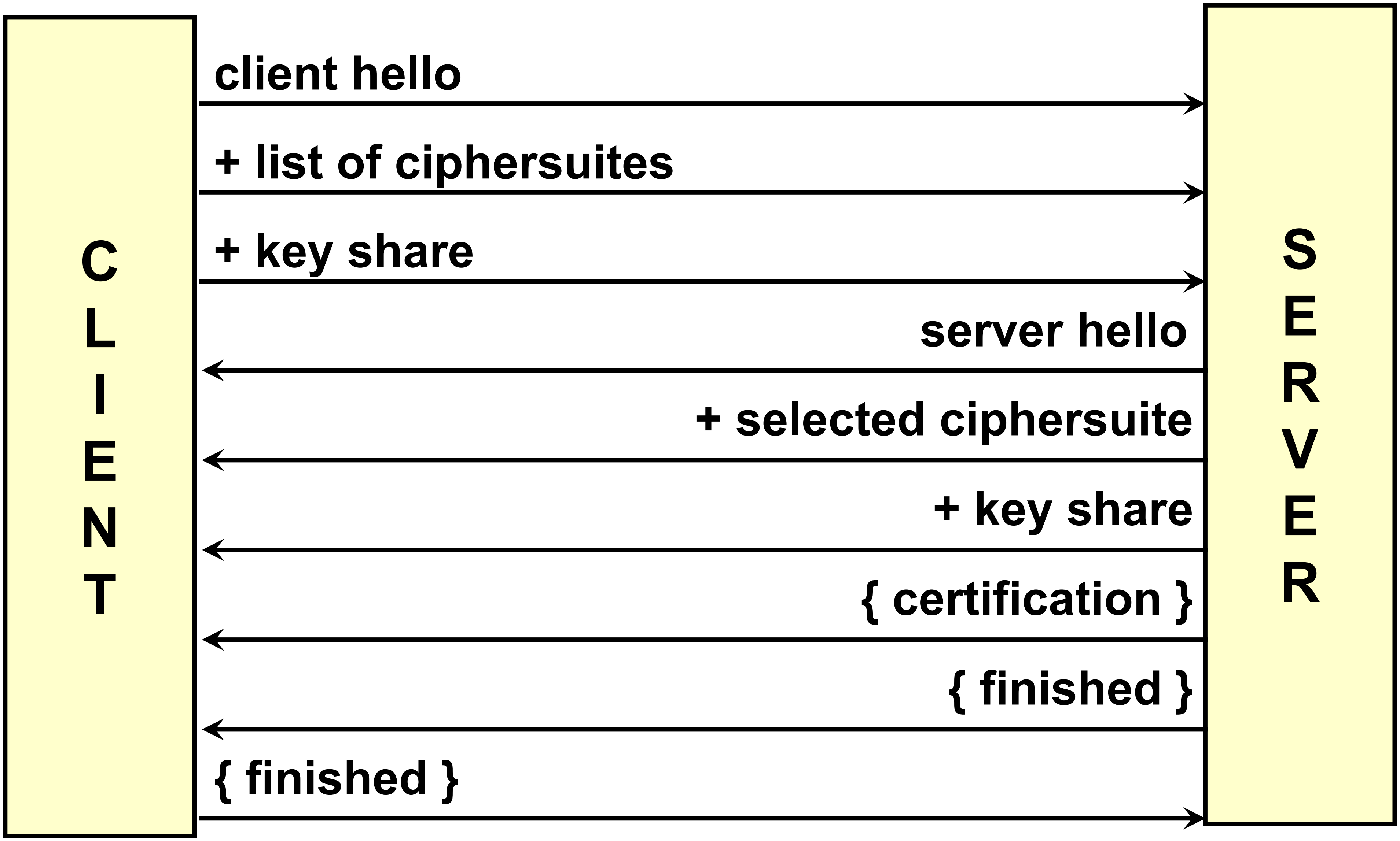
# TLS 1.3 key schedule (II)

```
         ...
          |
          v
(EC)DHE -> HKDF-Extract = Handshake Secret
          |
          +---> Derive-Secret(., "c hs traffic",
          |                   ClientHello...ServerHello)
          |                   = client_handshake_traffic_secret
          |
          +---> Derive-Secret(., "s hs traffic",
          |                   ClientHello...ServerHello)
          |                   = server_handshake_traffic_secret
          v
    Derive-Secret(., "derived", "")
          |
          v
```

# TLS 1.3 key schedule (III)

```
        ...
         |
         v
0 -> HKDF-Extract = Master Secret
         |
         +---> Derive-Secret(., "c ap traffic",
         |                ClientHello...server Finished)
         |                = client_application_traffic_secret_0
         |
         +---> Derive-Secret(., "s ap traffic",
         |                ClientHello...server Finished)
         |                = server_application_traffic_secret_0
         |
         +---> Derive-Secret(., "exp master",
         |                ClientHello...server Finished)
         |                = exporter_master_secret
         |
         +-----> Derive-Secret(., "res master",
                          ClientHello...client Finished)
                          = resumption_master_secret
```

# TLS-1.3 handshake



# TLS-1.3 handshake: notes

- **for backward compatibility, TLS-1.2 messages are also sent, and most TLS-1.3 features are in message extensions**
- **basically, it's a 1-RTT handshake**
- **… that can be reduced to 0-RTT upon resumption of a previous session (or by using true PSK, which is rare)**

- **notation:**
  - { data } = protected by keys derived from a [sender]_handshake_traffic_secret
  - [ data ] = protected by keys derived from a [sender]_application_traffic_secret_N

# TLS-1.3 handshake: client request

- **ClientHello**
  - client random
  - highest supported protocol version (note: TLS-1.2 !)
  - supported ciphersuites and compression methods
  - session-ID
- **contains extensions for key exchange**
  - key_share = client (EC)DHE share
  - signature_algorithms = list of supported algorithms
  - psk_key_exchange_modes = list of supported modes
  - pre_shared_key = list of PSKs offered

# TLS-1.3 handshake: server response

- **ServerHello (server random, selected version, ciphersuite)**
- **key exchange**
  - key_share = server (EC)DHE share
  - pre_shared_key = selected PSK
- **server parameters**
  - { EncryptedExtensions } = responses to non-crypto client ext
  - { CertificateRequest } = request for client certificate
- **server authentication**
  - { Certificate } = X.509 certificate (or raw key, RFC-7250)
  - { CertificateVerify } = signature over the entire handshake
  - { Finished } = MAC over the entire handshake
- **[ Application Data ]**

# TLS-1.3 handshake: client finish

- **client authentication**
  - { Certificate } = X.509 certificate (or raw key, RFC-7250)
  - { CertificateVerify } = signature over the entire handshake
  - { Finished } = MAC over the entire handshake
- **[ Application Data ]**

# TLS-1.3 / Pre-Shared Keys

- **PSK replaces session-ID and session ticket**
  - one or more PSKs agreed in a full handshake and re-used for other connections
- **PSK and (EC)DHE can be used together for forward secrecy**
  - PSK used for authentication, (EC)DHE for key agreement
- **PSK could also be OOB (e.g. generated from a passphrase)**
  - … but this is risky if have insufficient randomness (see RFC-4086) so that a brute-force attack could be possible
  - in general, OOB PSK is discouraged

# TLS-1.3 / 0-RTT connections

- **when using a PSK, client can send "early data" along with its first message (client request)**
- **early data protected with a specific key (client_early_traffic_secret)**
  - does not provide forward secrecy (because it depends only upon the PSK)
  - possible some kind of replay attack (partial mitigations are feasible but complex, especially in multi-instance servers)

# TLS-1.3 / Incorrect share

- client can send a list of (EC)DHE groups not supported by the server
- server will respond with HelloRetryRequest and the client must restart the handshake with other groups
- if also the new groups are unacceptable for the server, then the handshake will be aborted with an appropriate alert

# TLS and PKI

- **PKI needed for server (and optionally) client authentication**
  - unless PSK authentication is adopted
- **when a peer sends its certificate:**
  - the whole chain is needed (but the root CA - beware!)
  - validate the whole chain (not just the EE certificate)
  - revocation status needed at each step of the chain
- **to check revocation status:**
  - CRL can be used but big size and lengthy look-up
  - OCSP can be used but generates privacy problems (leaks client navigation history)
  - both require one additional network connection and add delay (e.g. for OCSP +300ms median, +1s average)

# TLS and certificate status

- **what if the URL for CRL or OCSP is unreachable?**
- **possible causes:**
  - server error
  - network error
  - access blocked by firewall (e.g. due to security policy or insecure channel – typical for OCSP)
- **possible approaches:**
  - hard fail – page is not displayed + security warning
  - soft-fail – page is displayed (assuming certificate is good)
  - both hard and soft fail require additional load time (wait for the connection to time out)

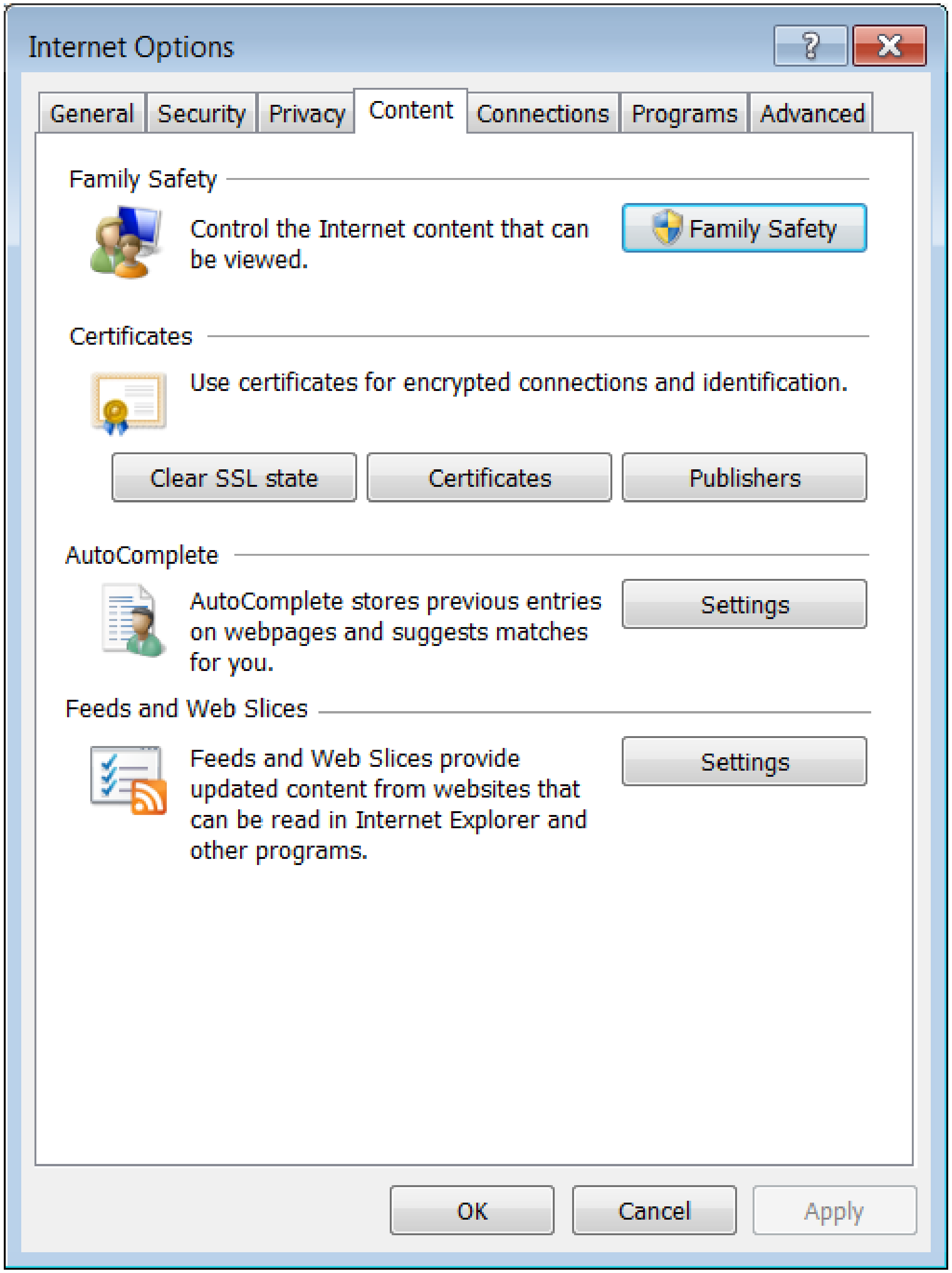
# TLS and certificate status: pushed CRL

- **revoked certificates are often originated by a compromised intermediate CA**
- **browser vendors thus decided to push (some) revoked certificates:**
  - Internet Explorer (with browser update – bad, can be blocked)
  - Firefox - oneCRL (part of the blocklisting process)
  - Chrome (also Edge, Opera) - CRLsets

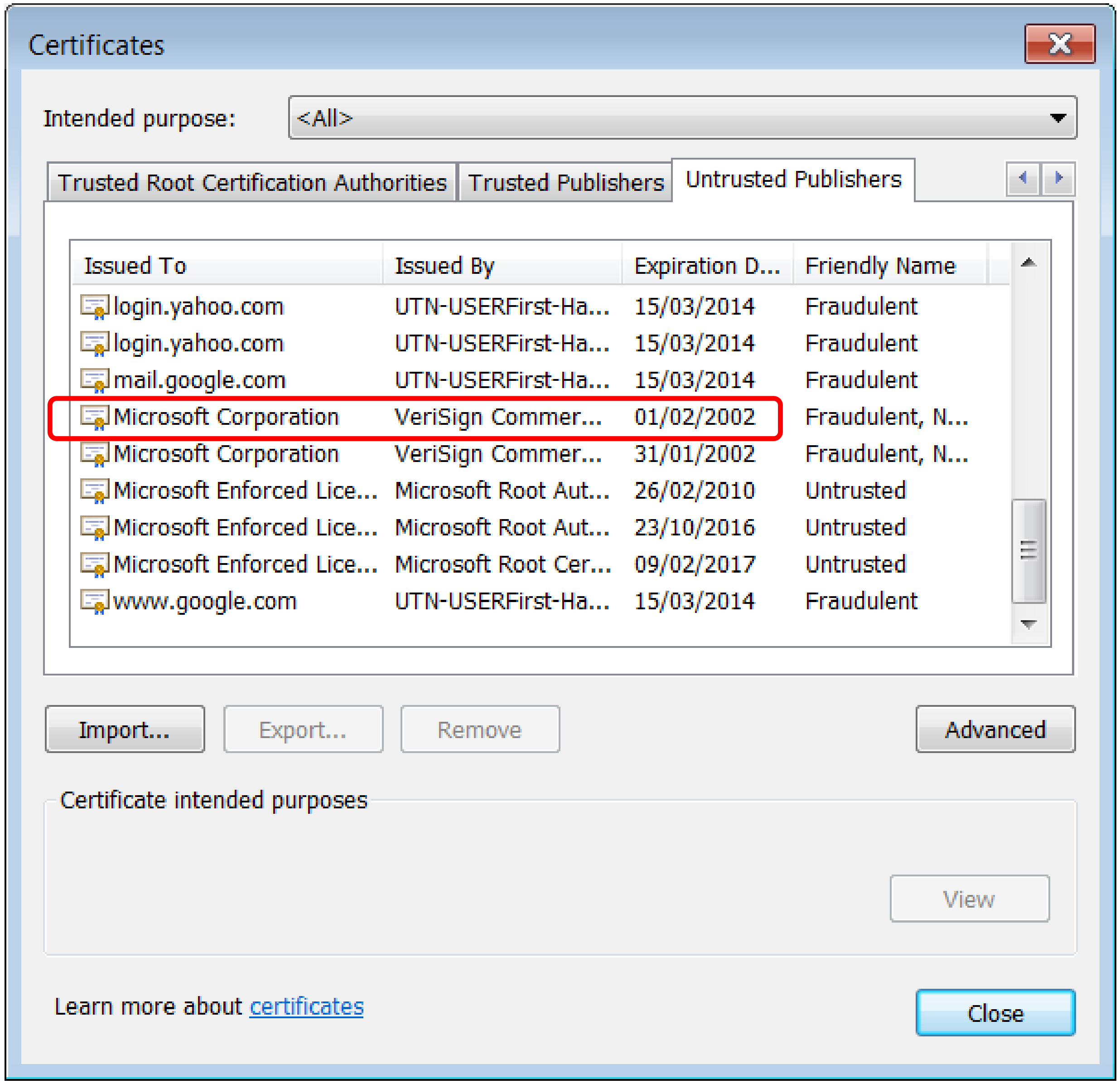
```
$ curl https://firefox.settings.services.mozilla.com/v1/buckets/
     security-state/collections/onecrl/records > crl.json
$ jq '[.data[] | {enabled,issuerName,serialNumber}
      | select(.enabled) | del(.enabled)]' crl.json > crl_short.json
```

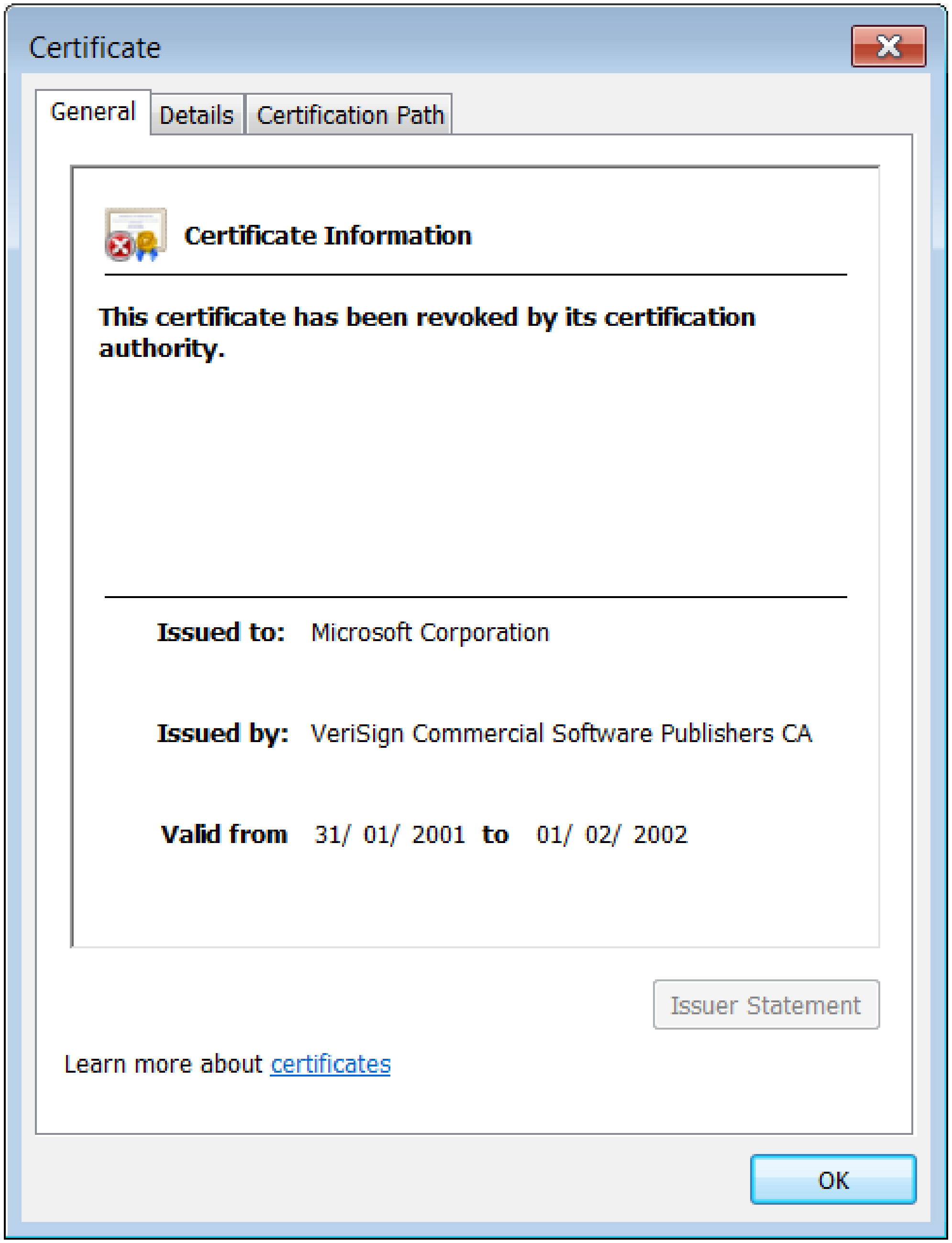
```
https://github.com/agl/crlset-tools
```

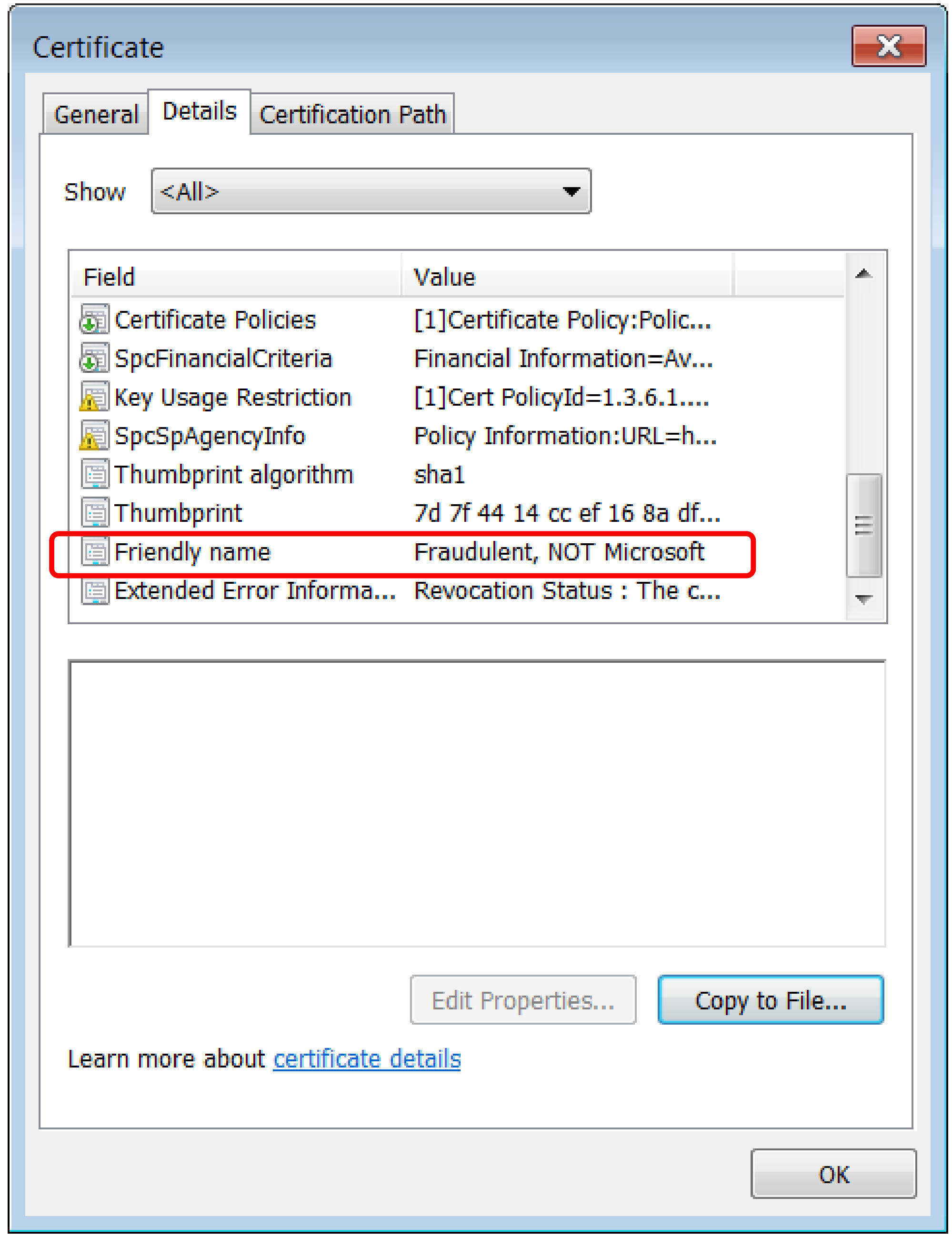
# Internet Explorer
# > Internet options

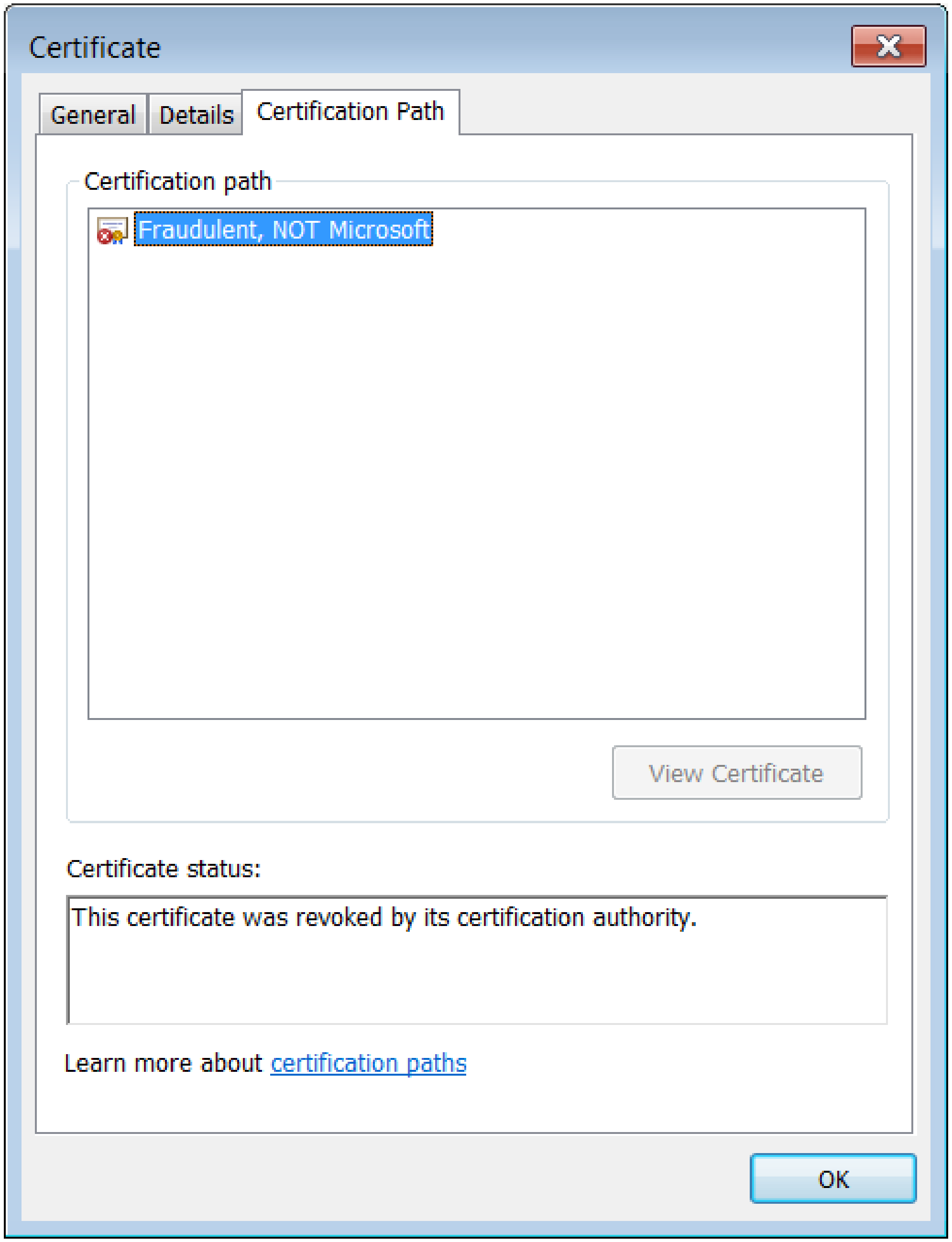


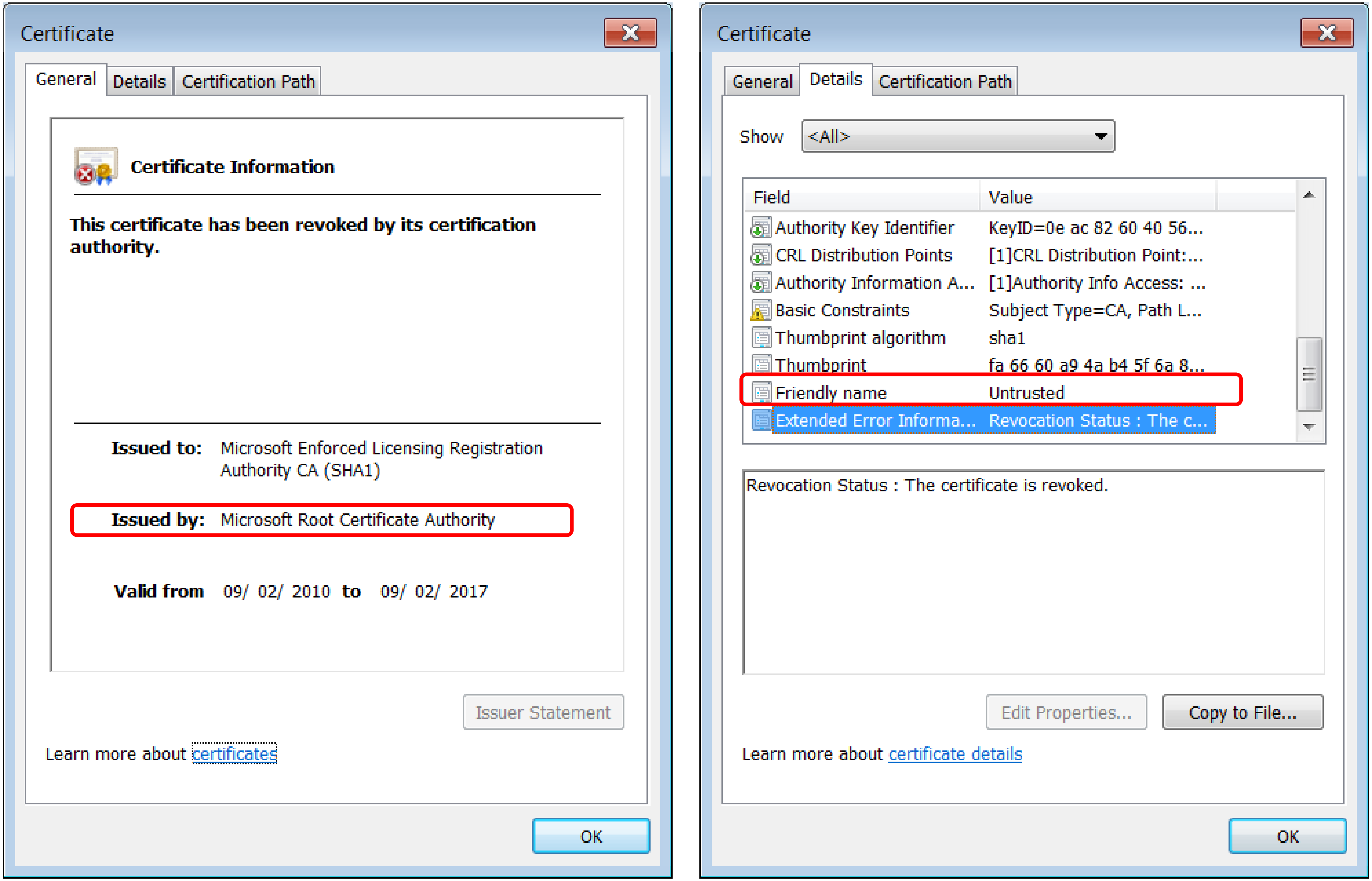
**Internet Explorer**
**> Internet options**
**> Certificates**
**- Untrusted (!) Publishers**

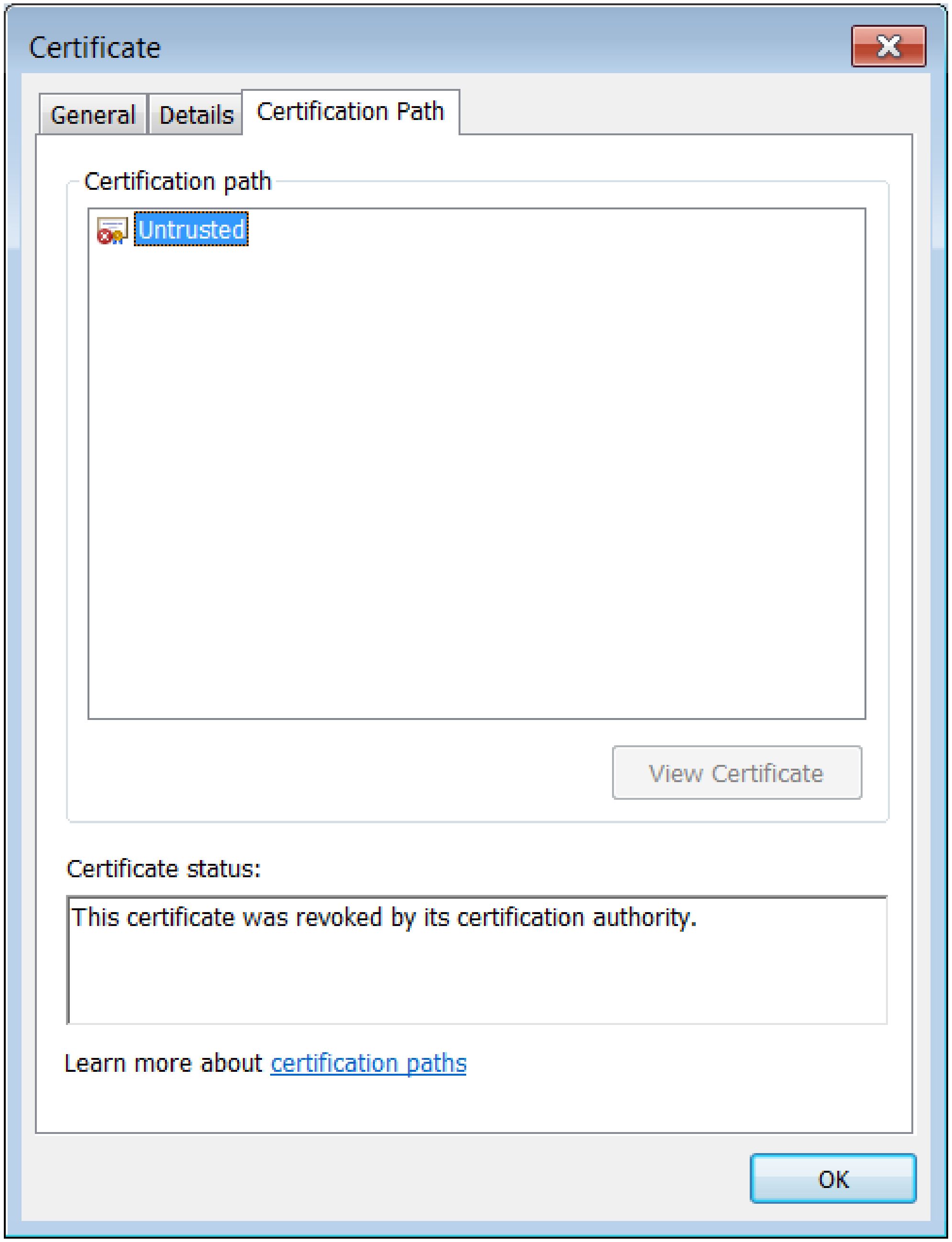



Certificate
General
Details
Certification Path
Certificate Information
This certificate has been revoked by its certification authority.
Issued to: Microsoft Corporation
Issued by: VeriSign Commercial Software Publishers CA
Valid from 31/ 01/ 2001 to 01/ 02/ 2002
Issuer Statement
Learn more about certificates
OK


Certificate
General
Details
Certification Path
Show <All>
Field
Value
Certificate Policies
[1]Certificate Policy:Polic...
SpcFinancialCriteria
Financial Information=Av...
Key Usage Restriction
[1]Cert PolicyId=1.3.6.1....
SpcSpAgencyInfo
Policy Information:URL=h...
Thumbprint algorithm
sha1
Thumbprint
7d 7f 44 14 cc ef 16 8a df...
Friendly name
Fraudulent, NOT Microsoft
Extended Error Informa...
Revocation Status : The c...
Edit Properties...
Copy to File...
Learn more about certificate details
OK


Certificate
General
Details
Certification Path
Certification path
Fraudulent, NOT Microsoft
View Certificate
Certificate status:
This certificate was revoked by its certification authority.
Learn more about certification paths
OK


Certificate
General
Details
Certification Path
Certificate Information
This certificate has been revoked by its certification authority.
Issued to: Microsoft Enforced Licensing Registration Authority CA (SHA1)
Issued by: Microsoft Root Certificate Authority
Valid from 09/ 02/ 2010 to 09/ 02/ 2017
Issuer Statement
Learn more about certificates
OK
Certificate
General
Details
Certification Path
Show <All>
Field
Value
Authority Key Identifier
KeyID=0e ac 82 60 40 56...
CRL Distribution Points
[1]CRL Distribution Point:...
Authority Information A...
[1]Authority Info Access: ...
Basic Constraints
Subject Type=CA, Path L...
Thumbprint algorithm
sha1
Thumbprint
fa 66 60 a9 4a b4 5f 6a 8...
Friendly name
Untrusted
Extended Error Informa...
Revocation Status : The c...
Revocation Status : The certificate is revoked.
Edit Properties...
Copy to File...
Learn more about certificate details
OK


Certificate
General
Details
Certification Path
Certification path
Untrusted
View Certificate
Certificate status:
This certificate was revoked by its certification authority.
Learn more about certification paths
OK

# MS certificate problems

- https://us-cert.cisa.gov/ncas/current-activity/2012/06/04/Unauthorized-Microsoft-Digital-Certificates
- https://docs.microsoft.com/en-us/security-updates/SecurityAdvisories/2012/2718704?redirectedfrom=MSDN

# TLS: OCSP stapling – concept

- **CRL and OCSP automatic download often disabled**
- **OCSP request from client creates a privacy problem**
- **pushed CRL contains only some revoked certificates**
- **browser behaviour is greatly variable in this area**
- **solution:**
  - OCSP stapling
  - i.e. have the server autonomously obtain the OCSP answer and pass it along with its certificate to the client

# TLS: OCSP Stapling – implementation (I)

- **OCSP Stapling is a TLS extension**
  - to be specified in the TLS handshake
- **v1 in RFC-6066 (extension "status_request"), v2 in RFC 6961 (extension "status_request_v2")**
  - with value CertificateStatusRequest
- **how it works:**
  - the TLS server pre-fetches the OCSP responses
  - ... and provides them to the client in the handshake, as part of the server's certificate message
  - the OCSP responses are "stapled" to the certificates
- **benefit: eliminates client privacy concern and need for the client to connect to an OCSP responder**
- **downside: freshness of the OCSP responses**

# TLS: OCSP Stapling – implementation (II)

- **TLS client MAY send the CSR to the server**
  - as part of ClientHello
  - request the transfer of OCSP responses in the TLS handshake
- **the TLS server that receives a client hello status_request_v2 MAY return OCSP responses for its certificate chain**
  - OCSP responses in a new message, CertificateStatus
- **problems:**
  - servers MAY ignore the status request
  - clients MAY decide to continue anyway the handshake even if OCSP responses are not provided
- **solution: OCSP Must Staple**

# OCSP Must Staple

- **X.509 certificates for servers MAY include a certificate extension**
  - named "TLSFeatures" (OID 1.3.6.1.5.5.7.1.24)
  - defined in RFC-7633
- **the extension informs the client that it MUST receive a valid OCSP response as part of the TLS handshake**
  - otherwise it SHOULD reject the server certificate
- **benefits:**
  - efficiency: the client does not need to query the OCSP responder
  - attack resistance as it prevents blocking OCSP responses (for a specific client) or DoS attack against OCSP responder

# OCSP Must Staple – actors and duties (I)

- **CA must include the extension into server certificates**
  - if requested by the server's owner
- **OCSP Responder must:**
  - be available 365x24 and return valid OCSP responses
- **TLS client must:**
  - send the CSR extension in the TLS in the ClientHello
  - understand the OCSP Must-Staple extension (if present in the server's certificate)
  - reject the server certificate without OCSP stapled response

# OCSP Must Staple – actors and duties (II)

- **TLS server must:**
  - support OCSP Stapling by prefetching and caching OCSP response
  - provide an OCSP response in the TLS handshake
  - handle errors in communication with OCSP responders
- **TLS server administrators should:**
  - configure their servers to use OCSP Stapling
  - request a server certificate with OCSP Must Staple extension

- **open issue (and potential pitfall):**
  - duration of OCSP stapled response (e.g. 7 days for cloudflare)

# TLS status

- F5 telemetry report (October 2021) for the top 1M servers
- 63% have TLS-1.3 (from 80% USA to 15% China and Israel)
- 25% certs use ECDSA and 99% choose non-RSA handshake
- 52% permit RSA, 2.5% expired certs, 2% permit SSL-3
- encryption is abused: 83% of phishing sites use valid TLS and 80% of sites are hosted by 3.8% hosting providers
- SSLstrip attacks still successful, so urgent need for HSTS or to completely disable plain HTTP
- cert revocation checking mostly broken, which pushes for very short-lived certs
- by TLS fingerprinting, 531 servers potentially match the identity of Trickbot malware servers, and 1,164 match Dridex servers