XCTF 逆向进阶区Windows_Reverse1——WriteUp

原创

Code Segment 于 2020-02-20 16:20:00 发布 731 收藏 5

分类专栏： CTF Reverse 文章标签： 安全 debug

本文链接：https://blog.csdn.net/qq_43547885/article/details/104412656

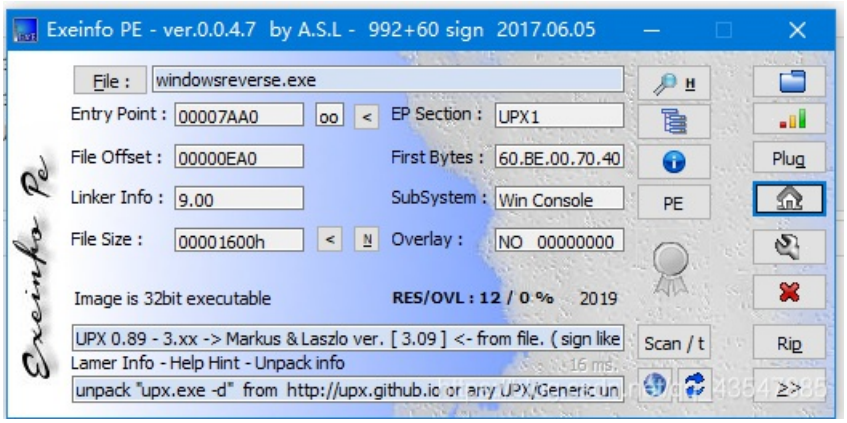
题目链接：

Windows_Reverse1

答题过程

脱壳

- 首先将程序拖入 Exeinfo PE 中查看相关信息：



- 发现程序有`upx`壳, 使用`upx -d`将壳脱去:

脱壳显示成功, 但奇怪的是程序一运行就闪退. 先不讨论该问题, 最后再来研究.

IDA PRO 分析程序

- 用 IDA PRO 打开该程序, 直接查看反编译代码, 发现程序逻辑非常清晰:

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
int result; // eax@2
char v4; // [sp+4h] [bp-804h]@1
char v5; // [sp+5h] [bp-803h]@1
char v6; // [sp+404h] [bp-404h]@1
char Dst; // [sp+405h] [bp-403h]@1

v6 = 0;
memset(&Dst, 0, 0x3FFu);
v4 = 0;
memset(&v5, 0, 0x3FFu);
printf("please input code:");
scanf("%s", &v6);
sub_401000(&v6);
if ( !strcmp(&v4, "DDCTF{reverseME}") )
{
    printf("You've got it!!%s\n", &v4);
    result = 0;
}
else
{
    printf("Try again later.\n");
    result = 0;
}
return result;
}
```

  大概过程为:

  1. 读入字符串, 存入变量`v6`中
  2. 将字符串用`sub_401000`进行变换, 变换后的字符串应位于`v4`变量处, 再与`DDCTF{reverseME}`进行比较, 相同则通过认证

`sub_401000`的分析

- 首先考虑: 为何IDA在反编译时不能正确的将传入`sub_401000`的参数识别出来
  - 查看`main`函数的反汇编代码, 发现了问题所在.

```
push    offset aS       ; "%s"
call    ds:scanf
lea     eax, [esp+82Ch+var_404]
push    eax
lea     ecx, [esp+830h+var_804]
call    sub_401000
```

    即出题人修改过该程序的反汇编代码, 使用了一个寄存器来进行了参数的传递

- 查看`sub_401000`的反编译代码

```
v2 = 0;
result = strlen(a1);
if ( result )
{
    v4 = a1 - v1;
    do
```

```
    {
    *v1 = byte_402FF8[v1[v4]];
    ++v2;
    ++v1;
    result = strlen(a1);
    }
    while ( v2 < result );
}
return result;
```

乍一看还有点懵. 但其实此处的代码逻辑并不复杂:

1. `a1` 是通过压栈的方式传递的参数; `v1` 是通过寄存器保存地址的方式传递的参数
2. 最令人迷惑的便是 `v1[v4]` 这个地方. `v1` 是一个地址, `v4` 是 `a1` 和 `v1` 两个地址间的差值. 地址的差值是怎么成为一个数组的索引的呢？这里卡了我好长时间, 之后我突然意识到, `v1[v4]` 和 `v1+v4` 是等价的, 而在循环刚开始的时候 `v1+v4` 等于 `a1`, 随着 `v1` 的递增, `v1[v4]` 也会遍历 `a1` 数组中的各个元素的地址
3. 而地址又怎么能作为数组的索引呢? 这里就是 IDA 背锅了, 换言之, 做题还是不能太依赖于反编译后的伪代码. 查看了反汇编代码后, 发现其实是将 `a1` 字符串中的字符本身作为 `byte_402FF8` 的索引, 取值后放入 `v1` 数组中
4. 查看位于 `byte_402FF8` 的值.

```
.rdata:00402FF8 ; char byte_402FF8[]
.rdata:00402FF8 byte_402FF8     db ?                    ; DATA XREF: sub_401000+24↑r
.rdata:00402FF9                 db    ? ;
.rdata:00402FFA                 db    ? ;
.rdata:00402FFB                 db    ? ;
.rdata:00402FFC                 db    ? ;
.rdata:00402FFD                 db    ? ;
.rdata:00402FFE                 db    ? ;
.rdata:00402FFF                 db    ? ;
.rdata:00402FFF _rdata          ends
.rdata:00402FFF
.data:00403000 ; Section 3. (virtual address 00003000)
.data:00403000 ; Virtual size                  : 000003E4 (    996.)
.data:00403000 ; Section size in file          : 00000200 (    512.)
.data:00403000 ; Offset to raw data for section: 00001600
.data:00403000 ; Flags C0000040: Data Readable Writable
.data:00403000 ; Alignment     : default
.data:00403000 ; ===========================================================================
.data:00403000
.data:00403000 ; Segment type: Pure data
.data:00403000 ; Segment permissions: Read/Write
.data:00403000 _data            segment para public 'DATA' use32
.data:00403000                  assume cs:_data
.data:00403000                  ;org 403000h
.data:00403000 ___security_cookie db 4Eh                ; DATA XREF: _main+6↑r
.data:00403000                                          ; __security_check_cookie(x)↑r ...
.data:00403001                  db 0E6h ;
.data:00403002                  db  40h ; @
.data:00403003                  db 0BBh ;
.data:00403004 byte_403004      db 0B1h                 ; DATA XREF: ___report_gsfailure+B0↑r
.data:00403004                                          ; ___security_init_cookie+2B↑w ...
.data:00403005                  db  19h
.data:00403006                  db 0BFh ;
.data:00403007                  db  44h ; D
.data:00403008                  db 0FFh
.data:00403009                  db 0FFh
.data:0040300A                  db 0FFh
.data:0040300B                  db 0FFh
.data:0040300C                  db 0FFh
.data:0040300D                  db 0FFh
.data:0040300E                  db 0FFh
.data:0040300F                  db 0FFh
.data:00403010 dword_403010     dd 0FFFFFFFEh           ; DATA XREF: .text:004013E2↑r
.data:00403014 dword_403014     dd 1                    ; DATA XREF: .text:004013C8↑r
.data:00403018 aZyxwvutsrqponm db '~}|{zyxwvutsrqponmlkjihgfedcba`_^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?>'
.data:00403018                  db '=<;:9876543210/.-,+*)(',27h,'&%$#"! ',0
```

我吊, 这一堆问号是什么. 这里就要考验你思维的连贯性了. 要知道 `ASCII` 编码表里的可视字符就得是32往后了, 所以, `byte_402FF8` 里凡是位于32以前的数统统都是迷惑项. 不会被索引到的. 往下看, 咦? 这一坨是什么. 再一算地址的差值. 没错, 这一坨就是与 `a1` 数组中的各个元素进行替换的字符！

脚本解密

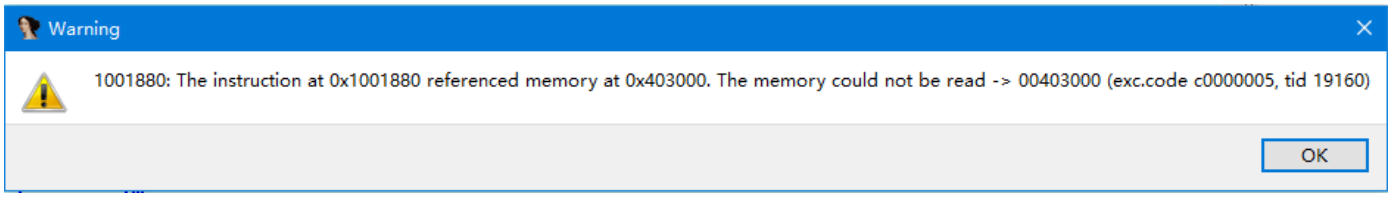
- 整道题的思路都理清了, 写个Python脚本解密就完事儿了~

```
a = [126, 125, 124, 123, 122, 121, 120, 119, 118, 117,
    116, 115, 114, 113, 112, 111, 110, 109, 108, 107,
    106, 105, 104, 103, 102, 101, 100,  99,  98,  97,
    96,  95,  94,  93,  92,  91,  90,  89,  88,  87,
    86,  85,  84,  83,  82,  81,  80,  79,  78,  77,
    76,  75,  74,  73,  72,  71,  70,  69,  68,  67,
    66,  65,  64,  63,  62,  61,  60,  59,  58,  57,
    56,  55,  54,  53,  52,  51,  50,  49,  48,  47,
    46,  45,  44,  43,  42,  41,  40,  39,  38,  37,
    36,  35,  34,  33,  32]
b = r"DDCTF{reverseME}"
for c in b:
    tmp = ord(c)
    for i in range(len(a)):
        if a[i] == tmp:
            print(chr(32+i), end='')
```

- 答案为`flag{ZZ[JX#,9(9,+9QY!}`

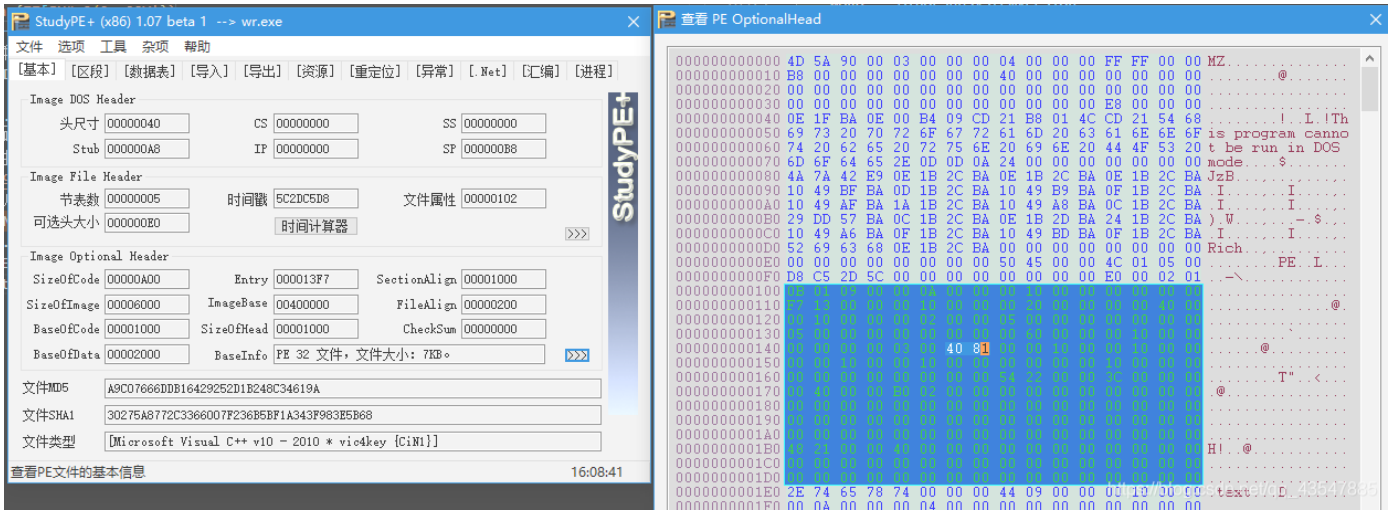
最后的一点疑惑

- Oh no, 还剩最后一点问题没有弄明白, 为什么脱壳后的程序无法直接执行呢. 用 OD 动态调试也不大灵, 于是菜鸡又打开了 IDA 进行动态调试, 看看能不能找到那行让程序崩溃的代码
- 哟! 报错了



这是神魔恋啊, 猜测是程序读取到了不在自己运行空间的内存地址? 为啥会这样呢

- 无奈上网去搜dalao们的writeup, 在这篇blog里看到了解决方案: http://blog.eonew.cn/archives/749
- 原来程序开启了ASLR(地址随机化), 并且出题人又在程序中采用了绝对地址的方式□, 所以程序才不能正常运行.
- 关掉它的方法很简单， 只需将该PE文件的IMAGE_OPTIONAL_HEADER\DllCharacteristics中的MAGE_DLLCHARACTERISTICS_DYNAMIC_BASE标志去掉即可. 即将PE中8140的数据改为8100
- 用Study_PE编辑该文件, 在 `IMAGE_OPTIONAL_HEADER` 偏移+5eh的地方(单位是Byte)即可找到DllCharacteristics. 修改为8100即可



- 修改完成后程序即可成功运行

总结

- 总的来说这题不难. 但是有点偏，小坑不断. 下次再遇到这种小坑要当心不要跳进去了~