# Cryptographie - PaddingOracle - 250 points

Kévin DUVERGER

## Table des matières

**1 Résolution PaddingOracle : 2**

# 1 Résolution PaddingOracle :

Avant de commencer ce write-up, je ne peux que vous conseiller d'aller lire la ressource associée au challenge sur la plateforme de CTF. Elle vous introduira au fonctionnement du mode d'opération CBC sur le chiffrement par bloc AES. Elle vous montre aussi la théorie derrière l'attaque et vous donne les pistes pour la réaliser. Nous ne referons donc pas cette introduction ici.

La première chose à faire est de tenter de se connecter au serveur pour voir comment il se comporte :

```
C:\Users\keked>ncat 146.59.227.136 23420
libnsock ssl_init_helper(): OpenSSL legacy provider failed to load.

--------------------------------
| PADDING ORACLE ATTACK TRAINER |
--------------------------------

I will be using AES-CBC with PKCS padding to encrypt the flag !
Here is the IV that I used     : d790b2b2bb1a9f8b0ee0f2679b43c67c
And here is the encrypted flag : 9663d7dffddb1802d036142e39b0280bda6536bbe323949f58bd2185652c32cb
e429cbf1659006a48f9a99545485ba21c518609f72704d81457fca49585754396fd443cf8667810902eb21ec32585ccf

Now, give me ciphertexts to decrypt (in hex format, IV first and then the "true" ciphertext)
Your ciphertext : d790b2b2bb1a9f8b0ee0f2679b43c67c9663d7dffddb1802d036142e39b0280bda6536bbe323949
f58bd2185652c32cbe429cbf1659006a48f9a99545485ba21c518609f72704d81457fca49585754396fd443cf86678109
02eb21ec32585ccf
Valid padding !
Your ciphertext : d790b2b2bb1a9f8b0ee0f2679b43c67cd790b2b2bb1a9f8b0ee0f2679b43c67c
Invalid padding !
Your ciphertext : ^C
```

On peut voir ici que le chiffrement utilisé est AES-CBC avec du padding PKCS, et après avoir annoncé cela le serveur nous envoie l'IV et le chiffré du flag (qui est la cible qu'il faudra déchiffrer).
On peut donc essayer de renvoyer le couple IV, chiffré donné par le serveur, et ici on obtient la réponse `Valid padding !`. Si l'on envoie en revanche un chiffré invalide, on obtient la réponse `Invalid padding !`.

Comme nous l'avons vu dans la ressource associée au challenge, on peut attaquer le chiffré bloc par bloc (sauf le premier). Il faut donc faire une fonction pour attaquer un bloc séparément, ce qui peut se faire de la façon suivante :

```python
def attackOnBlock(maSocket, previousBlock, block, IV, blockSize) :
    plaintextBlock = bytes.fromhex("00" * blockSize) # contains Dec(K, C[i])
    otherBlock     = bytes.fromhex("00" * blockSize) # contains what to send as C[0]
    for i in range(blockSize - 1, -1, -1) :
        print("Getting byte " + str(i))
        # Let's try all 256 possible values for byte i
        for j in range(0, 256) :
            otherBlock = otherBlock[:i] + j.to_bytes(1, "big") + otherBlock[i + 1:]
            maSocket.sendall((IV + otherBlock + block).hex().encode("utf-8") + b"\n")
            response = maSocket.recv(1024)
            if "Valid" in response.decode("utf-8") :
                print("\tFound j = " + str(j))
                break
        # Let's compute the plaintext block and the new bytes in otherBlock
        plaintextBlock = plaintextBlock[:i] + (otherBlock[i] ^ (blockSize - i)).to_bytes(1, "big") +
     plaintextBlock[i + 1:]
        if i > 0 :
            for j in range(i, blockSize) :
                otherBlock = otherBlock[:j] + (plaintextBlock[j] ^ (blockSize - (i - 1))).to_bytes
    (1, "big") + otherBlock[j + 1:]
    # Xoring with the previous block to get the result
    for i in range(0, blockSize) :
            plaintextBlock = plaintextBlock[:i] + (previousBlock[i] ^ plaintextBlock[i]).to_bytes(1,
     "big") + plaintextBlock[i + 1:]
    return plaintextBlock
```

L'idée est donc d'essayer toutes les valeurs du dernier octet de `otherBlock` pour faire apparaître `0x01` à la fin du déchiffré de `block`. Une fois cela fait, on peut modifier le `plaintextBlock` (qui stocke pour l'instant `Dec(K, block)`) et poursuivre avec les octets suivants (en modifiant `otherBlock[-1]` pour qu'il se déchiffre sur `0x02`). À la fin de tout ce processus, il ne reste plus qu'à xorer `plaintextBlock` avec le bloc de chiffré précédent pour obtenir le clair final !

Il ne reste maintenant plus qu'à se connecter au serveur, parser les valeurs et déchiffrer bloc par bloc :

```python
server = "146.59.227.136"
port = 23420

```

```
#Connection au serveur
maSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try :
    maSocket.connect((server, port))
except Exception as e :
    print("Probleme de connexion", e.args)
    sys.exit(1)

# Getting the data
data = maSocket.recv(2048)
data = data.decode("utf-8")
lines = data.split("\n")
# Parsing IV and ciphertext
IV = bytes.fromhex(lines[5].split(" : ")[1])
ciphertext = bytes.fromhex(lines[6].split(" : ")[1])
# Printing them
print("IV : " + IV.hex())
print("CT : " + ciphertext.hex())

# Let's divide the ciphertext by blocks
blockSize = 16
blocks = []
for i in range(0, len(ciphertext) // blockSize) :
    blocks.append(ciphertext[i * blockSize : (i + 1) * blockSize])

#Nous pouvons maintenant dechiffrer chaque bloc
plaintext = b""
for i in range(1, len(blocks)) :
    print("Decrypting block " + str(i))
    plaintext += attackOnBlock(maSocket, blocks[i - 1], blocks[i], IV, blockSize)
print(plaintext)
```

Ce qui nous permet d'obtenir le flag !