

ctf逆向解题——re1

原创

[FunkyPants](#) 于 2019-07-22 12:20:12 发布 6349 收藏 25

分类专栏: [CTF writeup](#) 文章标签: [CTF 逆向](#)

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/FunkyPants/article/details/96843483>

版权



[CTF writeup](#) 专栏收录该内容

13 篇文章 0 订阅

订阅专栏

ctf逆向解题——re1

- re1
 - 1.题目概述
 - 2.题目知识点
 - 2.1 脱壳
 - 2.2 反汇编
 - 2.3 位运算
 - 3.做题环境
 - 4.解题思路
 - 脱壳
 - 4.1 初步观察
 - 4.1.1 part1
 - 4.1.2 part2
 - 4.1.3 part3
 - 4.2 具体分析
 - 4.2.1 part1
 - 4.2.2 part2
 - 4.2.2 part3
 - 4.3 flag
 - 5.总结收获

1.题目概述

题目给出一个带壳的32位ELF文件, 要求通过对其进行分析, 得到输入flag为何值时, 程序可以得到正确的结果。

题目文件下载链接:

链接: <https://pan.baidu.com/s/1ns8LiGSODII313skD1Mi4w> 提取码: m43u

2.题目知识点

1. 脱壳
2. 反汇编
3. 位运算

3.做题环境

- Ubuntu16.04(32bit or 64bit)
- IDA6.8
- gdb

4.解题思路

脱壳

使用upx脱壳工具直接脱壳

- -d 脱壳
- -o 指定输出文件

```
E:\吾爱破解工具包\Tools\PETools\CFF_Explorer\Extensions\CFF Explorer\UPX Utility
$ upx.exe -d re1 -o re1_upx
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2011
UPX 3.08w Markus Oberhumer, Laszlo Molnar & John Reiser Dec 12th 2011
-----
File size      Ratio  Format      Name
-----
709894 <-    302828  42.66% netbsd/elf386  re1_upx
Unpacked 1 file.
https://blog.csdn.net/FunkyPants
```

4.1 初步观察

首先观察程序运行流程，在运行后，程序等待接收一个flag，如果超过5秒未输入或输入错误则退出。

使用IDA打开脱壳后的文件，可以看到函数名称均未被正确恢复，首先我们需要定位主函数。在运行程序时我们看到程序输出了一个字符串“Please input flag”，可在strings中找到该字符串，查看交叉引用，定位到主函数。

```

aThereSOnlyOneR db 0
aPleaseInputFla db 'There',27h,'s only one right flag.',0 ; DATA XREF: sub_8049CF0+30fo
aS              db 'Please input flag:',0 ; DATA XREF: sub_8049CF0+63fo
aCongratulation db '%s',0 ; DATA XREF: sub_8049CF0+7Cfo
aCongratulation db 'Congratulations ',0 ; DATA XREF: sub_8049CF0+2A7fo
xrefs to aPleaseInputFla
Typ Address Text
o sub_8049CF0+63 lea eax,(aPleaseInputFla - 80F3000h)[ebx]; "Please input flag."
OK Cancel Search Help
1
https://blog.csdn.net/FunkyPants
```

在定位到主函数后，可使用hex-rays插件进行反汇编（快捷键F5），查看程序大致流程。根据程序的执行流程、函数的参数可以大致猜测出部分函数的功能，使用快捷键n可以对其进行重命名。

可将程序大致分为三个部分：

4.1.1 part1

在接收输入字符串后，判断程序运行环境，之后选择对字符串从前向后或从后向前进行异或操作。

```
print(edi0, (int)"There's only one right flag.");
sub_804F9D0(14, (int)sub_804A12D);
sub_806D510(5);
print(edi0, (int)"Please input flag:");
scanf("%s", (unsigned int)input_str);
sub_806D550();
v3 = sub_806D570(v2, 135213056);
if ( v3 == sub_806D560() ) // program running environment
{
    if ( len((int)input_str) != 50 )
        exit(1);
    for ( i = 49; i >= 0; --i )
    {
        BYTE3(v14) = input_str[i];
        input_str[i] = byte_80F4DA0[i % 6] ^ BYTE3(v14);
    }
}
else
{
    for ( j = 0; len((int)input_str) > (unsigned int)j; ++j )
    {
        BYTE3(v14) = input_str[j];
        input_str[j] = byte_80F4DA0[j % 6] ^ BYTE3(v14);
    }
}

```

<https://blog.csdn.net/FunkyPants>

4.1.2 part2

开辟一段长度为100个字符的内存空间，对上一步异或后的字符串，按照一定规律将其放入新的内存空间。

```
,
memset(new_array, 0, sizeof(new_array));
v18 = 0;
v19 = 0;
while ( v19 <= 49 )
{
    sub_8049B55((int)&new_array[v18], (int)&input_str[v19]);
    v19 += 5;
    v18 += 8;
}

```

4.1.3 part3

对于程序中给定的一个长字符串，进行复杂的操作后，调用一个函数将该字符串与上一步得到的字符串进行比较，根据结果输出wrong或者congratulations。

```

00049F30 lea    eax, [ebp+var_D1]
00049F36 lea    edx, (a58_724gC3A5c3 - 80F3000h)[ebx] ; "-;/5,+:+0..724g/C,=3++A5C3>=/,;5?@++95"...
00049F3C mov    ecx, 51h
00049F41 mov    esi, [edx] ; get 4 char. -;/;
00049F43 mov    [eax], esi
00049F45 mov    esi, [edx+ecx-4] ; get last 4 char. ?CC
00049F49 mov    [eax+ecx-4], esi ; store last 4 char
00049F4D lea    edi, [eax+4]
00049F50 and    edi, 0FFFFFFCh
00049F53 sub    eax, edi
00049F55 sub    edx, eax ; delete first char
00049F57 add    ecx, eax
00049F59 and    ecx, 0FFFFFFCh
00049F5C mov    eax, ecx
00049F5E shr    eax, 2
00049F61 mov    esi, edx
00049F63 mov    ecx, eax
00049F65 rep  movsd
00049F67 sub    esp, 0Ch
00049F6A lea    eax, [ebp+var_D1]
00049F70 push   eax
00049F71 call  len
00049F76 add    esp, 10h
00049F79 sub    esp, 4
00049F7C push   eax
00049F7D lea    eax, [ebp+var_D1]
00049F83 push   eax
00049F84 lea    eax, [ebp+new_array]
00049F87 push   eax
00049F88 call  sub_8049058
00049F8D add    esp, 10h
00049F90 test   eax, eax
00049F92 jnz   short loc_8049FA8

00049F94 sub    esp, 0Ch
00049F97 lea    eax, (aCongratulation - 80F3000h)[ebx] ; "Congratulations."
00049F9D push   eax
00049F9E call  print
00049FA3 add    esp, 10h
00049FA6 jmp   short loc_8049FBA

00049FA8 loc_8049FA8:
00049FA8 sub    esp, 0Ch
00049FAB lea    eax, (aWrong_ - 80F3000h)[ebx] ; "Wrong."
00049FB1 push   eax
00049FB2 call  print
00049FB7 add    esp, 10h

```

4.2 具体分析

4.2.1 part1

根据程序执行流程，输入字符串长度应为50。输入字符串后，程序会调用一个函数对当前运行环境进行判断（此处有坑），决定程序向左或向右执行，这里可以先略过对环境判断函数的分析，选择使用判断失败的路径进行后续分析。

对于这条路径，程序对输入的字符串按照从前向后的顺序，使用字符串byte_80f4da0逐个字符进行异或操作，通过动态调试可以得到该字符为"zjgcjy"（暂时认为此值正确）。

```

else
{
    for ( j = 0; len((int)input_str) > (unsigned int)j; ++j )
    {
        BYTE3(v14) = input_str[j];
        input_str[j] = byte_80f4da0[j % 6] ^ BYTE3(v14);
    }
}

```

4.2.2 part2

该部分将异或后的字符串按照一定规则存入新开辟的地址空间，观察该函数逻辑。

```
while ( v19 <= 49 )
{
    sub_8049B55((int)&new_array[v18], (int)&input_str[v19]);
    v19 += 5;
    v18 += 8;
}

int __cdecl sub_8049B55(int output, int input)
{
    int i; // ST08_4@1
    int v3; // ST1C_4@1
    int result; // eax@1

    sub_804A15C();
    i = input;
    v3 = *MK_FP(__GS__, 20);
    *(_BYTE *)output = (*(_BYTE *)input >> 3) + unk_80F4E06; // 'w'
    *(_BYTE *)output + 1 = unk_80F4E06 + (4 * *(_BYTE *)i & 0x1C | (*(_BYTE *)i + 1) >> 6));
    *(_BYTE *)output + 2 = unk_80F4E06 + ((*(_BYTE *)input + 1) >> 1) & 0x1F);
    *(_BYTE *)output + 3 = unk_80F4E06 + (16 * *(_BYTE *)i + 1) & 0x10 | (*(_BYTE *)i + 2) >> 4));
    *(_BYTE *)output + 4 = unk_80F4E06 + (2 * *(_BYTE *)i + 2) & 0x1E | (*(_BYTE *)i + 3) >> 7));
    *(_BYTE *)output + 5 = unk_80F4E06 + ((*(_BYTE *)input + 3) >> 2) & 0x1F);
    *(_BYTE *)output + 6 = unk_80F4E06 + (8 * *(_BYTE *)i + 3) & 0x18 | (*(_BYTE *)i + 4) >> 5));
    *(_BYTE *)output + 7 = unk_80F4E06 + (*(_BYTE *)input + 4) & 0x1F);
    result = 0;
    if ( *MK_FP(__GS__, 20) != v3 )
        sub_806FE60();
    return result;
}
https://blog.csdn.net/FunkyPants
```

该函数接收原字符串input与新字符串output的地址，每次使用input中5个字符，通过操作后得到8个新字符并存入output中。完成一次操作后，input加5，output+8，直到对input中的所有字符完成操作。

通过分析函数的逻辑，可以得到该函数的功能是将input的5个字符按照每个字符对应8位二进制数据的形式，转化为40位二进制字符。将40位二进制字符按照每组5位取出8组，对每一组二进制数据加上一个unk字符，得到最后的8个output字符。

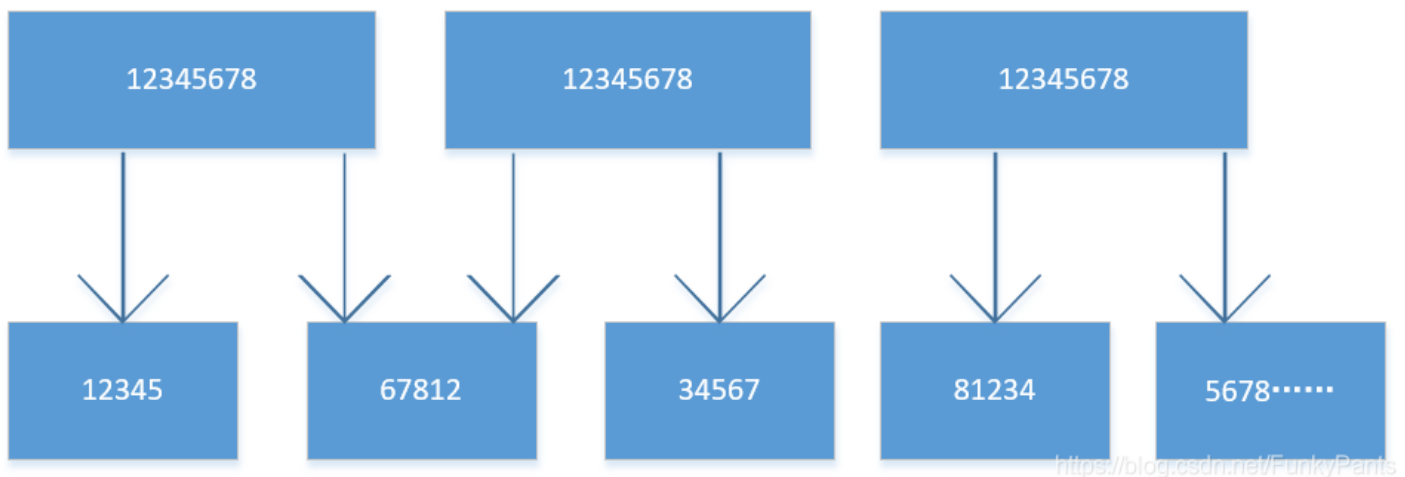
运算顺序：乘法运算高于逻辑运算，关于运算符的执行顺序也可以通过分析汇编代码得到。

output = unk + input,为了简化表达，以下分析过程省略unk,input省略为i

```
output0 = i0 >> 3
1234 5678 -> 0001 2345
相当于output0=i0(12345)

output1 = 4*i & 0x1c | i1 >> 6
i0(3456 7800) & 00011100 -> i(678)
i1(1200 0000) -> i1(12)
000678000 | 0000 0012 -> 0006 7812
相当于output1=i0(678)+i1(12)

output2 = i1 >> 1 & 0x1f
i1(1234 5678) -> i1(0123 4567)
i1(0123 4567) & 00011111 -> i1(34567)
```



4.2.2 part3

在IDA6.8中，反汇编插件得出的结果可以直观的看出是错误的，小伙伴的7.0似乎得到了正确的结果。我们可以通过分析汇编代码观察进行了什么操作。可以看到程序在一顿移位操作后使用了 `rep movsd` 命令进行字符串的复制，查找其具体用法：

```
rep指令的目的是重复其上面的指令。ECX的值是重复的次数。
movsd指令是移动一个双字（dword），将DS:ESI中的值拷贝到ES:EDI指向的地址。
```

但是在mov指令完成后，后面的操作中均未使用ES段的数据。在该汇编块的末尾，程序调用了一个函数，其参数是该段中定义的一个长字符串与part2中得到的字符串，最后根据函数返回结果，如果反汇结果为0则输出“wrong”，可以猜测该函数的功能是字符串比较。`rep movsd` 指令之前的操作不会影响最后的结果，在分析时可略过。

4.3 flag

在本题中，flag由我们输入，需要从根据程序的流程逆推得到。

在part2中计算的结果，应该与part3中的定义的长字符串相等。在前面已经分析过part2完成的功能，现在考虑如果由其得到的结果逆推出part2中输入的字符串。

直观上来看，part2中将50位ASCII字符按照每个字符对应8位二进制码的形式，转化为400位二进制字符串，按5位一组取出80组，加上一个unknown字符后的得到结果，也就是part3中的长字符串。现在已知part3长字符串，将其每一字符减去unknown字符，得到的应该是一个小于等于5位的二进制码，补全至5位，再将所有二进制码拼接为400位，按8位一组取出50组即可还原出part2中的输入字符串。

这里需要考虑的一个问题是unkown字符的值，在最初通过动态调试得到的unknown是“w”，用这个值不能计算得到结果，此时考虑是否另有其值。经过提醒，在part1中程序会调用一个函数对当前运行环境进行判断，从而决定unknown的值，该值属于ASCII字符，我们可以通过爆破的方法去猜测。另外，该函数也决定了part1中byte_80f4da0的值，对于byte_80f4da0查找交叉引用可以看到正确值是“redctf”。

0004A026	mov	eax, offset byte_80F4DA9	0004A076	loc_804A076:	0004A076	mov	eax, offset byte_80F4DA9	
0004A02C	mov	byte ptr [eax], 'z'	0004A076	mov	byte ptr [eax], 'r'	0004A07C	mov	byte ptr [eax], 'r'
0004A02F	mov	eax, offset byte_80F4DA9	0004A07F	mov	eax, offset byte_80F4DA9	0004A085	mov	byte ptr [eax+1], 'e'
0004A035	mov	byte ptr [eax+1], 'j'	0004A085	mov	byte ptr [eax+1], 'e'	0004A089	mov	eax, offset byte_80F4DA9
0004A039	mov	eax, offset byte_80F4DA9	0004A089	mov	byte ptr [eax+2], 'd'	0004A08F	mov	byte ptr [eax+2], 'd'
0004A03F	mov	byte ptr [eax+2], 'g'	0004A093	mov	eax, offset byte_80F4DA9	0004A093	mov	eax, offset byte_80F4DA9
0004A043	mov	eax, offset byte_80F4DA9	0004A099	mov	byte ptr [eax+3], 'c'	0004A099	mov	byte ptr [eax+3], 'c'
0004A049	mov	byte ptr [eax+3], 'c'	0004A09D	mov	eax, offset byte_80F4DA9	0004A09D	mov	byte ptr [eax+4], 't'
0004A04D	mov	eax, offset byte_80F4DA9	0004A0A3	mov	byte ptr [eax+4], 't'	0004A0A3	mov	byte ptr [eax+4], 't'
0004A053	mov	byte ptr [eax+4], 'j'	0004A0A7	mov	eax, offset byte_80F4DA9	0004A0A7	mov	eax, offset byte_80F4DA9
0004A057	mov	eax, offset byte_80F4DA9	0004A0AD	mov	byte ptr [eax+5], 'f'	0004A0AD	mov	byte ptr [eax+5], 'f'
0004A05D	mov	byte ptr [eax+5], 'y'	0004A0B1	mov	eax, offset byte_80F4DA9	0004A0B1	mov	eax, offset byte_80F4DA9
0004A061	mov	eax, offset byte_80F4DA9	0004A0B7	mov	byte ptr [eax+6], 0	0004A0B7	mov	byte ptr [eax+6], 0
0004A067	mov	byte ptr [eax+6], 0	0004A0BB	call	sub_8050E10	0004A0BB	call	sub_8050E10
0004A06B	mov	eax, offset unk_80F4E06	0004A0C0	shl	eax, 6	0004A0C0	shl	eax, 6
0004A071	mov	byte ptr [eax], 'w'	0004A0C3	mov	ecx, eax	0004A0C3	mov	ecx, eax
0004A074	jmp	short loc_804A0E7	0004A0C5	mov	edx, 40000001h	0004A0C5	mov	edx, 40000001h
			0004A0CA	mov	eax, ecx	0004A0CA	mov	eax, ecx
			0004A0CC	imul	edx	0004A0CC	imul	edx
			0004A0CE	sar	edx, 1Dh	0004A0CE	sar	edx, 1Dh
			0004A0D1	mov	eax, ecx	0004A0D1	mov	eax, ecx
			0004A0D3	sar	eax, 1Fh	0004A0D3	sar	eax, 1Fh
			0004A0D6	sub	edx, eax	0004A0D6	sub	edx, eax
			0004A0D8	mov	eax, edx	0004A0D8	mov	eax, edx
			0004A0DA	add	eax, 20h	0004A0DA	add	eax, 20h
			0004A0DD	mov	edx, eax	0004A0DD	mov	edx, eax
			0004A0DF	mov	eax, offset unk_80F4E06	0004A0DF	mov	eax, offset unk_80F4E06
			0004A0E5	mov	[eax], dl	0004A0E5	mov	[eax], dl

在计算得到part2中的字符后，根据异或运算的性质，进行与part1中相同的异或操作，即可还原得到flag。

```
flag{e4Sy_ReVer5e_bU1It_0n_h00K_Antldbg_anD_UpX?!}
```

代码如下：

```

compare = "-;/;5,+:+8..724G/C,=3++A5C3>=/,;5?@++95-,7H;A9EA2/H-5.3+,36+9+DG-?/.79<<-7A0=?CC"

total = ''
for w in range(0x20, 0x7e + 1): # w的范围属于ASCII码, w=43
    for s in compare:
        if ord(s) - w < 0: # 原字符的后5位不会小于0
            break
    # 用part3中的字符减去unknown, 转化为二进制, 替换前两位"0b"
    s5 = bin(ord(s) - w).replace('0b', '')

    if len(s5) != 5: # 补全至5位
        s5 = "0" * (5 - len(s5)) + s5
    total += s5 # 拼接得到400位二进制码

# 取出8位字符, 转化为ASCII字符
# ps: 好久没用忘了切片就自己写了一个
count = 0
s = ''
b = ''
for c in total:
    s += c
    count += 1
    if count == 8:
        b += chr((int('0b' + s, 2)))
        s = ''
        count = 0
total = ''

# part1
flag = ''
tag = 'redctf'
# tag = 'zjgcjy'
for i in range(0, 50):
    flag += chr(ord(b[i]) ^ ord(tag[i % 6]))
try:
    print(flag)
except Exception as e:
    print(e)
    continue

```

5.总结收获

1. 逆向分析时, 应该主要关注代码段(函数)完成的功能, 不要过多局限于单条指令功能的分析。
2. 在ctf中, 很多时候会遇到出题人故意留的坑, 比如这题中part1中用于异或的字符串根据环境的不同而不同, part3中进行复杂操作后却不影响结果, 遇到这种情况应拓宽思路。