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

```
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{dOoiN\nPwJ\x04\x14\x08/S}2G\x
[+] Found plausible flag with start byte 69: b"#1PaiG{W2OAU:\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, $\rightarrow$ **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* $\oplus$ **keybyte** $\neq$ *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

```
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!}`