Secure Programming 2020

HW1 write-up

POA

腳本solve.py附在zip /POA/solve.py

這題是 padding oracle attack,是針對AES的CBC mode, 主要的思路是透過server會回應padding格式是否正確來得到明文.

其中有幾個關鍵,第一是initial vector必須是已知,這題題目有提供IV.

假設我們要解密cipher i, CBC mode會先將cipher i作decrypt part得到middle值,然後再跟IV(cipher i-1)作xor得到明文.

由於xor是可以抵銷的,所以如果我們改動IV(cipher i-1)最後的byte來刻意製造padding的格式(透過server的回應我們可以知道我們是否成功),padding的值是已知的,而我們改動的byte也是已知的,將這兩個值作xor就能夠得到middle(cipher i作decrypt得到的值),將這個值再跟IV(cipher i-1)作xor就能夠得到明文.

```
                cipher i                                          cipher i
                   |                                                 |
                decrypt                                           decrypt
                   |                                                 |
IV(cipher i-1) -- xor          IV(bytes modified)(cipher i-1) -- xor
                   |                                                 |
                 plain                                       padding byte

bytes(modified) xor padding byte = middle
middle xor IV(cipher i-1) = plaintext
```

首先我們得先搞清楚題目的padding格式,這題的padding邏輯是要補一個byte就補0x80,如果是兩個byte就補0x80, 0x00,三個byte就是0x80, 0x00, 0x00,以此類推.

```
...
int('1' + '0' * (padlen * 8 - 1), 2).to_bytes(padlen, 'big')
...
```

所以我們從第一個cipher的最後一個byte開始暴搜,想辦法使得server回應yes,代表我們成功製造出padding 0x80, 透過上述的公式我們就可以得到這一個byte的明文.

```
...
for k in range(256):
    if oracle(iv[:16 - 1 - j] + bytes([k]) + xor(iv[-j:], ans) + block):
        ...
        ans = bytes([iv[16 - 1 - j] ^ k ^ (0x80)]) + ans # k ^ 0x80 會得到middle
                                                         # middle再跟原來的IV xor會得到
明文
        ...
```

以此類推往前一個byte作一樣的操作,由於補n個的byte的時候,除了第n個byte本身以外後面的byte都是0x00, 所以IV(cipher i-1)其實不用更動,所以上面的code只將明文跟IV作xor得到middle,但如果padding值不是0x00那就要跟padding值再作一次xor,這樣得到的結果才會是padding值.

然後做完第一個cipher會得到前半段的FLAG:

```
b'FLAG{31a7f10f131'
```

再用同樣的模式操作第二個cipher就能得到後半段的FLAG.....才怪,如果用上面的code對第二個cipher作的話會得到奇怪的結果.

我想了很久，才明白原因是因為第二個cipher真的有padding格式,這意味著什麼呢?意味著如果你改動的byte剛好等於cipher1對應的byte, paddinge格式也會是正確的,但找到的東西並不是真正的明文.

因為如果是剛好等於cipher1所觸發的padding, padding的值可能不是0x80而是0x00(因為cipher2本來就有padding,而且0x80的位置可能根本不是你現在改動的byte,但cipher1原本的byte會使這個byte變成0x00,也符合padding格式),那麼如果還向上面的code那樣跟0x80作xor就得不到正確的明文.

要避免這個問題就得在code裡再加上一行判斷式,使得改動的byte不會等於cipher的那個byte,這樣觸發的padding才不會是cipher1的.

```
...
for k in range(256):
    if oracle(iv[:16 - 1 - j] + bytes([k]) + xor(iv[-j:], ans) + block):
        if iv[16 - 1 - j] != k: # prevent k set to original iv value
            ans = bytes([iv[16 - 1 - j] ^ k ^ (0x80)]) + ans
            ...
```

然而改完後,程式在print出以下的結果後就crash了:

```
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00'
...
```

我trace了一下code發現到了從右數來第十個byte的時候，程式找不到除了cipher1以外的值,也就是說第十個byte只有cipher原來的值才能觸發,所以再把code改一下,如果找到除了cipher1以外的值能使padding正確那就用這個值,如果都找不到就用cipher1的值.

```
...
temp = b''
f = 1
for k in range(256):
    if oracle(iv[:16 - 1 - j] + bytes([k]) + xor(iv[-j:], ans) + block):
        if iv[16 - 1 - j] != k: # prevent k set to original iv value
            ans = bytes([iv[16 - 1 - j] ^ k ^ (0x80)]) + ans
            print(ans)
            f = 0
            break
        else:
            temp = bytes([iv[16 - 1 - j] ^ k ^ (0x80)])

if f == 1: # if cannot find value different from cipher1, then use cipher1 value

    ans = temp + ans
...
```

這樣就能夠找到正確的明文了,而且第十個byte就是0x80,這樣就完全合理了,因為第十個byte就是原來cipher2產生的0x80,所以你根本就找不到第二個值讓padding是0x80.

這樣就得到完整的FLAG:

```
FLAG{31a7f10f1317f622}
```

COR

腳本solve.py附在zip /COR/solve.py

這題是上課有提到的LFSR correlation attack,題目給的是跟範例一樣的mixed LFSR. 題目用FLAG的 0-1 bytes, 2-3 bytes, 4-5 bytes 作為LFSR的三個init value來得出用於LFSR的x1, x2, x3, 再用x1, x2, x3算出output, 然後我們也有100個LFSR的output.

要作correlation attack,首先得先算出x2, x3跟output之間的correlation,這題的operation跟範例一樣所以correlation是0.75.

```
x1 x2 x3 output = (x1 & x2) ^ ((not x1) & x3)
0  0  0  0   #equal
0  0  1  1   #equal
0  1  0  0   #equal
0  1  1  1   #equal
1  0  0  0   #equal
1  0  1  0
1  1  0  1
1  1  1  1   #equal

6/8 = 0.75
```

再來我們要先暴搜x3, x3的init value大小是2個bytes,所以暴搜需要2的16次方,把每一個init value得到的100個x3跟output比較,算出兩個值相同的比率,至少要超過0.7,然後選擇其中比率最高的.

```
...
for i in range(pow(2,16)):
    lfsr = LFSR([int(i) for i in f"{int.from_bytes(i.to_bytes(2, 'big'), 'big')
    :016b}"], [int(i) for i in f'{40111:016b}']) # x3 generated from init value
i
    count = 0
    for j in range(100):
        x2 = lfsr.getbit()
        if x2 == output[j]: # calculate the rate of correlation
            count += 1
    rate = count/100
    if rate > 0.7 and correlate < rate: # the rate should larger than 0.7,
select
            correlate = rate            # the largest as the possible init value
            init_x2 = i
...
```

然後用同樣的作法找出init x2的候選.

其實大於0.7的init value不只一個,而且不一定等於0.75,我把所有大於0.7的可能全部搜了一遍發現最大的那個值就是答案,所以我就把code改成找最大的那一個就好,以節省時間.

我們找到了最可能是x2, x3, init value的值:

```
init value(int): 26730 correlation: 0.81 // most possible init value for x3
init value(int): 30057 correlation: 0.78 // most possible init value for x2
```

最後再用得出的init value x2, init value x3暴搜x1,如果產生的output跟我們一開始得到的用flag算出的output一樣,那我們就得到了正確的init value也就是FLAG.

```
...
for i in range(pow(2,16)):#use the init value x2, x3 we find to find x1 init
value
    lfsr = MYLFSR([i.to_bytes(2, 'big'), init_X2.to_bytes(2, 'big')
    , init_X3.to_bytes(2, 'big')])
    lis = [lfsr.getbit() for _ in range(100)]
    if output == lis: # compare the output we generated and the output from FLAG
                      # if is equal, then we find the correct init value
        flag = b"FLAG{" + i.to_bytes(2, 'big') + init_X2.to_bytes(2, 'big') +
        init_X3.to_bytes(2, 'big') + b"}"
        print(flag)
...
```

```
b'FLAG{dfuihj}'
```