TLS Handshake Protocol

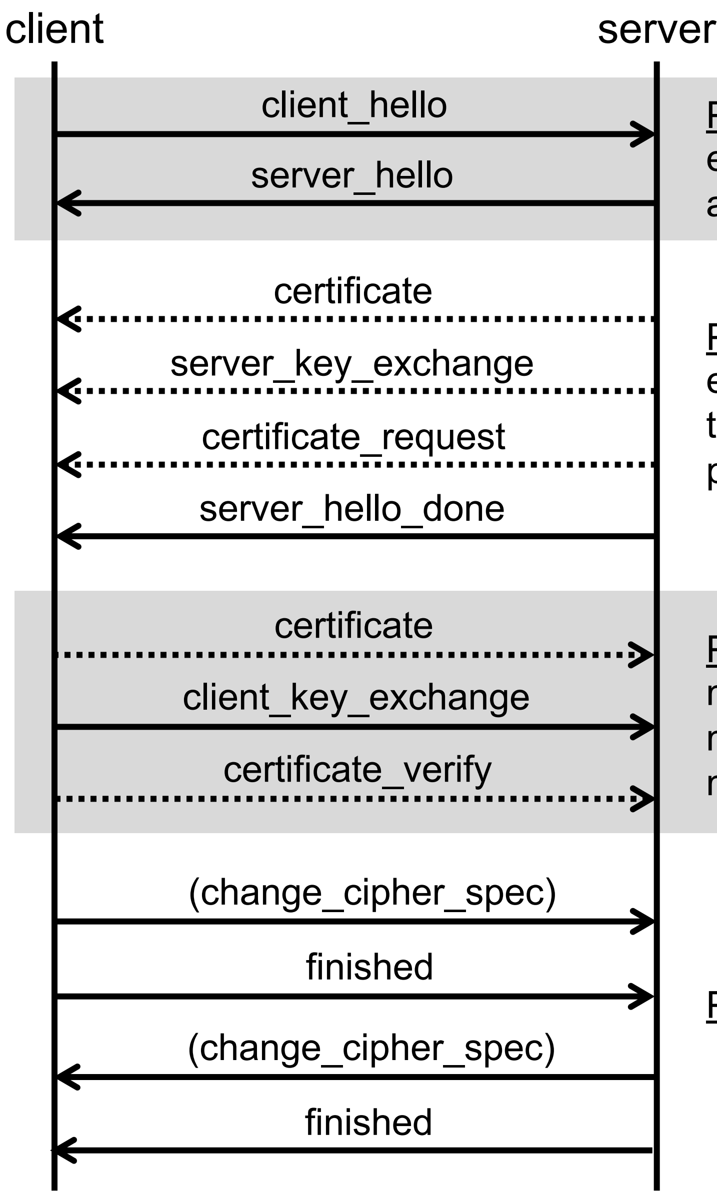
Levente Buttyán
CrySyS Lab, BME
buttyan@crysys.hu

TLS – Transport Layer Security

TLS provides a secure channel for applications (typically between a web server and a web browser → **https**)

- message confidentiality, integrity, and replay protection:
  - symmetric key cryptography is used for message encryption and MAC computation
  - MAC covers a message sequence number → replay protection
  - v1.2 supports a keyed MAC function and authenticated encryption modes, v1.3 only supports authenticated encryption
- (mutual) authentication of parties:
  - asymmetric key cryptography is used to authenticate parties to each other
  - however, client authentication is optional
- key exchange and key derivation:
  - multiple key exchange methods are supported
  - keys are generated uniquely for each connection
  - different keys are used for the encryption and the MAC (unless an authenticated encryption mode is negotiated) and in the two directions (client → server, server → client)
- negotiation of cryptographic algorithms and parameters

TLS v1.2 Handshake – overview



Phase 1: Negotiation of the session ID, key exchange algorithm, MAC algorithm, encryption algorithm, and exchange of initial random numbers

Phase 2: Server may send its certificate and key exchange message, and it may request the client to send a certificate. Server signals end of hello phase.

Phase 3: Client sends certificate if requested and may send an explicit certificate verification message. Client always sends its key exchange message.

Phase 4: Change cipher spec and finish handshake

Hello messages

- version
  - in client_hello: the TLS version the client wants to use (typically the highest version supported by the client)
  - in server_hello: same as client version, or lower if the server does not support that version

- random
  - current time (4 bytes) + pseudo random bytes (28 bytes)

- session_id
  - if a new session is opened:
    - session_id of client_hello is empty
    - session_id of server_hello is the new session ID
  - if the client wants to create a new connection in an existing session:
    - session_id of client_hello is the session ID of the existing session
    - if a new connection can be created in that session, then the server responds with the same session ID → parties can proceed to the "finished" messages
    - otherwise, the server responds with a new session ID → full handshake will take place

Hello messages (cont'd)

- cipher_suites
  - in client_hello: list of cipher suites supported by the client ordered by preference
  - in server_hello: the selected cipher suite

  - a cipher suite contains the specification of
    - » the key exchange method
    - » the encryption algorithm
    - » the MAC algorithm

  - exmaples:

    TLS_RSA_with_AES_128_CBC_SHA
    TLS_RSA_WITH_RC4_128_MD5
    TLS_RSA_WITH_NULL_SHA
    TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_DHE_DSS_WITH_AES_256_CBC_SHA

Supported key exchange methods

- RSA based (TLS_RSA_with...)
  - the secret key (pre-master secret) is encrypted with the server's public RSA key
  - the server's public key is made available to the client during the exchange

- fixed Diffie-Hellman (TLS_DH_RSA_with… or TLS_DH_DSS_with…)
  - the server has fix DH parameters contained in a certificate signed by a CA
  - the client may have fix DH parameters certified by a CA or it may send an unauthenticated one-time DH public value in the client_key_exchange message

- ephemeral Diffie-Hellman (TLS_DHE_RSA_with… or TLS_DHE_DSS_with…)
  - both the server and the client generate one-time DH parameters
  - the server signs its DH parameters with its private RSA or DSS key
  - the client sends an unauthenticated one-time DH public value in the client_key_exchange message
  - the client may authenticate itself (if requested by the server) by signing the hash of the handshake messages with its private RSA or DSS key

- anonymous Diffie-Hellman (TLS_DH_anon_with…)
  - both the server and the client use one-time DH parameters without authentication

certificate and server_key_exchange

- certificate
  - required for every key exchange method except for anonymous DH
  - contains one or a chain of X.509 certificates (up to a known root CA)
  - may contain
    - » public RSA key suitable for encryption, or
    - » public RSA or DSS key suitable for signature verification only, or
    - » fix DH parameters
- server_key_exchange
  - sent only if the certificate does not contain enough information to complete the key exchange (e.g., the certificate contains a signing key only)
  - may contain
    - » public RSA encryption key (exponent and modulus), or
    - » DH parameters (p, g, public DH value g^x mod p)
  - digitally signed

certificate_request and server_hello_done

- certificate_request
  - sent if the server wants the client to authenticate itself
  - specifies which type of certificate is requested

- server_hello_done
  - sent to indicate that the server is finished its part of the key exchange
  - after sending this message the server waits for client response
  - the client should verify that the server provided a valid certificate and the server parameters are acceptable

Client authentication and client_key_exchange

- certificate
  - sent only if requested by the server

- client_key_exchange
  - always sent
  - may contain
    - RSA encrypted pre-master secret, or
    - client one-time public DH value

- certificate_verify
  - sent only if the client sent a certificate
  - provides client authentication
  - contains signed hash of all the previous handshake messages from client_hello up to (not including) this message

Key exchange alternatives

- RSA / no client authentication
  - server sends its encryption capable RSA public key in server_certificate
  - server_key_exchange is not sent
  - client sends encrypted pre-master secret in client_key_exchange
  - client_certificate and certificate_verify are not sent

  or

  - server sends its RSA or DSS public signature key in server_certificate
  - server sends a temporary RSA public key in server_key_exchange
  - client sends encrypted pre-master secret in client_key_exchange
  - client_certificate and certificate_verify are not sent

Key exchange alternatives (cont'd)

- RSA / client is authenticated
  - server sends its encryption capable RSA public key in server_certificate
  - server_key_exchange is not sent
  - client sends its RSA or DSS public signature key in client_certificate
  - client sends encrypted pre-master secret in client_key_exchange
  - client sends signature on all previous handshake messages in certificate_verify

  or

  - server sends its RSA or DSS public signature key in server_certificate
  - server sends a one-time RSA public key in server_key_exchange
  - client sends its RSA or DSS public signature key in client_certificate
  - client sends encrypted pre-master secret in client_key_exchange
  - client sends signature on all previous handshake messages in certificate_verify

Key exchange alternatives (cont’d)

- fix DH / no client authentication
  - server sends its fix DH parameters in server_certificate
  - server_key_exchange is not sent
  - client sends its one-time DH public value in client_key_exchange
  - client_ certificate and certificate_verify are not sent

- fix DH / client is authenticated
  - server sends its fix DH parameters in server_certificate
  - server_key_exchange is not sent
  - client sends its fix DH parameters in client_certificate
  - client_key_exchange is sent but empty
  - certificate_verify is not sent

Key exchange alternatives (cont'd)

- ephemeral DH / no client authentication
  - server sends its RSA or DSS public signature key in server_certificate
  - server sends signed one-time DH parameters in server_key_exchange
  - client sends one-time DH public value in client_key_exchange
  - client_certificate and certificate_verify are not sent

- ephemeral DH / client is authenticated
  - server sends its RSA or DSS public signature key in server_certificate
  - server sends signed one-time DH parameters in server_key_exchange
  - client sends its RSA or DSS public signature key in client_certificate
  - client sends one-time DH public value in client_key_exchange
  - client sends signature on all previous handshake messages in certificate_verify

Finished messages

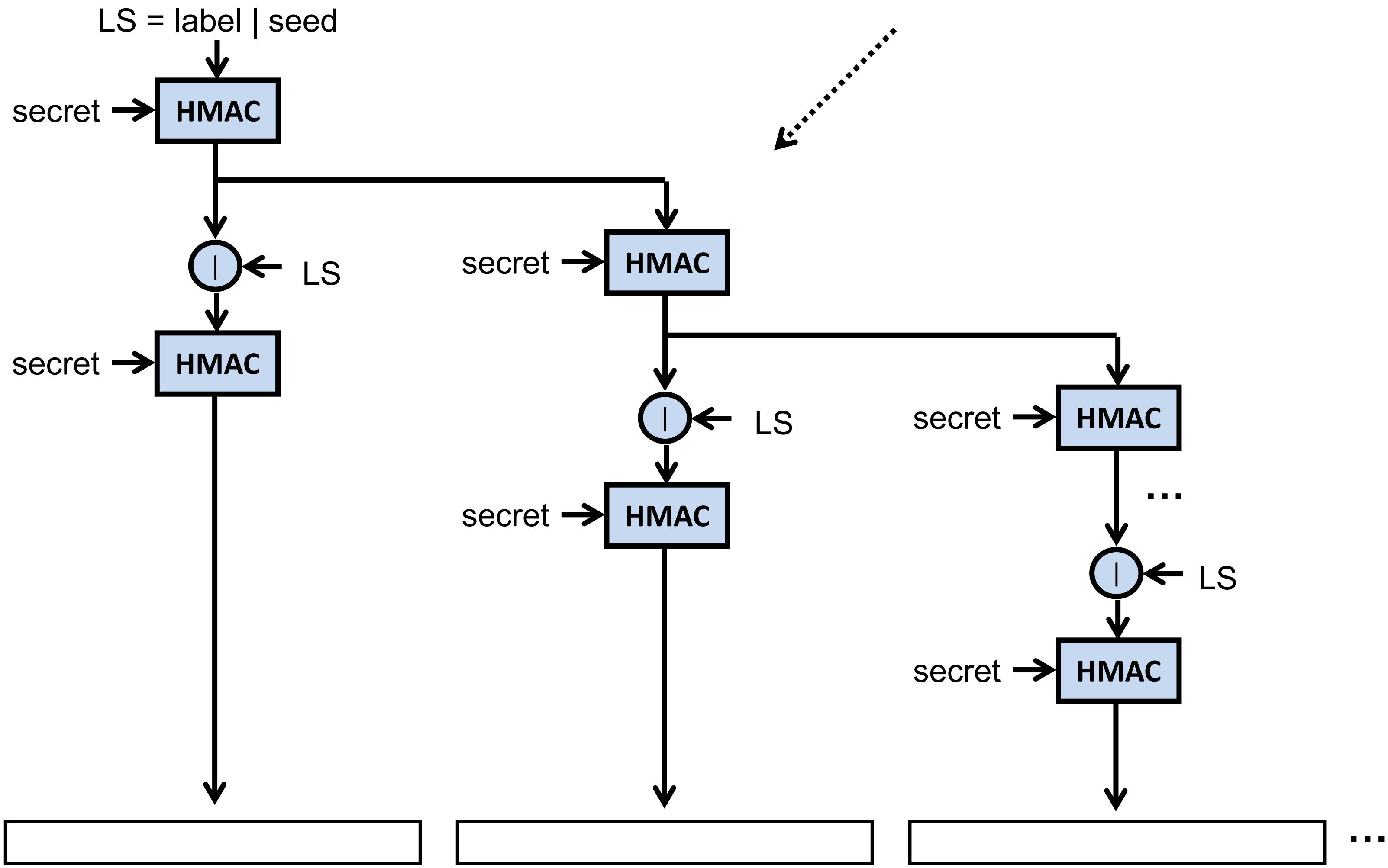
- sent immediately after the change_cipher_spec message
  - change_cipher_spec is not part of the handshake!
  - it triggers a state change (parties are supposed to start using the newly negotiated algorithms and parameters)
  - hence, the finished message is the first message that is protected with the newly negotiated algorithms and keys
- used to authenticate all previous handshake messages
- computed with a pseudo-random function (see definition later) from the master secret and the hash of all handshake messages

```
PRF( master_secret,
     “client finished”,
      hash(handshake_messages) )  → 12 bytes

PRF( master_secret,
     “server finished”,
      hash(handshake_messages) )  → 12 bytes
```

The pseudo-random function PRF

PRF(secret, label, seed) = P_hash(secret, label | seed)



blocks of pseudo-random bytes...

Key generation

- master secret:

  PRF(pre_master_secret,
  "master secret",
  client_random | server_random) → 48 bytes

- connection keys:
  - key block:

    PRF(master_secret,
    "key expansion",
    server_random | client_random) → as many bytes as needed

  - key block is then partitioned:

    client_write_MAC_key | server_write_MAC_key | client_write_key | server_write_key | client_write_IV | server_write_IV

Attacks on the TLS Handshake protocol (up to v1.2)

- (prevention of) version rollback attacks
- exploiting version downgrade implementations (POODLE)
- cross protocol (TLS vs SSLv2) attack (DROWN)
- dropping the Change_Cipher_Spec message
- key exchange algorithm confusion (LOGJAM)

Preventing rollback to SSL v2.0

- an attacker may change the client_hello message so that it looks like an SSL 2.0 client_hello
- if the server still supports SSL 2.0, it will accept the client's offer
- as a result the client and the server will run SSL 2.0
- SSL 2.0 has serious security flaws
  - among other things, there are no finished messages to authenticate the handshake
  - the version rollback attack will go undetected

- fortunately, TLS and SSL 3.0 can detect version rollback
  - pre-master secret generated on SSL 3.0 enabled clients:

```
struct{
  ProtocolVersion client_version; // latest version supported by the client
  opaque random[46];              // random bytes
} PreMasterSecret;
```

  - an SSL 3.0 enabled server detects the version rollback attack, when it runs an SSL 2.0 handshake but receives a pre-master secret that includes version 3.0 as the latest version supported by the client

POODLE attack (2014)

- rollback to SSL v3.0 can still work!
  - POODLE = Padding Oracle On Downgraded Legacy Encryption

- to work with legacy servers (no support for TLS), many TLS clients implement a *downgrade dance*
  - in a first handshake attempt, offer the highest protocol version supported
  - if this handshake fails, retry with earlier protocol versions

→ downgrade can also be triggered by active attackers
  - attacker controls the network between the client and the server
  - she interferes with any attempted handshake offering TLS 1.0 or later
  - client will finally attempt SSL 3.0
  - if they don't use RSA based key exchange, then the client cannot inform the server about the latest version supported (trick to prevent rollback to SSL v2.0 doesn't work)

- SSL 3.0 has severe problems
  - e.g., predictable IVs, features that allow for padding oracle attacks

POODLE countermeasures

- don’t enable SSL in your browser at all
- however, disabling SSL 3.0 entirely may not be practical if it is needed occasionally to work with legacy systems
- in that case, use the TLS_FALLBACK_SCSV mechanism
  - see: https://tools.ietf.org/html/draft-ietf-tls-downgrade-scsv-00
  - basic idea:
    - » client should include in any fallback handshake the TLS_FALLBACK_SCSV value in the list of proposed cipher suites
    - » server must reject connection if TLS_FALLBACK_SCSV is present, and server highest version is larger than the version proposed by the client (new fatal alert type: inappropriate_fallback)
  - attacks remain possible if both parties allow SSL 3.0 but one of them is not updated to support TLS_FALLBACK_SCSV

DROWN attack (2016)

- DROWN = Decrypting RSA using Obsolete and Weakened eNcryption

- exploits the weakness of SSL v2.0 to break TLS
  - SSL v2.0 is vulnerable to the Bleichenbacher attack (1998)
  - adaptive chosen ciphertext attack on RSA with PKCS #1 v1.5
  - sort of a padding oracle attack that allows for the decryption of RSA encrypted messages

- DROWN is a cross protocol attack assuming that the server...
  - still supports SSL v2.0 (besides TLS)
  - uses the same RSA key for both SSL and TLS

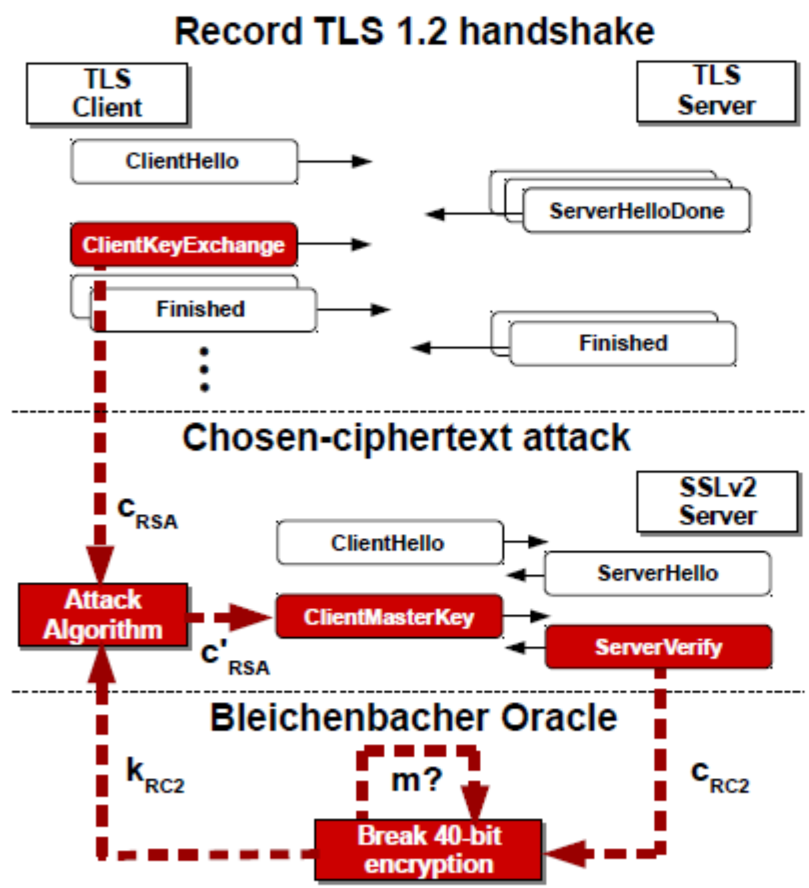
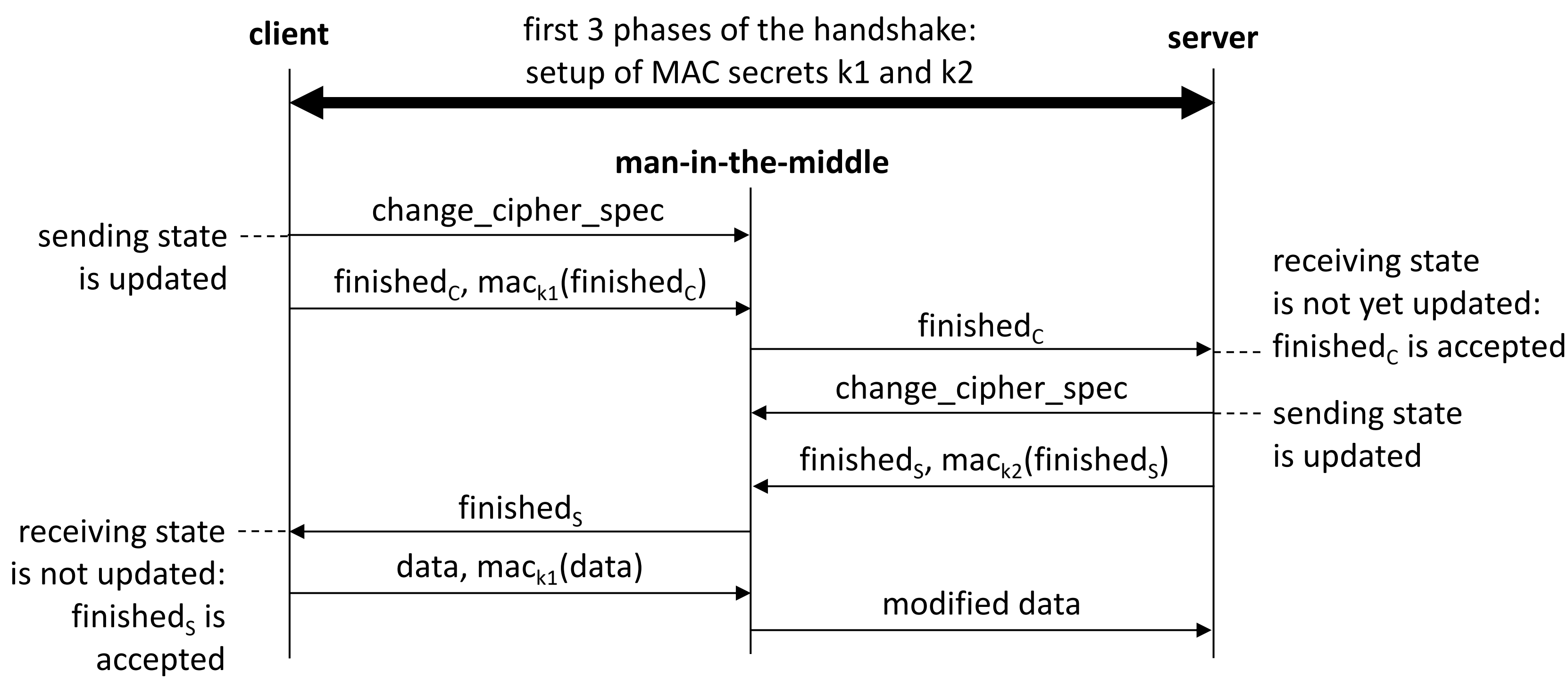


Figure 2: **Our SSLv2-based Bleichenbacher attack on TLS.** An attacker passively collects RSA ciphertexts from a TLS 1.2 handshake, and then performs oracle queries against a server that supports SSLv2 with the same public key to decrypt the TLS ciphertext.

Dropping change_cipher_spec

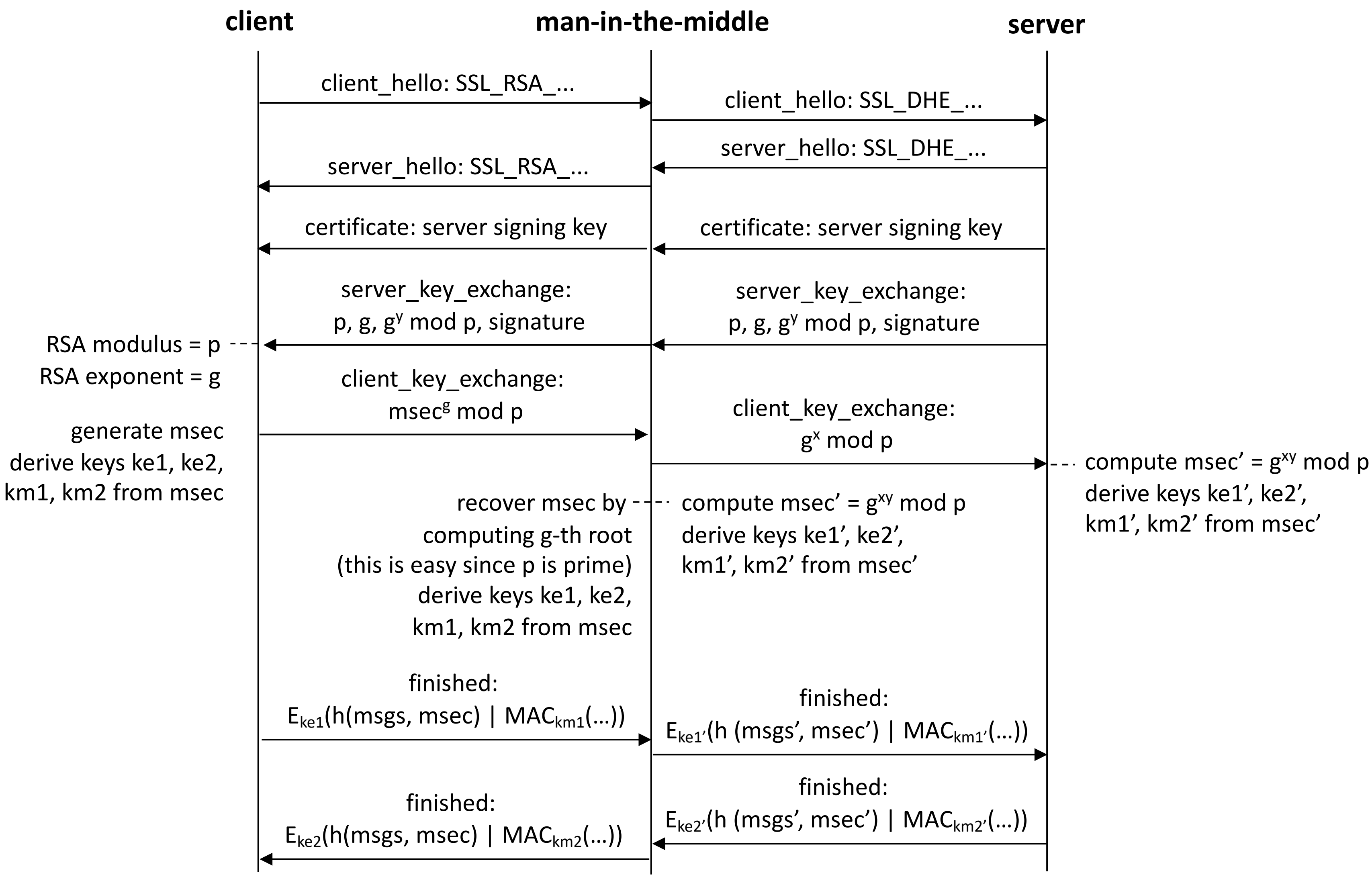
- authentication in the finished message does not protect the change_cipher_spec message (it is not part of the handshake protocol !)
- this may allow the following attack:
  – assume that the negotiated cipher suite includes only message authentication (no encryption)



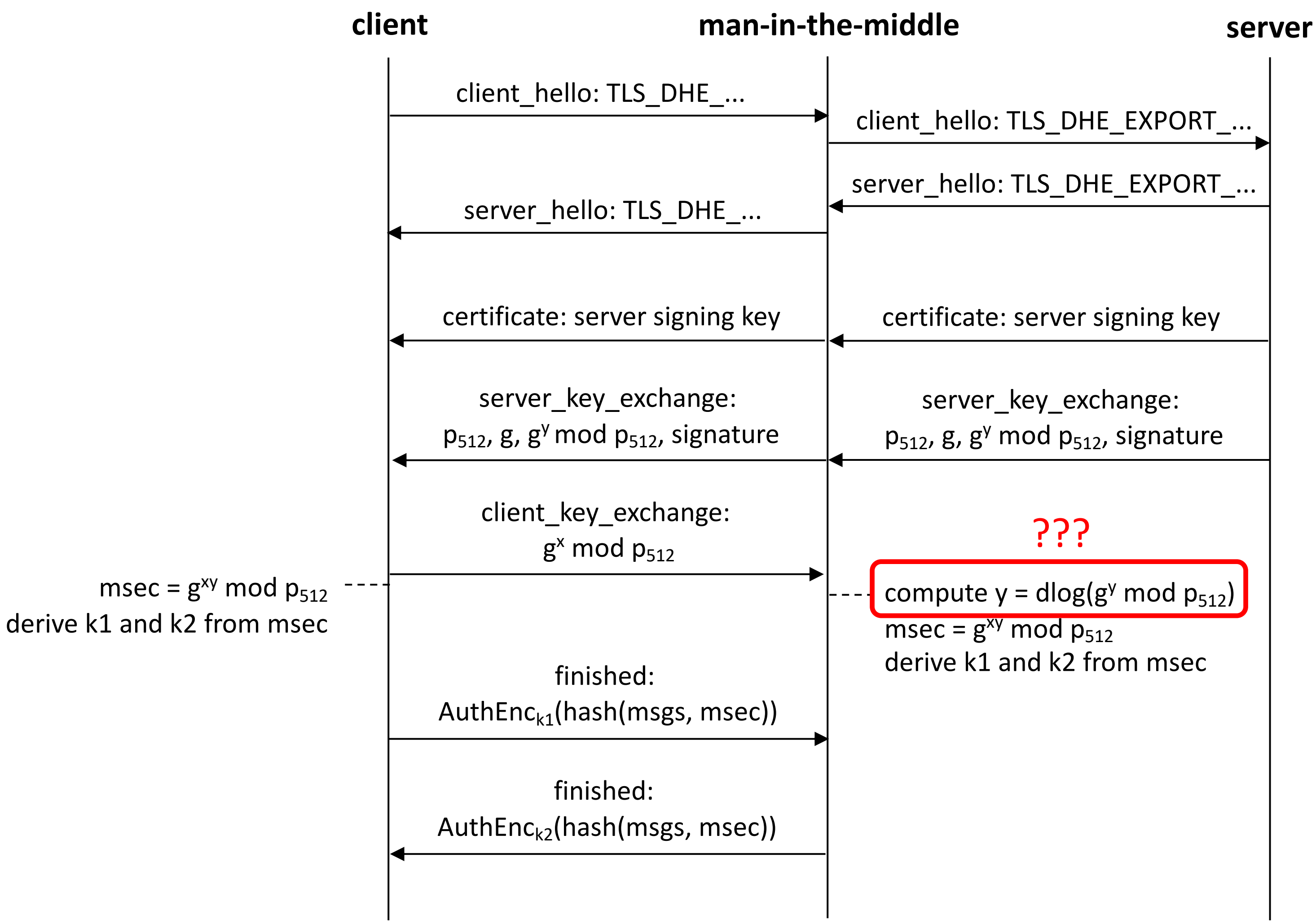
Dropping change_cipher_spec

- if the negotiated cipher suite includes encryption, then the attack doesn't work
  - client sends encrypted finished message
  - server expects clear finished message
  - the attacker cannot decrypt the encrypted finished message
- the attack is now prevented in TLS by requiring reception of change_cipher_spec before processing the finished message
  - this seems to be obvious, but…
  - even Netscape's reference SSL implementation SSLRef 3.0b1 allowed for processing finished messages without checking if a change_cipher_spec has been received
- another possible fix: include the change_cipher_spec message in the computation of the finished message
  - for some reason, this approach has not been adopted for a long time

Key-exchange algorithm confusion

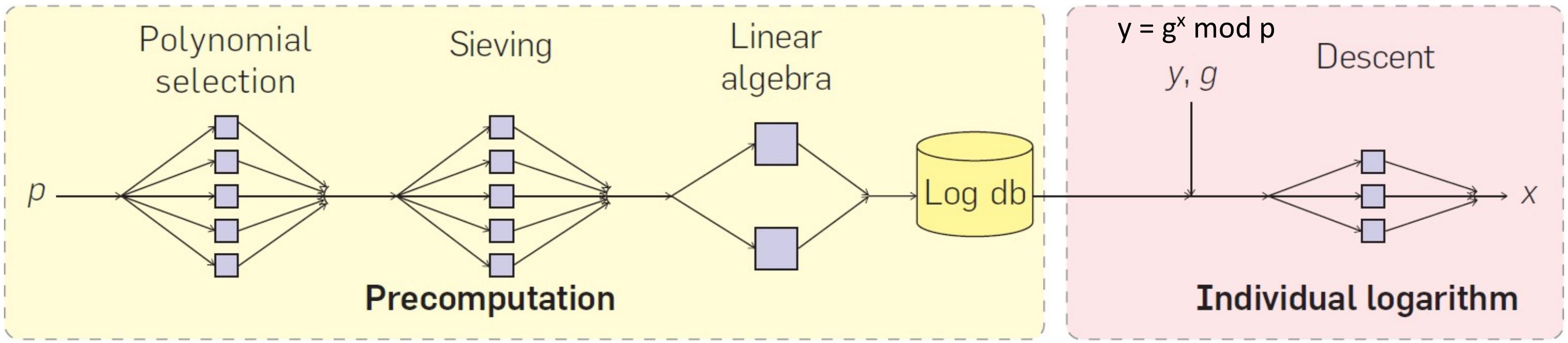


Logjam attack (2015)



Computing discrete logarithm (dlog)

- best known algorithm is the Number Field Sieve (NFS)
- NFS can take advantage of pre-computations



from article "Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice"
https://cacm.acm.org/magazines/2019/1/233523-imperfect-forward-secrecy/fulltext

- if primes are re-used, then pre-computation makes sense and discrete log of individual values can be computed faster
  - order of minutes on commodity PCs for 512-bit primes
  - even faster on high performance servers

Back to Logjam

- TLS servers in practice often don't generate fresh primes, but they use a static prime hard-coded in the implementation (e.g., Apache)
- some statistics from 2015:
  - 8.4% of Alexa Top Million sites enabled DHE_EXPORT (512 bit)
  - 92.3% of these sites used one of two well-known primes
- recommendations:
  - disable DHE-EXPORT
  - use primes of 2048 bits or larger
  - generate primes on-demand

Lessons form key-exchange confusion

- TLS/SSL authenticates only the server’s (RSA or DH) parameters in the server_key_exchange message
- it doesn’t authenticate the context (key exchange algorithm in use) in which those parameters should be interpreted

- a potential fix:
  - hash all messages exchanged before the server_key_exchange message and include the hash in the signature in the server_key_exchange message

Conclusions on TLS attacks

- in case of message encryption schemes, any kind of information leakage may be a problem in practice, even if you think that the amount of information leaked is small
  - information may be leaked through the protocol itself
    - e.g., error messages may leak information about correctness of the padding
  - or via side-channels
    - e.g., timing, message length (compression ratio), ...

- protocols with lot of flexibility are difficult to make secure
  - flexibility → many options → increased complexity →← security
  - in particular, supporting multiple versions (including old, potentially weak versions) of the protocol may easily lead to problems
  - supporting multiple key exchange protocols may allow for cross protocol attacks

- even if your protocol is secure at the design level, there can be implementation flaws
  - minimize the risk by being precise and explicit in the specification (don’t let implementers make decisions on their own)

TLS v1.3 Handshake

- faster and more secure than the v1.2 Handshake
- main features/differences:
  - 1-RTT full handshake
  - 0-RTT session resumption (with limitations on security)
  - RSA based kex echange (and fixed DH) removed (did not provide forward secrecy)
  - Change Cipher Spec message is removed (wasn’t included in Finished)
  - server certificate verify message introduced
  - handshake messages after the Hello are encrypted
  - better version downgrade protection
  - new key derivation function (HKDF)

1-RTT full handshake (simplified)

```
                 Client              Server

            ClientHello
           + key_share*  -------->

                                     ServerHello
                                     + key_share*
                                     {EncryptedExtensions}
                                     {Certificate*}
                                     {CertificateVerify*}
                                     {Finished}
                         <-------- [Application Data*]

              {Finished} -------->

       [Application Data] <-------> [Application Data]


        +  extension
        *  optional
       { } protected with keys derived from handshake_traffic_secret
       [ ] protected with keys derived from application_traffic_secret
```

1-RTT full handshake (simplified)

```
                 Client                 Server

            ClientHello
          + key_share*   -------->
                                        ServerHello
                                        + key_share*
                                        {EncryptedExtensions}
                                        {Certificate*}
                                        {CertificateVerify*}
                                        {Finished}
                         <--------      [Application Data*]

              {Finished} -------->

     [Application Data]  <------->      [Application Data]


      +   extension
      *   optional
     { }  protected with keys derived from handshake_traffic_secret
     [ ]  protected with keys derived from application_traffic_secret
```

list of KeyShareEntry values of the client (in different groups)

KeyShareEntry of the server in the selected group

1-RTT full handshake (simplified)

```
              Client                  Server

         ClientHello
        + key_share*  -------->
                                      ServerHello
                                      + key_share*
                                      {EncryptedExtensions}
                                      {Certificate*}
                                      {CertificateVerify*}
                                      {Finished}
                      <--------       [Application Data*]

          {Finished}  -------->

  [Application Data]  <------->       [Application Data]
```

hash of handshake messages signed by the server

```
 +   extension
 *   optional
{ } protected with keys derived from handshake_traffic_secret
[ ] protected with keys derived from application_traffic_secret
```

Key derivation (simplified)

- output of (EC)DHE
  - Handshake Secret
    - client_handshake_traffic_secret
    - server_handshake_traffic_secret
    - Master Secret
      - client_application_traffic_secret
      - server_application_traffic_secret

0-RTT session resumption

```
                      Client          Server

                 ClientHello
       + psk_key_exchange_modes*
               + pre_shared_key*
           (Application Data*) -------->

                                      ServerHello
                                      + pre_shared_key*
                                      {EncryptedExtensions}
                                      {Finished}
                               <-------- [Application Data*]

                   {Finished} -------->

            [Application Data] <-------> [Application Data]


  +  extension
  *  optional
 { } protected with keys derived from handshake_traffic_secret
 [ ] protected with keys derived from application_traffic_secret
```

0-RTT session resumption

```
                  Client               Server

             ClientHello
+ psk_key_exchange_modes*
        + pre_shared_key*
    (Application Data*)  -------->
                                       ServerHello
                                       + pre_shared_key*
                                       {EncryptedExtensions}
                                       {Finished}
                         <-------- [Application Data*]

              {Finished} -------->

     [Application Data]  <-------> [Application Data]


 +   extension
 *   optional
{ }  protected with keys derived from handshake_traffic_secret
[ ]  protected with keys derived from application_traffic_secret
```

list of key IDs of previous shared keys

selected key ID

early data protected with an early_traffic_secret derived from the first PSK in the list

Control questions

- What are the phases of the TLS v1.2 Handshake Protocol?
- What key exchange methods are supported by TLS v1.2? How do they work?
- How are ciphersuites negotiated in the TLS handshake?
- How is the server authenticated in the TLS handshake?
- What is the role of the Change_Cipher_Spec messages?
- How are connection keys derived from the session master secret?
- What is the role of the Finished messages and how are they constructed?
- Which parts of the handshake are kept when the parties create a new connection in an already existing session?

Control questions

- What are the main ideas of the following attacks on the Handshake Protocol?
  - Version rollback
  - POODLE attack
  - DROWN attack
  - Dropping the Change Cipher Spec messages
  - Key exchange confusion and the LOGJAM attack
- What are the main differences between the TLS v1.3 Handshake and the TLS v1.2 Handshake?
- How do the 1-RTT and 0-RTT handshakes work?
- What are the main lessons that we can learn from the history of TLS (attacks and re-designs)?