TLS Record 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**)

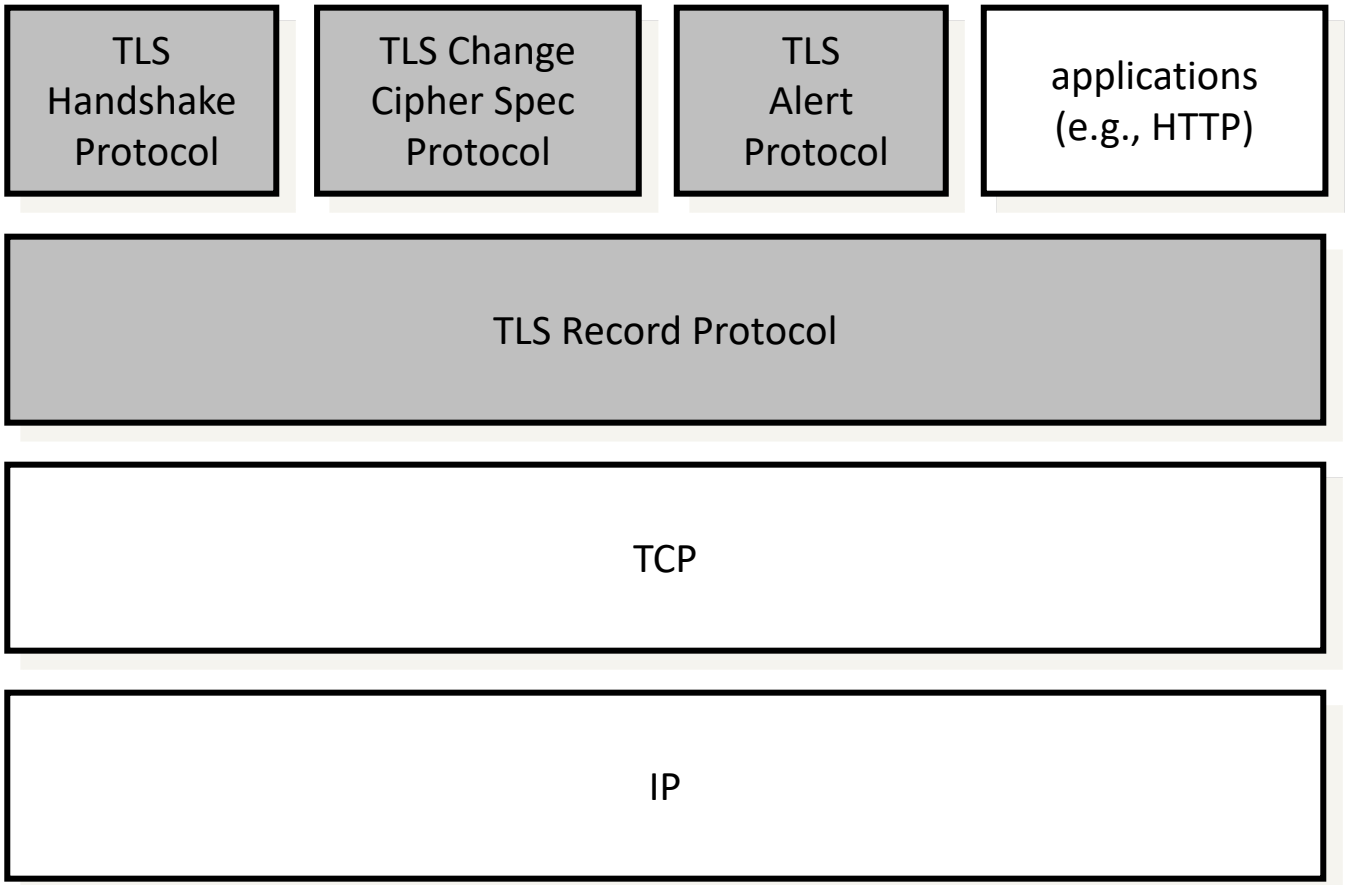
- message confidentiality, integrity, and replay protection:
  - » symmetric key cryptography is used for message encryption and MAC computation
  - » MAC covers a message sequence number → replay protection
  - » v1.2 supports a keyed MAC function and authenticated encryption modes, v1.3 only supports 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 history

- SSL – Secure Socket Layer
  - developed by Netscape in the mid 90's
  - SSL v3.0 has been implemented in many web browsers and web servers → become a de-facto standard

- TLS – Transport Layer Security
  - the IETF adopted SSL in 1999 under the name TLS
  - TLS v1.0 ~ SSL v3.0 with some design errors corrected
  - further modifications due to attacks discovered later → v1.1 → v1.2 → v1.3
  - TLS v1.2 is specified in RFC 5246
  - TLS v1.3 is specified in RFC 8446 (accepted in 2018)

TLS v1.2 sub-protocols



TLS v1.2 sub-protocols

- TLS Handshake Protocol
  - server authentication and optional client authentication
  - key exchange
  - negotiation of security algorithms and parameters
- TLS Record Protocol
  - fragmentation
  - compression
  - message authentication and integrity protection
  - replay protection
  - confidentiality
- TLS Alert Protocol
  - alert messages (fatal errors and warnings)
- TLS Change Cipher Spec Protocol
  - a single message that indicates the end of the key exchange
  - triggers a state change at the parties

TLS sessions and connections

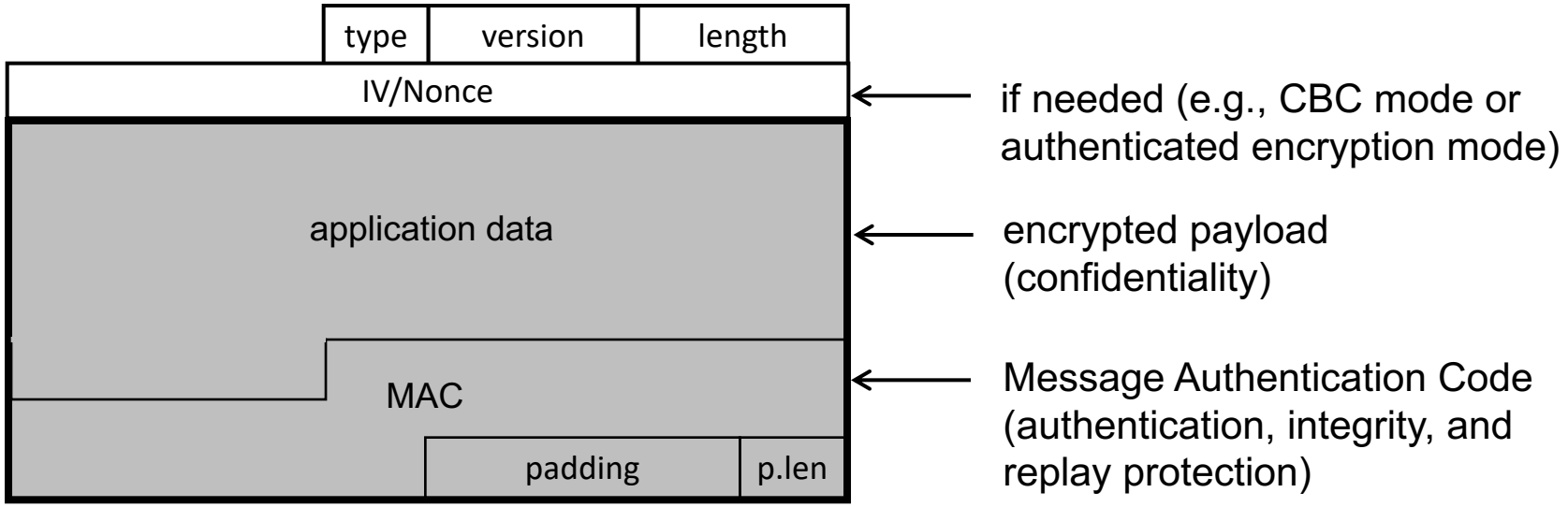
- a TLS session is a security association between a client and a server
- sessions are stateful; the session state includes:
  - the negotiated security algorithms and parameters
  - exchanged certificates (if any)
  - a master secret shared between the client and the server (established during a full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

TLS v1.2 Alert Protocol

- each alert message consists of 2 fields (bytes)
- first field (byte): “warning” or “fatal error”
- second field (byte):
  - in case of fatal errors:
    - » unexpected_message
    - » handshake_failure
    - » **decryption_failure** (failed padding verification in earlier versions of TLS)
    - » **bad_record_MAC** (failed MAC verification in earlier versions of TLS)
    - » …
  - in case of warnings:
    - » 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 v1.2 Record Protocol



- TLS padding:
  - last byte is the length n of the padding (not including the last byte)
  - padding length is random (0-255 bytes), but such that the length of the padded message is a multiple of the block length of the blockcipher used
  - all padding bytes have value n
  - examples for correct message tails: `x00`, `x01x01`, `x02x02x02`, …
  - verification: if the last byte is n, then verify if the last n+1 bytes are all n
  - if verification is successful, remove the last n+1 bytes, and proceed with the verification of the MAC

MAC function

- when not authenticated encryption is used, message integrity protection and origin authentication is provided by a keyed MAC
- the MAC value is computed before the encryption
- input to the MAC computation:
  - MAC_write_key
  - **seq_num** |type | version | length | payload (compressed fragment)
    - » sequence numbers are maintained at both sides of the connection
    - » they are not sent explicitly in the TLS Record header (implicit sequence numbering)
- supported algorithms: HMAC with MD5, SHA1, SHA256

Encryption

- stream ciphers
  - no IV and padding
  - no re-initialization of the cipher for individual messages
  - supported algorithm: (RC4), ChaCha20
- block ciphers in CBC mode
  - IV must be a random value
    - » until TLSv1.0, IV was the last cipher block of the previous message
  - padding (see previous slide)
  - supported algorithms: (3DES-EDE, IDEA), Camellia, ARIA, AES (128, 256)
- authenticated encryption modes
  - nonce is carried in the IV field
  - no padding
  - jointly encrypted and authenticated message is computed from
    - » client_write_key or server_write_key
    - » nonce
    - » cleartext payload
    - » seq_num | type | version | length as associated data
  - supported algorithms: AES-GCM (128, 256), AES-CCM (128, 256)

Attacks on the TLS Record Protocol (up to v1.2)

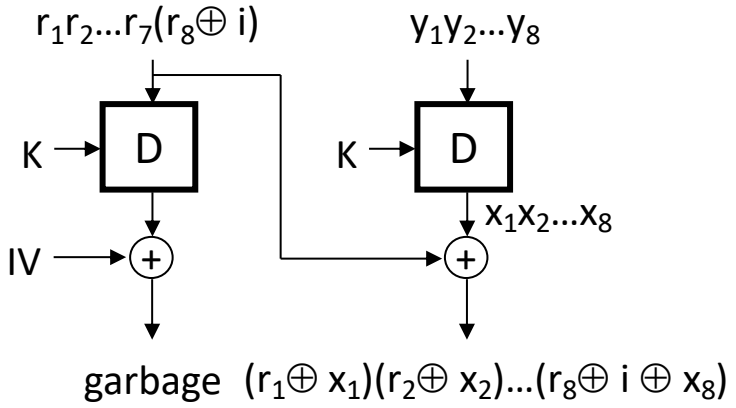
- chosen ciphertext (padding oracle) attacks
  - difficulties with the original PO attack in case of TLS
  - successful application of the PO attack in a multi-session setup
  - fixing TLS and the Lucky 13 attack

- chosen plaintext attacks
  - BEAST attack
  - CRIME, BREACH, and TIME attacks

Reminder on the padding oracle attack

- padding oracle
  - assume that the attacker can access a CBC decryption oracle
  - the oracle receives ciphertext messages
  - it decrypts them in CBC mode, and checks the padding
  - it responds with a single bit indicating whether the padding was correct or not
  - thus, we assume a chosen ciphertext attacker model, where the oracle does not return the full plaintext, but only some partial information about the result of the decryption

- by sending carefully crafted messges (R|Y) to the oracle, one can decrypt any encrypted block Y

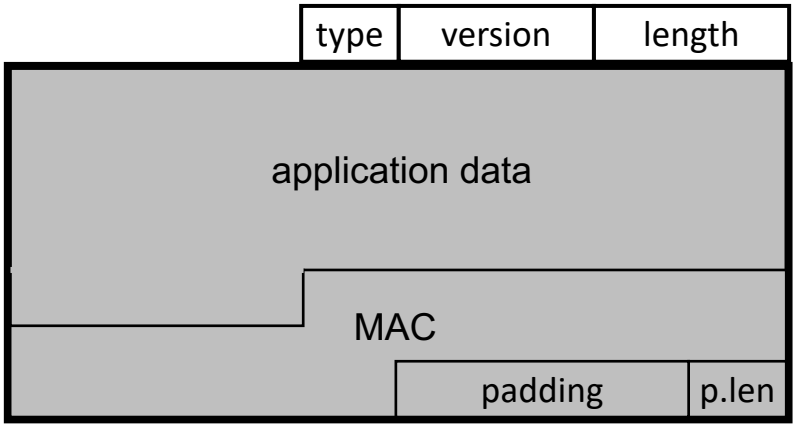


Padding oracle attack on TLS v1.0

- send a random message to a TLS server
- the server will drop the message with overwhelming probability
  - after decryption 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)

→ server works as a padding oracle !

- the attacker can try to apply the padding oracle attack on CBC
- there are some problems in practice, though



PO attack on TLS v1.0 in practice (2003)

- alert messages are encrypted → BAD_RECORD_MAC and DECRYPTION_FAILED cannot be easily distinguished

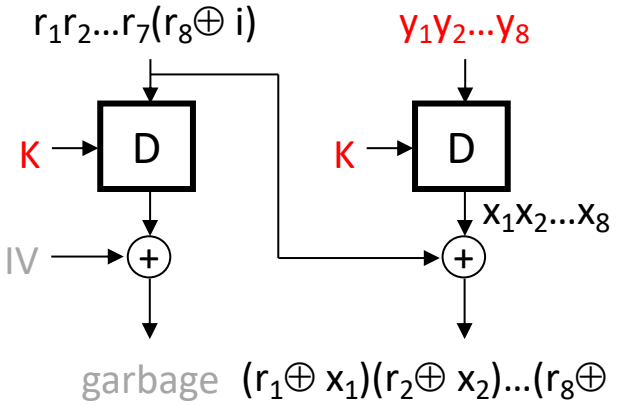
  solution:

  - measure timing between oracle call and oracle response
  - BAD_RECORD_MAC takes more time than DECRYPTION_FAILED, as it also involves the MAC verification time

- BAD_RECORD_MAC and DECRYPTION_FAILED are fatal errors → connection is closed after one oracle call

  solution:

  - mount a **multi-session attack**
  - works when the same secret (e.g., a password) is sent periodically to a server using TLS
  - each session uses a different key, but the secret is always the same!

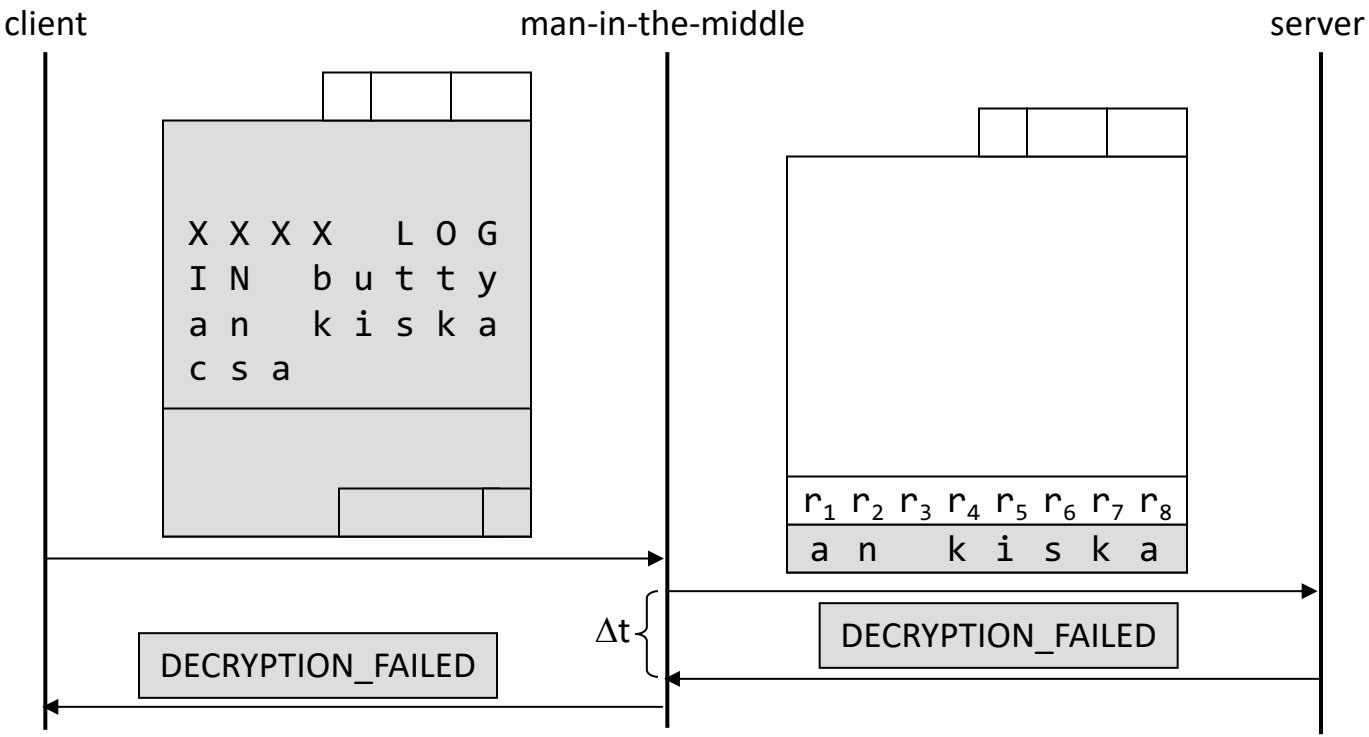


- padding is verified on the plaintext
- if X contains the password, we don't mind if K, and thus, Y changes in each session

Example

- IMAP over TLS
  - Outlook Express checked for new mail on the server periodically (e.g., every 5 minutes)
  - each time the same username / password pair was sent for every folder:

    XXXX LOGIN “username” “password” \x0D\x0A
- it is possible to break the password using the attack as follows:



Eliminating the timing channel in TLS v1.1

- when badly formatted padding is encountered during decryption, a MAC check must still be performed on ***some data*** to prevent known timing attacks
  - MAC verification is always done
  - only BAD_RECORD_MAC errors are sent

- which data?

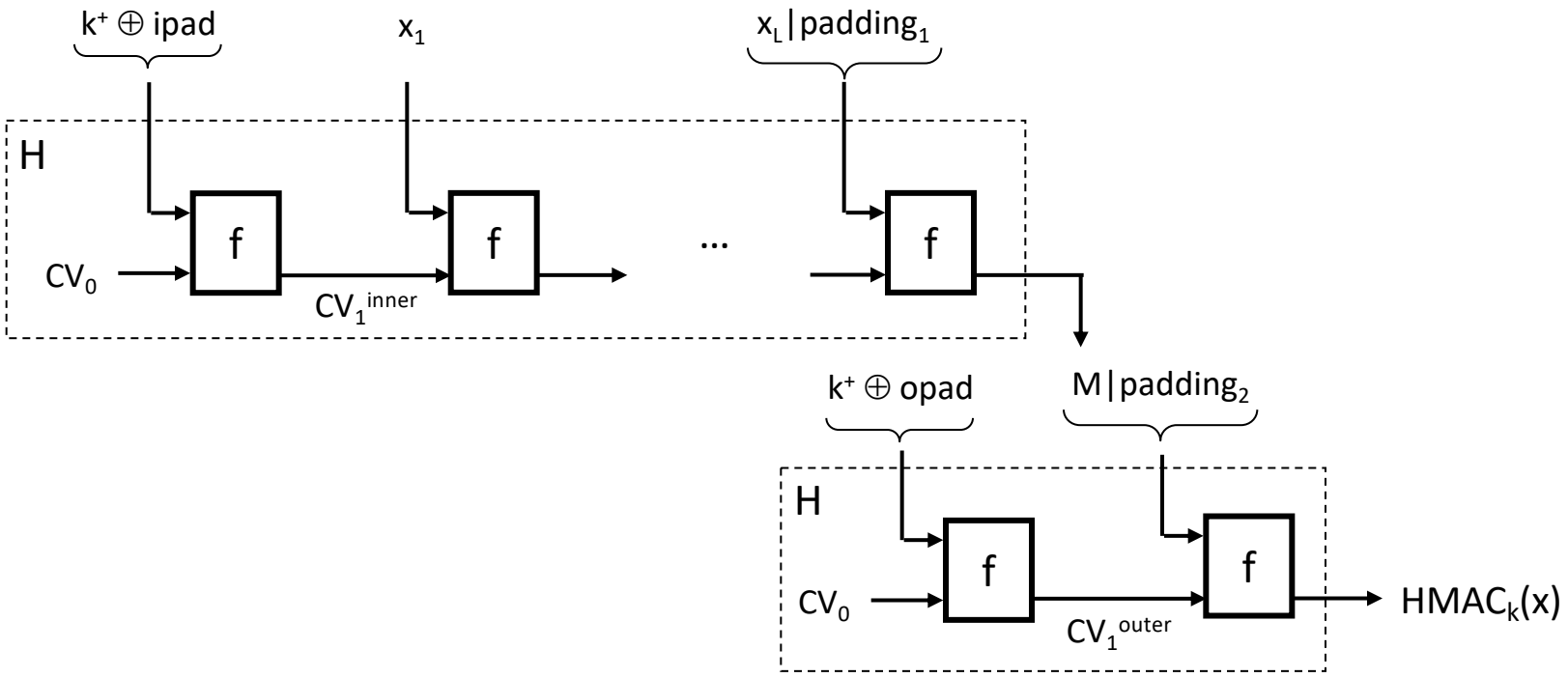
- TLS 1.1 and 1.2 recommend checking the MAC as if there was a zero-length padding in the message

  „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."

  wrong? → Lucky 13 attack (2013)

Reminder on HMAC

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



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 $\leq$ 55 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 function f
  - this is a timing channel!

Lucky 13 attack (2013)

- similar to the padding oracle attack, but uses the previously introduced 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)
  - R is a random block controlled by the attacker
- the oracle decrypts this into P1 | P2 | P3 | P4, where $P4 = D_K(C^*) + R = P^* + R$
- MAC verification fails, but there are 3 cases ...

Lucky 13 attack (2013)

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 computation
   - > 55 bytes → 5 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 interpreted as message
   - 43 + 13 bytes (sqn and header) = 56 bytes go into MAC computation
   - > 55 bytes → 5 calls to function f
3. P1 | P2 | P3 | 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 bytes
   - $\leq$ 42 + 13 bytes (sqn and header) = 55 bytes go into MAC computation
   - $\leq$ 55 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 `x01x01` has the highest probability
  - this means that the last two bytes of P* are equal to `x01x01` XORed to 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”, then change last two bytes of R
  - after at most 2^{16} trials, we get a short response time ...

Lucky 13 – 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 value of R
  - 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
- problem: guessing an entire block is infeasible in practice if the block length is large enough

BEAST idea

HTTP request:

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

P_1	G	E	T		/	d	o	c	u	m	e	n	t	1	2	3
P_2	.	h	t	m	l		H	T	T	P	/	1	.	1	\r	\n
P_3	H	o	s	t	:		w	w	w	.	e	x	a	m	p	l
P_4	e	.	o	r	g	\r	\n	C	o	o	k	i	e	:		s
P_5	e	c	r	e	t	v	a	l	u	e	\r	\n	\r	\n		

- in the HTTP request sent out by the browser, the 4th block will have a single unknown byte: the first letter of the cookie
- the attacker can obtain the first byte of the cookie by repeatedly checking different guesses and taking advantage of predictable IVs (as we saw before)

BEAST idea

HTTP request:

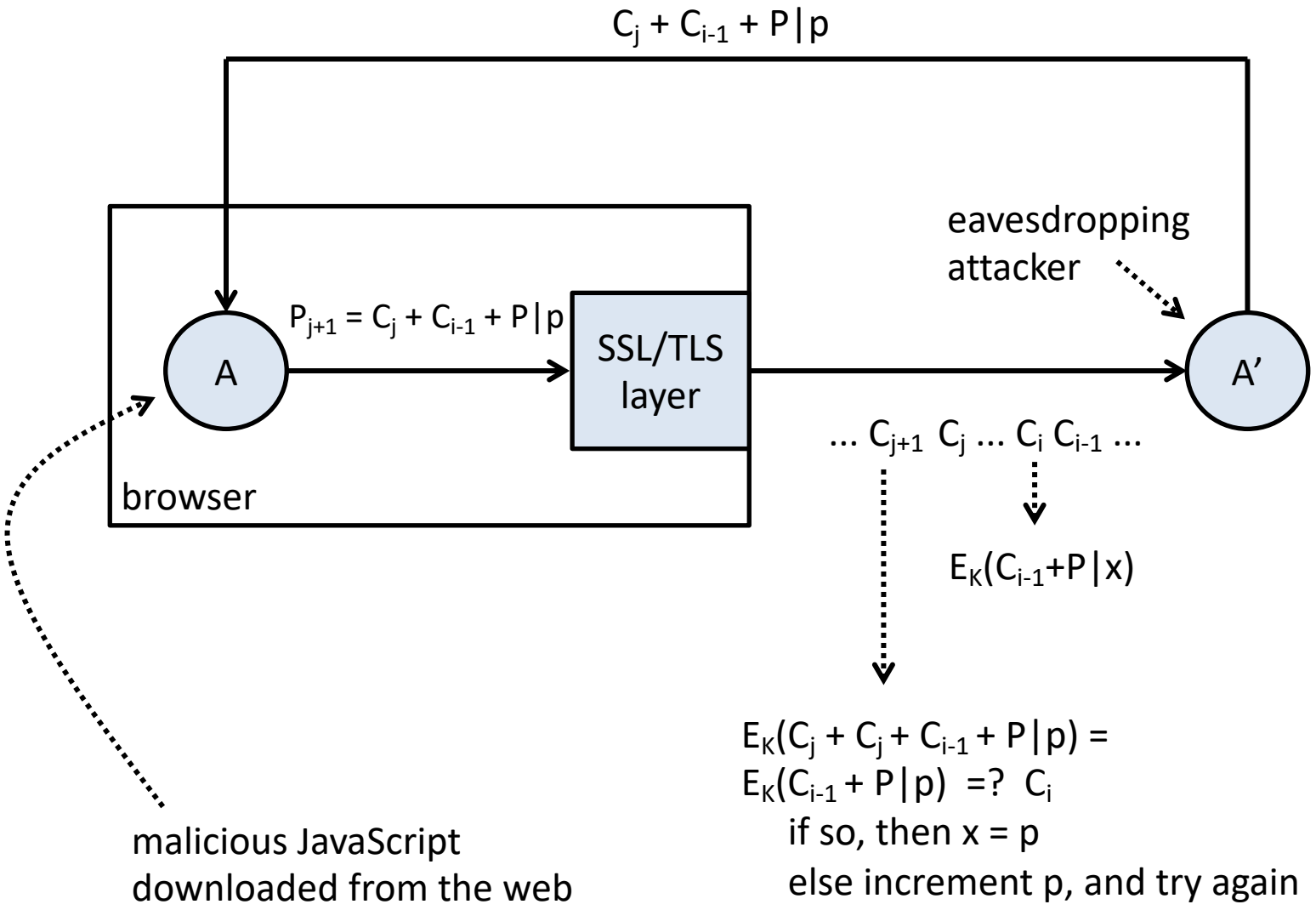
```
GET /document123.html HTTP/1.1
Host: www.example.org
Cookie: secretvalue
[blank line here]
```

the attacker may adjust the block boundary by controlling the length of this string

```
G E T   / d o c u m e n t 1 2 3
. h t m l   H T T P / 1 . 1 \r \n
H o s t :   w w w . e x a m p l
e . o r g \r \n C o o k i e :   s
e c r e t v a l u e \r \n \r \n
```

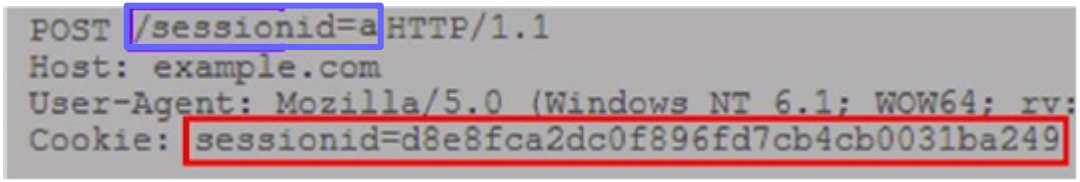
P_1	G	E	T		/	d	o	c	u	m	e	n	t	1	2	.
P_2	h	t	m	l		H	T	T	P	/	1	.	1	\r	\n	H
P_3	o	s	t	:		w	w	w	.	e	x	a	m	p	l	e
P_4	.	o	r	g	\r	\n	C	o	o	k	i	e	:		s	e
P_5	c	r	e	t	v	a	l	u	e	\r	\n	\r	\n			

BEAST attack



CRIME attack (2012)

- CRIME = Compression Ratio Info-leak Made Easy
- exploits information leakage through a side-channel provided by TLS compression (or compression of HTTP requests)
  - the attacker tries to guess the value of a cookie which is automatically appended by the browser to HTTP requests to a target site
  - the attacker can put his guesses in the requested path (which is part of the HTTP request) or in any request parameter that he can control
  - if the guess is correct, then the same string occurs twice in the request (in the path and in the cookie header) → compression ratio improves
  - the attacker can observe the size of the ciphertext → smaller request means good guess!

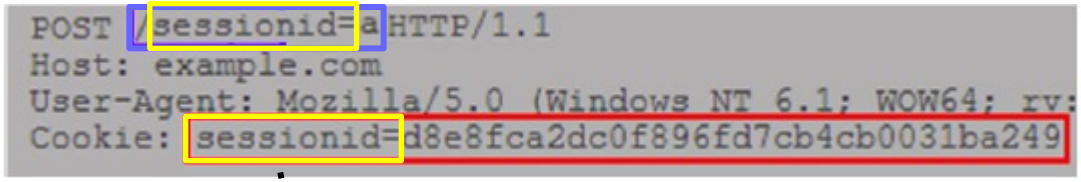


TLS compression

- based on DEFLATE
  - combination of LZ77 and Huffman coding
- LZ77 is used to eliminate the redundancy of repeating sequences
  - replace repeated occurrences of data with references to an earlier occurence
  - references are (distance, length) pairs
  - example: Google is so googley → Google is so g(-13, 5)y
- Huffman coding is used to eliminate the redundancy of repeating symbols
  - replace common bytes with shorter codes
  - examples:
    - » e and a are frequent chars, they can be encoded on 3 bits
    - » x and q rare chars, can be encoded on 9 or more bits

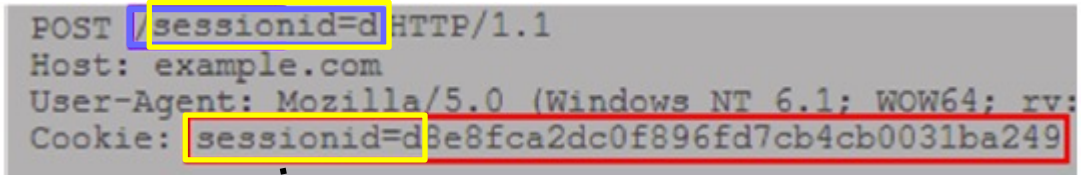
CRIME illustrated

wrong guess:



replaced with a (distance, length) pair by LZ77

good guess:



1 byte longer sub-string is replaced
with a (distance, length) pair by LZ77
→ compressed message will be 1 byte shorter

- after correctly guessing the first character, the attacker moves on to guess the next character in the same manner, until all the characters of the secret cookie are revealed

BREACH attack (2013)

- BREACH = Browser Reconnaissance and Exfiltration via Adaptive Compression of Hypertext

- similar to CRIME, but instead of HTTP requests, BREACH targets HTTP responses
  - after CRIME was published, TLS record and HTTP request compression were recommended to be disabled
  - however, most web servers still used HTTP response compression to save bandwidth

- main idea:
  - many web applications routinely deliver secrets (e.g., CSRF tokens) in the bodies of HTTP responses
  - it is also common for web applications to reflect user input (e.g., URL parameters) in HTTP response bodies
  - an attacker can make guesses for a secret in a URL parameter
  - if guess is correct, resulting HTTP response (which reflects the guess and contains the secret) can be compressed better

BREACH example

- Microsoft Outlook Web Access uses a CSRF token called ”canary”

- if a user with an active session makes the request:

```
GET /owa/?ae=Item&t=IPM.Note&a=New&id=canary=<guess>
```

- she receives the following HTTP response body:

```
<span id=requestUrl>https://malbot.net:443/owa/forms/
   basic/BasicEditMessage.aspx?ae=Item&amp;t=IPM.Note&
   amp;a=New&amp;id=canary=<guess></span>
...
<td nowrap id="tdErrLgf"><a href="logoff.owa?
   canary=d634cda866f14c73ac135ae858c0d894">Log
   Off</a></td>
```

TIME attack (2013)

- TIME = Timing Info-leak Made Easy

- similar to BREACH, but TIME uses timing information to infer the size of the compressed payload
  - better compression ratio → smaller message → shorter transmission time
- advantage:
  - no need for the attacker to observe the size of encrypted messages (needs eavesdropping privilege)
  - attacker's agent inside the victim browser can make a guess AND measure the response time of the server
  - enough to download a malicious JavaScript !!!

Summary on TLS Record protocol attacks

- chosen ciphertext (padding oracle) attacks
  - difficulties with the original PO attack in case of TLS
    - » encrypted alert messages --» figure out oracle response from response time
    - » decryption failure is a fatal error, connection is closed after one trial --» mount a multi-session attack (same secret encrypted under different keys)
  - newer versions of TLS try to eliminate timing channels from the oracle by requiring MAC verification even in case of a wrong padding
  - the Lucky 13 attack exploits the remaining small timing difference between a correct padding of length at least 2 and wrong padding

- chosen plaintext attacks
  - BEAST attack
    - » iterative guessing of secret cookies
    - » requires chosen boundary, blockwise, and eavesdropping privileges
    - » blockwise privilege may be provided by APIs to JavaScript code
  - CRIME, BREACH, and TIME attacks
    - » iterative guessing of secret cookies or CSRF tokens
    - » all exploit a side channel based on compression ratio (request or response)

TLS v1.3 Record Protocol

- main differences wrt. TLS v1.2:
  - old algorithms (RC4, DES, IDEA, Camellia, MD5, SHA1) are removed
  - compression is removed (was exploited by plaintext guessing attacks)
  - CBC mode is removed (was vulnerable to padding oracle attacks)
  - block ciphers must be used in authenticated encryption modes (GCM, CCM)

Cipher			Protocol version						Status
Type	Algorithm	Nominal strength (bits)	SSL 2.0	SSL 3.0 [n 1][n 2][n 3][n 4]	TLS 1.0 [n 1][n 3]	TLS 1.1 [n 1]	TLS 1.2 [n 1]	TLS 1.3	
Block cipher with mode of operation	AES GCM[54][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	Secure	Defined for TLS 1.2 in RFCs
	AES CCM[55][n 5]		N/A	N/A	N/A	N/A	Secure	Secure	
	AES CBC[n 6]		N/A	Insecure	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	Camellia GCM[56][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	N/A	
	Camellia CBC[57][n 6]		N/A	Insecure	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	ARIA GCM[58][n 5]	256, 128	N/A	N/A	N/A	N/A	Secure	N/A	
	ARIA CBC[58][n 6]		N/A	N/A	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	SEED CBC[59][n 6]	128	N/A	Insecure	Depends on mitigations	Depends on mitigations	Depends on mitigations	N/A	
	3DES EDE CBC[n 6][n 7]	112[n 8]	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	
	GOST 28147-89 CNT[53][n 7]	256	N/A	N/A	Insecure	Insecure	Insecure	N/A	Defined in RFC 4357
	IDEA CBC[n 6][n 7][n 9]	128	Insecure	Insecure	Insecure	Insecure	N/A	N/A	Removed from TLS 1.2
	DES CBC[n 6][n 7][n 9]	56	Insecure	Insecure	Insecure	Insecure	N/A	N/A	
		40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	Forbidden in TLS 1.1 and later
	RC2 CBC[n 6][n 7]	40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	
Stream cipher	ChaCha20-Poly1305[64][n 5]	256	N/A	N/A	N/A	N/A	Secure	Secure	Defined for TLS 1.2 in RFCs
	RC4[n 11]	128	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	Prohibited in all versions of TLS by RFC 7465
		40[n 10]	Insecure	Insecure	Insecure	N/A	N/A	N/A	
None	Null[n 12]	–	Insecure	Insecure	Insecure	Insecure	Insecure	N/A	Defined for TLS 1.2 in RFCs

https://en.wikipedia.org/wiki/Transport_Layer_Security

Control questions

- What security services does TLS provide?
- What are the TLS v1.2 sub-protocols, and how do they fit in the TCP/IP network stack?
- What are TLS sessions and connections? How are they related and what was the design motivation for this relationship?
- How does the TLS v1.2 Record Protocol work?
  - message format
  - MAC computation
  - encryption and supported padding format
- What are the main ideas of the following attacks on the Record Protocol?
  - Multi-session padding oracle attack
  - Lucky 13 attack
  - BEAST attack
  - BREACH, CRIME, and TIME attacks