Hw08

wishMachine

1. defeat gdb defender

try 了一下發現: binary 能直接跑起來, 但是 gdb 跑起來會爛掉, 所以猜有某種機制會擋掉 gdb 的 trace

而此種機制是 binary 利用 system call, 詢問 kernel 當前是否有 process 正在 trace 他, 而後做出如 exit 的處理

如果 binary 不想被 gdb trace, 會 call `ptrace(PTRACE_TRACEME, 0, 0, 0)`, 判斷 `ptrace(PTRACE_TRACEME, 0, 0, 0)` 是否小於 0 來判斷當前是否有 gdb attach 他

可以在 gdb 時, 輸入以下的 command, gdb 會在每個 ptrace 的地方下斷點 (bp 1), 並且 `commands 1` 可以指定在 bp 1 到達時做哪些行為, 在此為清空 eax, 也就是 `ptrace(PTRACE_TRACEME, 0, 0, 0)` 的 return value, 將其設為 0, 就會通過 `ptrace(PTRACE_TRACEME, 0, 0, 0)` < 0 的判斷

```
catch syscall ptrace
commands 1
set ($eax) = 0
continue
end
```

也有其他方法, 像是在 binary `ptrace(PTRACE_TRACEME, 0, 0, 0)` jle 的部分 patch 成 jmp, 就可以直接繞過

2. analyze structure

很快地發現, 他會將輸入的 70 長度字串存在某個地方, 並且 call 某個 function ptr, 在 function 中取出字串部分的字元 x, 求出其 `fibonacci( ord(x) )`, 並比對 data section 某個位置的值

try 了一下, 發現每次在 call fibonacci 前, 都會取出 data section 某位置的 40 大小, 推測為一個 structure, member 如下:

```
struct {
    int base; // 0-3
    int no[3]; // 4-15
    int offset; // 16-19
    int pos; // 20-23
    int num; // 24-27
    int fb_value1; // 28-31
    int fb_value2; // 32-35
    int no; // 36-39
}
```

- base+offset 會得到一個位置, 根據不同的位置會呼叫不同的 function, 有的是取出對應值的 fibonacci index, 有的是跟某值做 xor, 所有的 operation function 都寫在 wishMachine/exp.py 了

- no 不重要
- pos 為長度 70 的字串在 pos 位置的字元 (`str[pos-1]` 的感覺)
- fb_value1, fb_value2 為 function 的參數, 注意的是當 num = 2, fb_value2 才會有值
- num 若為 1, 則 `func(fb_value1)` 的結果會 assign 給 str[pos-1] ; 若為 2, 則還會**額外**將 `func(fb_value2)` 的結果會 assign 給 str[(pos+1)-1]

data section 放這些值的地方為 0xD5100 開始, 前 0x20 為 padding, 剩下的就直接 extract 出來即可

P.S. 因為有些時候會得到兩個 char, 有些時候只有一個, 但是總和一定會是 70 * 1000 個字元, 就寫個 script 讓他每 70 個就存成一筆, 寫到一個檔案中, 最後在 `echo serial.txt | ./patch_binary`, 得到 flag.

curse

因為上課有提過 pack, 所以大概有猜到此為加殼過的 windows binary. 而我對加殼過的 windows binary 認知即為:

1. 脫殼: 不知道是用什麼加殼, 所以就沒常識
2. 直接跟, 找大跳, 跳到的地方即為 OEP

我是採取第二種方式, 其中使用到的一些 tricks, 像是: for loop 的地方就下斷點在外圈、當 exit 時紀錄當前位置 (因為在下個斷點前就會到 OEP 了)、特別注意 jmp 或是 pop register 的地方等等

最後找到 OEP, 發現行為做的並不複雜:

- 有一個 target 會在最後做比對, 猜測為 flag
- 有一個很大的 data chunk, 內容沒什麼規律
- 某 function 會將 input 字串中的字元慢慢變成其他字元, 但是有其規律

追進去 function 後, 看到 function 會取出 data chunk, 並找出 input string 中字元在裡面出現的地方, 回傳後面一位的字元

於是拿 target 一個一個 char 丟進 data chunk 內找, 回傳前面一位, 得到原本的 input, 即是 flag

P.S. flag.txt 為 encrypt 後的 flag 字串, qq.txt 為那個大 data chunk (mapping table), exp.py 能把 input 做加解密

SecureContainProtect

一開始要先玩數讀, 大概玩了一個小時後, 直接找 online 數讀工具解決他, 並且 patch 起來XD

稍微追一下, 可以發現他是拿兩個已知的值 (數讀 sequence 跟固定 data), 與我們的 input 做 xor, 會得到某個結果, 而預期結果就是在我們輸入對的 input 時, 會有 ascii art 型式的 flag

因為兩個 key 都是已知, 可以想成: `result = input ^ const` 的感覺

如果 input 長度不夠, 他會 repeat input 做 xor, 像是 `input[i % len(input)]` 的感覺. 也因為這樣, 所以第一步會想 0~255 都 xor 一遍, xor 過程寫在 SCP/exp.py 內

而後來才想到 (input == key) `result = key ^ const` => `key = result ^ const`, 而 result 為 ascii art, 如果我們要求得 key, 可以先猜測 ascii art 會有哪些東西... 用來分隔的 ' ' (space)

雖然是一個 char 慢慢做 xor, 但是 ascii art 在字符與字符之間一定會需要用 ' ' (space) 隔開, 代表:

```
key = input('> ')
for i in range(len(key)):
    assert(key[i] ^ const[i] == ' ')
```

那我們可以反過來:

```
art = ' '*n
for i in range(n):
    assert(' ' ^ const[i] == key[i])
```

因此輸入 ' ' 當作 key, 觀察輸出結果, 能發現中間一大塊因為連續 ' ' 的關係, 跑出多次 real key, 而後我們拿 key 當作 input, 讓 ascii art 能夠完整的被 xor 出來, 就能拿到 flag 了