Transport Layer Security (TLS)

Security Protocols (bmevihim132)

Dr. Levente Buttyán

associate professor

BME Hálózati Rendszerek és Szolgáltatások Tanszék

Lab of Cryptography and System Security (CrySyS)

buttyan@hit.bme.hu, buttyan@crysys.hu

Outline

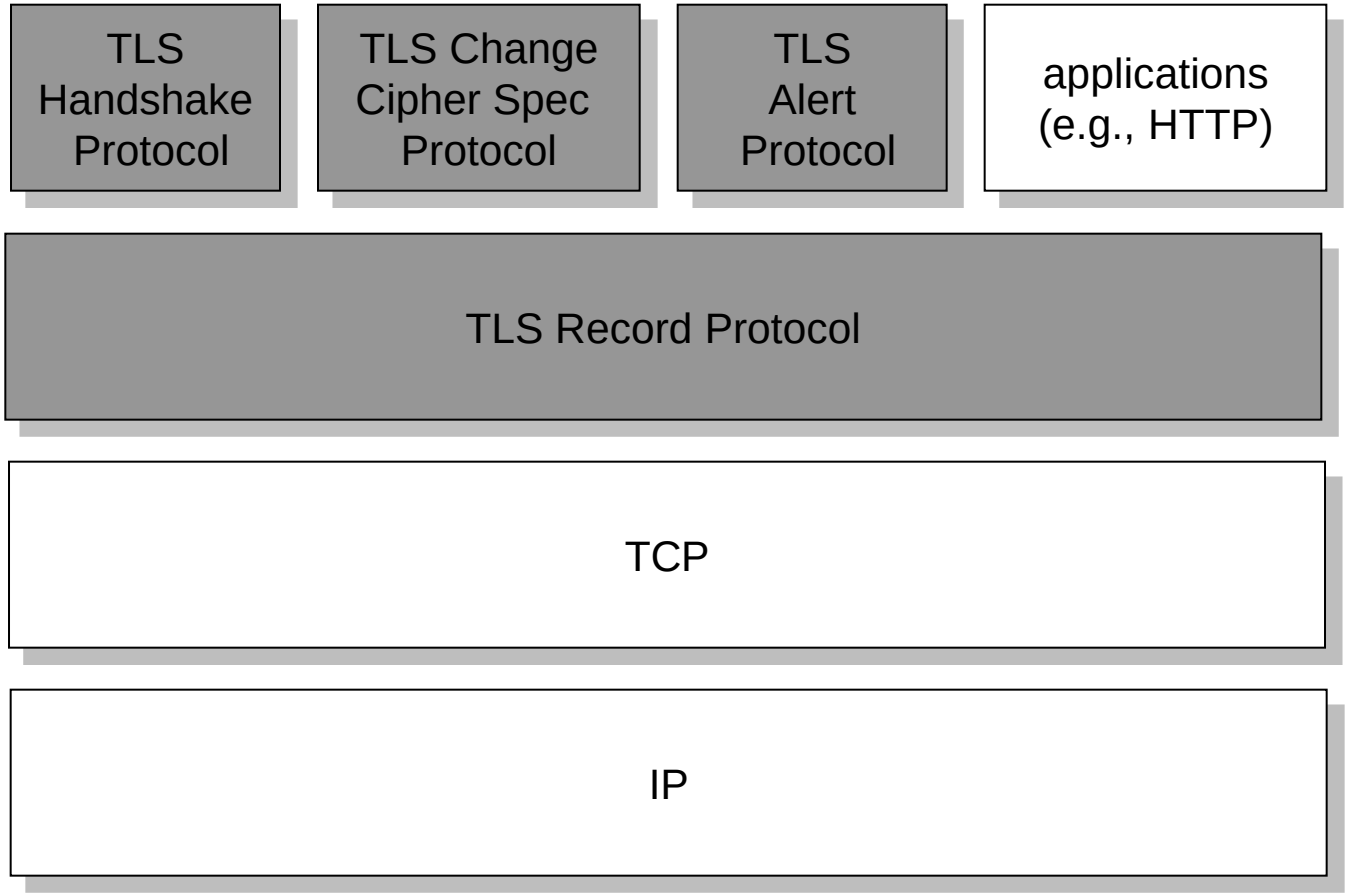
- architecture and services
- TLS Record Protocol
- TLS Handshake Protocol
- analysis and some potential attacks

TLS – Transport Layer Security

- objective
  - to provide a secure transport connection between applications (typically between a web server and a browser → https)
    - mutual authentication of parties
    - encryption, integrity and replay protection of all exchanged messages

- history
  - SSL – Secure Socket Layer
    - developed by Netscape in the mid 90's
    - version 3.0 has been implemented in many web browsers and web servers and become a de-facto standard
  - the IETF adopted it in 1999 under the name TLS
    - TLS v1.0 ~ SSL v3.0 with some design errors corrected
    - further modifications due to some recent attacks → v1.1 → v1.2
    - most recent specification is RFC 5746

TLS architecture



TLS components

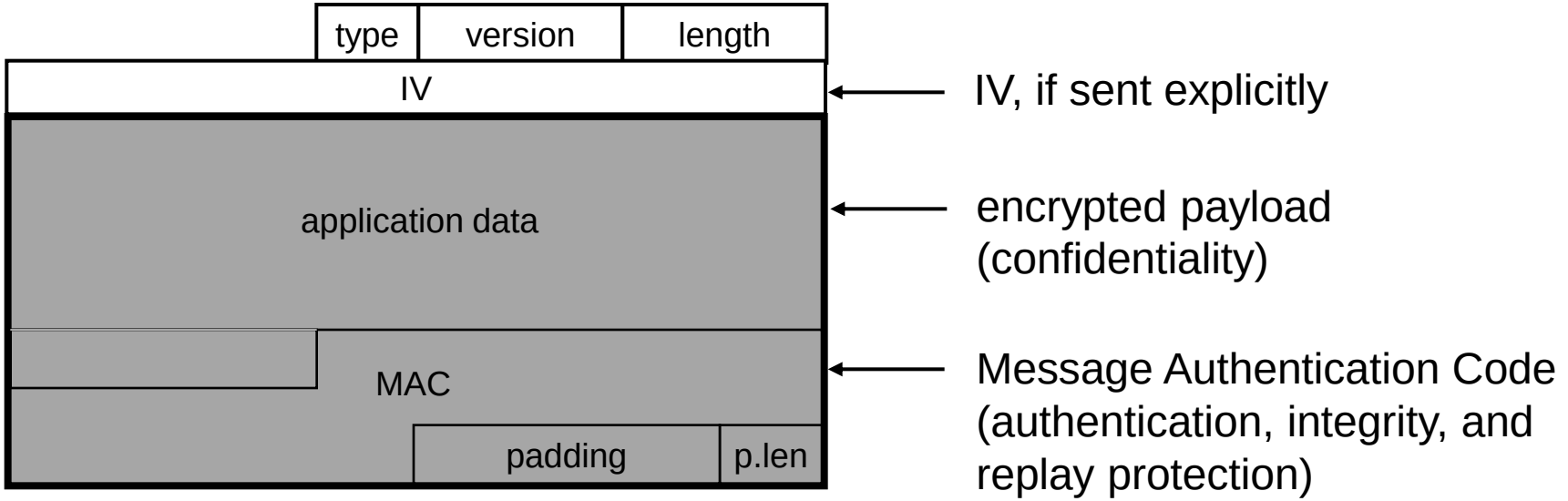
- TLS Handshake Protocol
  - negotiation of security algorithms and parameters
  - key exchange
  - server authentication and optionally client authentication
- TLS Record Protocol
  - fragmentation
  - compression
  - message authentication and integrity protection
  - encryption
- TLS Alert Protocol
  - error messages (fatal alerts and warnings)
- TLS Change Cipher Spec Protocol
  - a single message that indicates the end of the TLS handshake

Sessions and connections

- a TLS session is a security association between a client and a server
- sessions are stateful; the session state includes:
  - the negotiated security algorithms and parameters
  - certificates (if any)
  - a master secret shared between the client and the server (established during the TLS handshake)

- a session may include multiple secure connections between the same client and server
  - connections of the same session share the session state
  - in addition, each connection has its own connection keys (derived from the master secret) and connection specific random numbers
- factoring out the master secret into the session state helps to avoid expensive negotiation of new security parameters for each and every new connection

TLS Record Protocol



MAC:

- computed before the encryption
- input to MAC computation:
  - MAC_write_key
  - **seq_num** || type || version || length || payload (compressed fragment)
- supported algorithms: HMAC with MD5, SHA1, SHA256

Encryption in the Record Protocol

- stream ciphers
  - no IV and padding
  - no re-initialization of the cipher for individual messages
  - supported algorithm: RC4 (128)

- block ciphers in CBC mode
  - IV must be a random value (in TLSv1.0, IV was the last cipher block of the previous message)
  - padding:
    - last byte is the length *n* of the padding (not including the last byte)
    - padding length *n* can range from 0 to 255 bytes, but the total length of the encrypted text must be a multiple of the block size of the cipher
    - all padding bytes have value *n* → examples for correct padding: x00, x01x01, x02x02x02, …
    - after decryption, the padding format is verified; if correct, then MAC verification follows
  - supported algorithms: 3DES-EDE, AES (128, 256)

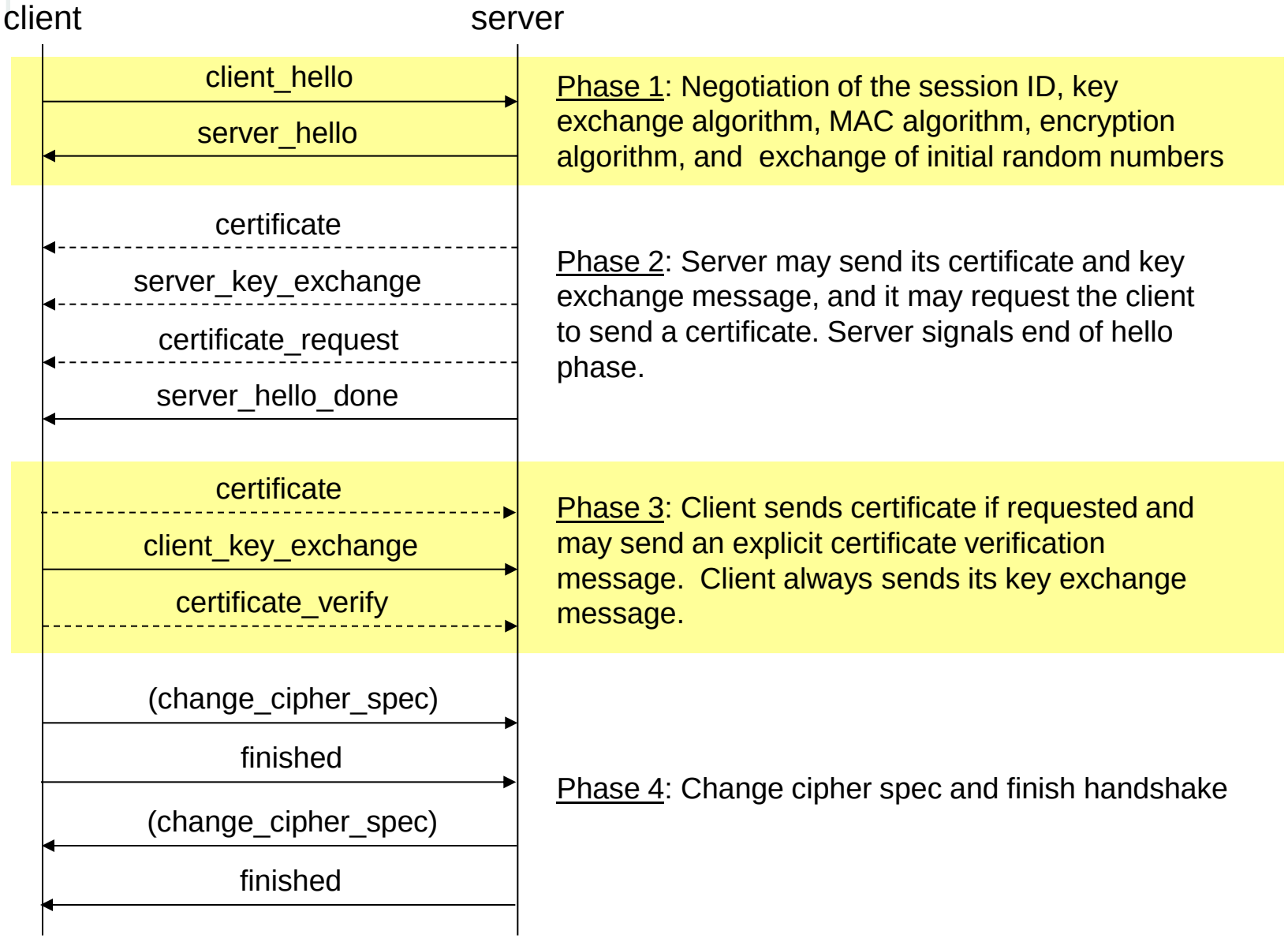
Auth-enc in the Record Protocol

- authenticated encryption modes
  - nonce is carried in the IV field, no padding
  - jointly encrypted and authenticated message is computed from
    - client_ or server_write_key
    - nonce
    - cleartext payload
    - seq_num || type || version || length
  - supported algorithms: AES-GCM (128, 256), AES-CCM (128, 256)

TLS Alert Protocol

- each alert message consists of 2 fields (bytes)
- first field (byte): “warning” or “fatal”
- second field (byte):
  - fatal
    - unexpected_message
    - bad_record_MAC
    - decryption_failure (!)
    - handshake_failure
    - …
  - warning
    - close_notify
    - user_canceled
    - no_renegotiation
    - …
- in case of a fatal alert
  - connection is terminated
  - session ID is invalidated → no new connection can be established within this session

TLS Handshake – overview



Hello messages

- version
  - the highest TLS version supported by the party
- 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 a new connection is created in an existing session:
    - session_id of client_hello is the session ID of the existing session
    - session_id of server_hello is the existing session ID if the connection can be created or empty otherwise
- cipher_suites
  - in client_hello: list of cipher suites supported by the client ordered by preference
  - in server_hello: selected cipher suite
  - a cipher suite contains the specification of the key exchange method, the encryption algorithm, and the MAC algorithm
  - exmaple: TLS_RSA_with_AES_128_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 generate one-time DH parameters
  - they send their parameters to the peer without authentication

Server certificate and 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 signing 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 an RSA signing key only)
  - may contain
    - public RSA key (exponent and modulus), or
    - DH parameters (p, g, public DH value)
  - digitally signed

Cert request and server hello done

- certificate_request
  - sent if the client needs 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 auth and 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 this message

Finished messages

- finished
  - sent immediately after the change_cipher_spec message
  - used to authenticate all previous handshake messages
  - first message that uses the newly negotiated algorithms and keys
  - 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)

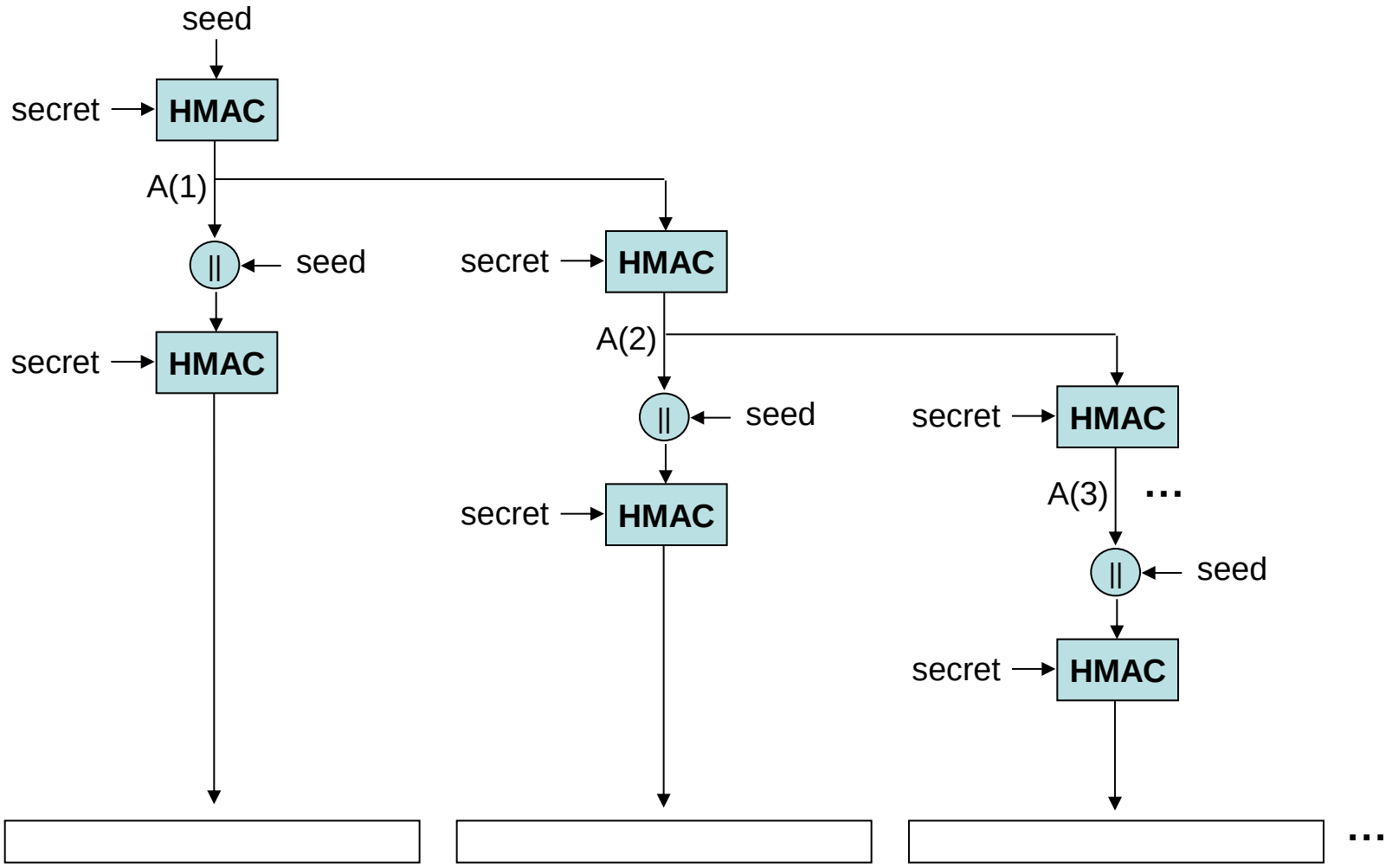
- P_hash(secret, seed) = HMAC_hash(secret, A(1) || seed) ||
  HMAC_hash(secret, A(2) || seed) ||
  HMAC_hash(secret, A(3) || seed) ||
  …

  where

  A(0) = seed

  A(i) = HMAC_hash(secret, A(i-1))

P_hash illustrated



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

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

Key exchange alternatives (cont'd)

- anonymous DH / no client authentication
  - server_certificate is not sent
  - server sends (unsigned) 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

- anonymous DH / client is authenticated
  - not allowed

Analysis and attacks

SSL/TLS Handshake protocol

- Cipher suite rollback attack
- Dropping the Change_Cipher_Spec message
- Key exchange algorithm rollback
- Version rollback

Reference:

D. Wagner, B. Schneier. Analysis of the SSL 3.0 protocol, November 1996.

Cipher suite rollback attack

- in SSL 2.0, an attacker could force the use of an export-weakened encryption algorithm by modifying the list of supported cipher suites in the hello messages
- this is prevented in TLS (and SSL 3.0) by authenticating all handshake messages with the master secret (in the finished messages)
- the master secret itself is authenticated by other means
  - for the client:
    - implicit authentication via the server certificate
      - only the server could decrypt the RSA encrypted pre-master secret
      - only the server could compute the pre-master secret from the client's public DH value
    - explicit authentication via the server_key_exchange message (if sent)
      - ephemeral parameters are explicitly signed by the server
  - for the server:
    - explicit authentication via the certificate_verify message (if sent)
      - certificate_verify contains a signed hash of all previous handshake messages including those that contain the key exchange parameters used to compute the master secret

Version rollback attacks

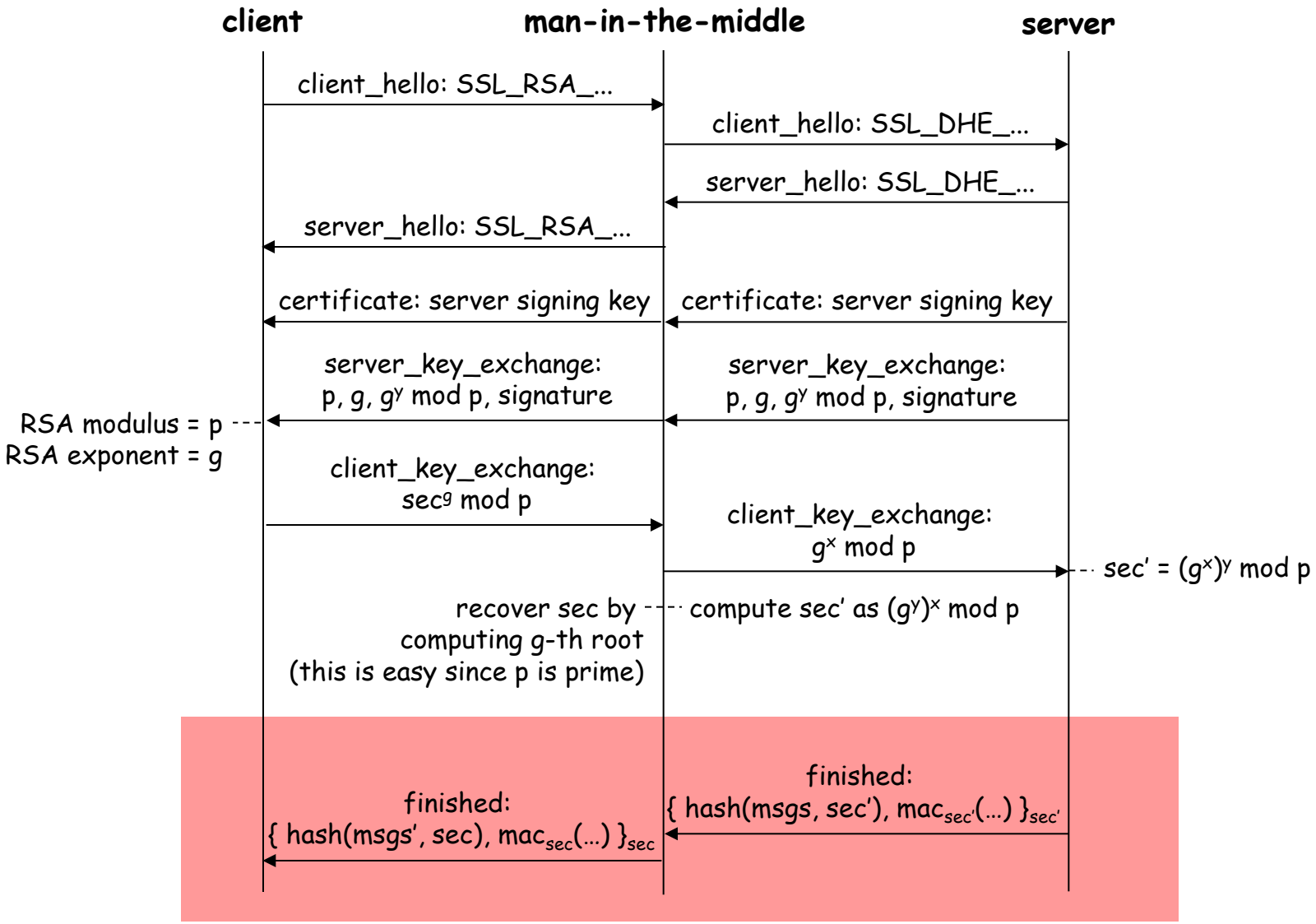
- on some servers, SSL 2.0 may still be supported
- an attacker may change the client_hello message so that it looks like an SSL 2.0 client_hello
- as a result the client and the server will run SSL 2.0
- SSL 2.0 has serious security flaws
  - among other things, there are no finished messages to authenticate 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

Key-exchange algorithm rollback



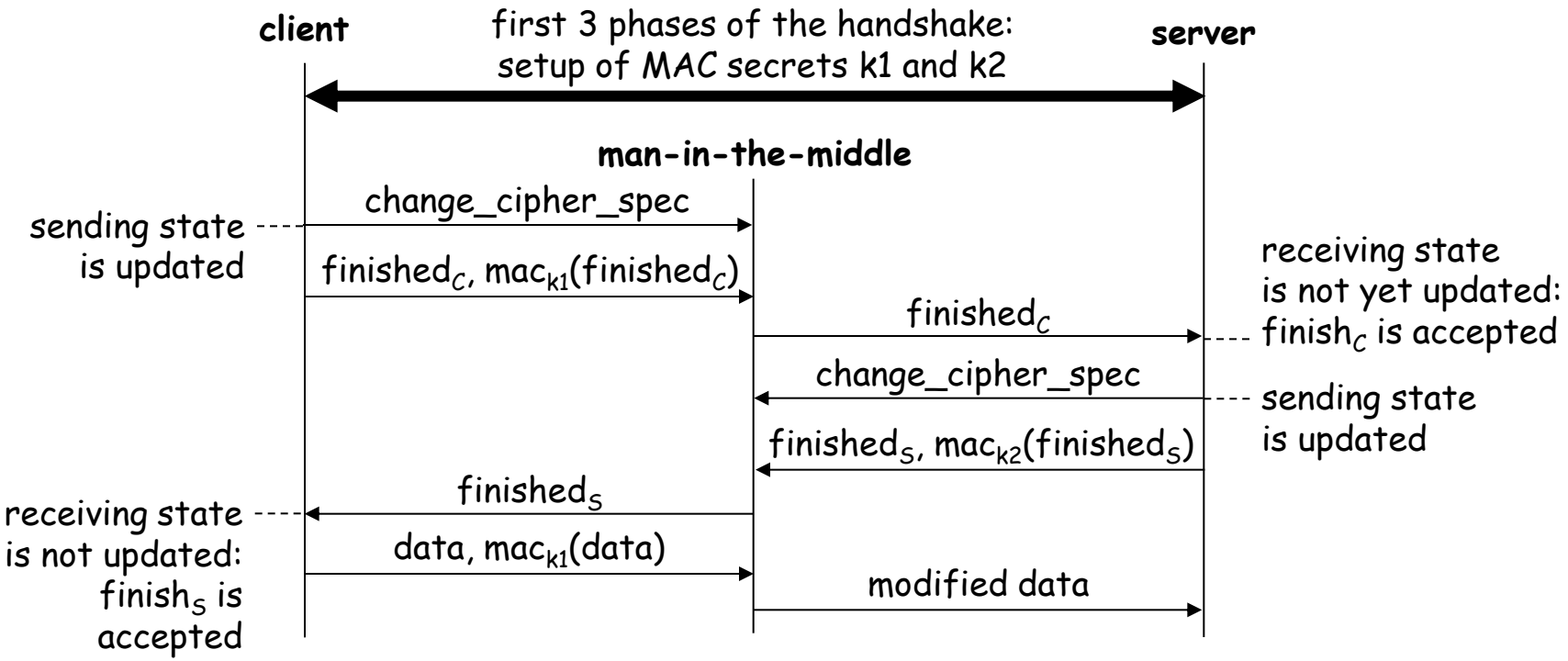
Key-exchange algorithm rollback

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

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 the 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 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

SSL/TLS Record protocol

- A distingushing attack
- Padding oracle attack
- Lucky 13 attack
- BEAST attack

References:

S. Vaudenay. Security Flaws Induced by CBC Padding – Applications to SSL, IPSEC, WTLS ... EUROCRYPT 2002.

B. Canvel, A. P. Hiltgen, S. Vaudenay, and M. Vuagnoux. Password Interception in a SSL/TLS Channel. CRYPTO 2003.

N. J. AlFardan, K. G. Paterson. Lucky 13: Breaking the TLS and DTLS Record Protocols. February 27, 2013.

T. Duong and J. Rizzo. Here come the XOR Ninjas. Unpublished manuscript, 2011.

Padding oracle attack (2002)

- send a random message to a TLS server
- the server will drop the message with overwhelming probability
  - either the padding is incorrect (the server responds with a DECRYPTION_FAILED alert)
  - or the MAC is incorrect with very high probability (the server responds with BAD_RECORD_MAC alert)
- if the response is BAD_RECORD_MAC, then the padding was correct → we get 1 bit of information !
- such an oracle can be used to decrypt any encrypted message (padding oracle attack)

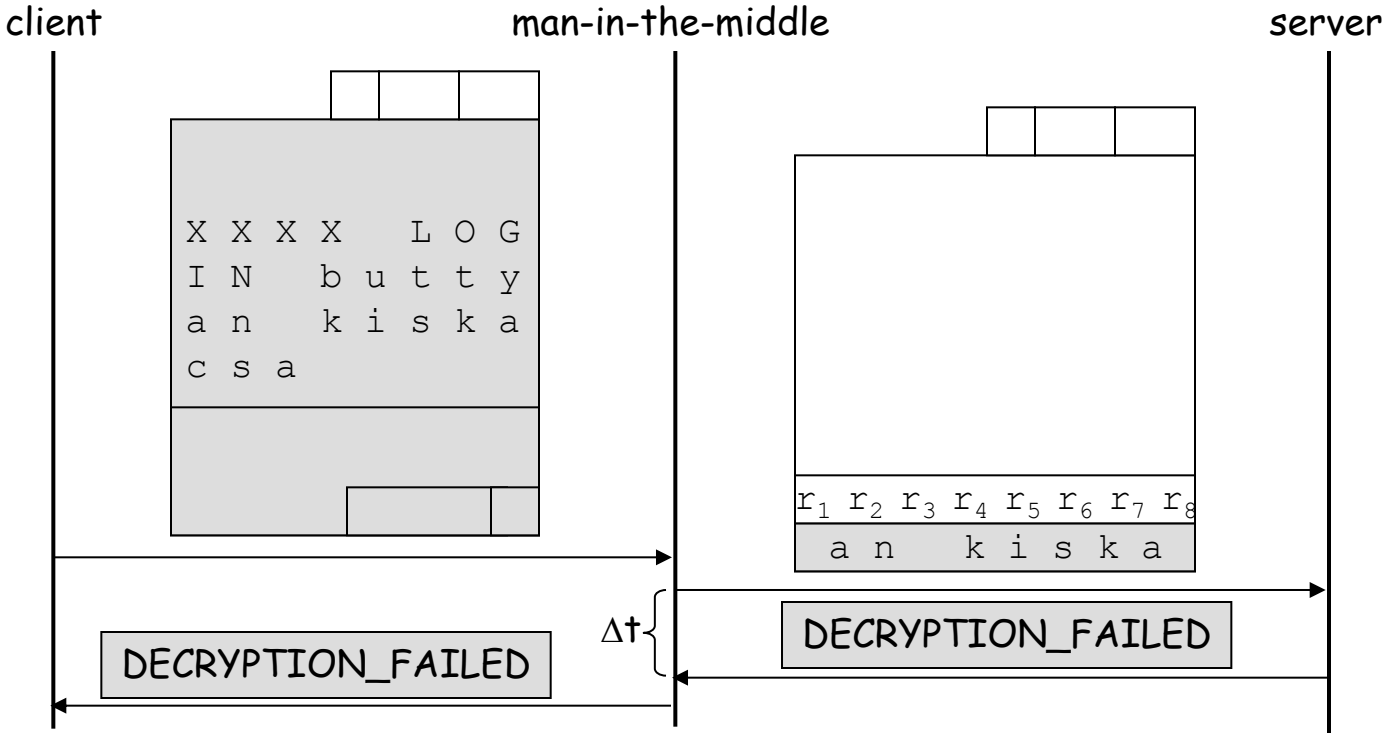
Problems in practice

- alert messages are encrypted → BAD_RECORD_MAC and DECRYPTION_FAILED cannot be distinguished
  - measure timing between oracle call and oracle response
  - BAD_RECORD_MAC takes more time than DECRYPTION_FAILED

- BAD_RECORD_MAC and DECRYPTION_FAILED are fatal errors → connection is closed after one oracle call
  - a password can still be broken if it is sent periodically to a server using TLS (a different session (and key) is used each time the password is sent, but the password is always the same)

Example: IMAP over TLS (2003)

- Outlook Express checks for new mail on the server periodically (every 5 minutes)
- each time the same password is sent for every folder
  XXXX LOGIN “username” “password”<x0D><x0A>
- it is possible to uncover the password using the attack as follows:



Eliminating the timing channel

- when badly formatted padding is encountered during decryption, a MAC check must still be performed on *some* data to prevent known timing attacks
- which data?
- TLS 1.1 and 1.2 recommend checking the MAC as if there was a zero-length pad

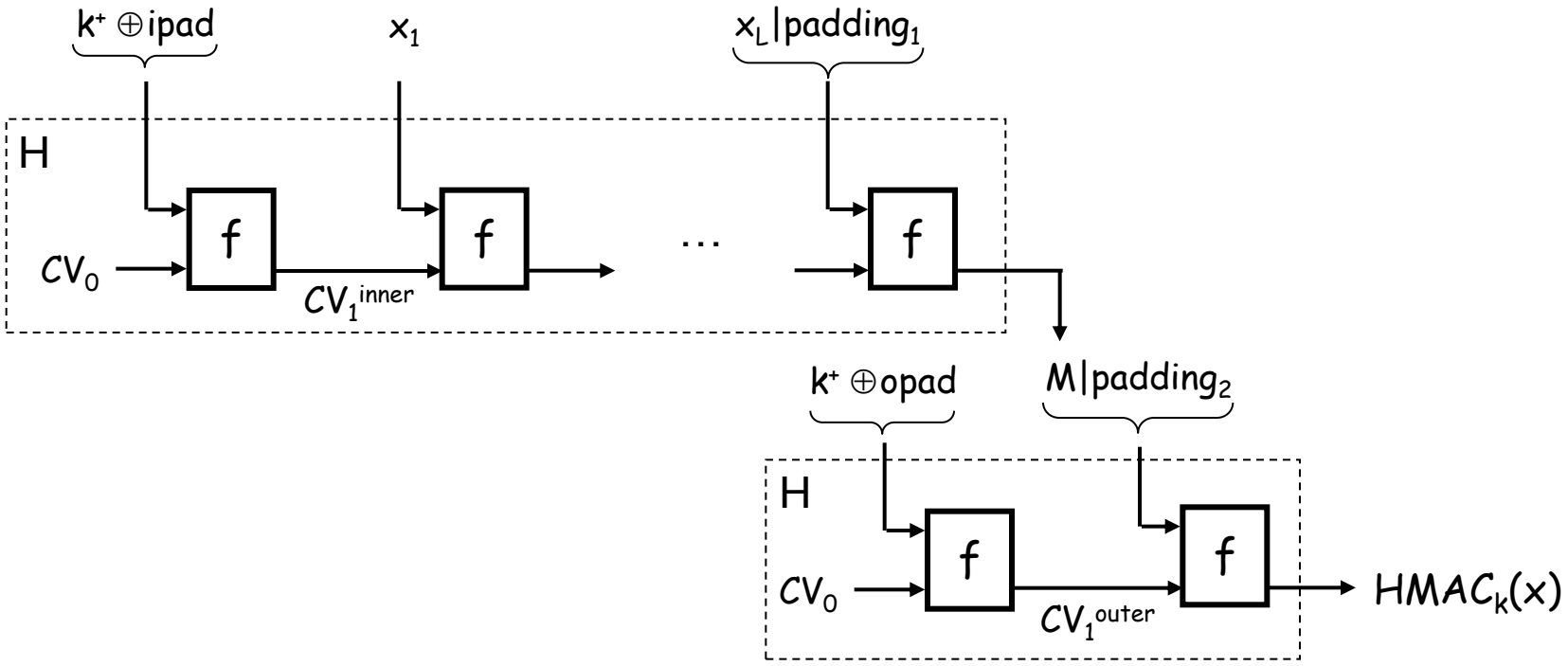
> „This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal.”

Reminder on HMAC

$HMAC_k(x) = H((k^+ \oplus opad) \mid H((k^+ \oplus ipad) \mid x))$

where

- h is a hash function with input block size b and output size n
- k^+ is k padded with 0s to obtain a length of b bits
- ipad is 00110110 repeated b/8 times
- opad is 01011100 repeated b/8 times



Hash functions used by HMAC in TLS

- MD5, SHA1, SHA256

- block size for all of these is 512 bits = 64 bytes
- padding for hashing
  - 8-byte message length field (Merkle-Damgard strengthening)
  - followed by at least 1 byte of padding

- key observation:
  - if message length < 56 bytes
    - then message + 8 byte length + padding fits in a single 64-byte block
    - hashing needs 4 invocations of the compression function f
  - if message length > 55 bytes
    - then padded message needs at least two 64-byte blocks
    - hashing needs at least 5 invocations of f
  - this is a timing channel!

A distinguishing attack

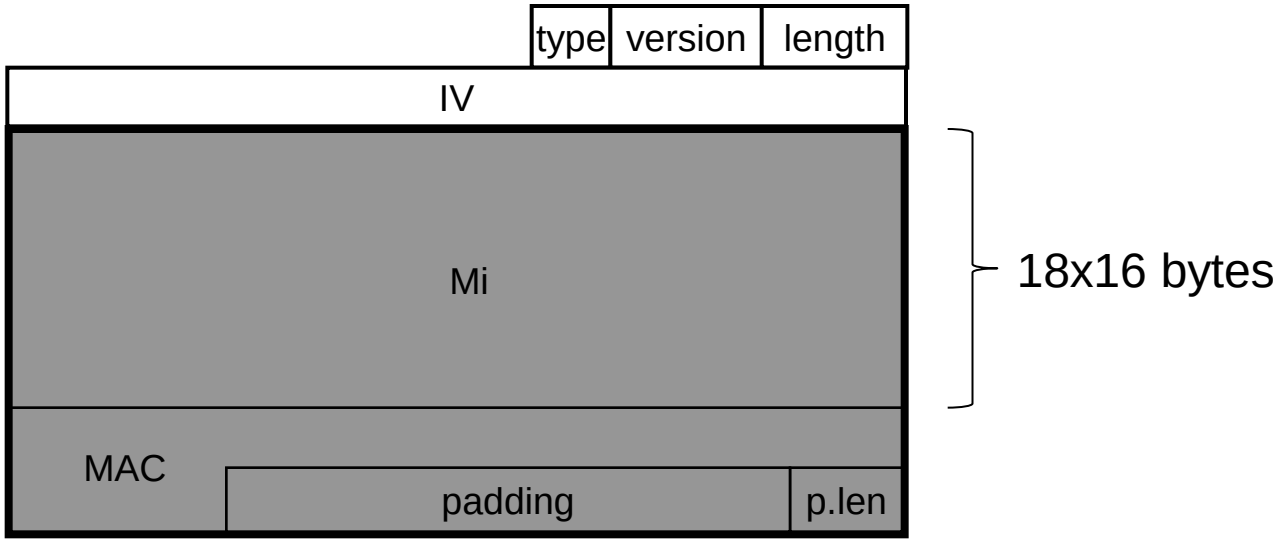
- abstract description
  - the attacker chooses a pair of messages (M0, M1) of equal length
  - one of these is encrypted and given back to the attacker
  - the attacker must tell which message was encrypted (can use a decryption oracle on other messages)

- particular parameters in case of TLS
  - encryption is AES in CBC mode (enc block size is 16 bytes)
  - MAC and padding (in this order) before encryption
    - let's assume the MAC is HMAC with SHA1 (20 bytes)
  - TLS Record header (5 bytes)

- let
  - M0 be 32 arbitrary bytes followed by 256 xFF (max length padding)
  - M1 be 287 arbitrary bytes followed by a single x00 (min pad)
- note that both messages are 288 bytes long (= 18x16 bytes)

A distinguishing attack

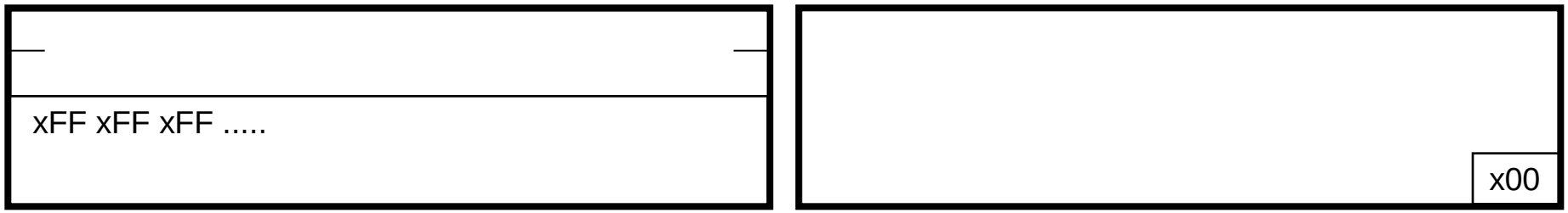
- after encryption:



- now the attacker removes the part containing the MAC and the padding, and submits the remaining message to a decryption oracle (length field in header is set accordingly)

A distinguishing attack

- after decryption:



- in case of M0:
  - xFF xFF ... valid padding → removed
  - remaining 32 bytes are interpreted as 12-byte message + 20-byte MAC
  - MAC is verified on 8-byte sequence number + 5-byte header + 12 byte message = 25 bytes < 56 bytes → 4 calls to function f

- in case of M1:
  - x00 valid padding → removed
  - remaning 287 bytes are interpreted as 267-byte message + 20-byte MAC
  - MAC is verified on 8-byte sequence number + 5-byte header + 267-byte message = 280 bytes → 8 calls to function f

Experimental results

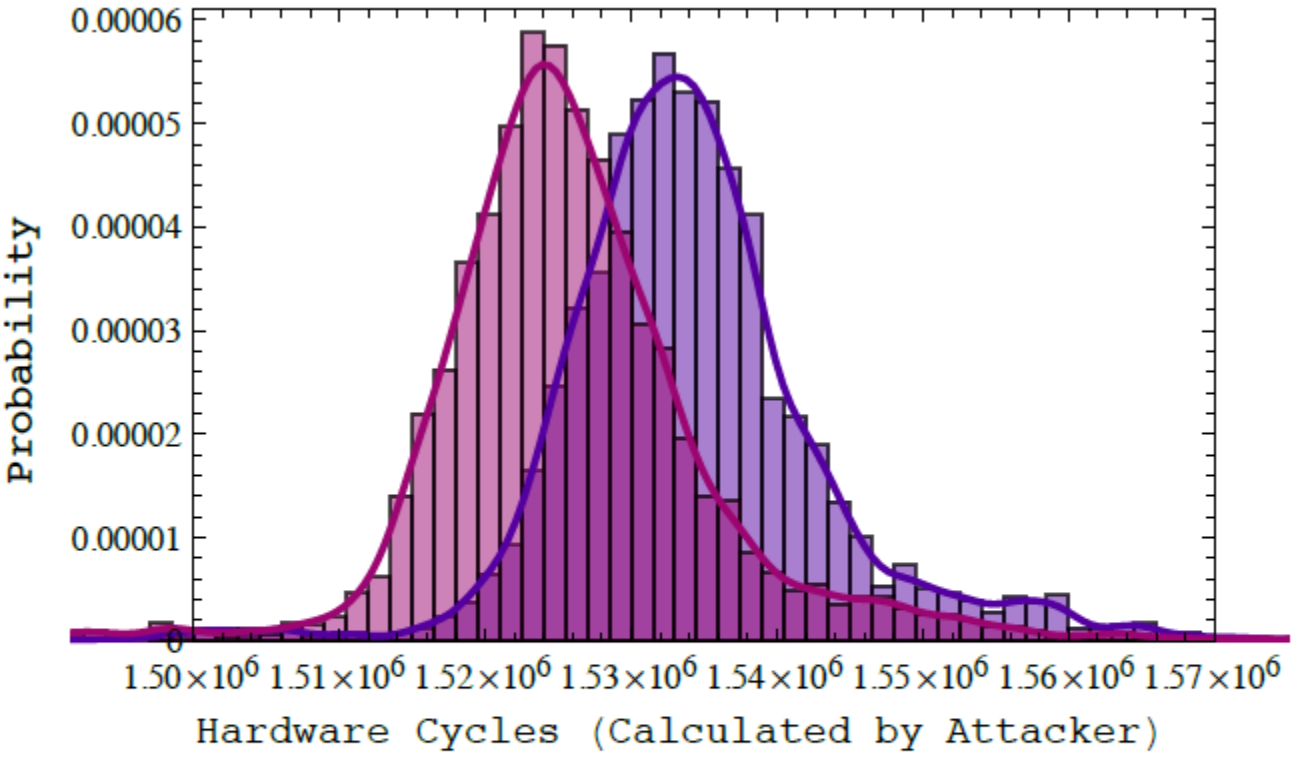


Figure 2: Distribution of timing values (outliers removed) for distinguishing attack on OpenSSL TLS, showing faster processing time in the case of M_0 (in red) compared to M_1 (in blue).

Lucky 13 attack (2013)

- similar to the padding oracle attack, but uses a different timing channel

- let C* be an encrypted block (16 bytes); $C^* = E_K(P^*)$
- the attacker wants to find P*
- the attacker sends C1 | C2 | R | C* (64 bytes) with proper TLS record header to a decryption oracle (e.g. TLS server)
- the oracle decrypts this into P1 | P2 | P3 | P4, where $P4 = D_K(C^*) + R = P^* + R$
- MAC verification fails, but ...

Lucky 13 attack (2013)

- there are 3 cases:
  1. P1 | P2 | P3 | P4 ends with invalid padding:
     - last 20 bytes are interpreted as MAC
     - remaining 44 bytes are interpreted as message
     - 44 +13 bytes (sqn and header) = 57 bytes go into MAC verification
     - >55 bytes → >4 calls to function f
  2. P4 ends with x00:
     - x00 removed as valid padding
     - next 20 bytes interpreted as MAC
     - remaining 64-21 = 43 bytes are interpreted as message
     - 43 + 13 bytes = 56 bytes go into MAC computation
     - > 55 bytes → >4 calls to function f
  3. P4 ends with a valid padding of length at least 2:
     - at least 2 bytes are removed as valid padding
     - message length is at most 42
     - 42 + 13 bytes (sqn and header) = 55 bytes go into MAC computation
     - < 56 bytes → 4 calls to function f

Lucky 13 attack (2013)

- if response time is „short”, then there is a valid padding of length at least 2 at the end of P1 | P2 | P3 | P4
- the case x01|x01 has the highest probability
- this means that the last two bytes of P* = x01|x01 XOR the last two bytes of R
- the rest can be figured out in the same way as in the Vaudenay attack

- if response time is „long”, than change last two bytes of R
- after at most 2^{16} trials, we get a short response time

Lucky 13 attack (2013)

- practical considerations
  - the TLS session is destroyed as soon as the attacker submits his very first attack ciphertext
    - mount a multi-session attack, where the same plaintext is repeated in the same position over many sessions (similar to the attack on the IMAP password)
  - the timing difference between the cases is very small, and so likely to be hidden by network jitter and other sources of timing difference
    - iterate the attack many times for each R value (similar to the distinguishing attack)
    - then perform statistical processing of the recorded times to estimate which value of R is most likely to correspond to case 3 („short” response time)

The problem of predictable IVs in CBC

- let $C_i = E_K(C_{i-1} + P_i)$ for some i (part of a CBC encrypted message), and let us assume that the attacker suspects that $P_i = P^*$
- the attacker predicts the IV of the next message, and submits a message with IV + C_{i-1} + P* as the first block to the oracle
- the oracle outputs a ciphertext with $E_K(IV + IV + C_{i-1} + P^*) = E_K(C_{i-1} + P^*)$ as the first block
- if the attacker's guess was correct (i.e., $P_i = P^*$), then the above first block is equal to C_i
- thus, the attacker can confirm his guess

BEAST attack (2011)

- in SSL v3.0 and TLS v1.0, the IV of the next message is the last ciphertext block of the previous message → IV is predictable (known)
- Can we take advantage of the known weakness of predictable IVs in practice?
- basic idea:
  - if all bytes of Pi are known except one byte, then that missing byte can be figured out by guessing and confirming the guess, and repeating this until it succeeds (at most 256 trials only)

BEAST attack (2011)

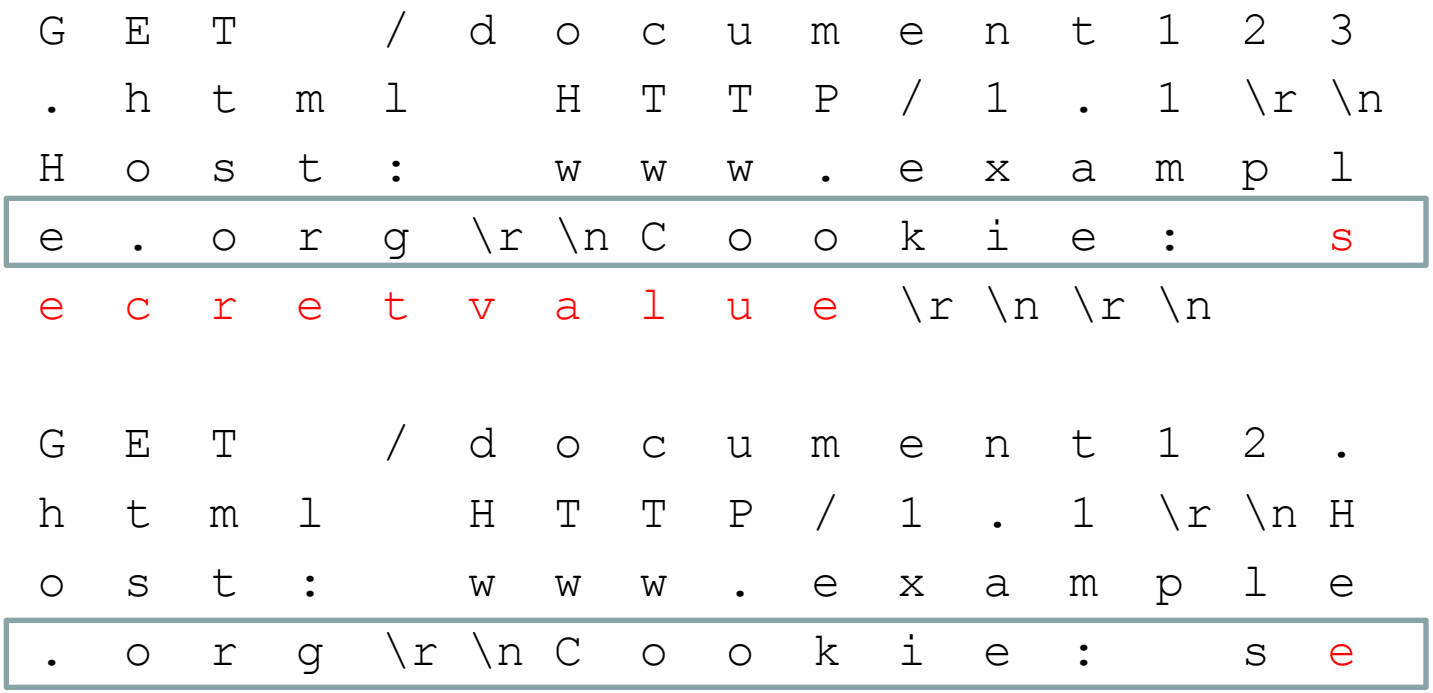
- algorithm to figure out the last byte:
  - let $C_i = E_K(C_{i-1} + P_i)$ for some i (part of a CBC encrypted message), and let us assume that the attacker knows all but the last byte of P_i ($P_i = P|p^*$, where p^* is the last byte)
  - let p = 0, and repeat the following:
    - predict the next IV, and submit a message with IV + C_{i-1} + P|p as the first block to the oracle
    - (the oracle outputs a ciphertext with $E_K(IV + IV + C_{i-1} + P|p) = E_K(C_{i-1} + P|p)$ as the first block)
    - check if the first block of the oracle output is equal to C_i
    - if so, then p is the unknown byte ($p = p^*$), and stop
    - otherwise increment p

- this algorithm finds p^* in at most 256 steps

BEAST attack (2011)

```
GET /document123.html HTTP/1.1
Host: www.example.org
Cookie: secretvalue
[blank line here]
```

attacker may be able to control the length of this string



BEAST attack (2011)

- Step 1:
  - Mallory forces Alice's browser to send an HTTPS POST (or GET) request to http://bob.com/AAAAAA.
  - Alice's browser uses SSL/TLS to derive a shared secret key k with Bob's web server, then it utilizes a block cipher in CBC mode to compute the encrypted request
  - AAA... is chosen in such a way that the first byte x of the unknown cookie header falls in the last position of one of the plaintext blocks, which then looks like P|x, where P is known to the attacker.
  - The encrypted request is sent to Bob's server by Alice's browser.

BEAST attack (2011)

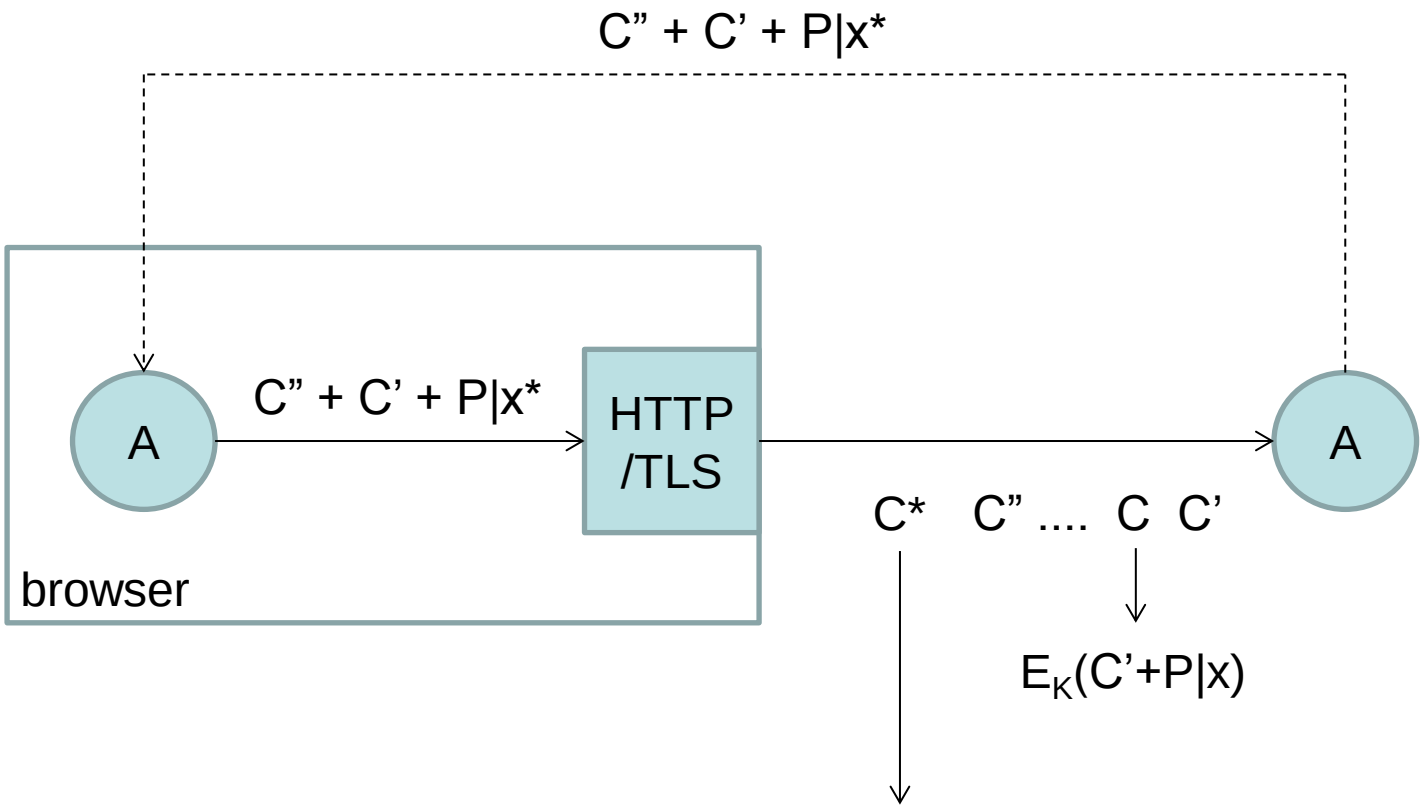
- Step 2:
  - Mallory captures the encrypted request.
  - Let the encrypted block containing the first unknown byte x of the cookie header be C, and let the preceding block be C'.
  - Mallory's goal is to obtain x.
  - Let W[1..L] be the set of allowed bytes in HTTP header and let i = 1.

BEAST attack (2011)

- Step 3:
  - Let IV be the last ciphertext block that Mallory captures, e.g., IV is the last block of the request captured in Step 2.
  - Mallory computes IV + C' + P|W[i], and arranges that this is encrypted and sent by Alice's browser (e.g., appends it to the existing request)
  - Alice's browser would compute and send C* = E_K(IV+IV+C'+P|W[i]) = E_K(C'+P|W[i])

- Step 4:
  - Mallory captures C*.
  - If C* = C, then she knows that x is equal to W[i].
  - Otherwise she increases i and goes back to step 3.

Illustration



E_K(C” + C”+ C’+P|x*) = E_K(C’+P|x*)
=? C
if so, then x = x*
else increament x*, and try again

BEAST attack (2011)

- summary of assumptions:
  - Mallory can capture encrypted HTTPS requests sent by Alice
  - Mallory can force Alice's browser to make arbitrary cookie-bearing requests to Bob's web server over HTTPS. Moreover, Mallory can also control the path value in these requests.
    - HTML5 WebSocket API
    - Java URLConnection API
    - Silverlight WebClient API
  - After making Alice's browser open some HTTPS requests to the server, Mallory can append arbitrary plaintext blocks to each ongoing request.

- a browser exploit that leverages browser features or plugins to help Mallory gain his desired privileges and run an agent inside the browser
  - BEAST = Browser Exploit Against SSL/*TLS*
- a network sniffer that intercepts SSL traffic, and sends ciphertext blocks to the agent
  - the agent can communicate with the network sniffer via basic socket programming

Summary on TLS

- overall:
  - TLS is a well designed protocol providing strong security
  - current version evolved by considering many known problems of previous versions
  - extremely important for securing web transactions, and for other applications

- TLS Record Protocol
  - uses strong algorithms (HMAC, AES)
  - good protection against passive eavesdropping
  - some protection against traffic analysis (random length padding)
  - possibly vulnerable to some padding oracle attacks
  - may be vulnerable to the BEAST attack (needs a browser exploit)

- TLS Handshake Protocol
  - uses strong algorithms (RSA, DH, DSS, well designed PRF based on HMAC)
  - protection against passive and active attacks
  - protection against some rollback attacks (cipher suite rollback, version rollback) and against dropping change_cipher_spec messages