hw1 Writeup

School/Grade: 交大資料工所 碩一
Student ID: 309551004 (王冠中)
ID: aesophor

POA (Padding Oracle Attack)

- **Overview**:

  My exploit gives the following output:

  ```
  > ./exploit.py
  [+] Opening connection to 140.112.31.97 on port 30000: Done
  b'1'
  b'31'
  b'131'
  b'f131'
  -- trimmed --
  b'LAG{31a7f10f131'
  b'FLAG{31a7f10f131'
  b'\x00'
  b'\x00\x00'
  b'\x00\x00\x00'
  -- trimmed --
  b'\x00\x00\x00\x00\x00\x00\x00\x00\x00'
  b'\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00'
  b'}\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00'
  b'2}\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00'
  -- trimmed --
  b'7f622}\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00'
  b'FLAG{31a7f10f1317f622}\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00'
  ```

- **Observation**:
  - `server.py` uses AES-CBC cipher to encrypt the flag
    - if len(flag) is not an integral multiple of 16, it will be padded with '\x80\x00\x00…' s.t. the resulting length is an integral multiple of 16

- after encrypting the flag, the program will prompt the user for an input, where
  - the first 16 bits will be used as IV
  - the remaining 32 bits will be used as ciphertext
  - if no PaddingError occur, it will send 'YESS' to user
  - if a PaddingError occurs, it will send 'NOOO' to user

- **Exploitation**:
  1. Find the position of \x80
  2. Exploit the property of XOR to create a fake IV
  3. Bruteforce each block from the end to the beginning
  4. Result: FLAG{31a7f10f1317f622}\x80\x00\x00\x00\x00\x00\x00\x00\x00\x00

- **Flag**:

  flag: `FLAG{31a7f10f1317f622}`
  exploit: please see the attachment

COR (Correlation Attack)

- **Overview**:

  My exploit gives the following output:

```
> ./exploit.py
[*] bruteforcing lfsr3 ...
[*] found possible init state: [0, 1, 1, 0, 1, 0, 0, 0, 0,
    1, 1, 0, 1, 0, 1, 0]

[*] bruteforcing lfsr2 ...
[*] found possible init state: [0, 1, 1, 1, 0, 1, 0, 1, 0,
    1, 1, 0, 1, 0, 0, 1]

[*] launching final stage bruteforce attack
[*] FLAG{dfuihj}
```

- **Observation**:
    - Suppose the flag is `FLAG{abcdef}` , it will be stripped and become `abcdef`

    ```
    FLAG = open('./flag', 'rb').read()
    assert len(FLAG) == 12
    assert FLAG.startswith(b'FLAG{')
    assert FLAG.endswith(b'}')
    FLAG = FLAG[5:-1]
    ```

    - `MYLFSR` combines three LFSRs to generate a bit stream

# lfsr	initial state (8-bit)	feedback (16-bit)
lfsr 1	0x??	39989
lfsr 2	0x??	40111
lfsr 3	0x??	52453

    - `MYLFSR` uses the following method to generate a bit

    ```
    def getbit(self):
        x1 = self.l1.getbit()
        x2 = self.l2.getbit()
        x3 = self.l3.getbit()
        return (x1 & x2) ^ ((not x1) & x3)
    ```

    - the boolean function `(x1 & x2) ^ ((not x1) & x3)` contains a vulnerability where bruteforce searching can be used

- **Bruteforce Initial States of LFSR3 and LFSR2**:
    - we can use a two-level for loop to try all possible initial states to generate 100 bits
    - if its output is ~75% similar to the output from `output.txt` , then it's probably the actual value used as LFSR3's initial state.

- LFSR2's initial state can be bruteforced using the same approach

```
for byte1 in range(256):
      for byte2 in range(256):
          init = [int(i) for i in f"{int.from_bytes(bytes([byte1])
                  bytes([byte2]), 'big'):016b}"]

          lfsr = LFSR(init, [int(i) for i in f'{feedback:016b}'])

          lfsr_output = [lfsr.getbit() for _ in range(100)]

          if get_similarity(lfsr_output, output) > 0.75:
              log.info('found possible init state: {}'.format(init)
              result.append({'byte1': byte1, 'byte2': byte2, 'bits'
```

- **Bruteforce the Initial State of LFSR1**:
    - at this point we should have found the initial states of LFSR3 and LFSR2
    - now we can bruteforce the "accurate" initial state of LFSR1
    - if the output of MYLFSR is 100% similar to the output from `output.txt` , then we've found the flag (just decode the bytes with `chr()`)

- **Flag**:

  flag: `FLAG{dfuihj}`
  exploit: please see the attachment