

# 攻防世界Reverse进阶区-EasyRE-writeup

原创

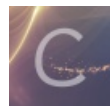
y4ung 于 2020-09-19 09:17:32 发布 532 收藏 1

分类专栏: [ctf](#) 文章标签: [ctf](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: [https://blog.csdn.net/qq\\_35056292/article/details/108676664](https://blog.csdn.net/qq_35056292/article/details/108676664)

版权



[ctf 专栏收录该内容](#)

35 篇文章 0 订阅

订阅专栏

## 1. 介绍

本题是xctf攻防世界中Reverse的进阶区的题EasyRE。

下载下来以后是一个exe文件: `210f1e18ac8d4a15904721a2383874f5.exe`

## 2. 分析

首先看下文件格式为windows 下的32位程序

```
$ file 210f1e18ac8d4a15904721a2383874f5.exe
210f1e18ac8d4a15904721a2383874f5.exe: PE32 executable (console) Intel 80386, for MS Windows
```

先在windows下运行一下, 输入时提示: `input:`, 输入以后按下回车却没有输出。

```
C:\Users\hzy\Downloads>210f1e18ac8d4a15904721a2383874f5.exe
input: 123

C:\Users\hzy\Downloads>
```

### 2.1 静态分析

先拖进IDA 里看看吧! 找到之前命令行中的提示字符串 `input`, 找到交叉引用该字符串的地方为函数`sub_401080`。(在查看strings windows的时候眼前一亮发现有个 `flag{NP2NiaNXx1C1GYVQ50}`, 但是输入以后发现并不是真正的flag...

F5查看函数`sub_401080`的伪代码。字符串 `input` 的地址为`0x402150`, 结合代码, 可以判断`sub_401020`函数为`printf`函数。再往下发现地址`0x402158`开始的字符串为`%s`, 说明`sub_401050`应该是`scanf`函数, 将用户输入保存到`v7`中。

```
● 14 sub_401020((int)&unk_402150); // printf
● 15 v9 = 0;
● 16 v10 = 0;
● 17 v7 = 0i64;
● 18 v8 = 0i64;
● 19 sub_401050((const char *)&unk_402158, &v7); // scanf
```

得到输入以后, 对`v7`的长度进行检查, 从代码中可知`v7`长度必须为24个字符。

接下来以 v8地址+7 位置处的字符赋值给v2，根据IDA的分析提示，v7的起始地址为ebp-24h(ebp-36)，v8的起始地址为ebp-14h(ebp-20)。假设我输入的是 `abcdefghijklmnopqrstuvwx`，那么在栈中应该是下面的情况。也就是v2的初始值为用户输入的最后一个字符 `x`。

高地址	ebp-13	x
	ebp-14	w
	...	...
	ebp-20	q
	...	...
↓	ebp-35	b
低地址	ebp-36	a

每次循环中通过v1控制对用户输入字符串的遍历，将v2的值赋值给v3，然后v2地址自减1，也就是逆序取下一个字符。将v3保存的当前字符赋值给数组 `byte_40336C[v1]`。所以这个部分其实就是逆序提取用户输入，保存到数组`byte_40336C`的过程。

```

23 | v1 = 0;
24 | v2 = (char *)&v8 + 7;
25 |
26 |
27 | do
28 | {
29 |     v3 = *v2--;
30 |
31 |     byte_40336C[v1++] = v3;
32 | }
33 | while ( v1 < 24 );
34 |

```

接下来对数组`byte_40336C`的每个值x进行  $(x+1)^6$ 的操作。

```

35 | v4 = 0;
36 |
37 | do
38 | {
39 |     byte_40336C[v4] = (byte_40336C[v4] + 1) ^ 6;
40 |     ++v4;
41 | }
42 | while ( v4 < 24 ); // 对数组中的每一个值进行处理
43 |

```

最后将数组`byte_40336C`，也就是一个字符串，与地址`0x402124`开始的字符串进行比较。如果相同，即`strcmp`返回值为0，则调用`printf`函数输出 `right\n`

```

44 | v5 = strcmp(byte_40336C, (const char *)&unk_402124);
45 | if ( v5 )
46 |     v5 = -(v5 < 0) | 1;
47 | if ( !v5 ) // v5必须为False
48 | {
49 |     sub_401020((int)"right\n");
50 |     system("pause");
51 | }
52 | }
53 | return 0;
54 | }

```

双击`unk_402124`，选中24个字符，按下 `Shift + E` 提取，选择 `string literal`，得到的字符串为：`xIrcj~<n|2tWsv3PtI\x7Fzndka`。

```
.rdata:00402100 ; DATA XREF: sub_40140D+ED10
.rdata:00402108 aFlagNp2nianxxx1 db 'flag{NP2NiaNXXiC1GVVQ50}',0
.rdata:00402108 ; DATA XREF: .rdata:00402160+0
.rdata:00402121 align 4
.rdata:00402124 unk_402124 db 78h ; x ; DATA XREF: sub_401080+85f0
.rdata:00402124 ; .rdata:0040215C+0
.rdata:00402125 db 49h ; I
.rdata:00402126 db 72h ; r
.rdata:00402127 db 43h ; C
.rdata:00402128 db 6Ah ; j
.rdata:00402129 db 7Eh ; ~
.rdata:0040212A db 3Ch ; <
.rdata:0040212B db 72h ; r
.rdata:0040212C db 7Ch ; |
.rdata:0040212D db 32h ; 2
.rdata:0040212E db 74h ; t
.rdata:0040212F db 57h ; W
.rdata:00402130 db 73h ; s
.rdata:00402131 db 76h ; v
.rdata:00402132 db 33h ; 3
.rdata:00402133 db 50h ; P
.rdata:00402134 db 74h ; t
.rdata:00402135 db 49h ; I
.rdata:00402136 db 7Fh ;
.rdata:00402137 db 7Ah ; z
.rdata:00402138 db 6Eh ; n
.rdata:00402139 db 64h ; d
.rdata:0040213A db 60h ; k
.rdata:0040213B db 61h ; a
.rdata:0040213C db 0
.rdata:0040213D db 0
.rdata:0040213E db 0
.rdata:0040213F db 0
.rdata:00402140 aRight db 'right',0Ah,0
.rdata:00402147 align 4
.rdata:00402148 ; char Command[]
.rdata:00402148 Command db 'pause',0
.rdata:0040214E align 10h
.rdata:00402150 unk_402150 db 69h ; i
.rdata:00402151 db 6Eh ; n
.rdata:00402152 db 70h ; p
.rdata:00402153 db 75h ; u
.rdata:00402154 db 74h ; t
.rdata:00402155 db 0A3h
.rdata:00402156 db 0BAh
.rdata:00402157 db 0
.rdata:00402158 unk_402158 db 25h ; % ; DATA XREF: sub_401080+2E10
.rdata:00402159 db 73h ; s
```

**Export data**

Export as

- hex string (unspaced)
- hex string (spaced)
- string literal
- C unsigned char array (hex)
- C unsigned char array (decimal)
- initialized C variable
- raw bytes

Save data to clipboard

Preview

```
x\rCj~\r|2t#sv3Ft|\x7Fzrndka
```

Line:1 Column:1

Output file: export\_results.txt

0000133B 0040213B: .rdata:0040213B (Synchronized with Hex View-1) [https://blog.csdn.net/qq\\_35056292](https://blog.csdn.net/qq_35056292)

至此，整个程序的逻辑很清楚了：

- 在第1个部分中，读取用户输入
- 在第2部分中，判断用户输入的长度。逆序提取用户输入，保存到数组中（其实是个字符串）
- 在第3部分中，对数组每个值x进行  $(x+1)^6$  的操作
- 在第4部分中，检查得到的数组（字符串）与 `xIrCj~<r|2tWsv3PtI\x7Fzndka` 是否相等，相等则成功解决。

```

1 int sub_401080()
2 {
3     unsigned int v0; // kr00_4
4     signed int v1; // edx
5     char *v2; // esi
6     char v3; // al
7     unsigned int v4; // edx
8     int v5; // eax
9     __int128 v7; // [esp+2h] [ebp-24h]
10    __int64 v8; // [esp+12h] [ebp-14h]
11    int v9; // [esp+1Ah] [ebp-Ch]
12    __int16 v10; // [esp+1Eh] [ebp-8h]
13
14    sub_401020((int)&unk_402150); // printf
15    v9 = 0;
16    v10 = 0;
17    v7 = 0i64;
18    v8 = 0i64;
19    sub_401050((const char *)&unk_402158, &v7); // scanf
20    v0 = strlen((const char *)&v7);
21    if ( v0 >= 16 && v0 == 24 ) // 用户输入字符串长度为24个字符
22    {
23        v1 = 0;
24        v2 = (char *)&v8 + 7; // 取v8的地址，再偏移7。 这里需要调试
25        // 这里取的是输入的最后一个字符x
26
27        do
28        {
29            v3 = *v2--; // uVar2 = *puVar9;
30            // puVar9 = puVar9 + -1;
31            byte_40336C[v1++] = v3;
32        }
33        while ( v1 < 24 ); // 将用户输入逆序，对数组进行赋值，
34
35        v4 = 0;
36
37        do
38        {
39            byte_40336C[v4] = (byte_40336C[v4] + 1) ^ 6;
40            ++v4;
41        }
42        while ( v4 < 24 ); // 对数组中的每一个值进行处理
43
44        v5 = strcmp(byte_40336C, (const char *)&unk_402124);
45        if ( v5 )
46            v5 = -(v5 < 0) | 1;
47        if ( !v5 ) // v5必须为False
48        {
49            sub_401020((int)"right\n");
50            system("pause");
51        }
52    }

```

00000490 sub\_401080:18 (401090) [https://blog.csdn.net/qq\\_35056292](https://blog.csdn.net/qq_35056292)

用python写脚本逆出正确的输入 `flag{xNqU4otPq3ys9wkDsN}` :

```

# user_input逆序，存到arr数组中
# arr中的每个字符，进行 (each+1)^6 的操作
# 将arr与target比较，相同的时候输出"right"
res = ""
for each in target:
    tmp = (ord(each) ^ 6) - 1 # 异或的优先级!!
    tmp_char = chr(tmp)
    # print("tmp:{}, tmp_char:{}".format(tmp, tmp_char))
    res += tmp_char

res = res[::-1]
print("flag:", res) # flag{xNqU4otPq3ys9wkDsN}

```

编写脚本的过程中有两个需要注意的：

### 1. 异或运算的逆运算还是异或，比如：

```
x = 5
y = x^6 # 3, 0b101 ^ 0b110 => 0b011

# 已知y, 求x
x = y^6 # 0b011 ^ 0b110 => 0b101
```

### 2. 异或运算的优先级是低于减号的：

```
5^6 -1 # => 0
(5^6) -1 # => 2
```

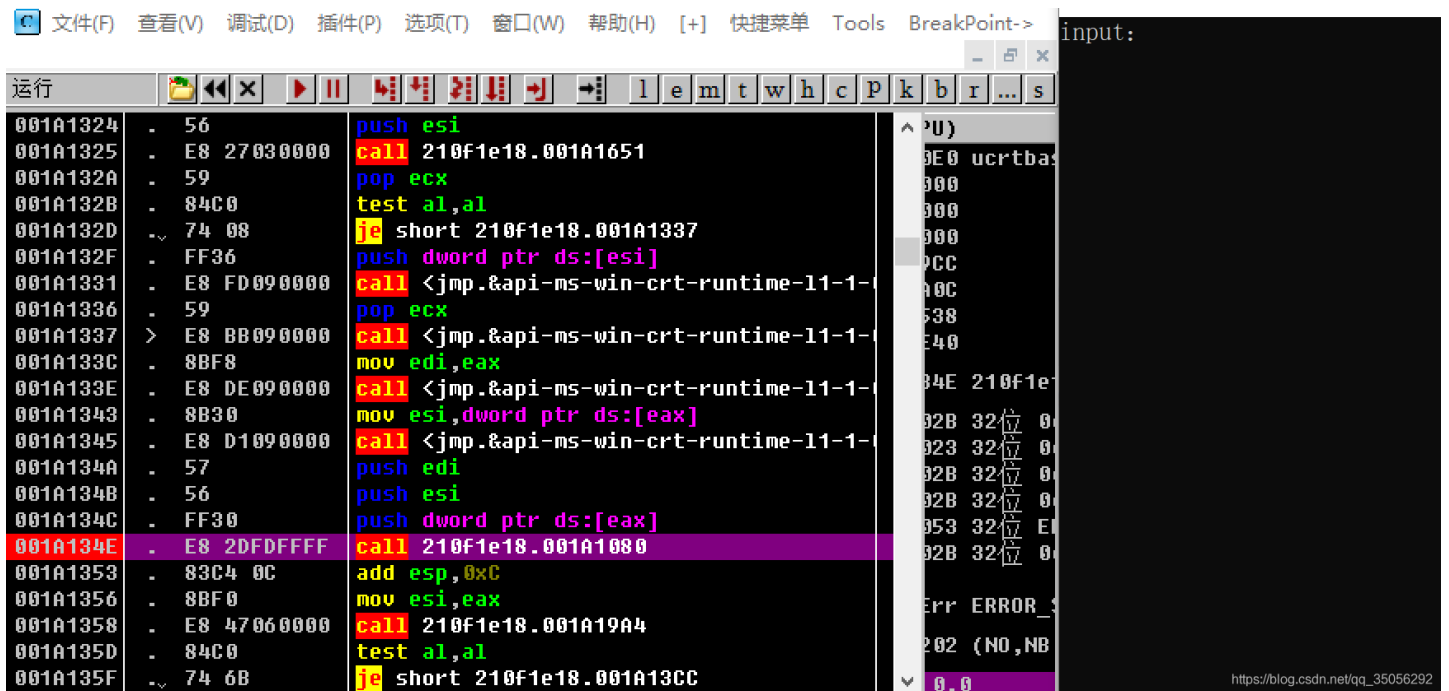
## 2.2 动态调试

一开始逆序提取那个部分有点迷，主要是不清楚那个v8的含义，也没注意到IDA在函数的前面有提示v8的地址。于是我通过Ollydbg动态调试帮助理解。

首先需要在动态调试过程中定位之前IDA分析的sub\_401080函数。可以用字符串来定位。

也可以通过程序运行的方式来定位：

运行到0x1A134E时，控制台输出了 `input:`，说明这里可能就是我们需要分析调试的地方，但是不知道是printf函数还是就是IDA里的sub\_401080函数。因此我输入以后直接F8运行，发现没有经过之前分析过的循环逻辑，程序就终止了。说明0x1A134E调用的函数应该是sub\_401080函数。



在0x1A134E按 `F2` 打断点，按 `ctrl+F2` 重新运行。当再次运行到这里的时候，按 `F7` 也就是Step Into，进入到函数中去调试。

0x1A1080就是sub\_401080函数的起始地址。

```

001A107F L. C3      retn
001A1080 $ 55      push ebp
001A1081 . 8BEC    mov ebp,esp
001A1083 . 83EC 24  sub esp,0x24
001A1086 . A1 04301A00 mov eax,dword ptr [0x1A0004]
001A108B . 33C5    xor eax,ebp
001A108D . 8945 FC  mov [local.1],eax
001A1090 . 68 50211A00 push 210F1e18.001A2158
001A1095 . E8 86FFFFFF call 210F1e18.001A2158
001A109A . 8D45 DC  lea eax,[local.1]
001A109D . C745 F4 0000 mov [local.3],eax
001A10A4 . 0F57C0  xorps xmm0,xmm0
001A10A7 . 66:C745 F8 00 mov word ptr [local.2],ax
001A10AD . 50      push eax
001A10AE . 68 58211A00 push 210F1e18.001A2158
001A10B3 . 0F1145 DC movups dqword ptr [local.4],xmm0
001A10B7 . 660Fd645 ec movq qword ptr [local.5],xmm0
001A10BC . E8 8FFFFFFF call 210F1e18.001A2158
001A10C1 . 8D4D DC  lea ecx,[local.6]
001A10C4 . 83C4 0C  add esp,0xC
001A10C7 . 8D51 01  lea edx,dword ptr [local.7]
001A10CA . 66:0F1F4400 nop word ptr [local.8]

```

需要分析IDA中 `v2 = (char *)&v8 + 7;` 的值是什么。通过调试可知是输入的最后一个字符开始，也就是 x

<pre> 001A10AD . 50      push eax 001A10AE . 68 58211A00 push 210F1e18.001A2158 001A10B3 . 0F1145 DC movups dqword ptr ss:[ebp-0x24],xmm0 001A10B7 . 660Fd645 ec movq qword ptr ss:[ebp-0x14],xmm0 001A10BC . E8 8FFFFFFF call 210F1e18.001A2158 001A10C1 . 8D4D DC  lea ecx,[local.9] 001A10C4 . 83C4 0C  add esp,0xC 001A10C7 . 8D51 01  lea edx,dword ptr ds:[ecx+0x1] 001A10CA . 66:0F1F4400 nop word ptr ds:[eax+eax] 001A10D0 &gt; 8A01    mov al,byte ptr ds:[ecx] 001A10D2 . 41      inc ecx 001A10D3 . 84C0    test al,al 001A10D5 ^ 75 F9   jnz short 210F1e18.001A10D0 001A10D7 . 2BCA    sub ecx,edx 001A10D9 . 83F9 10 cmp ecx,0x10 001A10DC ~ 0F82 91000000 jb 210F1e18.001A1173 001A10E2 . 83F9 18 cmp ecx,0x18 001A10E5 ~ 0F85 88000000 jnz 210F1e18.001A1173 001A10EB . 33D2    xor edx,edx 001A10ED . 85C9    test ecx,ecx 001A10EF ~ 7E 20   jle short 210F1e18.001A1111 001A10F1 . 56      push esi 001A10F2 . 8D740D DB lea esi,dword ptr ss:[ebp+ecx-0x25] 001A10F6 . 66      data16 001A10F7 . 66:0F1F8400 nop word ptr ds:[eax+eax] 001A1100 &gt; 8A06    mov al,byte ptr ds:[esi] 001A1102 . 8D76 FF lea esi,dword ptr ds:[esi-0x1] 001A1105 . 8882 6C331A00 mov byte ptr ds:[edx+0x1A336C],al 001A110B . 42      inc edx 001A110C . 3BD1    cmp edx,ecx 001A110E ^ 7C F0   jl short 210F1e18.001A1100 001A1110 . FF      pop esi </pre>	<pre> UNICODE "猥"  scanf  取用户输入的长度，存到ecx寄存器  edx是v1，通过xor edx, edx置为0  需要观察一下这个地址的字符串是啥  取esi中的当前字符x，赋值给寄存器al esi地址减1的地址赋值给esi，也就是w al的值赋值给数据段的数组[0x1a336c]，edx控制数组的偏移 </pre>
---	---

### 3. 总结

可以根据输出字符串找到需要分析的函数，然后通过静态分析寻找用户输入在函数中的变换，如果有不懂的先跳过，后续用Ollydbg动态调试辅助理解。先对整个函数的逻辑，每个块的作用有个宏观的了解。