**=============THE FOLLOWING GIVEN TO THE PARTICIPANTS=============**

## **The Alien Oracle**

## **Challenge Category:** Crypto

## **Challenge Description:**

*Once upon a time, you are stuck in an alien planet,*

*with your only means to return home hidden within the alien message:*

|  |
| --- |
|  |
| *b';e\xc6\xd3\xb5\xed\xcaz\xd82\x97{`\x02\xd0\xee\xdf%\x18\xeaf\xaa/,\'3\xael\x83\xd9\xf2u\xda\'\xf5\xb0\xad"q\xfa\xf1\n\xecRZ?rh\x92{\x07\xaf@J4Y\xd2\x9a\xad9\xf0\xf4\x90\xf1'* |
|  |

*The message comes in four 16-symbol blocks, and the answer is hidden in the second and third blocks of the message. The message originated from the Cryogenic-Bezier-Curve and not the primitive version, Ectoplasmic-Cuckoo-Boogaloo. Fortunately, there is an alien Oracle that lives in the planet, but it only speaks in alien tongue. It only will say "****ree****" if your alien message submitted has proper Planetary-Kebab-Cosmic-Solomon-7 padding (like one used on Earth) and "****yon****" if it does not. A means of communicating with the alien Oracle has been provided in challenge-client.py. You realise you are missing essential information for the first block, but the Oracle says it is irrelevant.*

*You have only 75 hours before nightfall in this alien planet, where you will freeze over in its subzero temperatures.*

*Will you be able to return home?*

**Challenge Source Files for participants:**

challenge-client.py (python3 skeleton code for attack **given** to participants)

**NOTE:**

Please use python3 to communicate with the Oracle. ;)

**=============THE FOLLOWING IS FOR PROF NILS EYES ONLY=============**

## **Team Reyon**

Members:

* Kenneth ,1001092
* Weihan ,1001003
* Wei Sheng ,1000978
* Lim Zhi Han Ryan ,1000985

Challenge Name: The Alien Oracle

Category: Crypto

Topics: Chain Block Cipher (CBC) & Padding Oracle Attack

**Challenge Server running in Amazon EC2 Instance:**

team-reyon-server.py (Code for running the server, **not given** to participants)

## **Summary of Challenge:**

In this CTF, we will provide a ciphertext that attackers need to decrypt to obtain the flag, and a server that acts as an oracle which will take in a ciphertext, and outputs a positive result if the padding of the encrypted message is correct, and a negative result otherwise. The padding we are using will be PKCS7 and in the form of the following:

|  |  |  |  |
| --- | --- | --- | --- |
| N-3 byte | N-2 byte | N-1 byte | Nth byte |
| … | … | … | 0x01 |
| … | … | 0x02 | 0x02 |
| … | 0x03 | 0x03 | 0x03 |

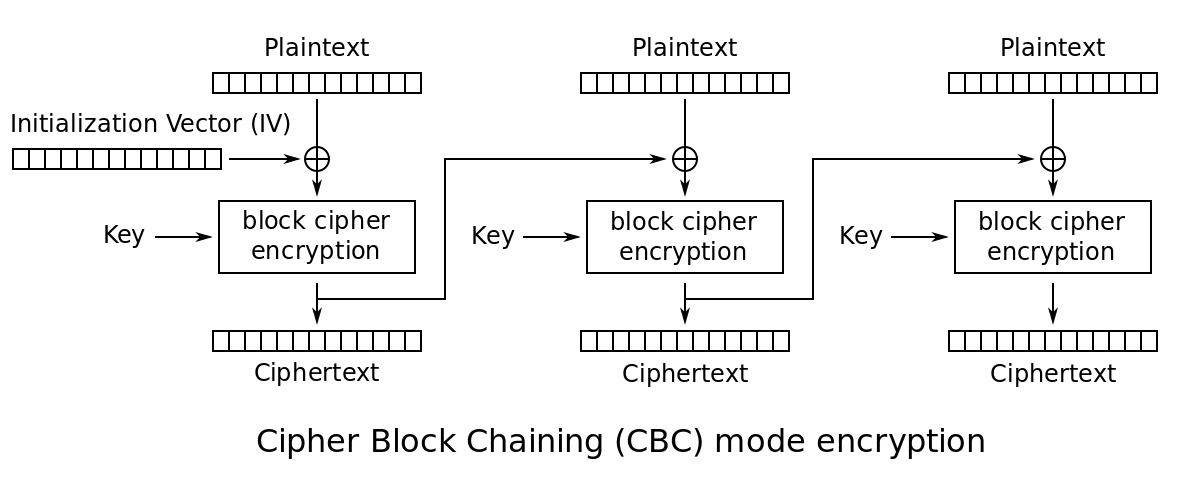
And so on…

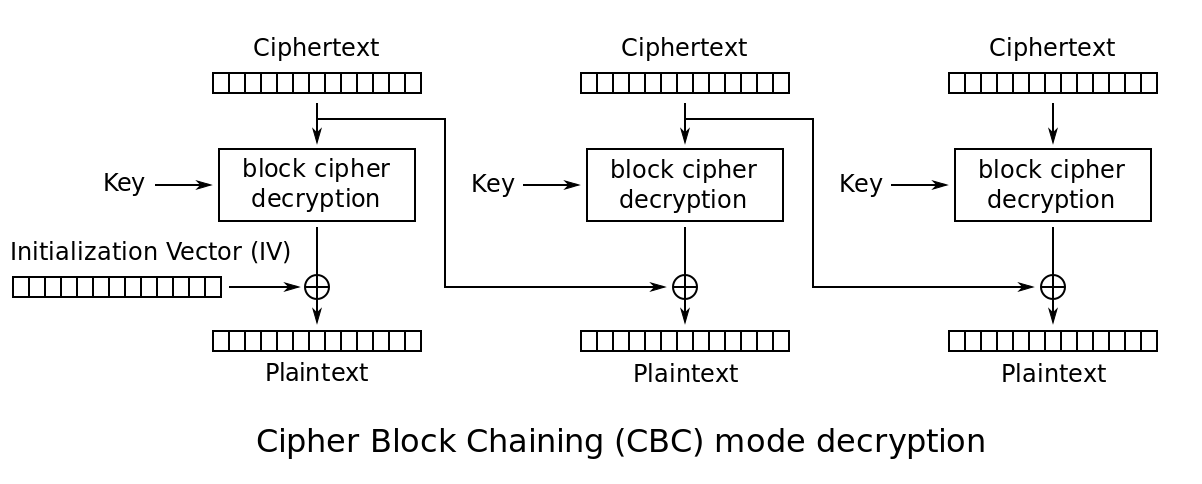
The objective of the challenge is to have attackers understand how CBC works and how the leakage of information (whether the padding of an encrypted message is correct) can lead to the **confidentiality** of the ciphertext being compromised.

Users will perform a padding oracle attack on our system to obtain the flag. Details of an attacker’s approach and likely pseudocode are below.

**Summary of Chain Block Cipher (CBC):**

In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all the plaintext blocks processed up to that point. To make each message unique, an initialization vector must be used in the first block.





## 

## **Summary of Attack:**

We can pass any ciphertext to the oracle, and the oracle will reply whether the decrypted ciphertext is a plaintext with a valid padding or not. We can use this padding information to exploit CBC.

To find out the value of P2[15], we first craft a manipulated ciphertext C1’, which will be used to replace C1. We then submit our modified ciphertext (C1’+C2) to the Oracle.

C1’ will contain the first 15 characters of the original ciphertext C1 concatenated with the last character of the original ciphertext XORed with our guess G and the character 0x01 (refer to excerpt of PKCS7 padding above). Our guess G can be any 8-bit character (256 values):

C1’ = C1[0]…C1[14] + C1[15] ^ G ^ 0x01

Since we know that the Oracle will say our padding is valid if the decrypted plaintext’s last character is 0x01, when we submit the original ciphertext with C1 replaced with the crafted ciphertext C1’, if the response is a valid padding we know our guess G is the correct plaintext P2[15] (since, according to the XOR logic table, XOR-ing two bytes of the same value will cancel out each other, leaving only the padding 0x01).

Example:

P2[15] = I2[15] ^ C1[15]

During decryption, if G == (I2[15] ^ C1[15]):

I2[15] ^ C1[15] ^ G ^ 0x01 = 0x01

If our guess G is the same value as (I2[15] ^ C1[15]), these values will cancel out leaving a value of 0x01, which results in a valid padding. The oracle will then reply with a positive response and we know that our guess G was P2[15].

If the oracle replies that it is a wrong padding, we know that we have made a wrong guess which resulted in the decrypted plaintext padding not being 0x01. This is where we vary our guess from 0 to 255, until we get a reply where the submitted ciphertext has valid padding, which meant that guess is P2[15].

Storing the correct guess G as P2[15], we will move on to the second last character in the block, C1[14].

We will now craft the ciphertext C1’: the first 14 characters remain unchanged, and the 15th character is XORed with our guess and the padding character 0x02. The 16th character is XORed with our previous guess value P2[15] and the padding character 0x02:

C1’ = C1[0]…C1[13] + C1[14] ^ G ^ 0x02 + C1[15] ^ P2[15] ^ 0x02

Since we know that the Oracle will say our padding is valid if the last two characters of the decrypted plaintext are 0x02 (refer to excerpt of PKCS7 padding above), when we submit our modified ciphertext (C1’+C2), if the response is a valid padding we know our guess G is the correct plaintext P2[14].

If the response is an invalid padding, we vary our guess G from values 0 to 255, until a receive a valid padding response, which means that guess G is the correct plaintext P2[14].

Storing the correct guess G as P2[14], we will repeat this process “up” the block, incrementing the padding each time. After 16 iterations, we will obtain the all 16 bytes of the plaintext block P2. After doing this for all the blocks except the first\*, the attacker should be able to identify the flag in the plaintext.

**\* Note:** The first block of the plaintext cannot be decrypted as we did not provide the Initialisation Vector to the participants, however it is irrelevant.

**Pseudocode:**

Create empty array correctByteArray to hold correctly deciphered bytes

Each block is 16 bytes, possible pad values are 1-16

Guess last byte, with pad fixed to 0x01:

For each 'guess' byte value, G, from 0-255:

C1’ = C1[0]…C1[14] + C1[15] ^ G ^ 0x01

MutatedCipher = C1’ + C2

Send MutatedCipher to check padding

Correct padding result means guess is the plaintext's byte value at that position

Insert correctly guessed byte into correctByteArray, break for loop

Repeat for rest of the bytes in the block from index 14 - 0, with pad values increasing 0x02 - 0x16. Each previously guessed byte must be set to the current pad value for subsequent guesses to ensure padding stays consistent.

E.g In order to find P2[14], we must ensure that P2[15] always decrypts to 0x02, which can be done by modifying C1’[15] = C1[15] ^ P2[15] ^ 0x02.

Repeat above steps for all blocks except for the first block

**Complexity:**

Since there are 256 possibilities per byte of ciphertext, enumerating all possibilities to find the intermediate and subsequently plaintext will take on average 128 iterations per byte, and given 16 bytes per block is will take 2048 iterations per block, which should not take very long. This attack essentially negates the need to find out what the key is or attacking the encryption algorithm(AES).

**Working Solution**:

* solution-client.py