HW1 Writeup

R10922043 黃政瑋

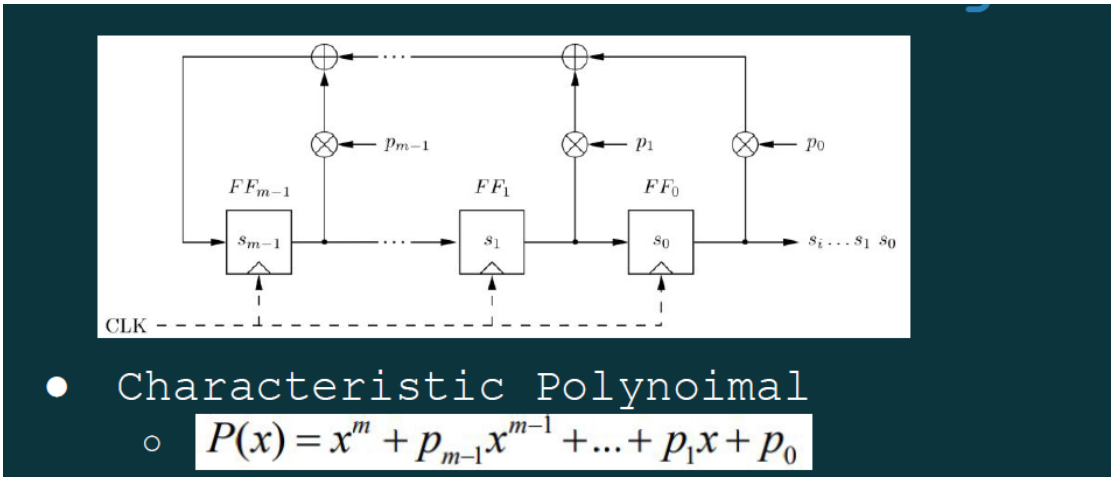
CTF account: cwhuang1937

1. nLFSR

(1) 轉成 Characteristic Polynomial

首先觀察 step function，可以發現他的過程相似於 nonlinear-LFSR，又已知 state 的初始值是隨機產生的 64bits，因此我們可以將這題轉成在 **GF(2^64) 上的 Characteristic Polynomial**。接著將’1’ + poly reverse 來當作 GF 的 modulus。

modulus 的由來: 之所以 reverse 是因為 GF 吃的係數是降冪的形式，而 state 的 bits 是升冪的形式，1 則是補上 x^64 項的係數。(這邊跟組員 **R10922046 李柏漢與 R10944011 吳添毅**討論非常久，這邊最一開始是參考到網路上某份相似題目的 writeup，但對他如何取 modulus 這部分一直很不明白，我們三個討論了很久才弄懂為何要這樣取)



```
In [10]: poly = bin(0xaa0d3a677e1be0bf)
         print(f'poly: {poly[2:]}')
         MOD = '1' + poly[-1:1:-1]
         print(f'MOD: {MOD}')
         MOD = int(MOD, 2)

         poly: 1010101000001101001110100110011101111111000011011111000001011111
         MOD: 1111111010000011111011000011111101110011001011100101100000101010 1
```

(2) 求出 state-transition matrix

由 source code 可知，每次的 random function 會經過 43 次 step，也就是 state 會轉移 43 次，因此這邊可以列出 64 條 state-transition 的式子分別對應初始 state 做了 random function 1~64 次，此 64 條式子形成的係數即為 state-transition matrix。

(3) 求出 init_state

接著用 pwntools 連上 server 後，全部故意猜 0 並根據 money 的變化來取得前 64 次的結果，接著用前面的 state-transition matrix 做矩陣運算後可以得到 init_state。

```
# M*ret = seq
M = Matrix(FF, total)
seq = Matrix(FF, seq).T
ret = (M^-1)*seq
ret = ret.T[0]
print(f'ret: {ret}')


init_state = 0
for i in range(64):
    init_state += ret[i]*x^i
```

(4) 取得 flag

取得 init_state 後，先做 64 次自己定義 gf_step(一次 gf_step 等同於做了 43 次 step)，來讓 init_state 來到我們前面亂猜完 64 次後的 state，接著照 gf_step 每次回傳的結果來往下猜，即可 100%預測出之

後的 0、1，最終即可超過 2.4 元拿到 flag。

```
def gf_step():
    global init_state
    init_state = init_state * (x^43)
    ret = init_state.integer_representation()
    return (ret >> 63) & 1
```

Reference:

https://hackmd.io/@cXpZn6ltSku4Vwx_OL0bqA/SyyxioFgI?fbclid=IwAR2rdKeIMoHmXtc3dpou8babWXKhVR4zp2PatfipAHJB9z6lBUT1xjGFcJE#winner-winner-chicken-dinner

2. Single

(1) 求係數 a、b

首先這題並不知道橢圓曲線的係數 a、b，因此先將題目提供的三點(G、A、B)取其中兩點代入方程式，可以得二元一次聯立方程組。由於這兩個式子都是同餘方程，這邊用 **sage 的 solve()**來解，接著做一些數學運算後即可得到係數 a、b。

```
a, b = var('a, b')
eq1 = (Ay**2 - (Ax**3 + a*Ax + b)).mod(p) == 0
eq2 = (By**2 - (Bx**3 + a*Bx + b)).mod(p) == 0
res = solve([eq1, eq2], a, b)
a = Integer(Mod(res[0][0].rhs().numerator(), p) / Mod(res[0][0].rhs().denominator(), p))
b = Integer(Mod(res[0][1].rhs().numerator(), p) / Mod(res[0][1].rhs().denominator(), p))
print(f'a = {a}')
print(f'b = {b}')
```

```
a = 960527526587963100872646774064653712569216779434164082270231305661193843299 4
b = 783983860770749446375804983051536938377893194811495567698518099356920037548 0
```

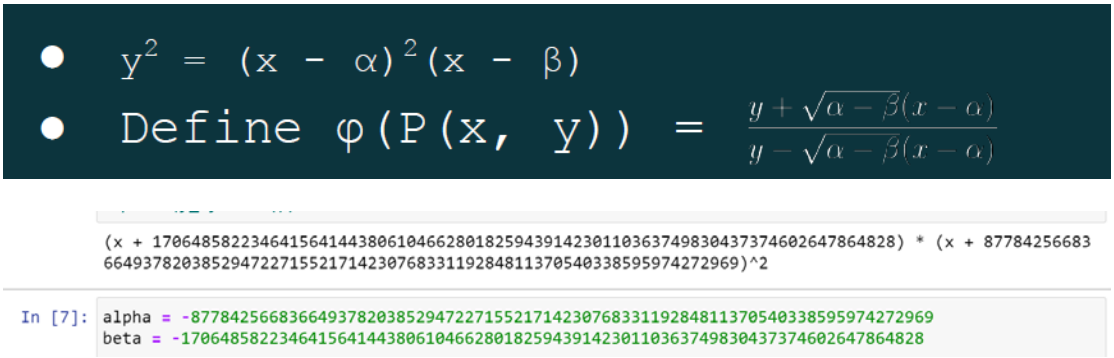
(2) 判斷是否為 singular curve

將求出來的 a、b 帶入(4*a^3 + 27*b^2)%p，會得到 0，即可確定這

是個 **singular curve**。

(3) 求出 phi function 中的 alpha、beta

對原式 call .**factor()** 後，比對係數即可得到 alpha、beta。




```
(x + 17064858223464156414438061046628018259439142301103637498304373746026478648 28) * (x + 87784256683
6649378203852947227155217142307683311928481137054033859597427296 9)^2
```

```
In [7]: alpha = -87784256683664937820385294722715521714230768331192848113705403385959742729 69
        beta = -17064858223464156414438061046628018259439142301103637498304373746026478648 28
```

(4) 由公式回推出 dA

由 A = G*dA，可以推得 phi(A) = phi(G*dA) = phi(G)^dA (singular curve 中 node 的性質)，接著帶入 **discrete_log()**即可解出 dA。

```
t = GF(p)(alpha-beta).square_root() # get the (alpha-beta)^(1/2) % p
phi = (y + t*(x-alpha)) / (y-t*(x-alpha))

u = phi(A[0], A[1]) % p
v = phi(G[0], G[1]) % p
dA = discrete_log(u, v)
dA
```

```
: 153248752161246289457951716360635928598956820351573408309956740278043319005 2
```

最後取出 B*dA 的 x，即可以得到 key 值 k，拿 k 來還原出 flag。

Reference:

https://crypto.stackexchange.com/questions/61302/how-to-solve-this-ecdlp

3. HNP-revenge

(1) 判斷題目

觀察題目所給的 k 會發現，他是一個 256bits 所形成的 integer，其中**前 128bit 為一定值**，而後 128bit 來自一**隨機產生**的 prikey 與 bytes Kuruwa 所組成，由此可知，這個 h 的 high bits 是已知的，故可套用講義上的 lattice basis，但這邊 **k 的 high bits 並不是都為 0**，故下面求解 u、t 時需做些代換。

```
else:
    h = sha256(msg.encode()).digest()
    k = int(md5(b'secret').hexdigest() + md5(long_to_bytes(prikey.secret_multiplier) + h).hexdigest(), 16)
    sig = prikey.sign(bytes_to_long(h), k)
    print(f'({sig.r}, {sig.s})')
```

- Construct lattice basis

$$B = \begin{bmatrix} n & 0 & 0 \\ t & 1 & 0 \\ u & 0 & K \end{bmatrix}$$

(2) 取得 r1、s1、r2、s1

與本週的 lab1 相同，前兩次 option 先用不同的兩個 message 來得到兩組 r 與 s，以便第三次 option 的計算。

(3) 計算 lattice basis 中的 u、t

進入第三次 option 時，首先將 k 中前 128bit 的值用一個變數 w 來替換，接著照著講義的公式移項之後帶入數值，即可求得 u、t。

- Eliminate the variable d
  - $k_1 - s_1^{-1}s_2r_1r_2^{-1}k_2 + s_1^{-1}r_1h_2r_2^{-1} - s_1^{-1}h_1 \equiv 0 \bmod n$
- Let $t = -s_1^{-1}s_2r_1r_2^{-1}$, $u = s_1^{-1}r_1h_2r_2^{-1} - s_1^{-1}h_1$
  - $k_1 + tk_2 + u \equiv 0 \bmod n$
- We wish to solve k_1 and k_2, both small.
  - Let $|k_1|, |k_2| < K$

```
    # calculate some value for lattice
    w = int(md5(b'secret').hexdigest(), 16) << 128
    t = -s2*r1*inverse_mod(s1, n)*inverse_mod(r2, n) % n
    u = (r1*h2*inverse_mod(s1, n)*inverse_mod(r2, n) - h1*inverse_mod(s1, n) + w + t*w) % n
```

(4) 用 LLL 求解 approximated SVP

這邊 K 取用 n 的開根號-1(也就是 2^128-1)，找出符合條件的 vector，即 abs(k1)和 abs(k2)皆小於 K。求出來後加上原本的 w 即可還原出原始的 k，接著帶入公式後即可還原出 d。

```
# use LLL to get the SV
k1 = k2 = 0
K = int(n^(1/2))-1
B = matrix(ZZ, [[n, 0, 0], [t, 1, 0], [u, 0, K]])
for V in B.LLL():

    if V[2] == K and abs(V[0]) < K and abs(V[1]) < K:
        print(V)
        k1 = -V[0]
        k2 = V[1]
        break
# resotre the k1 and k2
k1 += w
k2 += w
```

- Two singature (r_1, s_1), (r_2, s_2), both use small nonces k
  - $s_1 \equiv k_1^{-1}(h_1 + dr_1) \bmod n$
  - $s_2 \equiv k_2^{-1}(h_2 + dr_2) \bmod n$

(5) 由 d 反推出題目的 prikey

用題目固定的 G 與推出來的 d 即可產生 pubkey、prikey，接著再用該 prikey 去簽 bytes Kuruwa 的 sha256，即可得到正確對應的 r、s，最終即可通過認證並得到 flag。

(6) 解決不能每次成功得到 flag 的問題

每次的 prikey 產生都是 random 的，因此每次對應到的 lattice basis 也都不同。**LLL 只能保證求出一個近似 SVP 的 vector**，而這 vector 不一定剛好對應到題目所要的 k，會導致之後推導出來的 d 未必是題目一開始簽所用的 d，因此用這個 d 產生的 prikey 來簽會得到 Bad signature。這邊將 pwntools 連上 server 前用個無限迴圈包住，直到求出 flag 才跳出迴圈，解決無法每次成功求解的問題。