# Secure Programming 2020

# HW8 write-up

## wishMachine

## 腳本附在/wishMachine/exploit.sh(需要跟wishMachine在同一個目錄下)

用cutter看會看到main,main裡有一個1000次的loop,跑過了就有flag.

```
while ((int32_t)(uint32_t)var_a4h < 1000) {
        printf(param_1, param_2, param_3, param_4, param_5, param_6, param_7,
param_8,
               (int64_t)"One by one, What is the serial number of coin%d ?",
(uint64_t)(uint32_t)var_a4h, iVar2, arg4,
               in_R8, in_R9);
        iVar2 = memset((int64_t)&var_a4h + 4, 0, 0x46);
        scanf(iVar2, param_2, param_3, param_4, param_5, param_6, param_7,
param_8, (int64_t)"%70s",
                      (int64_t)&var_a4h + 4, arg3, arg4, in_R8, in_R9);
        fcn.00400e0a((int64_t)&var_a4h + 4);
        param_1 = fcn.00400f69((int64_t)&var_50h, (int64_t)&var_a4h + 4);
        var_a4h._0_4_ = (uint32_t)var_a4h + 1;
        iVar2 = extraout_RDX;
    }
    printf(param_1, param_2, param_3, param_4, param_5, param_6, param_7,
param_8, (int64_t)"Ok, the flag is %s",
           (int64_t)&var_50h, iVar2, arg4, in_R8, in_R9);
```

會先印coin的訊息,然後要求輸入,輸入的長度是70個byte,之後會進到func_400e0a:

```
for ( i = 0; i <= 69; i += loop_count )
  {
    input = a1;
    check_index = *((_DWORD *)&unk_6D5114 + 10 * dword_8A1070);
    loop_count = *((_DWORD *)&unk_6D5118 + 10 * dword_8A1070);
    check_value = *((_DWORD *)&unk_6D511C + 10 * dword_8A1070);
    dword_8A2120 = dword_6D5120[10 * dword_8A1070];
    dword_8A2110 = *((_DWORD *)&unk_6D5110 + 10 * dword_8A1070);
    main_calculate_func_call = *((_QWORD *)&unk_6D5100 + 5 * dword_8A1070) +
dword_8A2110;
    ((void (*)(void))main_calculate_func_call)();
    ++dword_8A1070;
    result = (unsigned int)loop_count;
  }
```

前面先經過一連串的賦值,會call一個function pointer,用disassembly看就是call rdx,而pointer的值是經過細算而成,可能會不一樣.

實際用gdb trace一下,結果程式直接結束,後來發現是ptrace在搞鬼,查了一下才知道這是防止debug的手段,因為gdb也是用ptracex來attach程式,只要程式自己先call ptrace,gdb就不能再attatch上去,解決的發法很簡單,只要直接跳過不執行ptrace,再把return值的rax設成0記可以繼續debug了.

用gdb執行,到了call function pointer的地方,發現第一個function pointer是0x4011d6.

```
__int64 result; // rax
  int v1; // [rsp+Ch] [rbp-14h]
  int i; // [rsp+10h] [rbp-10h]
  int v3; // [rsp+14h] [rbp-Ch]
  signed int v4; // [rsp+18h] [rbp-8h]
  signed int j; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = (unsigned int)loop_count;
    if ( i >= loop_count )
      break;
    v3 = 0;
    v4 = 1;
    for ( j = 0; j < *(char *)(&qword_8A2108 + check_index + i); ++j )
    {
      v1 = v3 + v4;
      v3 = v4;
      v4 = v1;
    }
    if ( v1 != *((_DWORD *)&qword_8A2108 + i + 4LL + 1) )
      sub_40F130(0LL);
  }
```

追進去看會發現這其實就是個fibonacci的function,程式會從input中的某個byte取值當成fibonacci算的次數,最後再拿這個值跟memory中的某個檢查值比對,通過才能繼續,沒有就直接exit,而且這個function執行的次數被一個loop count決定(有可能不是1).

照理來說這些檢查值應該都放在binary裡,所以應該可以在某些區段找到這些值.先用gdb的awatch去跟0x8a2108也就是檢查值會放的地方,結果發現賦值的地方就在func_400e0a前面那一串賦值的操作.

```
for ( i = 0; i <= 69; i += loop_count )
  {
    input = a1;
    check_index = *((_DWORD *)&unk_6D5114 + 10 * dword_8A1070);
    loop_count = *((_DWORD *)&unk_6D5118 + 10 * dword_8A1070);
    check_value = *((_DWORD *)&unk_6D511C + 10 * dword_8A1070);
```

看一下0x6d5110一帶的區塊,會發現確實是這樣,每一次檢查的值都剛好差4*10(bytes).

```
0x6d5100:   0x00000000003fa21e  0x0000000000000000
0x6d5110:   0x0000003f00006fb8  0x15f2ac4800000001 //3f是第一次檢查的input byte
index
0x6d5120:   0x0000000000000000  0x00000000003f95c1 //0x15f2ac48是第一個檢查值
0x6d5130:   0x0000000000000000  0x0000001d00007c15 //1d是第二次檢查的input byte
index
0x6d5140:   0x2989ced100000002  0x0000000095cb62f5 //0x2989ced1是第二個檢查值
0x6d5150:   0x00000000003fb2ec  0x0000000000000000
0x6d5160:   0x0000001300005d41  0x0000014300000002
0x6d5170:   0x0000000000000208  0x00000000003fea93
0x6d5180:   0x0000000000000000  0x0000000f000026a5
0x6d5190:   0xfab1380c00000002  0x00000000fab13884
```

3f是第64個input byte index,0x15f2ac48是第一個檢查值,1d是第二個input byte index,0x2989ced1是第二個檢查值.

1d跟3f剛好差40byte,而0x15f2ac48跟0x2989ced1也差40 bytes符合discompile的結果.

另外第二次檢查的loop count是2(0x6d5140),所以會算兩次fibonacci,第二次的值剛好在上一個值的後面也就是0x95cb62f5.也是對的.

所以可以歸納出來,func_400e0a前面那塊賦值應該是一個類似struct的結構,每一個element都差40 bytes, 裡面包含檢查值,檢查input的第n個byte,loop count(每個檢查function跑幾遍),同時還有計算出這一次檢查function的pointer.

歸納一下總共有5個檢查function,分別是:

```
//0x4011d6

//0x40102d

//0x401138

//0x400fbe

//0x4010c8
```

每一次loop會根據算出的function pointer來call function.

因此其實可以把整塊儲存檢查資料的memory直接dump下來,然後直接模仿這些function的算法,反算出正確的input,就可以通過檢查了.

先用objdump把需要的memory區塊dump下來,從0x6d5100開始,總共需要40*70(input長度)*1000(loop次數) bytes.

```
objdump -s --start-address=0x6d5100 --stop-address=0x8a1008 ./wishMachine
```

再寫一個python腳本把資料整理成連續的signed int array,存在一個txt裡.

再用c把檢查function的全部複製下來.

這裡有幾個坑,像是fibonacci程式是用int存的,所以會overflow,因此在寫的時候type一定要跟程式邏輯一致,int就int,signed int就signed int.

```
oid fibonacci(int i) {
  for(int j = 0; j < mem[10*i + 6]; j++) {
    unsigned int v3 = 0;
```

```
    unsigned int v4 = 1;
    unsigned int v1 = 0;
    for ( int k = 0; ; ++k ) {
      v1 = v3 + v4;
      v3 = v4;
      v4 = v1;
      //程式用int會overflow,所以要也用int
      //直接跟檢查值比對找正確的input
      if(v1 == mem[10*i + 7 + j]) {
        input[mem[10*i + 5] + j] = (char)(k+1);
        break;
      }
    }
  }
}
```

剩下4個function也差不多,程式邏輯跟type都要跟程式完全一致.

另外func_400e0a的loop count也是從memory中來的,要記得邏輯也要完全符合(不小心加錯地方,debug好久...).

最後完全仿照程式邏輯找出1000次loop的正確input再喂給wishMachine,就可以得到flag了.

```
...
...
One by one, What is the serial number of coin998 ?
One by one, What is the serial number of coin999 ?
Ok, the flag is
FLAG{7hes3_func710n_ptrs_g1v3_m3_l0t_0o0of_f4n_I_w4n7_m00r3_11!!l1!|!}
```

# SecureContainProtect

# 腳本附在/SecureContainProtect/exploit.py

這題要先解數讀,解完數讀題目會要求輸入:

```
數獨的input:
l1l2l7l5l3l6l4l9j5h7h1h2h8hhh4h9j6ll5l4ll1ll8l3j6h9h8hh3h2h4hh1j3l6l9l8llll2l1j4h
h5h9h6hh7h8h2j5l2ll9l7l4l3llj7hh9h6h2hhh3h4j7ll6l3l1l8ll5l2z
```


根據逆向完的code,輸入的input會跟兩個array作xor,再把每一個byte加總,等於一個值後就會印出xor完的array,這個array很可能就是flag.

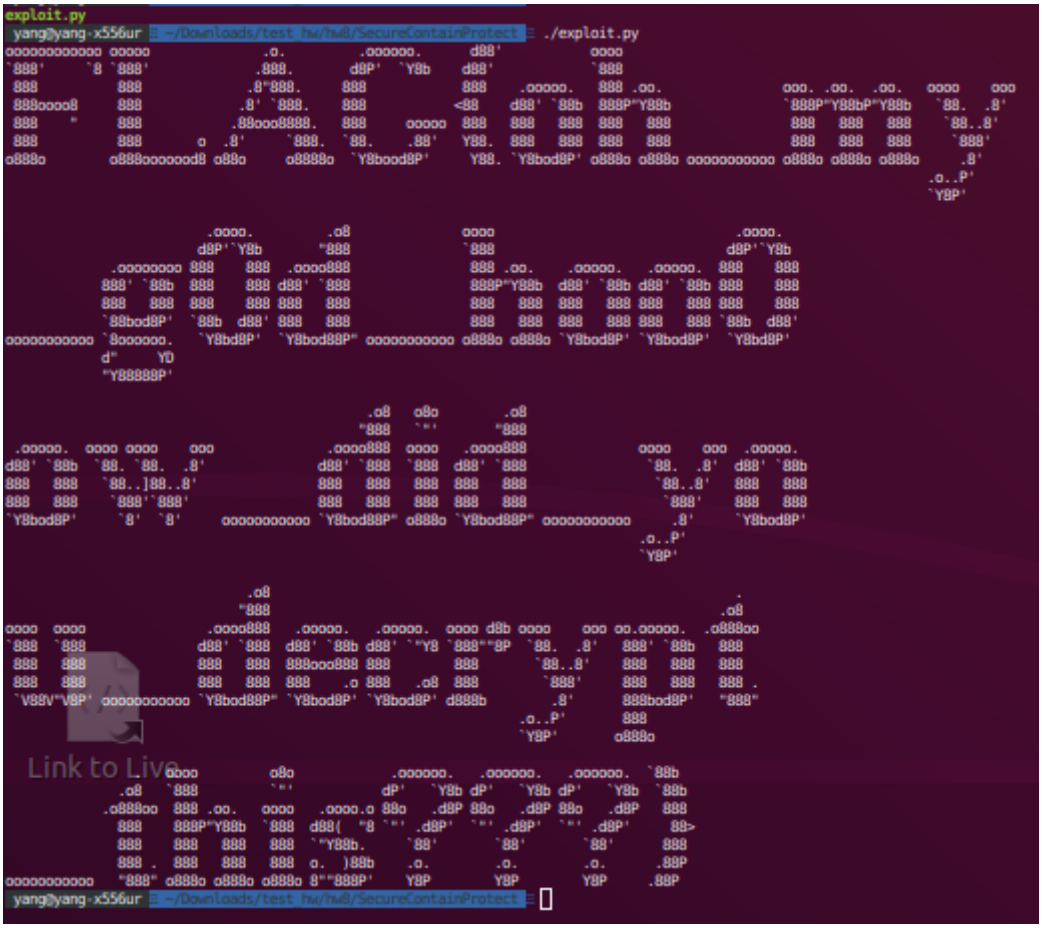
```
__isoc99_scanf("%39s", &v8); //ask input
    for ( k = 0; k <= 6014; ++k )
    {
      v0 = byte_202E00[k]; // array in binary
      v1 = byte_202020[k % 81]; //sudoku的值
      //三個值作xor
      byte_202E00[k] = v1 ^ *((_BYTE *)&v8 + k % strlen((const char *)&v8)) ^ v0;

      v6 += byte_202E00[k];
    }
    if ( v6 == 257498 )
      puts(byte_202E00);
```

可以在binary中找到byte_202e00的值,我們直接把這個array跟判斷式拉出來寫成一個腳本,方便測試.

根據提示,flag會是ascii art的形式,如果將input(v8[])設成'\0',再把byte_202e00跟byte_202020作xor的結果用ascii art印出來:

:DECRYPTTHEDOCUMENTOFscp,可能是個提示.
在沒有跟key作xor的情況下,array就已經是ascii art的格式,稍微統計一下會發現空白是所有用到的字元中最多的,key跟byte_202e00跟byte_202020作很有可能是空白,由於xor是可逆operation,因此把key設成全空白輸入會得到:

decrypt_the_document_of_SCP-2521頻繁的出現,把這個當成key輸入,會發現判斷式過了,把印出的結果縮小印出來就得到flag了:


exploit.py
yang@yang-x556ur ~/Downloads/test hw/hw8/SecureContainProtect ./exploit.py
Link to Liva
yang@yang-x556ur ~/Downloads/test hw/hw8/SecureContainProtect