IT Security - Task 2

Group G41

Name	E-Mail
LORENZ Peter	peter.lorenz@student.tugraz.at
STRAMER Jorrit	jstramer@student.tugraz.at

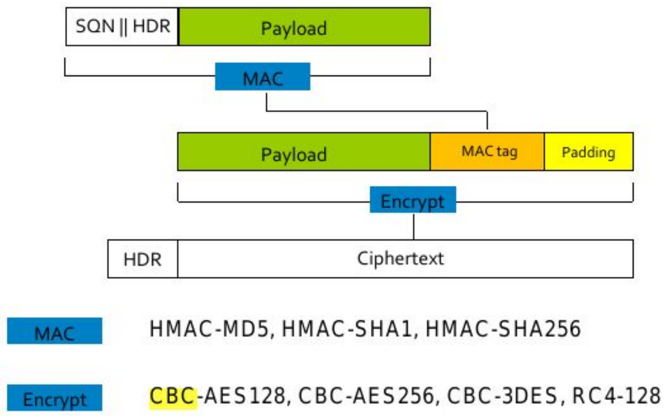
Theoretical Questions

a. What purpose fulfill AES-CBC and HMAC in TLS?

TLS works basically two protocols. The first one is the “handshake protocol” to establish a connection to a server. During this phase 3 important steps are done:

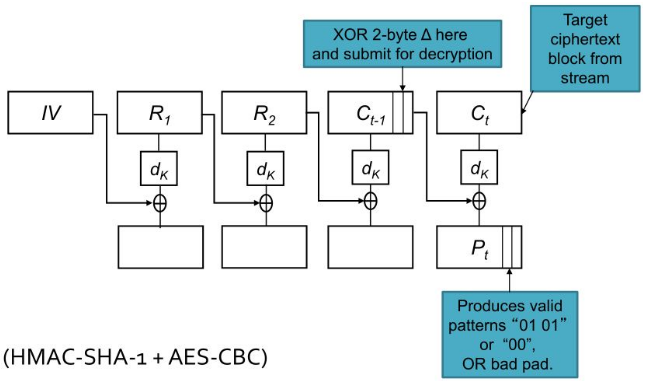
1. Negotiate ciphersuite
2. Authenticate
3. Establish key used in the Record Protocol

The second protocol, called “record protocol”, provides confidentiality and authenticity of application layer data using keys from handshake protocol. In the graphic below is shown when is used HMAC and when CBC.


SQN || HDR
Payload
MAC
Payload
MAC tag
Padding
Encrypt
HDR
Ciphertext
MAC
HMAC-MD5, HMAC-SHA1, HMAC-SHA256
Encrypt
CBC-AES128, CBC-AES256, CBC-3DES, RC4-128

HMAC: To compute the MAC, take the sequence number, the header and the message, then HMAC them using the shared integrity key.
CBC: To encrypt the message aka payload, MAC Tag and Padding.

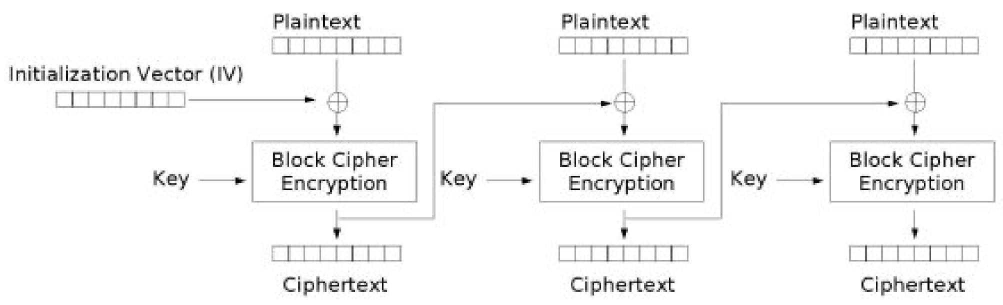
b. Explain how ciphersuites using AES-CBC and HMAC work in principle.


XOR 2-byte Δ here and submit for decryption
Target ciphertext block from stream
IV
R_1
R_2
C_{t-1}
C_t
d_K
P_t
Produces valid patterns “01 01” or “00”, OR bad pad.

(HMAC-SHA-1 + AES-CBC)

AES-CBC + HMAC is quite slow.

AES-CBC:



You have an initialization vector which you XOR the first block of plaintext against. Then this block of plaintext is encrypted. The next block of plaintext is XORed against the last encrypted block before this block is encrypted.

AES always works on 128 bit blocks. CBC mode means that the 128 bit message block first XORed with the 128 bit cipher block of the previous block and then AES encrypted.

HMAC:

- Computed on SQN | HDR | Payload
- Adding >9 bytes of padding and iteration of hash compression function, e.g. MD5, SHA-1, SHA-256
- Running time of HMAC depends on L, the byte length of SQN || HDR || Payload
  - L ⇐ 55 bytes: 4 compression functions
  - 56 ⇐ L ⇐ 119: 5 compression functions
  - 120 ⇐ L ⇐ ?: 6 compression functions

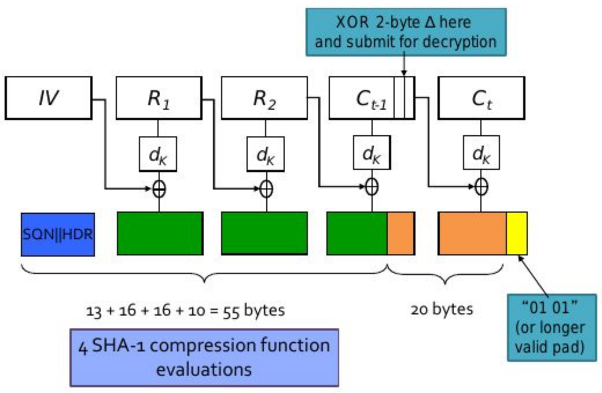
To compute HMAC over the data `text' we perform: H(K XOR opad, H(K XOR ipad, text))[1]

1. Append zeros to the end of K to create a B byte string (e.g., if K is of Length 20 bytes and B=64, then K will be appended with 44 zero bytes 0x00)
2. XOR (bitwise exclusive-OR) the B byte string computed in step (1) with ipad
3. Append the stream of data 'text' to the B byte string resulting from step (2)
4. Apply H to the stream generated in step (3)
5. XOR (bitwise exclusive-OR) the B byte string computed in step (1) with opad
6. Append the H result from step (4) to the B byte string resulting from step (5)

c. Explain the principles of the attack[2].

Lucky Thirteen: The name comes from the fact that for the computation of the TLS HMAC, additionally to the data, 13 more bytes are used (5 bytes of TLS header and 8 bytes of TLS sequence number).

[1] Copied from https://www.ietf.org/rfc/rfc2104.txt
[2] http://www.isg.rhul.ac.uk/tls/Lucky13.html


XOR 2-byte Δ here and submit for decryption
IV
R1
R2
Ct-1
Ct
dK
SQN||HDR
13 + 16 + 16 + 10 = 55 bytes
4 SHA-1 compression function evaluations
20 bytes
"01 01" (or longer valid pad)

Given is an encrypted message. First the message is split into multiple 16 byte blocks (because AES has a blocksize of 128 bits = 16 bytes). To recover one plaintext block, two consecutive ciphertext blocks are needed. According to the paper we use 4 block of 16 bytes each (C1 || C2 || C3 || C4) for the attack, whereby C3 and C4 are the two consecutive ciphertext blocks and C1 and C2 are irrelevant (we used all 0's) and the attack tries to recover P4.

As shown in the picture above, C3 is XORed with the decrypted C4 block, which results in the plaintext block P4. Therefore if an attacker changes C3, P4 will also be changed accordingly.

To recover the last two bytes, an attacker tries to guess two deltas, which are XORed with the last two bytes of C3, such that the last two bytes of P4 are 0x01 || 0x01 aka. a valid padding. This is done by bruteforcing the deltas and measuring the time it took for the decryption of the blocks. If a valid padding is used, the decryption is faster than otherwise.

To recover the following bytes, first the attacker has to change all previous d

After receiving an encrypted messages. This messages is divided in four 16 bytes packages. And tried via brute force to find the right delta. There are 2^{16} possibilities for the last two bytes in 3th package and then submitted it for decryption. Via rdtsc it is taken the time into account. After calculating the mean of the measurements, it yields the last two bytes for decryption. Then, all 16 bytes of the encrypted message can be used as 4th block and decrypted.

d. Why is it important to perform all checks during the decryption in constant time? If constant-time decryption was not possible, would random delays prevent the attack?

It is really needed constant-time decryption for TLS-CBC. Because of the time measurements, it can be found out the right two last bytes of the third block for decryption. Otherwise, the mean would be wrong.

e. While using the same ciphersuite, is it possible to remove the timing side-channel?

Yes, it is difficult, but not impossible[3]. It is recommend in the paper to use AES-GCM instead of the CBC mode.

f. **Compare the use of the block cipher construction form this Task to the use of Authenticated Encryption in Task 1. Is it possible to model the Encrypt-Then-MAC paradigm as Authenticated Encryption scheme?**
Yes, it is possible.

g. **Explain the difference between Encrypt-then-MAC and MAC-then-Encrypt. Would MAC-then-Encrypt prevent the attack?**

Encrypt-then-MAC = Encrypt the plaintext, MAC the ciphertext + IV then append it to the ciphertext.
MAC-then-Encrypt = MAC the plaintext then append the MAC to the plaintext then encrypt it all. (TLS)

- Encrypt-then-MAC:
    - Provides integrity of Ciphertext. Assuming the MAC shared secret has not been compromised, we can state whether the public key is authentic or not. E.g., in public key cryptography anyone can send you messages. EtM ensures you only read valid messages.
    - Plaintext integrity.
    - The MAC does not provide any information on the plaintext.
- MAC-then-Encrypt:
    - Does not provide any integrity on the ciphertext, since we have no way of knowing until we decrypt the message whether it was indeed authentic or spoofed.
    - Plaintext integrity.
    - The MAC cannot provide any information on the plaintext either, since it is encrypted.

No, it does not work with MtE, which is TLS doing and where the padding oracle occurs, because the Decryption Algorithm has to work out how much there should be.

The goal is, to protect against padding oracles, that there is not notifiable difference if the padding is wrong. It is possible by using EtM construction, where MAC is applied to the ciphertext. If MAC fails, the padding does not change length. If MAC is correct, it is unlikely that the padding has been tampered.

h. **POSIX.1-2008 defines multiple clocks which measure different notions of “time”. It specifies system wide clocks to measure real time, per-process or per-thread CPU-time clocks, and others. Compare the clocks specified in POSIX.1-2008 and**

[3] http://www.isg.rhul.ac.uk/tls/TLStiming.pdf

compare to the timing information retrieved using the timestamp counter. Can the timestamp counter be replaced with any of the standardized clocks?

The type clockid_t represents a specific POSIX clock:

1. **CLOCK_MONOTONIC**: a monotonically increasing clock, that is not settable by any process.
2. **CLOCK_PROCESS_CPUTIME_ID**: This clock uses the TSC register.
3. ***CLOCK_REALTIME***: It is the system-wide real-time (wall-time) clock. Setting this clock requires special privileges. *This clock cannot be used for the timing attack*: *the real-world time elapsed between method entry and method exit. If there are other threads/processes concurrently running on the system, they can affect the results.*

   Whereas, the others are CPU time - the time actually spent by CPU executing method code.

4. **CLOCK_THREAD_CPUTIME_ID**: Similar to the per-process clock, but unique to each thread in a process.

Notes:

http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf p15

“The RDTSCP instruction waits until all previous instructions have been executed before reading the counter. However, subsequent instructions may begin execution before the read operation is performed.”