PADDING ORACLE ATTACK LAB

SEED Lab #5

Identification

- **Group nº3**
  - **José Rodrigues** : 201708806
  - **Paulo Ribeiro** : 201806505
  - **Pedro Ferreira** : 201806506

Task 1:

- After creating three files of different sizes (5, 10, and 16 bytes), we encrypted them using the 128-bit AES with **CBC** mode and verified that their sizes had changed to be an exact multiple of the block size (composed of 16 bytes). Notice that in the last file, despite the size of the file already being a multiple of the block size, padding still occurs, and the encrypted file ends up with the size of the next multiple of the AES block size, which in this case is 32. This happens since PKCS padding rules say that padding is always applied.

Original file size	Encrypted file size
5 bytes	16 bytes
10 bytes	16 bytes
16 bytes	32 bytes

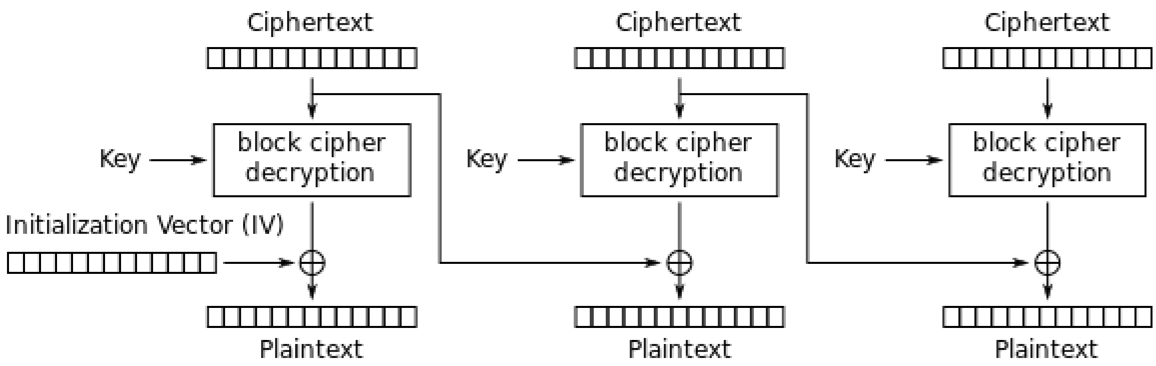
- After decrypting the three files without removing the padding that was added (using `-no-pad`), we verify that each padding byte has a value equal to the total number of padding bytes that are added. In the first case, the value `0b` was used as the padding value since it corresponds to the eleven extra bytes added during the padding. The same happens for the other two files (`06` for 6 added bytes and `10` for 16 added bytes).

```
[04/03/22]seed@VM:~/.../Files$ hexdump -C df1.txt
00000000  31 32 33 34 35 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |12345...........|
00000010
[04/03/22]seed@VM:~/.../Files$ hexdump -C df2.txt
00000000  31 32 33 34 35 36 37 38  39 30 06 06 06 06 06 06  |1234567890......|
00000010
[04/03/22]seed@VM:~/.../Files$ hexdump -C df3.txt
00000000  31 32 33 34 35 36 37 38  39 30 31 32 33 34 35 36  |1234567890123456|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
```

Task 2:

- The attack starts by modifying the last byte of the first block of the ciphertext (`C1[15]`) into a new version `CC1[15]`, which will be sent to the oracle for decryption, which is obtained by iterating from 0 to 255 (all possible byte values) and checking which one of them returns a signal of valid padding.

- After knowing this value, the last byte of the plaintext is obtained by calculating `P2[15] = PAD ⊕ C1[15] ⊕ CC1[15]`, where `C1` is the known ciphertext of the secret message and `PAD` is the value we want the last byte of the plaintext to decrypt to (for the padding to wind up to that value upon decryption - in the case of this last byte, the value 0x01 is used. For the second-to-last byte, the value 0x02 is used and so on). This formula works as a result of the Cipher Block Chaining (CBC) mode decryption method, shown in the following picture:



Cipher Block Chaining (CBC) mode decryption

- For the second iteration forward, the values of CC1 of the current iteration's byte to the last byte (first iteration) have to be changed accordingly. To do so, we have to use the following equation `CC1 = PAD ⊕ P2 ⊕ C1` so that when sending `CC1` back to the oracle for decryption, we obtain the next wanted byte of the plaintext, which gives back the valid padding signal.

- The above operations are repeated until all bytes in the plaintext message P2 are decrypted. Below we show the first seven iterations that allowed us to discover the last seven bytes of the secret message P2. The results we obtained match the plaintext solution presented in the lab guide.

```
# CC1 = PAD ⊕ P2 ⊕ C1

# Iteration 1:
    # In the first iteration, the first CC1 is returned by the oracle
    # It is used to calculate the value of the first byte of P2, necessary for
iteration 2

# Iteration 2:
    CC1[15] = 0xcc # 0x02 ⊕ 0x03 ⊕ 0xcd

# Iteration 3:
    CC1[14] = 0x38 # 0x03 ⊕ 0x03 ⊕ 0x38
    CC1[15] = 0xcd # 0x03 ⊕ 0x03 ⊕ 0xcd

# Iteration 4:
    CC1[13] = 0xf5 # 0x04 ⊕ 0x03 ⊕ 0xf2
    CC1[14] = 0x3f # 0x04 ⊕ 0x03 ⊕ 0x38
    CC1[15] = 0xca # 0x04 ⊕ 0x03 ⊕ 0xcd

# Iteration 5:
    CC1[12] = 0x19 # 0x05 ⊕ 0xee ⊕ 0xf2
```

```
    CC1[13] = 0xf4 # 0x05 ⊕ 0x03 ⊕ 0xf2
    CC1[14] = 0x3e # 0x05 ⊕ 0x03 ⊕ 0x38
    CC1[15] = 0xcb # 0x05 ⊕ 0x03 ⊕ 0xcd

# Iteration 6:
    CC1[11] = 0x43 # 0x06 ⊕ 0xdd ⊕ 0x98
    CC1[12] = 0x1a # 0x06 ⊕ 0xee ⊕ 0xf2
    CC1[13] = 0xf7 # 0x06 ⊕ 0x03 ⊕ 0xf2
    CC1[14] = 0x3d # 0x06 ⊕ 0x03 ⊕ 0x38
    CC1[15] = 0xc8 # 0x06 ⊕ 0x03 ⊕ 0xcd

# Iteration 7:
    CC1[10] = 0xeb # 0x07 ⊕ 0xcc ⊕ 0x20
    CC1[11] = 0x42 # 0x07 ⊕ 0xdd ⊕ 0x98
    CC1[12] = 0x1b # 0x07 ⊕ 0xee ⊕ 0xf2
    CC1[13] = 0xf6 # 0x07 ⊕ 0x03 ⊕ 0xf2
    CC1[14] = 0x3c # 0x07 ⊕ 0x03 ⊕ 0x38
    CC1[15] = 0xc9 # 0x07 ⊕ 0x03 ⊕ 0xcd

# The last seven bytes of P2, calculated using:
# P2 = PAD ⊕ C1 ⊕ CC1 (CC1 that comes from the oracle)
    P2[9]  = 0xbb # 0xbb = 0x07 ⊕ 0x21 ⊕ 0x9d
    P2[10] = 0xcc # 0xcc = 0x06 ⊕ 0x20 ⊕ 0xea
    P2[11] = 0xdd # 0xdd = 0x05 ⊕ 0x98 ⊕ 0x40
    P2[12] = 0xee # 0xee = 0x04 ⊕ 0xf2 ⊕ 0x18
    P2[13] = 0x03 # 0x03 = 0x03 ⊕ 0xf2 ⊕ 0xf2
    P2[14] = 0x03 # 0x03 = 0x02 ⊕ 0x38 ⊕ 0x39
    P2[15] = 0x03 # 0x03 = 0x01 ⊕ 0xcd ⊕ 0xcf
```

Task 3:

- For this final task, we automated the attack process described in **Task 2** so that we could get all the blocks of the plaintext. The strategy used consists of looping through the three pairs of the *(current block/previous block)*, as that is what we need to decipher each one of the blocks since the *previous* one is used in the decryption of the *current* one.
- For each of these pairs, we loop through all 16 bytes, from last to first, repeating the same calculations as described in **Task 2** to obtain the plaintext of the current block of the secret message.
- In the end, we organize the deciphered blocks and print them out, revealing the secret message, as shown in the screenshot below:

```
[04/12/22]seed@VM:~/.../Labsetup$ python3 auto_attack.py
IV:  c9a88b259a61840d4a5822097998d24b
C1:  19b4e57181becbf5c95ee6ddfff6fe59
C2:  fc2ad0b4a906cd5d3461bb155b56640d
C3:  35fe2e6f6cefaefca83ed3bf337c0277

Secret Message: 285e5f5e29285e5f5e2920546865205345454420 4c61627320617265206772656174212028 5e5f5e29285e5f5e290202

Decoded Secret Message: (^_^)(^_^) The SEED Labs are great! (^_^)(^_^)
[04/12/22]seed@VM:~/.../Labsetup$ python3 auto_attack.py
IV:  6c356eb9aef92f0b64930e02ba89ed9a
C1:  d7d53abda62d25b512daf24ee0a46c4f
C2:  f5cb63875c93013512888e0f48332792
C3:  72e38c9769352ba376407f2e38f71123

Secret Message: 285e5f5e29285e5f5e2920546865205345454420 4c61627320617265206772656174212028 5e5f5e29285e5f5e290202

Decoded Secret Message: (^_^)(^_^)_The SEED Labs are great! (^_^)(^_^)
```

- As we can see from the image, in each execution, the encryption process used different IV and key, resulting in different generated blocks of ciphertext. Even so, the final result is the same, as the secret message doesn't change. This demonstrates that our attack works.

This is the code we developed:

```
(...)

sM = ""

for (current_block, previous_block) in [(C3, C2), (C2, C1), (C1, IV)]:
    for K in range(1, 17):
        for i in range(256):
            CC1[16 - K] = i
            status = oracle.decrypt(IV + CC1 + current_block)
            if status == "Valid":
                D2[16 - K] = K ^ previous_block[16 - K] ^ i
                for c in range(1, K + 1):
                    CC1[16 - c] = (K + 1) ^ D2[16 - c] ^ previous_block[16 - c]
                break
    sM = D2.hex() + sM

print("Secret Message: " + sM)
print("\nDecoded Secret Message: " + bytes.fromhex(sM).decode("ASCII"))
```