CS409 Assignment2

Namish Bansal

September 2025

1 Challenge 1:

1.1 Understanding the problem:

We are given three files `ciphertext.bin, key.hex, iv.hex` containing ciphertext, key, and IV, respectively.
We know the encryption scheme is `aes-128-cbc` from the problem statement, and we can use OpenSSL to decrypt it with the help of the given key and IV.

1.2 Approach:

We will use the following OpenSSL command in the terminal.
`openssl enc -d -aes-128-cbc -in ciphertext.bin -K $(cat key.hex) -iv $(cat iv.hex)`

Here following flags are used:

- `-d`: for decryption mode.
- `-aes-128-cbc`: for specifying encryption scheme.
- `-in`: specifying ciphertext file
- `-K`: to give key
- `-iv`: to give IV

2 Challenge 2:

2.1 Understanding the problem:

- We are given two files `ciphertext.bin, encryptor.py`, which contain ciphertext and encryption method, respectively.
- In this, EBC mode of encryption is used, and we have to exploit its redundancy.

2.2 Vulnerability:

- In this, we were encrypting the Header and the flag both.
- Moreover, when encrypting, we are repeating each character to 16 bytes to make it a block and encrypting it afterwards using AES.
- This exposes the main vulnerability of the EBC method, as each character will now have a particular ciphertext block.
- Moreover, every alphabet and every digit is also present in the Header. Additionally, and _ are also present, which are usually present in a flag.

2.3 Approach:

- Firstly, we will split the ciphertext into blocks of length 16.
- Then we know if n is the length of the header, then the first n blocks of ciphertext will contain the header encryption.
- Then, iterating through the header, we will map each block of the ciphertext to the character it encodes from the header using a dictionary.
- Then, we will iterate over the remaining blocks and each block will map to a character. These characters will give us the flag.

3 Challenge 3:

3.1 Understanding the problem:

- Here we are given two files, `server.py, solution_template.py` for understanding the server behaviour and for writing the solution, respectively.
- This challenge involves interacting with a server. In this, we have the following two choices:
  - Choice 1: Giving parameters as input and getting their encrypted form back. There are some restrictions on input, such as `admin=true` should not be present in the input.
  - Choice 2: Giving encrypted parameters as input. The server decrypts internally, and if it doesn't find a valid decryption, it throws an error. If the decryption contains `admin=true`, it will give us the encrypted flag.
- Major mistake in the encryption in this problem is using the key as IV.

3.2 Vulnerability:

- As said earlier major vulnerability of this encryption is using the IV same as the key.
- Also, when the server decrypts the given ciphertext as per choice 2, if it doesn't find it valid parameter, it throws an error, but in the error message, it gives the decryption also.

3.3 Approach:

- Main idea of the approach to get the key is as follows:
  - Firstly, we will try to get a ciphertext from a given input of more than 16 bytes, in which the first 16 bytes don't contain "=" (say P_0 (first block) gives cipher C_1(first block) and key is k).

$$C_1 = AES_k(IV \oplus P_0) = AES_k(k \oplus P_0) \implies P_0 = decr_k(C_1) \oplus k$$

  - Then we will have choice 2 and give input of the form $C_1||(00)*16||C1$. As we can see, this will not give a valid decryption as there is no padding. Thus, we can get this decryption (say $P_1||P_2||P_3$). So,

$$P_1 = decr_k(C_1) \oplus k$$

$$P_2 = decr(\text{"00"} * 16) \oplus C_1$$

$$P_3 = decr(C_1) \oplus \text{"00"} * 16 = decr(C_1)$$

- Here $P_0 = P_1$, Thus

$$P_1 = P_3 \oplus k \implies k = P_1 \oplus P_3$$

  In this, we can get the key.
- In my approach in the code, I used P_1 as null bytes, thus giving $k = P_3$ directly.

- Once we get the key, then we can encrypt `admin=true` ourselves using AES.
- Then we will give this encrypted text to the server(choice 2), and thus we will get admin access and the encrypted flag.
- Finally, We can decrypt the flag easily as we have the key.

4 Challenge 4:

4.1 Understanding the problem:

- Here we are given two files `server.py, solution_template.py` for understanding the server behaviour and for writing the solution, respectively.
- In this problem, we are given a new mode of encryption, namely CTR.
- When we give a plaintext as input to the server, it outputs two ciphertexts, one being the input's ciphertext with the current counter value and the other being the ciphertext of flag (if input is "!flag") or of input with countervalue incremented by 10.

4.2 Vulnerability:

- In this, we are keeping a global counter variable named ctr.
- The major drawback of this encryption is that we are incrementing ctr by 10 in each case, irrespective of the length of the plaintext.
- Thus, if we give a long text in input, we can find the ciphertext corresponding to `nonce||ctr_val`, thus giving us the idea about the keystream corresponding to the next inputs also.

4.3 Approach:

- Firstly, we will input a string of null bytes of 480 length, having 30 blocks of length 16. Then we will get two ciphertexts (say c_1 and c_2) starting with `ctr_val=0` and `ctr_val=10`, respectively. And as our input is null bytes, the ciphertexts obtained are directly the keystreams.
- Then we will input "!flag" and the second ciphertext is the encryption of flag with `ctr_val` starting from 30.
- So, after skipping the first 20 blocks of c_2, the keystream of flag will be aligned with c_2.
- Then, XORing the encrypted flag with the new c_2 (after skipping 20 blocks), will give us the flag. (As c_2 is the keystream itself)

5 Challenge 5:

5.1 Understanding the problem:

- Here we are given two files `server.py, solution_template.py` for understanding the server behaviour and for writing the solution, respectively.
- This is the challenge based on the padding oracle attack, in which we connect with a server and we pass two arguments, IV and ciphertext.
- Then the server decrypts the given ciphertext and tells us if the plaintext has valid padding or not.

5.2 Vulnerability

- Major vulnerability of this encryption is in telling us whether the plaintext has valid padding or not.
- As we know, a plaintext has valid padding if at the last byte it has the same value as the number of bytes padded.
- Thus, if the plain text has `\x01` at the last, it will give valid padding, and similarly if the last two bytes are `\x02`, it will give valid padding, and so on.

5.3 Approach:

- Firstly, we connect to the server to get the iv and the encrypted flag i.e. the ciphertext.
- We break the ciphertext into blocks of 16 bytes each.
- Consider the previous block of the first block to be the iv.
- For each target block C_i, we can get a "previous block" C' and then we sent (C', C_i) to the server. Now, we start recovering the bytes from right to left.
- Consider padding length p where p ranges from 1 to 16, we can set the first $p-1$ bytes of C' so that after decryption they produce the value p using previously recovered plaintext.
- We brute force all possible values of the current byte in C', send (C', C_i) to validate_padding and wait for a valid response.
- when we get valid response, we compute plaintext byte using

$$P_i[j] = p \oplus C_{i-1}[j] \oplus C'[j]$$

- We then store the recently discovered byte, update C' such that last p bytes produce $p+1$, and continue to the next byte.

- After recovering all 16 bytes of current block, we do the same for the next ciphertext block.
- Once all blocks are recovered, remove the padding to get the flag.