# 攻防世界 （reverse）no-strings-attached

原创

上山砍大树 于 2019-11-15 19:26:10 发布 

分类专栏： 逆向



本文链接：https://blog.csdn.net/qq_43656475/article/details/103089528

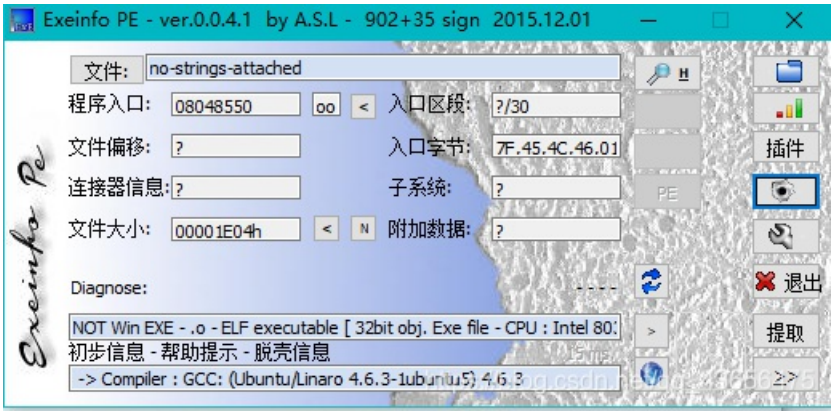




最近发现creak越来越好玩了，已经从吾爱上面下载了160个creakme，准备没事creak一下。

那么废话不多说了，直接搞题吧。

## 1.看文件类型




ELF格式的可执行文件，或许一会要用到linux环境（本人是在巨硬的windows10环境下搞的:）。

## 2.IDA静态分析

放到ida中进行分析，查看`main`函数反汇编后的伪代码是这样的：

通过各位无聊地探索，终于发现，除了`authenticate（）函数`有点意思外，其他函数对你没啥感觉。
所以直接进`authenticate（）`进行分析，而`authenticate（）`的反汇编代码形式是这样的：

```
1  void authenticate()
2  {
3    wchar_t ws[8192]; // [esp+1Ch] [ebp-800Ch] ws[]为字符型局部变量
4    wchar_t *s2; // [esp+801Ch] [ebp-Ch] s2为字符类型的指针
5
6    s2 = (wchar_t *)decrypt(&s, &dword_8048A90);  // 点开s和dword_8048A90后发现数据保存在rodata段
7    if ( fgetws(ws, 0x2000, stdin) )              // fgetws函数类似fgets(char *str, int n, FILE *stream)
8                                                  // 参数含义：
9                                                  // str    --> 这是指向一个字符数组的指针，该数组存储了要读取的字符串。
10                                                 // n      --> 这是要读取的最大字符数（包括最后的空字符）。通常是使用以 str 传递的数组长度。
11                                                 // stream --> 这是指向 FILE 对象的指针，该 FILE 对象标识了要从中读取字符的流。
12                                                 // --from 菜鸟教程
13                                                 // stdin就是标准输入，即键盘输入，这里证明了ws是需要从键盘键入的输入
14   {
15     ws[wcslen(ws) - 1] = 0;
16     if ( !wcscmp(ws, s2) )                      // ws和s2进行对比再进行下一步骤
17       wprintf(&unk_8048B44);                    // print: success! welcome back!
18     else
19       wprintf(&unk_8048BA4);                    // print: access denied!
20   }
21   free(s2);
22 }
```

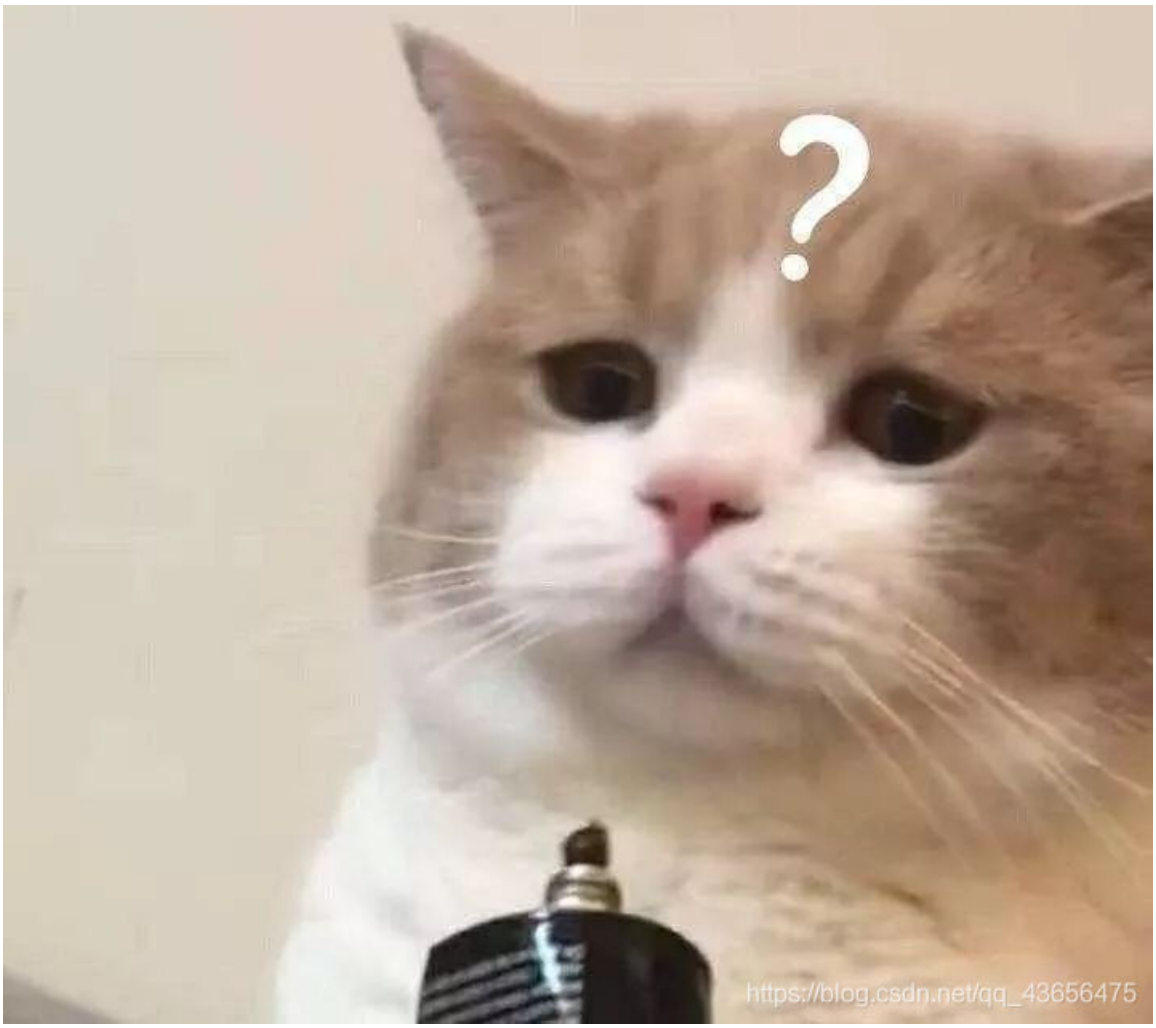


具体的注释我已经边分析边注释到代码段了。那么这个函数要告诉我们什么呢？**ws**是通过标准输入得到的字符串，然后需要通过和**s2**字符串进行对比判断。如果两段字符串相同的话，那么就`print：success！welcome back！`。
那么到了这一步，我们也就差不多明白了本题的大体思路：键入数据和flag进行对比，产生**success**或者**denied**，所以如何寻找flag就是看键入的数据跟哪个变量进行对比就ok了。由于键入数据需要和
**s2**保存的字符串机型比对，所以可以判定**s2**保存的就是flag。分析**s2**就需要分析`decrypt`函数了 。
进入`decrypt`后是这样的：

```
1  wchar_t *__cdecl decrypt(wchar_t *s, wchar_t *a2)
2  {
3    size_t v2; // eax
4    signed int v4; // [esp+1Ch] [ebp-1Ch]
5    signed int i; // [esp+20h] [ebp-18h]
6    signed int v6; // [esp+24h] [ebp-14h]
7    signed int v7; // [esp+28h] [ebp-10h]
8    wchar_t *dest; // [esp+2Ch] [ebp-Ch]
9
10   v6 = wcslen(s);
11   v7 = wcslen(a2);
12   v2 = wcslen(s);
13   dest = (wchar_t *)malloc(v2 + 1);
14   wcscpy(dest, s);
15   while ( v4 < v6 )
16   {
17     for ( i = 0; i < v7 && v4 < v6; ++i )
18       dest[v4++] -= a2[i];
19   }
20   return dest;
21 }
```



我们可以看到，函数最后返回的**dest**（也就是我们需要的**s2**）需要参数 **s**，**a2**，我们分析函数的话也需要分析传过来的参数，而返回上一级看传入的参数（上一张图），发现参数**s**和**dword_8048A90**都是 .rodata 段的数据，而 .rodata(read only data)存放的数据两种：要么存放C中的字符串和要么存放#define（摘自互联网，是否正确未知）定义的常量（文章最后放几张我对.rodata的测试）。
也就是说，**dest**是通过两个已知可读类型的变量来确定的，所以我们不妨点开**s**和**a2**看看里面的数据是什么，方不方便分析。
点开后聪明的兄弟就会发现，




这tm啥玩意？？？

- 
- 
- 

没错，参数读懂都如此艰难了，那么参数传进`decrypt`函数后再进行分析那么就是难如上青天的难度了（李白：`decrypt`难，难于上青天~）
不过我们可以不用像个铁憨憨去分析`decrypt`，而是想想**s2**会保存在哪里，我们直接找到程序运行时**s2**的保存地址dump分析不就ok了？这样做就不需要再硬着头皮分析了啊！
带着这个快乐的想法，我们就直接去看看`decrypt`的汇编代码：

```
text:08048658 ; int __cdecl decrypt(wchar_t *s, wchar_t *)
.text:08048658                 public decrypt
.text:08048658 decrypt         proc near               ; CODE XREF: authenticate+18↓p
.text:08048658
.text:08048658 var_1C          = dword ptr -1Ch
.text:08048658 var_18          = dword ptr -18h
.text:08048658 var_14          = dword ptr -14h
.text:08048658 var_10          = dword ptr -10h
.text:08048658 dest            = dword ptr -0Ch
.text:08048658 s               = dword ptr  8
.text:08048658 arg_4           = dword ptr  0Ch
.text:08048658
.text:08048658 ; __unwind {
.text:08048658                 push    ebp
.text:08048659                 mov     ebp, esp
.text:0804865B                 push    ebx
.text:0804865C                 sub     esp, 34h
.text:0804865F                 mov     eax, [ebp+s]
.text:08048662                 mov     [esp], eax      ; s
.text:08048665                 call    _wcslen
.text:0804866A                 mov     [ebp+var_14], eax
.text:0804866D                 mov     eax, [ebp+arg_4]
.text:08048670                 mov     [esp], eax      ; s
.text:08048673                 call    _wcslen
.text:08048678                 mov     [ebp+var_10], eax
.text:0804867B                 mov     ebx, [ebp+s]
.text:0804867E                 mov     eax, [ebp+s]
.text:08048681                 mov     [esp], eax      ; s
.text:08048684                 call    _wcslen
.text:08048689                 add     eax, 1
.text:0804868C                 mov     [esp], eax      ; size
.text:0804868F                 call    _malloc
.text:08048694                 mov     [ebp+dest], eax
.text:08048697                 mov     [esp+4], ebx    ; src
.text:0804869B                 mov     eax, [ebp+dest]            //从这里可以发现dest保存的值传递给了寄存器eax
.text:0804869E                 mov     [esp], eax      ; dest
.text:080486A1                 call    _wcscpy
.text:080486A6                 mov     [ebp+var_18], 0
.text:080486AD                 jmp     short loc_80486F7
.text:080486AF ; ---------------------------------------------------------------------------
```

通过汇编代码，了解到**eax**保存**dest**的值后传递给**s2**。我们刚刚头大的事情仿佛迎刃而解，直接动态调试，然后在`decrypt`处下断点单步执行到保存**dest**的值于**eax**代码后，查看**eax**保存的值就可以了。

而我们前面也分析到了，ELF文件格式在linux打开运行，所以我们要暂时抛弃微软，进入开源的怀抱~

## 3.gdb调试

自己百度gdb命令及用途，不用都记住，记住几个用得到就行了，以后用到再说。
通过gdb打开刚才分析的文件。
以下实现读取**eax**的过程代码：

*1.在decrypt下断点：*

```
b decrypt
```

*2.执行到断点所在的位置：*

```
r
```

3.执行一行源程序代码，此行代码中的函数调用也一并执行，就是执行`decrypt`函数

```
n
```

4.此时`decrypt`执行完毕，然后查看寄存器保存的内容即可

```
i r
```

5.查看内存地址（寄存器）中的值

```
x/5sw $eax
```

**参数含义：**
**x:examine–>检测内存地址中保存的值**
**5:显示5行目标数据**
**s:以字符串形式打印**
**w:以双字打印**
最后出来的格式：

```
0x8ca9800: U"9447{you_are_an_international_mystery}"
0x8ca989c: U"W\001\xf7a161e8\xf7a161ea\xf7a161ec\xf7a161ee\xf7a161f0\xf7a161f2\xf7a161f4\xf7a161f6\xf7a161f8\xf7
a161fa\001\xf7a16200\xf7a16204\xf7a16208\xf7a1620c\xf7a16210\xf7a16214\xf7a16218\xf7a1621c\xf7a16220\xf7a16224\x
f7a16228\xf7a1622a\xf7a1622c\xf7a1622e\xf7a16230\xf7a16232\xf7a16234\xf7a16236\xf7a16238\xf7a1623a\060\061\062\0
63\064\065\066\067\070\071\x175a\xf7a16268\xf7a1bfd0\xf7a27aa0\xf7a2d808\001\xf7a3ff4c"
0x8ca9968: U"\xf7a3ff54"
0x8ca9970: U""
0x8ca9974: U"\xf7a3ff7c\xf7a408b8\xf7a41234\xf7a42a74\xf7a42cec\xf7a42f64\xf7a4329c\xf7a45194\xf7a4708c\xf7a473c
4\xf7a4767c\xf7a48f74\xf7a4a7b4\xf7a4b8e4\xf7a4c854\xf7a4cb6c\xf7a52988\xf7a57ba4\021\x435f687a\x54552e4e\x382d4
6\021\x435f687a\x54552e4e\x382d46!\x8ca9a00\001\x8ca9a80\001"
```

看看哪个像是flag，你就填进去吧。
这个逆向题就这么完成了，我是跟着某个大哥的writeup做的，思路大体一样，在这里谢谢这位大哥了（这位大哥的博客排版很有感觉，是个宝贝博客~）地址：https://www.cnblogs.com/Mayfly-nymph/p/11403297.html

最后放几张看看就行的图（const和#define的讨论）

```c
#include <stdio.h>
#include <stdlib.h>
#define muagua_num 101
int max_num=100;
int min_num=20;
const int medium_num=50;
int main()
{
    int mmp=19;
    printf("%d %d %d %d %d\n",max_num,min_num,medium_num,muagua_num,mmp);
    return 0;
}
```



它编译后的反汇编的代码：

```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
  __main();
  printf("%d %d %d %d %d\n", max_num, min_num, 50, 101, 19);
  return 0;
}
```