CS 409: Lab 2 Report

Y Harsha Vardhan
24b1069

September 5, 2025

Contents

1 Challenge 1: openssl Decryption

The first challenge involves decrypting a ciphertext file using openssl with the AES-128-CBC mode. The key and IV are also provided in seperate files.

1.1 Process

Using openssl, we can decrypt a ciphertext just using the terminal, the following is the bash code that is used to generate a flag.txt, which contains the flag in it.

```
# This is a bash script to decrypt the given ciphertext using openssl
# The following is the description of the flags used:
# enc -> uses the symmetric cipher functions
# -d  -> decrypt
# -aes-128-cbc  => specifies the encryption scheme
# -in -> input file
# -out -> output flag text file
# -K  -> specifies the key value
# -iv -> specifies the IV value

openssl enc -d -aes-128-cbc \
    -in ciphertext.bin \
    -out flag.txt \
    -K $(cat key.hex) \
    -iv $(cat iv.hex)
```

1.2 Result

The decrypted flag inside flag.txt is:

`cs409{op3n551_2_d3crypt10n_1_4m}`

2 Challenge 2: The Electron Code

To solve this challenge, we will exploit the vulnerability of ECB Encryption scheme to come up with an attack. We know the following information from the given `encryptor.py` and ECB in general:

- In the `new_encrypt` function, the plaintext is modified to have each character repeated `AES.block_size` (denoting by n) times.
- Also, since the AES is in ECB Mode, we know that each block of the ciphertext (length n) will be 1 block per character in the original plaintext
- Since, the ECB scheme encrypts each block independently, if two plaintext blocks are same, then their ciphertext blocks will also be the same.

2.1 Process

From the `encryptor.py`, we also know that the plaintext has a `HEADER` befor the flag. So, if the flag consists of any of the same characters as the HEADER, we can map them. So, this code will implement the above mentioned idea:

```
from Crypto.Cipher import AES

# The value of the block size used in Encryption
n = AES.block_size
# This is the header given to us in the encryptor.py
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: "

# Reading the ciphertext.bin
with open("ciphertext.bin", "rb") as f:
    ciphertext = f.read()

# First we split the ciphertext into n-byte blocks
blocks = [ciphertext[i:i+n] for i in range(0, len(ciphertext), n)]

# Then we can build a mapping from known HEADER to the unknown
   plaintext
mapping = {}
for i, ch in enumerate(HEADER):
    mapping[blocks[i]] = ch

# Recovering the flag characters from the mapping
flag = ''.join(mapping[b] for b in blocks[len(HEADER):])

print("Flag:", flag)
```

Listing 1: Python Script to exploit ECB vulnerability

2.2 Result

The flag obtained by exploiting the ECB Mode is:

`cs409{r3dund4nt_134k4g35}`

3 Challenge 3: The Catastrophic Equality

In this challenge, we exploit a rookie mistake in cryptography, which is using the key as the IV. The following section shows why this type of scheme is vulnerable to attacks

3.1 Vulnerability Analysis

- Suppose we have a ciphertext C which consists of blocks: C0, C1, C2, ...
- Let `n = AES.block_size` (The length of each ciphertext block)
- When we decrypt the first block C0, we have:

```
P0 = Dec(C0) ^ IV = Dec(C0) ^ k  (as we know that IV = key)
```

- So, if we can get the server to decrypt

  `C = C0 || 0^n || C0`

- The ciphertext after decryption will be:

```
P0 = Dec(C0) ^ k
P1 = Dec(C1) ^ C0
P2 = Dec(C0) ^ C1 = Dec(C0) ^ (0^n) = Dec(C0)
So, P0 ^ P2 gives the key, using which we can find the entire plaintext
```

- The below code utilizes the above mentioned idea to find the flag

```
n = AES.block_size

def xor_bytes(a: bytes, b: bytes) -> bytes:
    return bytes([x^y for x, y in zip(a, b)])

# Choosing a simple plaintext to get a ciphertext:
cipher_HEX = choice1("a=b")
cipher = bytes.fromhex(cipher_HEX)

C0 = cipher[:n] # The first block of the ciphertext

# Now creating a malicious ciphertext which uses the above mentioned
   structure
mal = C0 + (b"\x00" * n) + C0
mal_HEX = mal.hex()

# Sending this to the server:
ok, leaked_HEX = choice2(mal_HEX)
assert not ok and leaked_HEX != "" # Ensuring that we got a HEX back
leaked = bytes.fromhex(leaked_HEX)

# Finding the key
P0 = leaked[:n]
```

```
P2 = leaked[n*2 : n*3]
key = xor_bytes(P0, P2)


# After this our task is relatively simple
# We need to first encrypt a payload containing "admin=true" using the
    obtained key
# then make the server decrypt it to get the key
payload = "a=b&admin=true"
payload_bytes = payload.encode()

padded_payload = pad(payload_bytes, n)

cipher_loc = AES.new(key, AES.MODE_CBC, iv=key)
forged_cipher = cipher_loc.encrypt(padded_payload)
forged_cipher_HEX = forged_cipher.hex()

# Sending this to the server, we get back the encrypted flag
got, enc_flag = choice2(forged_cipher_HEX)

# Now we need to decrypt this flag
flag_cipher = bytes.fromhex(enc_flag)
cipher_loc = AES.new(key, AES.MODE_CBC, iv=key)
flag = unpad(cipher_loc.decrypt(flag_cipher), n).decode()

print(f"Flag: {flag}")
```

Listing 2: Python Script to exploit equality of key and IV in CBC

3.2 Result

The flag is recovered by sending an intelligently crafted payload to the server. The payload and resulting flag are:

```
Payload: C0 || 0^n || C0
Flag: cs409{fu11_k3y_recovery_ftw_1mpl3m3nt_w1th_c4r3}
```

4 Challenge 4: Never Painted by the Numbers

In this challenge, the encryption used is in CTR Mode, so we need to identify a vulnerability in the server's implementation of this mode and exploit it to find the flag.

4.1 Vulnerability Analysis

- We know that, in CTR Mode:

```
C = P ^ KS (where KS is the key stream)
```

- If the server reuses KS for two encryptions, then we get:

```
P1 ^ C1 = KS
P2 = C2 ^ KS
```

4.2 Methodology

Using the above vulnerability, we can come up with this attack:

- First we will send a known P1 and get C1 -> using these two we can compute the KS (but for this we need to have a P1 with the byte length larger than the flag's)
- Then we can request the encrypted flag: Cflag, and compute the flag using the KS, and the fact that our flag starts with cs409{

```
n = AES.block_size

def xor_bytes(a: bytes, b: bytes) -> bytes:
    return bytes(x ^ y for x, y in zip(a, b))

# We need to choose a known plaintext which is long enough to recover
   the keystream
# Let the known flag length be l, we need to choose plaintext length
   greater than this
# Lets try 2048 bytes
l = 2048
known_plaintext = b'A' * l

# Sending this known plaintext to the server:
enc_inp_HEX, enc_out_HEX = send_to_server(known_plaintext.decode())
enc_inp = bytes.fromhex(enc_inp_HEX)

# Now using the enc_inp and enc_out, we can find the keystream =
   enc_inp ^ known_plaintext
ks = xor_bytes(enc_inp, known_plaintext)

# Now we can request the encryped flag by the command: !flag
enc_flag_in_HEX, enc_flag_out_HEX = send_to_server("!flag")
flag_cipher = bytes.fromhex(enc_flag_out_HEX)

# Now we just need to decrypt the flag by XORing it with the keystream
# Actually, for l = 1024, it didn't work so I then made it l = 2048
```

```
if len(ks) < len(flag_cipher):
    print ("The length l is small")
else:
    for shift in range(len(ks) - len(flag_cipher)):
        XOR_text = xor_bytes(flag_cipher, ks[shift:shift+len(
           flag_cipher)])
        try:
            text = XOR_text.decode()
        except UnicodeDecodeError:
            text = XOR_text.decode(errors="ignore")

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

Listing 3: Python Script to attack CTR

4.3 Result

The recovered flag is:

`cs409{y0u_kn0w_th3_gr34t35t_f1lm5_of_4ll_t1m3_w3re_n3v3r_m4d3}`

5 Bonus Challenge: Canis Lupus Familiaris

In this challenge, we need to implement the idea discussed in the class, when there is a padding oracle that gives us information about whether a given padding is valid or not. Using this information, we need to come up with an attack.

5.1 Vulnerability Analysis

In the CBC Decryption, we know that:

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

So, if we flip bytes in C_{i-1}, we can influence the last byte of P_i. Using the `validate_padding` (padding oracle), we can test our guesses until we find the correct padding.

5.2 Methodology

- First we need to split the ciphertext into n-byte blocks (let n = Block Size)
- Then we need to attack each block C_i (with preceding block C_{i-1}) to recover the plaintext
- So, for each byte (from last to first):
  - Take a guess of C_{i-1}
  - Send (C_{i-1}', C_i) to `validate_padding`
  - If it is a valid padding, then deducing the plaintext byte
- Finally, we will concatenate all recovered plaintext and strip the padding

```
n = 16 # Block Size

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

def decrypt_block(prev: bytes, curr: bytes) -> bytes:
    # Decrypting one block using the padding oracle, (validate_padding
        function)
    intermediate = [0] * n
    plaintext = [0] * n

    for pad_len in range(1, n + 1):
        i = n - pad_len
        found = False

        for guess in range(256):
            # Forging the block (C') to send to the oracle
            forged = bytearray(b'\x00' * n) # Starting with all zeroes

            for j in range(i + 1, n):
                forged[j] = intermediate[j] ^ pad_len

            # For the current position i, inserting our guess.
```

```
            forged[i] = guess

            # Checking if this forged block works or not using the
               validate_padding function
            if validate_padding(forged.hex(), curr.hex()):
                if i == 0:
                    is_verified = True
                else:
                    forged_check = bytearray(forged)
                    forged_check[i - 1] ^= 0x01  # Flip a bit in a
                       preceding byte
                    is_verified = validate_padding(forged_check.hex(),
                        curr.hex())

                if is_verified:
                    # The guess is correct! We can now calculate the
                       intermediate and plaintext byte.
                    intermediate[i] = guess ^ pad_len
                    plaintext[i] = intermediate[i] ^ prev[i]
                    found = True
                    break

        if not found:
            raise Exception(f"Byte not found at position {i} with
               pad_len={pad_len}")

    return bytes(plaintext)

# Decryption
blocks = [IV] + split_blocks(flag_enc, n)
recovered = b""

for i in range(1, len(blocks)):
    plaintext_block = decrypt_block(blocks[i-1], blocks[i])
    recovered += plaintext_block

# Removing the padding:
try:
    flag = unpad(recovered, n).decode()
except:
    flag = recovered.decode(errors="ignore")

print(f"Flag: {flag}")
```

Listing 4: Python Script to attack CBC with padding oracle

5.3 Result

It took around 30 min. to finish. The recovered flag is:

`cs409{sid3_ch4nn3l_danger!}`