Lab2 Report

Rishi Vardhan Majji
24B0969

September 2025

1 Challenge 1: openssl Decryption

You've been provided a file ciphertext.bin which contains an encryption of the flag encrypted using aes-128-cbc with the key given in key.hex and the IV given in iv.hex.

1.1 Approach

Using openssl, this qsn can be solved using the terminal.

enc = uses the symmetric cipher functions
-d = decrypt
-aes-128-cbc = specifies the encryption scheme
-in = input file
-K = specifies the key value
-out = output flag text file

```
openssl enc -aes-128-cbc -d \
 -in ciphertext.bin \
 -K "$(cat key.hex | tr -d '\n')" \
 -iv "$(cat iv.hex | tr -d '\n')" \
 -out -
```

```
flag-->cs409{op3n551_2_d3crypt10n_1_4m}
```

we get the flag **cs409{op3n551_2_d3crypt10n_1_4m}**

2 Challenge 2: The Electron Code

2.1 Description

The ECB (Electronic Code Book) mode of encryption is a "bad" mode of encryption because it always maps the same block of plaintext to the same block of ciphertext.
Thus, if you have a long message such that the same block repeats twice in the plaintext, the same block would repeat in the exact same position in the ciphertext too. This leaks information about the plaintext.

2.2 Approach

Since identical plaintext block produces the same ciphertext block,and each character is modified to repeat **16** times.
The byte block sizze of AES, so each **16** sized block of the cipher text corresponds to the a single character from the HEADER.
as we know the HEADER, we try to create a mapping for each character in the header to its corresponding ciphertext block.
Now, as the flag is encrypted using the same encryption, we can use the mapping we found from the HEADER part and find the corresponding character in the flag.

......The code goes as follows......

Mentioning the HEADER and the AES block size.

```
from Crypto.Cipher import AES

n = AES.block_size
HEADER = "_Have you heard about the \\{quick\\} brown fox which jumps over the
lazy dog?\n__The decimal number system uses the digits 0123456789!\n___The flag
is: "

ciphertext = open("ciphertext.bin","rb").read()
```

Initiating the map and filling it with the characters in HEADER

```
mymap = {}
for i in range(len(HEADER)):
    mymap[ciphertext[i*n:(i+1)*n]] = HEADER[i]
```

Looping through the ciphertext for the flag and finding the corresponding characters of it using the map.

```
    flag = ""
    for i in range(len(HEADER), len(ciphertext)//n):
        flag += mymap[ciphertext[i*n:(i+1)*n]]

    print(flag)


    #flag--> cs409{r3dund4nt_l34k4g35}
```

We get the flag **cs409{r3dund4nt_l34k4g35}**

3 Challenge 3: The Catastrophic Equality

3.1 Approach

In this challenge, the server uses AES in CBC mode, and the IV happens to be the same as the secret key.
First, we send some normal parameters to the server to get a real ciphertext block. Then we trick the server by sending a special ciphertext in the form C0 || zero_block || C0.
When the server tries to decrypt it, it ends up leaking part of the plaintext. Using this leaked data, we can figure out the key because P0 XOR P2 gives us the key (As IV = key).
After we get the key, we can locally encrypt a new parameter string that includes admin=true using the same key and IV. Finally, we send this ciphertext to the server, get admin access, and decrypt the flag to read it.

The code goes as follows...........

```
n = AES.block_size  # 16 bytes
```

We set `n` to 16 as AES works with 16-byte blocks.

```
ct_hex = choice1("a=b")
ct = bytes.fromhex(ct_hex)
C0 = ct[:n]
```

We send some normal parameters (‘a=b‘) to the server using `choice1`. It gives us a ciphertext. We convert it from hex to bytes and take the first block, ‘C0‘, which we’ll use in the next step.

```
# Make the server decrypt and leak plaintext by sending: C0 || zero_block || C0
mal = C0 + (b"\x00" * n) + C0
ok, leaked_hex = choice2(mal.hex())

leaked = bytes.fromhex(leaked_hex)

P0 = leaked[0:n]
P2 = leaked[2*n:3*n]
```

We create a special ciphertext: ‘C0 || zero_block || C0‘. When the server tries to decrypt it, it rejects it but leaks part of the plaintext. We convert the leaked hex back to bytes and split it: ‘P0‘ is the first block, ‘P2‘ is the third block. These will help us recover the key.

```
# Recover key (because IV == key): key = P0 XOR P2
key = strxor(P0, P2)
```

Since the IV is the same as the key, we can recover the key with a simple XOR: ‘key = P0 XOR P2‘.

```
# Locally encrypt a params string that contains admin=true using key as IV too.
payload = b"a=1&admin=true"
made = AES.new(key, AES.MODE_CBC, iv=key).encrypt(pad(payload, n))
```

We make a new parameters string containing ‘admin=true‘. We pad it to 16 bytes and encrypt it locally with AES in CBC mode, using the recovered key both as the key and as the IV. This gives us a crafted ciphertext that should give us admin access.

```
# Submit made ciphertext and print only the flag if we get admin.
got_admin, flag = choice2(made.hex())
flag_cipher = bytes.fromhex(flag)
flag = unpad(AES.new(key,AES.MODE_CBC,iv=key).decrypt(flag_cipher), n).decode()
print(flag)
```

We send the crafted ciphertext to the server using ‘choice2‘. If it works, the server returns the flag in encrypted form. We decrypt it locally with the same key and IV, remove padding, and print the flag.

```
    flag-->cs409{fu11_k3y_recovery_ftw_1mpl3m3nt_w1th_c4r3}
```

4 Challenge 4: Never Painted by the Numbers

4.1 Description

In this challenge, you will interact with a server which essentially acts as an echo server- it outputs back to you exactly what you input to it– except that the response is encrypted.
However, if you input !flag to it, the server instead outputs the flag to you– again, encrypted. The encryption is done using CTR mode, using a key you don't know. But you suspect that the implementation of the CTR mode encryption in the server has some vulnerability

4.2 Approach

CTR turns AES into a bytewise keystream XOR, so reusing the same key+nonce just shifts the same keystream around.
The server echoes our chosen plaintexts and encrypts them with predictable counters, so we can get ciphertexts for plaintexts we already know.
XORing a known plaintext with its ciphertext directly gives that segment of the keystream.
The flag is encrypted with the same keystream sequence but starting at some unknown offset, so we don't need every counter, just the right part
By sliding the keystream made across the flag ciphertext and XORing, one alignment will produce readable text.
When that alignment yields the expected flag pattern **cs409{** we've recovered the flag.

The code goes as follows..............

```
x=1000
known_plaintext=b'0'*x
```

Start by creating a long text thats known to us, we use long to get the key stream as long as possible so we can find our required chunk in it.

```
enc_inp_hex,enc_out_hex=send_to_server(known_plaintext.decode())
enc_inp=bytes.fromhex(enc_inp_hex)
```

Giving it to the server to get two encryptions with different ctr and convert to its byte form.

```
 k=strxor(enc_inp,known_plaintext)
```

This gives us the key stream used to encrypt this long message.

```
enc_flag_in_hex,enc_flag_out_hex=send_to_server("!flag")
flag_ciph=bytes.fromhex(enc_flag_out_hex)
```

Giving **!flag** as input to get the encryption of the required flag, as the required cipher text is in the second one(made by output encryption) we store it.

```
    for i in range(len(k)-len(flag_ciph)):
        text=strxor(flag_ciph,k[i:i+len(flag_ciph)])
```

We take the flag ciphertext and try XORing it with slices of the long keystream k. keeping each slice length the same length as the flag.

```
        try:
            text = text.decode()
        except UnicodeDecodeError:
            text = text.decode(errors="ignore")
```

There will be many texts(the ones we dont need) who will have random bytes which arent valid in ASCII, so we try to ignore them.

```
        if "cs409{" in text:
            print(text)
            break

    #flag-> cs409{y0u_kn0w_th3_gr34t35t_f1lm5_of_4ll_t1m3_w3re_n3v3r_m4d3}
```

We print the text if it has the beginning portion **CS409{** in it, as it will be our flag

5 Bonus Challenge: Canis Lupus Familiaris

5.1 Description

A padding oracle attack is a cryptographic attack that targets the padding scheme used in block cipher modes, particularly in Cipher Block Chaining (CBC) mode.
In practice, padding is often added to plaintext messages to ensure that the length of the message is a multiple of the block size of the cipher being used.
A padding oracle is a term used to describe a vulnerability that arises when an attacker can determine whether a given ciphertext has a valid padding or not. The padding is typically added to ensure that the plaintext can be properly aligned into blocks before encryption. The padding scheme we often use is PKCS#7.
The padding oracle attack can be extremely dangerous in the sense that one could completely recover the plaintext given access to a padding oracle
In this challenge, you will interact with a server which will act as a padding oracle

5.2 Approach

AES in CBC mode encrypts each plaintext block P_i by first XORing it with the previous ciphertext block C_{i-1} (or the IV for the first block), then applying the block cipher:

$$C_i = E_K(P_i \oplus C_{i-1}).$$

During decryption we first get the intermediate block $I := D_K(C_i)$, then recover the plaintext as

$$P_i = I \oplus C_{i-1}.$$

We exploit a *padding oracle* that tells us whether the decrypted plaintext has valid PKCS#7 padding. To recover a single byte $I[j]$:

- pick a padding value pad (1, 2, 3, ...), and build a forged previous block C' where we set the current test byte $C'[j] = I_{\text{guess}} \oplus \text{pad}$;
- for any bytes to the right of j that we already solved, adjust them as $C'[k] = I[k] \oplus \text{pad}$ so the padding stays valid;
- send (C', C_i) to the oracle — if it says "Valid Padding!", the guess I_{guess} is correct.

Once we know $I[j]$, the real plaintext byte is recovered as

$$P_i[j] = I[j] \oplus C_{i-1}[j].$$

Repeat this from the last byte to the first, and for every block. After decrypting all blocks, remove the PKCS#7 padding to get the full plaintext, including the hidden flag.

The code is as follows.................
....

```
n = 16  # AES block size
```

AES works with 16-byte blocks, so we set `n` to 16. We use this whenever we split data or remove padding.

```
def split_blocks(data: bytes, size: int = n):
    return [data[i:i+size] for i in range(0, len(data), size)]
```

This helper just cuts bytes into 16-byte pieces. We use it to turn the ciphertext into blocks so we can handle one block at a time.

```
def decrypt_block(prev_block: bytes, curr_block: bytes) -> bytes:
    intermediate = [0] * n   # will hold I = D_k(C) values
    plaintext = [0] * n      # will hold recovered P bytes
```

This function tries to recover one plaintext block. - `intermediate` stores the AES-decrypted bytes of the current ciphertext block (before XOR). - `plaintext` will store the real bytes we want after XORing with the previous block.

```
# work from last byte to first
for pos in range(n - 1, -1, -1):
    pad_val = n - pos
```

We solve bytes from the end of the block to the start. At each position we want the padding to look like `pad_val` (e.g. 1, 2, 3, ...).

```
for I_guess in range(256):
    forged = bytearray(b'\x00' * n)
```

We try all 0..255 values for the intermediate byte (call it `I_guess`) and build a fake previous block called `forged`.

```
for j in range(pos + 1, n):
    forged[j] = intermediate[j] ^ pad_val
```

For bytes we already found (the tail of the block), we set the forged block so those positions will decrypt to `pad_val`. This keeps the tail valid while we test the current byte.

```
forged[pos] = I_guess ^ pad_val
```

To test our guess, we set the current byte in the forged block to I_guess$\oplus$pad_val. If the real intermediate byte equals `I_guess`, the server will see valid padding.

```
if validate_padding(forged.hex(), curr_block.hex()):
    intermediate[pos] = I_guess
    plaintext[pos] = intermediate[pos] ^ prev_block[pos]
    break
```

We send the forged block and the current ciphertext to the server. If it replies "Valid Padding!", our guess is right. We then record the intermediate byte and compute the real plaintext byte by XORing with the previous block.

```
blocks = [IV] + split_blocks(flag_enc, n)
recovered = b""
for i in range(1, len(blocks)):
    recovered += decrypt_block(blocks[i-1], blocks[i])
```

We split the whole encrypted flag into blocks (IV first). Then we decrypt each ciphertext block using the previous block and collect the plaintext blocks into `recovered`.

```
flag = unpad(recovered, n).decode()
print(flag)
```

Finally, we remove PKCS#7 padding and print the flag as a readable string.

```
    flag-->cs409{sid3_ch4nn3l_danger!}
```