

攻防世界reverse新手区writeup

原创

[a370793934](#) 于 2019-11-27 15:55:30 发布 1026 收藏 3

分类专栏: [WriteUp](#) 文章标签: [攻防世界](#) [reverse](#) [writeup](#) [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/a370793934/article/details/103277329>

版权



[WriteUp](#) 专栏收录该内容

20 篇文章 2 订阅

订阅专栏

第一题 **re1.exe**

0x01.运行程序

可以看到需要输入正确的flag

那么现在, 我们需要判断程序是多少位的, 有没有加壳

0x02.exeinfope查详细信息

可以看到程序是32位的, 是Microsoft Visual c++编译的, 并且没有加壳

注: 查壳工具还有PEID, EID, 但是推荐EID或者exeinfope, 因为, PEID查壳的时候有时候不准确

那么, 我们可以用静态分析神器 IDA 打开, 进一步分析了

0x03.

然后, 查找主函数main, 可以看到右侧的是反汇编的汇编代码, 这时候, 我们可以直接分析汇编语言, 但是, 汇编语言看起来太多, 费劲。这个时候就可以是有IDA是最强大的功能F5了, 它能够直接将汇编代码生成C语言代码, 虽然和这个程序的源码不完全一样, 但是逻辑关系是一样的

F5查看伪代码

这是整个main函数的运算逻辑

可以看到一个关键的字符串，`print(aFlag)`，那么证明这就是输入正确flag，然后，会输出aFlag证明你的flag正确，然后，继续往上分析，可以看到v3的值，是由`strcmp()`决定的，比较v5和输入的字符串，如果一样就会进入后面的if判断，所以,我们继续往上分析，看看哪里又涉及v5，可以看到开头的`_mm_storeu_si128()`，对其进行分析发现它类似于`memset()`,将`xmmword_413E34`的值赋值给v5，所以，我们可以得到正确的flag应该在`xmmword_413E34`中，然后，我们双击413E34进行跟进

可以看到一堆十六进制的数

这时，我们使用IDA的另一个功能 R ，能够将十进制的数转换为字符串。

这就是我们最后的flag了

注：这里要跟大家普及一个知识了，及大端与小端

假设一个十六进制数0x12345678

大端的存储方式是：12,34,56,78，然后读取的时候也是从前往后读

小端的存储方式是：78,56,34,12，然后读取的时候是从后往前读取

所以，最后的flag应该是：DUTCTF{We1c0met0DUTCTF}

0x04.运行程序输入正确的flag

DUTCTF{We1c0met0DUTCTF}

第二题 game.exe

0x01.运行程序

可以看到程序应该是输入正确顺序使八个图案都变亮

0x02.查看详细信息

我们直接使用神器IDA

0x04.IDA

F5查看伪代码

然后，我们跟进Main函数

可以看到这是整个函数的运算逻辑

先是判断是输入的是否是1-8，然后进入后面的if判断然后进行循环，这个时候应该就是程序的亮暗的显示，然后，如果byte_532E28每一位都是1，那么，就会进入sub_457AB4,然后我们猜测这里应该就是最后的flag的地方。

然后我们跟进 sub_457AB4

注：这里说明一下，如果IDA不能正确的获得自定义函数的名字，那么IDA会用sub__加上自定义函数的起始地址来定义函数的名字

这里只截取了后面的部分，发现函数进行了两次xor运算，xor的逆运算也是xor，那么我们就可以根据这个运算来写脚本得到最后的flag

这里看到v2和v59这就证明了这是两个数组的运算，所以我们应该将上面的字符串分成两个数组，分别从v2和v59开始

0x05.写EXP

这里先是通过循环，将a和b数组的值进行xor运算，然后再将数组a的值与0x13xor运算

chr(): 是将十六进制转换为字符串

0x05.运行脚本

```
#coding:utf-8
```

```
b=[18, 64, 98, 5, 2, 4, 6, 3, 6, 48, 49, 65, 32, 12, 48, 65, 31, 78, 62, 32, 49, 32, 1, 57, 96, 3, 21, 9, 4, 62, 3, 5, 4, 1, 2, 3, 44, 65, 78, 32, 16, 97, 54, 16, 44, 52, 32, 64, 89, 45, 32, 65, 15, 34, 18, 16, 0]
```

```
a=[123, 32, 18, 98, 119, 108, 65, 41, 124, 80, 125, 38, 124, 111, 74, 49, 83, 108, 94, 108, 84, 6, 96, 83, 44, 121, 104, 110, 32, 95, 117, 101, 99, 123, 127, 119, 96, 48, 107, 71, 92, 29, 81, 107,
```

```
90, 85, 64, 12, 43, 76, 86, 13, 114, 1, 117, 126, 0]
```

```
c = ""
```

```
i = 0
```

```
while (i<56):
```

```
a[i]^=b[i]
```

```
a[i]^=19
```

```
c = c + chr(a[i])
```

```
i += 1
```

```
print c
```

得到最后的flag: zscftf{T9is_tOpic_1s_v5ry_int7resting_b6t_others_are_n0t}

第三题 Hello, CTF

0x01.运行程序

输入正确的flag,才会显示正确

0x02.查壳

是32位的程序，并且是Microsoft Visual C++编译，而且没有加壳

0x03.IDA

照旧，依旧先从main开始分析，然后，对main函数进行F5查看伪代码

首先，可以看到先是将字符串复制到v13的位置，

然后，后面对输入进行了判断，输入的字符串不能大于17

接着，将字符串以十六进制输出，然后，再将得到的十六进制字符添加到v10

最后，进行比较，看输入的字符串是否和v10的字符串相等，如果相等，则得到正确的flag

0x04.将字符串转换为十六进制

16进制"437261636b4d654a757374466f7246756e"转换为字符串

```
#coding:utf-8
```

```
a = '437261636b4d654a757374466f7246756e'
```

```
c = ""
```

```
for i in range(0,len(a),2):
```

```
s = '0x' + a[i]+a[i+1]
```

```
print s
```

```
c += chr(int(s,16))
```

```
print c
```

得到了最后的flag是：CrackMeJustForFun

第四题 open-source.c

1.打开源码

打开源码

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[]) {
5     if (argc != 4) {
6         printf("what?\n");
7         exit(1);
8     }
9
10    unsigned int first = atoi(argv[1]);
11    if (first != 0xcafe) {
12        printf("you are wrong, sorry.\n");
13        exit(2);
14    }
15
16    unsigned int second = atoi(argv[2]);
17    if (second % 5 == 3 || second % 17 != 8) {
18        printf("ha, you won't get it!\n");
19        exit(3);
20    }
21
22    if (strcmp("h4cky0u", argv[3])) {
23        printf("so close, dude!\n");
24        exit(4);
25    }
```

```
26
27  printf("Brr wrrr grr\n");
28
29  unsigned int hash = first * 31337 + (second % 17) * 11 + strlen(argv[3]) - 1615810207;
30
31  printf("Get your key: ");
32  printf("%x\n", hash);
33
34  return 0;
35 }
```

2. 分析

很明显，第29行计算flag，第32行代码输出十六进制形式。第29行代码就是利用argv[1]~argv[3]的数据进行计算。

2.1 argv[1]

```
if (first != 0xcafe) {
    printf("you are wrong, sorry.\n");
    exit(2);
}
```

不等于0xcafe就退出，那first=0xcafe

2.2 argv[2]

```
if (second % 5 == 3 || second % 17 != 8) {
    printf("ha, you won't get it!\n");
    exit(3);
}
```

满足if条件就退出，我想到第一个不满足的数就是25，second = 25

2.3 argv[3]

```
if (strcmp("h4cky0u", argv[3])) {  
    printf("so close, dude!\n");  
    exit(4);  
}
```

相等strcmp返回0，退出if条件，那argv[3]="h4cky0u"

3.get flag!

综上，写出解flag代码

```
#include <stdio.h>
```

```
#include <string.h>
```

```
int main(int argc, char* argv[]) {
```

```
    int first = 0xcafe;
```

```
    int second = 25;
```

```
    argv[3] = "h4cky0u";
```

```
    printf("Brr wrrr grr\n");
```

```
    unsigned int hash = first * 31337 + (second % 17) * 11 + strlen(argv[3]) - 1615810207;
```

```
    printf("Get your key: ");
```

```
    printf("%x\n", hash);
```

```
    system("PAUSE");
```

```
    return 0;
```



```
}
```

运行得到flag:

c0ffee

第五题 simple-unpack

拿到文件先查壳，发现是upx的壳，elf文件

直接放到kali里面构造命令脱壳

```
Upx -d simple-unpack
```

脱壳后的文件丢进IDA，明文flag

```
flag{Upx_1s_n0t_a_d3liv3r_c0mp4ny}
```

第六题 logmein

然后进入主函数，

经过分析，可以得出：输入的字符要等于 经过处理的v7和v8的异或。v8很明显，但是v7是怎么回事呢，新手没有遇到过，所以上网查看资料，LL是长长整型，v7要转换为16进制然后在转换为字符串，而且字符是小端序，所以把得到的字符翻转然后和v8的每一位进行异或。贴出脚本：

```
a = 'harambe'
```

```
b = '!"AL_RT^L*.?+6/46'
```

```
print(b)
```

```
tmp = "
```

```
for i in range(len(b)):
    c = ord(a[i % 7]) ^ ord(b[i])
    tmp += chr(c)
print(tmp)
```

运行得flag:

RC3-2016-XORISGUD

第七题 insanity

拿到的同样是elf文件，无壳，直接放到IDA，定位主函数

逻辑简单，没啥好说的，直接跟踪strs，得到flag

9447{This_is_a_flag}

第八题 no-strings-attached

[分析过程]

0x01.查壳和查看程序的详细信息

说明程序是ELF文件，32位

0x02.使用静态分析工具IDA进行分析

然后对main函数使用F5查看伪代码

然后，对每个函数进行跟进，最后发现authenticate(),符合获得flag的函数，对其进行跟进

然后我们发现一个特殊的函数decrypt,根据字面的意思是加密,那么我们可以大概的猜测是一个对dword_8048A90所对应的字符串进行加密,

加密得到的就应该是我们需要的flag,后面的判断应该就是将字符串输出。

这里我们有两种思维方式:

第一种就是跟进decrypt然后分析它的运算逻辑,然后,自己写脚本,得到最后的flag

第二种就涉及逆向的另一种调试方式,及动态调试,这里我就用动态调试了,之前的一直是静态调试

0x03.GDB动态调试

`gdb ./no_strings_attached` 将文件加载到GDB中

既然是动态调试,那么如果让它一直不停,那我不就相当于运行了嘛,所以,我们就需要下断点,断点就是让程序运行到断点处就停止

之前通过IDA,我们知道关键函数是decrypt,所以我们将断点设置在decrypt处, `b`在GDB中就是下断点的意思,及在decrypt处下断点

`r`就是运行的意思,这里运行到了我们之前下的断点处,停止。

我们要的是经过decrypt函数,生成的字符串,所以我们这里就需要运行一步,GDB中用 `n` 来表示运行一步

然后我们就需要去查看内存了,去查找最后生成的字符串

通过IDA生成的汇编指令,我们可以看出进过decrypt函数后,生成的字符串保存在EAX寄存器中,所以,我们在GDB就去查看 `eax` 寄存器的值

`x`:就是用来查看内存中数值的,后面的200代表查看多少个, `wx`代表是以word字节查看, `$eax`代表的 `eax` 寄存器中的值

在这里我们看到0x00000000，这就证明这个字符串结束了，因为，在C中，代表字符串结尾的就是"\0",那么前面的就是经过decrypt函数生成的falq

那我们就需要将这些转换为字符串的形式

0x04.Write EXP

首先将寄存器中的值提取出来，然后利用Python的decode函数，通过"hex"的方式转化为字符串，然后输出

0x05.运行脚本

得到最后的flag: 9447{you_are_an_international_mystery}

第九题 csaw2013reversing2

[分析过程]

0x01.查壳和查看程序的详细信息

程序是32位的并且没有加壳

0x02.运行程序

发现提示Flag，但是乱码了

载入IDA，静态分析

0x03.IDA

依旧从main函数开始分析，F5查看伪代码

可以看到有个关键函数，意思是如果在动态调试器中就进入判断运行，如果没有直接弹窗，显示乱码的值

在这里我们看到了一个int 3中断，所以，我们直接OD动态调试，到达中断的位置，应该就能得到正确的flag了

这里我们对比上图的IDA，可以看出mov edx,[ebp+lpMem]对应的汇编指令地址,单步运行F8

执行了mov指令，接下来调用call，F8继续执行，执行完，edx存的就是flag的地址

最后的flag是：flag{reversing_is_not_that_hard!}

还有另外一种做法ida修改：

- sub_40100 为解密函数，必须经过，所以修改 jz short loc_4010B9 为 jmp short loc_401096.
- loc_4010B9 输出flag的函数，解密完应跳转到loc_4010B9
- 具体修改步骤：
- 修改int 3为 NOP.
- 修改 jmp short loc_4010EF 为 jmp short loc_4010B9.
- 修改 jz short loc_4010B9 为 jmp short loc_401096.

第十题 getit

ida64静态分析，按F5生成pseudocode

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char v3; // al
    __int64 v5; // [rsp+0h] [rbp-40h]
    int i; // [rsp+4h] [rbp-3Ch]
    FILE *stream; // [rsp+8h] [rbp-38h]
    char filename[8]; // [rsp+10h] [rbp-30h]
```

```
unsigned __int64 v9; // [rsp+28h] [rbp-18h]
```

```
v9 = __readfsqword(0x28u);
```

```
LODWORD(v5) = 0;
```

```
while ( (signed int)v5 < strlen(s) )
```

```
{
```

```
  if ( v5 & 1 )
```

```
    v3 = 1;
```

```
  else
```

```
    v3 = -1;
```

```
  *(&t + (signed int)v5 + 10) = s[(signed int)v5] + v3;
```

```
  LODWORD(v5) = v5 + 1;
```

```
}
```

```
strcpy(filename, "/tmp/flag.txt");
```

```
stream = fopen(filename, "w");
```

```
fprintf(stream, "%s\n", u, v5);
```

```
for ( i = 0; i < strlen(&t); ++i )
```

```
{
```

```
  fseek(stream, p[i], 0);
```

```
  fputc(*(&t + p[i]), stream);
```

```
  fseek(stream, 0LL, 0);
```

```
  fprintf(stream, "%s\n", u);
```

```
}
```

```
fclose(stream);
```

```
remove(filename);
```

```
return 0;
```

```
}
```

下面是重点代码

```
LODWORD(v5) = 0;
```

```
while ( (signed int)v5 < strlen(s) )
```


第十一题 python-trade.py

是一个pyc文件，用python反编译工具反编译后可以看到代码<https://tool.lu/pyc/>

```
import base64
```

```
def encode(message):
```

```
    s = ""
```

```
    for i in message:
```

```
        x = ord(i) ^ 32
```

```
        x = x + 16
```

```
        s += chr(x)
```

```
    return base64.b64encode(s)
```

```
correct = 'XINkVmtUI1MgXWBZXCFeKY+AaXNt'
```

```
flag = ""
```

```
print 'Input flag:'
```

```
flag = raw_input()
```

```
if encode(flag) == correct:
```

```
    print 'correct'
```

```
else:
```

```
    print 'wrong'
```

编写相应的脚本

```
#coding:utf-8
```



```
import base64

buf = base64.b64decode('XINkVmtUI1MgXWBZXCFeKY+AaXNt')

flag = ""

for i in buf:
    i = ord(i)-16
    i ^= 32
    flag += chr(i)

print flag
```

输出flag:

```
nctf{d3c0mpil1n9_PyC}
```

第十二题 maze

0x01.查壳和详细信息

可以看到程序是ELF文件,64位

0x02.IDA

对main函数使用F5,查看伪代码

从这里可以看出先是进行判断,如果满足则进入判断,开头必须是以nctf{开头的

然后往下分析

从这里可以看出进行了四个判断,然后,进入四个函数()

这里就涉及逆向的一个有意思的问题那就是迷宫问题:

迷宫问题可以参考:

<https://ctf-wiki.github.io/ctf-wiki/reverse/maze/maze/>

我们输入的应该是'!', '0', 'o', 'O', 并以此来确定上下左右移动

从上往下以此追踪, 可以发现这些函数会跳到lable15的位置, 然后, 对lable15分析, 发现特殊的字符串

然后, 猜测可能是一个8*8的迷宫

根据迷宫最后得到的flag:

```
nctf{o0oo00O000oooo..OO}
```