CS409 Lab 1 – CTF Writeup

Y Harsha Vardhan
24b1069

August 9, 2025

Contents

1 Challenge 1: Two-Time Pad

1.1 Problem Understanding

This problem requires us to find the flag using the 2 given cipher texts, which are of the flag and a normal english sentence, also we know that the encryption algorithm is XOR

1.2 Given Data

- `ciphertext1.enc`
- `ciphertext2.enc`
- `encrypt.py`

1.3 Approach

- Using the cipher texts, if we XOR them, we will get s = m1 $\oplus$ m2
- Now using the fact that we know the starting few characters of the flag, (i.e., cs409{) we can XOR this with the starting 6 characters of s to get the first 6 characters of the message.
- Now using this, if we have any incomplete words, we can check and round it down to one word considering the fact that both the message and flag consist of meaningful words and symbols.
- So, using this method recursively to find a few characters of the message and then complete any incomplete words and then mapping this to the flag and so on, we can achieve our goal

1.4 Outputs and Flag

```
b'Crypta'
b'cs409{one_time'
b'Cryptanalysis freq'
b'cs409{one_time_pad_key_re'
b'Cryptanalysis frequently invo'
b'cs409{one_time_pad_key_reuse_compr'
b'Cryptanalysis frequently involves statistical att'
b'cs409{one_time_pad_key_reuse_compromises_security!!!'
b'Cryptanalysis frequently involves statistical attacks'

Flag: cs409{one_time_pad_key_reuse_compromises_security!!!}
```

1.5 Implementation

Listing 1: Python script to solve Challenge 1

```python
from Crypto.Util.strxor import strxor
from Crypto.Random import get_random_bytes

with open("ciphertext1.enc", "rb") as f:
    Cipher_Flag = f.read()
with open("ciphertext2.enc", "rb") as f:
    Cipher_Message = f.read()

M1_XOR_M2 = strxor(Cipher_Flag, Cipher_Message)
def Helper(a):
    n = len(a)
    extra = b"0"*(l - n)
    var = a + extra
    print(strxor(var, M1_XOR_M2)[0:n])

l = 53 # This is the length of the flag and message
flag = b"cs409{"
n = len(flag)
extra = b"0"*(l - n)
flag = flag + extra
print(strxor(flag, M1_XOR_M2)[0:n])

# Here we get that the first 6 characters of the MSG are: Crypta
# The possible words relevant are: Cryptanalysis, Cryptanalytic,
   Cryptanalyst
# So, lets assume that the first few characters of the message are: "
   Cryptanalysis "
message = b"Crypta" + b"nalysis "
n = len(message)
extra = b"0"*(l - n)
message = message + extra
print(strxor(M1_XOR_M2, message)[0:n])

flag = b"cs409{one_time_pad"
Helper(flag)

message = b"Cryptanalysis frequently "
Helper(message)

flag = b"cs409{one_time_pad_key_reuse_"
Helper(flag)

message = b"Cryptanalysis frequently involves "
Helper(message)

flag = b"cs409{one_time_pad_key_reuse_compromises_security"
Helper(flag)

message = b"Cryptanalysis frequently involves statistical attack"
Helper(message)

flag = b"cs409{one_time_pad_key_reuse_compromises_security!!!}"
Helper(flag)
# Message: "Cryptanalysis frequently involves statistical attacks"
```

2 Challenge 2: One-Time Pad over Groups

2.1 Problem Understanding

In this question, the OTP scheme is implemented using modulo 128 operation instead of XOR. The encryption is performed by adding each plaintext byte to the corresponding key byte modulo 128. Its flaw is in the fact that, instead of using a truly random key, it starts from a random bit and then deterministically expands it using SHA-26, ensuring key length matches plaintext length.

2.2 Given Data

- `ciphertext.enc`
- `encrypt.py`

2.3 Approach

Now since we know that the encryption algorithm deterministically generates the entire key just from the starting byte, which is generated by random. We can just brute force our way as there are only 128 possible first bits. For each case we can generate a key and then try to decrypt the message by reversing the modulo 128 operation (subtraction instead of addition). Then we can find the plausible plaintext using the fact that the flag contains a { and } in it.

2.4 Implementation

```python
import hashlib

def group_sub(m1: bytes, m2: bytes) -> bytes:
    assert len(m1) == len(m2)
    res = b""
    for i in range(len(m1)):
        res += chr((m2[i] - m1[i]) % 128).encode()
    return res

def generate_key(start_byte: int, length: int) -> bytes:
    key = chr(start_byte).encode()
    for _ in range(1, length):
        key += chr(hashlib.sha256(key).digest()[0] % 128).encode()
    return key

with open("ciphertext.enc", "rb") as f:
    ciphertext = f.read()

# A brute force which checks all possible 128 combinations of the key (
    because it can be generated using the first byte)
# Then comparing it with the flag format, which must contain { and }
for start_byte in range(128):
    key = generate_key(start_byte, len(ciphertext))
    plaintext = group_sub(key, ciphertext)
    if b"{" in plaintext and b"}" in plaintext:
        print(f"[+] Found plausible flag with start byte {start_byte}:
            {plaintext}")

# Hence the flag is: cs409{algebra_enters_the_picture!}
```

2.5 Output and Flag

```
[+] Found plausible flag with start byte 5: b'cs409{algebra_enters_the_picture!}'
[+] Found plausible flag with start byte 48: b'8{A[pQ\x18\x0f\x00fUk,{==\\54sxm(9\x07}$l\x1
[+] Found plausible flag with start byte 59: b'-\x19\x1ad\x16{d0oiN\nPwJ\x04\x14\x08/S}2G\x
[+] Found plausible flag with start byte 69: b"#1PaiG{W20AU:\x05hBQ_c\x0c'(\x19y\x12\x18}tF
[+] Found plausible flag with start byte 78: b'\x1aN\x03s}04]5,CMk\x1bd:{zQfU\x08-\x1d\x14q

Flag: cs409{algebra_enters_the_picture!}
```

3 Challenge 3: Faulty One-Time Pad Distinguishing Attack

3.1 Problem Understanding

This encryption scheme is faulty because the key space is reduced from what it should be to behave as a perfectly secure encryption. We are only using the keys that do not have the 0 bit, thus it is only [0x01, 0xff], and not [0x00, 0xff] which is secure.
Since Bob doesn't use 0x00, the **ciphertext** byte can never be equal to the **plaintext** byte

3.2 Approach

Due to the above mentioned flaw, this scheme is vulnerable to this type of attack:
We can distinguish between encryptions of a known plaintext vs encryption of a random message by checking if any **ciphertext byte equals the corresponding plaintext byte:**

- If yes, → **normal OTP is possible**, but Bob's scheme never produces that as the key doesn't contain 0x00.
- So, if we pick our plaintext as **all zero bytes** then:
  ciphertext byte = *0x00* ⊕ **keybyte** ≠ *0x00*

So, ciphertext bytes produced using this scheme will always be from **0x01** to **0xff** and **never zero**. But if the ciphertext is an encryption of a **random message**, then **cipher bytes** can be anything, including **zero**.

So, our payload should be a large string of 0's and the logic for deciding whether to send c1 or c2 to the server, will be based on choosing the one without 0x00. We are choosing a large string of zero's to ensure that we reduce the possibility of getting cipher texts such that, both c1 and c2 don't have 0x00. (As the cipher text generated from a random message will have lesser probability that it will not contain a 0x00 byte if the payload length is longer)

3.3 Output and Flag

When I have chosen the payload to be: "00"*1024, it only worked till Level 81 in the server, so then I chose it to be 2048 instead of 1024, to make the probability that both c1 and c2 don't have a 0x00 to be even lower.

```
Flag: cs409{y0u_h4d_fu11_4dv4nt4g3}
```

3.4 Implementation

```
## TODO 1 ##
payload = "00" * 2048  # 2048 zero-bytes

## TODO 2 ##
b1 = bytes.fromhex(c1)
b2 = bytes.fromhex(c2)

has0_1 = (b'\x00' in b1)
has0_2 = (b'\x00' in b2)

# If one contains 0x00 and the other doesn't, choosing the one without
    0x00
if has0_1 and (not has0_2):
    guess = 2
elif has0_2 and (not has0_1):
    guess = 1
else: # Highly unlikely as we are taking a very large string as payload
    guess = 1

sendline(f"c{guess}")
```

4 Challenge 4: Fixing the Fault

4.1 Problem Understanding

The encryption scheme takes a plaintext byte sequence $m = m_1 m_2 \dots m_n$, interprets it as a base-256 number, and then converts this number into base-255 digits $p_1 p_2 \dots p_{n'}$, where each digit $p_i \in [0, 254]$. The ciphertext digits c_i are computed as:

$$c_i = (p_i + k_i - 1) \bmod 255$$

where $k_i \in [1, 255]$ are the key bytes. The ciphertext digits are then converted back from base-255 to base-256 bytes, producing the encrypted message. This scheme ensures perfect secrecy as the key digits k_i act as a one-time pad on base-255 digits.

4.2 Approach

To decrypt, we reverse the process:

1. Convert the ciphertext bytes into an integer and then to base-255 digits c_i.
2. Use the key digits k_i to recover the original base-255 plaintext digits by computing:

$$p_i = (c_i - k_i + 1) \bmod 255$$

3. Convert the recovered base-255 digits p_i back to an integer and then to the original base-256 plaintext bytes.

This process reverses the encryption step and recovers the original plaintext perfectly due to the perfect secrecy properties of the scheme.

4.3 Implementation

```python
def bytes_to_int(b):
    return int.from_bytes(b, byteorder='big')

def int_to_bytes(x, length):
    return x.to_bytes(length, byteorder='big')

def int_to_base(n, base):
    digits = []
    while n > 0:
        digits.append(n % base)
        n //= base
    digits.reverse()
    return digits if digits else [0]

def base_to_int(digits, base):
    n = 0
    for d in digits:
        n = n * base + d
    return n

def decrypt(ciphertext_bytes, key_bytes):
    # Converting ciphertext to integer
    c_int = bytes_to_int(ciphertext_bytes)

    # Converting ciphertext integer to base-255 digits
    c_digits = int_to_base(c_int, 255)

    key_digits = list(key_bytes[:len(c_digits)])

    # Recover plaintext digits p_i = (c_i - k_i + 1) mod 255
    p_digits = [(c - k + 1)%255 for c, k in zip(c_digits, key_digits)]

    # Converting plaintext digits back to integer
    p_int = base_to_int(p_digits, 255)

    # Converting integer back to bytes
    # Assuming plaintext length approx ciphertext length
    plaintext_len = len(ciphertext_bytes)
    plaintext = int_to_bytes(p_int, plaintext_len)

    return plaintext

if __name__ == "__main__":
    with open("ciphertext.enc", "rb") as f:
        ciphertext = f.read()
    with open("keyfile", "rb") as f:
        key = f.read()

    plaintext = decrypt(ciphertext, key)
    print("Recovered plaintext:")
    print(plaintext)
```

4.4 Output and Flag

Flag: cs409{r4d1x_ch4ng1ng_f0r_th3_w1n!}

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:

```python
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_l34k4g35}`

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{

```python
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!}`

CS 409: Lab 3 Report

MAC and Hash

Y Harsha Vardhan
24b1069

October 6, 2025

Contents

1 Challenge 1: openssl Sorcery

1.1 Problem Statement

We need to compute the following MACs on `message.txt` using the provided key and IV:

1. HMAC with SHA256 (MAC1)
2. CMAC with AES-128 (MAC2)
3. GMAC with AES-128-GCM (MAC3)

Then store the digests in hex, and the flag format is: `cs409{MAC1_MAC2_MAC3}`

1.2 Approach

To find the individual parts of the flag, we need to run each of these lines of code in the terminal:

```
# MAC1: HMAC-SHA256
MAC1=$(openssl dgst -sha256 -mac HMAC -macopt hexkey:$(cat key.hex) message.txt)

# MAC2: CMAC (AES-128)
MAC2=$(openssl dgst -mac CMAC -macopt cipher:AES-128-CBC -macopt hexkey:$(cat key.hex) m

# MAC3: GMAC (AES-128-GCM)
MAC3=$(openssl mac -cipher AES-128-GCM -macopt hexkey:$(cat key.hex) -macopt hexiv:$(cat
```

We are going to get the following outputs:

1.3 Results

We need to use the hex outputs from here as flags:

```
MAC1 (HMAC-SHA256): 03D6DCD9DE01619EECC1E9EB064096492A78198CD6748B1F72768927C814C400
MAC2 (CMAC-AES-128): 087D547376AD3A2C0B1AA2322E1AB066
MAC3 (GMAC-AES-128-GCM): E4B75011F57EAB1A2DA5E3E8BA161922

Flag: cs409{<MAC1>_<MAC2>_<MAC3>}
```

1.4 Discussion

These commands initially didn't work in my Windows WSL, becasue the openssl version in it didn't have the necessary modules to find MAC3
So, I used these commands in my dual-booted Ubuntu which had the latest version of the openssl, then some problems I faced were that the flag didn't accept small cased answer, also if we run this code, then in the MAC1 and MAC2, some message will also be printed along with the flag, which I had to manually remove before submission.

2 Challenge 2: Extension of Deception (CBC-MAC length-extension)

2.1 Problem Statement

Given the MAC of `DATA` (from server), we need to forge a MAC for a string that starts with `DATA` and contains `admin=true` somewhere, leveraging CBC-MAC length-extension vulnerability.

2.2 Vulnerability

The weakness in the server's CBC-MAC is that the MAC for a message is simply the last block of the CBC encryption. This means if I have the original message's MAC, which lets call: T, I can use it as the starting point to calculate the MAC for an extended message. To forge a new message $M'||X$, I just need to compute a new tag, T', which would be $E_k(X \oplus T)$.
The server's 'Möbius Hacker' oracle was the key, as it allows us to perform the final encryption step without knowing the key.

```
# Defining the original data and the desired extension
original_cred = b"user=cs409learner&password=V3ry$3cur3p455"
extension = b"&admin=true"
block_size = 16

# Padding both of them to a full block size (16)
padded_original = pad(original_cred, block_size)
padded_extension = pad(extension, block_size)

# Converting the original MAC and IV from hex string to bytes
original_mac_BYTES = bytes.fromhex(original_mac)
iv_bytes = bytes.fromhex(iv)

# Next we XOR the padded extension with the original MAC and the IV to
    create the block, then we need the oracle to encrypt this for us
temp_xor = strxor(padded_extension, original_mac_BYTES)
data_to_send = strxor(temp_xor, iv_bytes)
mobius_data = data_to_send.hex()

# So, now the fully forged credentials are the original padded block
   plus our new padded block
forged_cred_BYTES = padded_original_cred + padded_extension

creds = forged_cred_BYTES.hex()
forged_mac = mobius_mac
```

Listing 1: Python Snippet for the code used in solution_template.py

2.3 Result

The forged message is constructed by taking the original, known data (DATA), padding it to a 16-byte block, and appending the desired extension (‘admin=true‘), which is also padded to a 16-byte block.

`Forged message (hex): cca2a938c5d0937693018536764 5e9a2`

```
Forged MAC (hex): 1ace2791c629f2d88912958a2b9873d3
Final Flag: cs409{53cur1ty_f0r_4ll_t1m3_4lw4y5}
```

3 Challenge 3: Tick-Tock on the HMAC (Timing attack)

3.1 Problem Statement

We need to exploit a timing leak in an insecure digest comparison routine. The target compares a user-supplied HMAC hexdigest prefix against the true HMAC prefix using a naive byte-by-byte comparison that sleeps for 1 second on every matching byte. So our goal is to recover the first 10 hex characters of the HMAC hexdigest for an attacker-chosen message by exploiting the timing side channel.

3.2 Approach

To approach this problem, we need to exploit the 1-second delay that the server adds for every correct character in the HMAC guess. This timing leak means that a longer server response time corresponds to a more accurate prefix.
So to find the correct HMAC, one character at a time. For each position, from 0 to 9, we will loop through all 16 possible hex characters ('0' through 'f').
We can append the test character to the known-correct prefix and send it to the server for verification. The character that causes the longest delay has to be the correct one for that position. To handle network delays and improving the accuracy of our result, we will average the time over 3 trials for each guess.

```python
# Goal: to recover the first 10 hex chars
hexchars = "0123456789abcdef"
known = ""
TRIALS = 3    # We will take the average of three trials


for pos in range(10):
    best = None
    best_time = -1.0

    for ch in hexchars:
        candidate = known + ch + "0"*(9-pos)
        # measuring the average response time across TRIALS (here 3)
        times = []
        for _ in range(TRIALS):
            t0 = time.time()
            send_guess(candidate)  # waiting for server response
            times.append(time.time() - t0)
        avg = sum(times)/len(times)

        if avg > best_time:
            best_time = avg
            best = ch

    known += best
    print(f"Recovered HMAC so far: {known} (pos {pos} chosen: {best},
       avg {best_time}s)")
```

```python
print("Recovered prefix of HMAC:", known)
```

Listing 2: Snippet that shows the Timing Attack on the HMAC

3.3 Implementation Details

- **Averaging:** Because network latency can vary, using multiple trials per attempted character and using the mean as the score for that candidate is better.
- **Early success:** If the server responds with success while testing a candidate (it accepts the prefix), we need to stop immediately and record the prefix.

3.4 Results

Recovered prefix (first 10 hex chars): `97906ea158`
Flag: `cs409{k3$h4_0r_t4yl0r_5w1ft?}`

4 Challenge 4: Commitment Issues (Merkle Tree)

4.1 Problem Statement

Server stores a Merkle tree where each leaf is a single character (one byte) of the flag. The length of the flag is n, which is guaranteed to be an integral power of two. The server allows up to $n/4$ Merkle proof queries: for a chosen leaf index the server returns the leaf value (raw byte) and the usual Merkle proof (list of sibling hashes up the tree).
Our task is to recover the entire flag using at most $n/4$ proof queries.

4.2 Approach

With a limit of n/4 queries for an n-byte flag, we need to get more than one character's worth of information from each query. The main observation here is to realize that a Merkle proof for one leaf also reveals the hashes of its siblings on the path to the root.
My approach was to group the flag into 4-byte blocks: [A, B, C, D]. By querying for leaf 'A', the server will give me A's value directly, but the proof will also contain the hash for 'B' and the combined hash for the '[C, D]' subtree. This approach helps to recover four bytes with just one query.

Recovering bytes from the proof

1. **A:** the server already returns A directly; we store it at `recovered[base_INDX]`.
2. **B:** the immediate sibling at leaf level is `proof_BYTES[-1]`. We can find B with a hash lookup:
$$B = \texttt{leaf\_hash\_to\_BYTE[proof\_BYTES[-1]]}$$
and store it at `recovered[base_INDX + 1]`.
3. **C,D:** the next proof element up, `proof_BYTES[-2]`, is the hash of the 2-leaf subtree for C and D:
$$\texttt{cd\_subtree\_hash} = \texttt{proof\_BYTES[-2]}$$
We can brute-force all 256×256 pairs (c, d) until
$$H\big(H(c) \parallel H(d)\big) = \texttt{cd\_subtree\_hash}, \quad \text{where } H(x) = \text{SHA256}(x).$$

Therefore, we performed **brute-force preimage recovery on very small domains** instead of violating the collision resistance of hash function

```
for b in range (256):
    h = sha256(bytes([b])).digest()
    leaf_hash_to_BYTE[h] = b

recovered = [0]*DATA_LEN

for block_INDX in range(blocks):
    base_INDX = block_INDX*4 # choosing the farst element of A of the
       block
    print(f"\n[*] Querying index {base_INDX} (block {block_INDX})")
    A_val , proof_hex = get_proof(base_INDX)
    # Converting proof hex list to bytes; the order is top->down, so
       last elements are in lower levels
    proof_BYTES = [bytes.fromhex(x) for x in proof_hex]

    # A is directly returned (as an integer 0 ... 255)
    A = A_val
    recovered[base_INDX] = A

    # immediate sibling at leaf-level is the last element in the proof
    sibling_leaf_hash = proof_BYTES[-1]

    # finding B by looking up in precomputed leaf hashes
    B = leaf_hash_to_BYTE[sibling_leaf_hash]
    recovered[base_INDX + 1] = B

    # the next proof element up is the hash of the 2-leaf subtree (C
       and D)
    cd_subtree_hash = proof_BYTES[-2]

    # brute-forcing to find C and D (256^2 = 65536 combinations)
    found_c = None
    found_d = None
    # Precomputing all sha256(d)
    right_hashes = [sha256(bytes([d])).digest() for d in range(256)]

    for c in range(256):
        left_hash = sha256(bytes([c])).digest()
        # trying all possible d's
        for d in range(256):
            candidate = sha256(left_hash + right_hashes[d]).digest()
            if candidate == cd_subtree_hash:
                found_c = c
                found_d = d
                break
        if found_c is not None:
            break

    recovered[base_INDX + 2] = found_c
    recovered[base_INDX + 3] = found_d

# Building final data bytes
data = bytes(recovered)
```

Listing 3: Code Snippet containing the main idea

4.3 Result

Recovered flag:

`cs409{cs409{maybe_you_don't_know_what's_lost_'til_you_find_it_merkle!}}`

4.4 Did we violate collision resistance of the hash function?

No. The attack does not produce collisions or contradict the collision-resistance property of sha256. Instead, the attack relies on these two weaknesses in the protocol:

- Each leaf is one byte (only 2^8 possibilities), so brute-forcing is easy; collision resistance of the hash function doesn't prevent the search over such small domains.
- Proofs reveal small subtree hashes which, for 1 to 2 byte subtrees, can be brute-forced due to low number of its possibilities (2^8 per leaf), hence it is not a weakness of the hash itself.

CS 409: Lab 4 Report

PKEs and Signatures

Y Harsha Vardhan
24b1069

November 1, 2025

Contents

1 Challenge 1: Public Encryption

1.1 Approach

We are given the RSA-OAEP private and public keys of a user in `pub.pem` and `priv.pem` respectively, we need to decrypt this using `openssl`.
On using the following bash code line on the terminal, we will get the flag printed in the terminal.

```
openssl pkeyutl -decrypt -inkey priv.pem -in cipher.bin -pkeyopt rsa_padding_mode:oaep
```

We are going to get the following output:

1.2 Results

```
    Output Flag: cs409{r54_043p_3t_0p3n_55l}
```

2 Challenge 2: Is This Certified ?

2.1 Problem Statement

Here, we need to find the common name of the issuer of the digital certificate of the given website: `https://www.cse.iitb.ac.in`

2.2 Approach

- First we need to open a browser, (in my case firefox) and go to the website of CSE IIT Bombay
- Then we need to click on the lock symbol beside the website link in the topbar
- Then we need to click on the topdown option named "Connection secure" and then click on More Information
- Now in the pop-up box that appears, we can see the "View Certificate" option, which on clicking will open a new tab in the browser.
- In this we can see that the common name is: RapidSSL TLS RSA CA G1

2.3 Result

Thus the flag is given by:

```
cs409{RapidSSL_TLS_RSA_CA_G1}
```

3 Challenge 3: ECDSA Nonce Reuse

3.1 Approach

To find the flag, we will exploit a critical vulnerability in ECDSA when the same random nonce (k) is used to sign two different messages. The ECDSA signature equation for a message hash (h) and private key (d) is given by: (here n is the order of the curve)

$$s = k^{-1}(h + d \cdot r) \pmod{n}$$

Since the same k is used for two messages, the r value is also same for both signatures. This will give us a system of two linear equations with two unknowns (k and d):

1. $s_1 = k^{-1}(h_1 + d \cdot r) \pmod{n}$
2. $s_2 = k^{-1}(h_2 + d \cdot r) \pmod{n}$

We can then solve these 2 linear equations to get the nonce k and private key d:

$$k = (h_1 - h_2) \cdot (s_1 - s_2)^{-1} \pmod{n}$$

$$d = (k \cdot s_1 - h_1) \cdot r^{-1} \pmod{n}$$

3.2 Implementation:

```python
n = ecdsa.SECP256k1.order
h1 = int(hashlib.sha256(msg1.encode()).hexdigest(), base=16)
h2 = int(hashlib.sha256(msg2.encode()).hexdigest(), base=16)

r = r_1

# k = (h1 - h2) * (s1 - s2)^-1 mod n
s_diff = (s_1 - s_2) % n
h_diff = (h1 - h2) % n

# (s1 - s2)^-1 mod n
inv_s_diff = inverse(s_diff, n)
nonce_rec = (h_diff * inv_s_diff) % n # Got the nonce

# d = (k*s1 - h1) * r^-1 mod n
# r^-1 mod n
inv_r = inverse(r, n)
k_s1 = (nonce_rec * s_1) % n
k_s1_minus_h1 = (k_s1 - h1) % n

privkey_rec = (k_s1_minus_h1 * inv_r) % n # Got the private key
```

Listing 1: Code Snippet of Solution

3.3 Result

The retrieved flag is given by:

`cs409{n0nc3_5h0uld_b3_u53d_0nc3}`

4 Challenge 4: EdDSA Variants

4.1 Approach

To solve this challenge, we should exploit the vulnerabilities in two different custom variants of EdDSA and recover their private keys and be able to forge signatures.

4.1.1 Variant 1: Deterministic Nonce

The flaw of this variant is that, it is generating its nonce r deterministically using only public information:

$$r = \text{hash}(\text{msg} + \text{VARIANT1\_PUBKEY.x}) \pmod q$$

And since the signature equation is $s = (r + h \cdot d) \pmod q$), and we can compute r (the nonce) and h (the hash) ourselves for any message we send, and we can easily solve the equation for the private key d.
After requesting a signature (R, s) for a known message ‘msg‘, we need to rearrange the equation as follows:

$$s - r = h \cdot d \pmod q$$

$$d = (s - r) \cdot h^{-1} \pmod q$$

Once d (the private key) is recovered, we can follow the server's exact signing algorithm to forge a valid signature for the challenge message.

4.1.2 Variant 2: Colliding Nonce via Message Prefix

This variant's flaw was that it is generating its nonce r using the private key and only the **first half** of the message:

$$r = \text{hash}(\text{msg}[:\text{len}(\text{msg})//2] + \text{str}(\text{VARIANT2\_PRIVKEY})) \pmod q$$

This causes a collision vulnerability, i.e., if we send two different messages that share the same first half (e.g., ‘msgA‘ and ‘msgB‘), we will force the server to unknowingly use the exact same nonce r and thus the same point R for both signatures.
This provides us with a system of two linear equations, which we can easily solve:

1. $s_a = (r + h_a \cdot d) \pmod q$
2. $s_b = (r + h_b \cdot d) \pmod q$

To find the private key d:

$$s_a - s_b = (h_a - h_b) \cdot d \pmod q$$

$$d = (s_a - s_b) \cdot (h_a - h_b)^{-1} \pmod q$$

Now since we have the d, we can compute the required nonce r_{forge} for the challenge message and forge the final signature.

```
# --- Exploit for Variant 1 ---
q = ecdsa.NIST256p.generator.order()
msgKnown = msgs[0].encode()
Rknown, Sknown = sigs[0]

# Calculating the rKnown (nonce):
rKnownHashInp = msgKnown + str(VARIANT1_PUBKEY.x()).encode()
rKnown = int(hashlib.sha256(rKnownHashInp).hexdigest(), base=16) % q

# Calculating the hKnown (hash):
hKnownHashInp = str(Rknown.x()).encode() + str(VARIANT1_PUBKEY.x()).
    encode() + msgKnown
hKnown = int(hashlib.sha256(hKnownHashInp).hexdigest(), base=16) % q

# Recovering the private key:
hKnownInv = inverse(hKnown, q)
privkeyRec1 = ((Sknown-rKnown)*hKnownInv) % q


# --- Exploit for Variant 2 ---
msgAbytes = msgs[0].encode()
msgBbytes = msgs[1].encode()
R_a , s_a = sigs[0]
R_b, s_b = sigs[1]
Rknown = R_a # R_a will be equal to R_b because of the vulnerability

# Calculating h_a and h_b
h_aHashInp = str(Rknown.x()).encode() + str(VARIANT2_PUBKEY.x()).
    encode() + msgAbytes
h_a = int(hashlib.sha256(h_aHashInp).hexdigest(), base=16) % q
h_bHashInp = str(Rknown.x()).encode() + str(VARIANT2_PUBKEY.x()).
    encode() + msgBbytes
h_b = int(hashlib.sha256(h_bHashInp).hexdigest(), base=16) % q

# Recovering private key d, (privkeyRec2):
sDiff = (s_a - s_b) % q
hDiff = (h_a - h_b) % q
hDiffInv = inverse(hDiff, q)
privkeyRec2 = (sDiff * hDiffInv) % q
```

Listing 2: Code Snippet showing exploit idea

4.2 Result

Since both private keys were successfully recovered, we can forge the signatures on the two challenge messages. The server accepted both forged signatures and returned the flag:

`cs409{3dd54_g0t_m3_t4lk1ng_n0nc353nc3}`

5 Challenge 5: Grover's Cipher

5.1 Approach

The given challenge is vulnerable to a meet-in-middle attack. I have read about this attack from the internet, it is a clever trick to reduce the effort it takes to break a "double" secret, by attacking the two halves separately and then matching their results in the middle, this way we can say that we are using extra memory for fewer computations, which will help us to be far more better than a naive brute-force algorithm.
The challenge has the following data: the plaintext m is a 4-byte (32 bit) value, the encryption function c is the product of a subset of a 32-bit element public vector$v$, based on the bits of m. So, a naive brute-force attack will require checking all 2^{32} possible plaintexts, which is too slow for the server's time limit.

So, using the idea from the meet-in-the-middle attack, we will start by splitting the 32-bit problem into two 2^{16} problems:

1. The 32-element vector is split into two 16-element halves: $v_L = v[0:16]$; $v_R = v[16:32]$
2. The ciphertext c can be seen as a product of two ciphertexts: $c = c_L \cdot c_R \pmod p$
3. **Precomputation:** We will iterate through all 2^{16} possible values of the left-half plaintext and compute their corresponding c_L values and store them in a look up table (Dictionary)
4. **Search:** We iterate through all 2^{16} possible values for the right-half plaintext (m_R) and for each one, we compute its c_R, calculate its modular inverse and find the required target value which is given by: $c_L^{\text{target}} = c \cdot (c_R)^{-1} \pmod p$
5. Then we check if this target c_L^{target} exists as a key in the precomputed table or not.
6. **Reconstruction:** If we get a match, we will retrieve the m_L from the table. So, the full 32-bit plaintext is reconstructed by combining the two halves: $m = (m_L \ll 16)|m_R$

This meet-in-the-middle attack idea, reduces the time complexity from $O(2^{32})$ to $O(2^{17})$, which is enough to solve the challenge.
After using this idea, there was a small setback that was caused because of a tiny issue the recvuntil function usage near to the end of the code, the comparisions were beteen a string and a raw byte value, to make it uniform I converted them all into string comparisions only.

5.2 Result

The script successfully implemented this attack, solving all 25 rounds correctly. The server then provides the flag.
Recovered flag:

`cs409{4174_br34k5_unbr34k34bl3_c1ph3r5}`

5.3 Implementation:

```
# Normalizing the ciphertext:
c = c % pub
vLeft = v[0:16]
vRight = v[16:32]

# Creating the lookup table
leftTable = {}
for m_L in range(65536): #2^16 = 65536
    c_L = 1
    bits_L = bin(m_L)[2:].zfill(16) # Converting m_L into a 16-bit
       binary string
    for i in range(16):
        if bits_L[i] == '1':
            c_L = (c_L * vLeft[i]) % pub

    leftTable[c_L] = m_L

# Searching in the right half:
message = None
for m_R in range(65536):
    c_R = 1
    bits_R = bin(m_R)[2:].zfill(16)
    for i in range(16):
        if bits_R[i] == '1':
            c_R = (c_R * vRight[i]) % pub
    if (gcd(c_R, pub) != 1):
        continue # Since c_R is not Invertible
    # We need c_L_target = c * (c_R)^-1 mod pub
    c_Rinv = pow(c_R, -1, pub)
    c_L_target = (c * c_Rinv) % pub

    # Checking if this target c_L is there in our precomputed table:
    if c_L_target in leftTable:
        m_L_found = leftTable[c_L_target]
        m_R_found = m_R

        # Reconstructing the full 32-bit message:
        msgFull = (m_L_found << 16) | m_R_found

        # Converting it into a 4-byte hex string:
        msgBytes = long_to_bytes(msgFull, 4)
        message = msgBytes.hex()
        break

if message == None:
    message = "00000000"
```

Listing 3: Code Snippet implementing the above mentioned idea