

# 32C3之PWN题目Readme的解法

原创

HappyOrange2014 于 2016-01-04 23:22:14 发布 5453 收藏 2

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

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/HappyOrange2014/article/details/50459201>

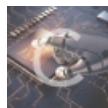
版权



CTF 同时被 2 个专栏收录

3 篇文章 0 订阅

订阅专栏



逆向工程

3 篇文章 0 订阅

订阅专栏

题目描述:

```
Can you read the flag?  
nc 136.243.194.62 1024
```

## 题目下载

本题是2015年32C3CTF比赛中一道300分的pwn题目, 也是我第一次在赛后看着writeup做出的pwn题目, 希望下次能够在赛中做出pwn题目。话说, 忙碌的工作和生活之余, 学习一点安全分析的知识, 也是很快乐的一件事情!

我这里所用的工具包括: IDAx64、kali2.0下的edb。

注: 整个解题过程参考了多篇writeup, 并非完全原创, 但每一步都经过实际测试, 特此说明。如有错误, 请留言指出, 不胜感激。

观察文件的基本情况, 说明是linux下的x64程序。远程连接后可知(本地运行也可), 可以输入两个字符串, 分别是用户名(设为user)和红框处的内容(设为str)。

```
root@kali:~/Desktop# file readme.bin  
readme.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24, BuildID[sha1]=7d3dcaal7ebe1662eec1900f735765bd990742f9, stripped
```

```
root@kali:~/Desktop# nc 136.243.194.62 1024  
Hello!  
What's your name? 123456  
Nice to meet you, 123456.  
Please overwrite the flag: 123456  
Thank you, bye!
```



```

00000000:00400844 83 f8 ff          cmp eax, 0xff
00000000:00400847 74 56            jz 0x000000000040089f
00000000:00400849 83 f8 0a        cmp eax, 10
00000000:0040084c 74 12            jz 0x0000000000400860
00000000:0040084e 88 83 20 0d 60 00  mov byte ptr [rbx+0x00600d20], al
00000000:00400854 48 83 c3 01      add rbx, 1
00000000:00400858 48 83 fb 20      cmp rbx, 32
00000000:0040085c 75 da            jnz 0x0000000000400838
00000000:0040085e eb 18            jmp 0x0000000000400878
00000000:00400860 ba 20 00 00 00  mov edx, 32
00000000:00400865 48 63 fb        movsxd rdi, rbx
00000000:00400868 31 f6            xor esi, esi
00000000:0040086a 29 da            sub edx, ebx
00000000:0040086c 48 81 c7 20 0d 60 00  add rdi, 0x00600d20
00000000:00400873 e8 f8 fd ff ff   call 0x0000000000400670
00000000:00400878 bf 4e 09 40 00  mov edi, 0x0040094e
00000000:0040087d e8 be fd ff ff   call 0x0000000000400640
00000000:00400882 48 8b 84 24 08 01 00 00  mov rax, qword ptr [rsp-264]
00000000:0040088a 64 48 33 04 25 28 00 00 00  xor rax, qword ptr fs:[0x28]
00000000:00400893 75 14            jnz 0x00000000004008a9
00000000:00400895 48 81 c4 18 01 00 00  add rsn, 0x0118

```

al = 0000000000000061  
byte ptr [rbx+0x00600d20] = [0000000000600d20] = 0x33

Data Dump

```

000000000000600000-000000000000601000
00000000:00600d00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00600d10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00600d20 33 32 43 33 5f 54 68 65 53 65 72 76 65 72 48 61 32C3 TheServerHa
00000000:00600d30 73 54 68 65 46 6c 61 67 48 65 72 65 2e 2e 2e 00 sTheFlagHere....
00000000:00600d40 a0 f2 07 bf 3e 7f 00 00 e0 f4 07 bf 3e 7f 00 00 b.¿>...a0.¿>...
00000000:00600d50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

```

00000000:0040083f e8 5c fe ff ff   call 0x00000000004006a0
00000000:00400844 83 f8 ff          cmp eax, 0xff
00000000:00400847 74 56            jz 0x000000000040089f
00000000:00400849 83 f8 0a        cmp eax, 10
00000000:0040084c 74 12            jz 0x0000000000400860
00000000:0040084e 88 83 20 0d 60 00  mov byte ptr [rbx+0x00600d20], al
00000000:00400854 48 83 c3 01      add rbx, 1
00000000:00400858 48 83 fb 20      cmp rbx, 32
00000000:0040085c 75 da            jnz 0x0000000000400838
00000000:0040085e eb 18            jmp 0x0000000000400878
00000000:00400860 ba 20 00 00 00  mov edx, 32
00000000:00400865 48 63 fb        movsxd rdi, rbx
00000000:00400868 31 f6            xor esi, esi
00000000:0040086a 29 da            sub edx, ebx
00000000:0040086c 48 81 c7 20 0d 60 00  add rdi, 0x00600d20
00000000:00400873 e8 f8 fd ff ff   call 0x0000000000400670
00000000:00400878 bf 4e 09 40 00  mov edi, 0x0040094e
00000000:0040087d e8 be fd ff ff   call 0x0000000000400640
00000000:00400882 48 8b 84 24 08 01 00 00  mov rax, qword ptr [rsp+264]
00000000:0040088a 64 48 33 04 25 28 00 00 00  xor rax, qword ptr fs:[0x28]

```

edx = 000000000000000a  
possible jump from 0x000000000040084c

Data Dump

```

000000000000600000-000000000000601000
00000000:00600d00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00600d10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
00000000:00600d20 61 62 63 64 65 66 67 68 53 65 72 76 65 72 48 61 abcdefghServerHa
00000000:00600d30 73 54 68 65 46 6c 61 67 48 65 72 65 2e 2e 2e 00 sTheFlagHere....
00000000:00600d40 a0 f2 07 bf 3e 7f 00 00 e0 f4 07 bf 3e 7f 00 00 b.¿>...a0.¿>...
00000000:00600d50 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....

```

如图所示，由于我输入的字符串str为abcdefgh，故原flag的前8位被覆盖成了abcdefgh。  
需要说明的是，该flag只是题目给出的二进制程序中保存的flag，远程服务器运行的虽然是功能相同的二进制程序，但flag肯定不是现在这个，需要通过远程溢出来获取。

**Step3:** 将0x600D20处的flag未被覆盖的部分覆盖为0x00。

下面我们依次回答上面提到的3个问题。

### 1、如何造成栈溢出？



显然，溢出点在用户名字符串user处，后面的str字符串只接收前32位，无法溢出。

假设我输入的user是12345678，则整个0x4007E0函数的栈空间如下图所示：

```
Stack
00007ffc:8dd756c0 3837363534333231 12345678
00007ffc:8dd756c8 00007f3ebf2a3400 .4*¿>...
00007ffc:8dd756d0 00007ffc8dd75820 Xx.ü...
00007ffc:8dd756d8 00007f3ebf2a6500 .e*¿>...
00007ffc:8dd756e0 00007ffc8dd75848 HXx.ü...
00007ffc:8dd756e8 00007f3ebf2a61a8 `a*¿>...
00007ffc:8dd756f0 0000000000000001 .....
00007ffc:8dd756f8 00007f3ebf08d77d }x.¿>... return to 00007f3ebf08d77d
00007ffc:8dd75700 0000000000000000 .....
00007ffc:8dd75708 00007f3ebf2a3a10 .:*¿>...
00007ffc:8dd75710 0000000000000001 .....
00007ffc:8dd75718 0000000000000000 .....
00007ffc:8dd75720 00007ffc00000001 ..... ü...
00007ffc:8dd75728 0000000000600c00 A. `...
00007ffc:8dd75730 0000000000000000 .....
00007ffc:8dd75738 00007ffc8dd758d0 ÐXx.ü...
00007ffc:8dd75740 0000000000000000 .....
00007ffc:8dd75748 0000000000000000 .....
00007ffc:8dd75750 0000000000000000 .....
00007ffc:8dd75758 00007f3ebf091e07 .. ¿>... return to 00007f3ebf091e07
00007ffc:8dd75760 00007ffc00000001 ..... ü...
00007ffc:8dd75768 0000000000000000 .....
00007ffc:8dd75770 00000000f63d4e2e .N=ö...
00007ffc:8dd75778 00007f3ebecb028 (*¿>...
00007ffc:8dd75780 0000000000000000 .....
00007ffc:8dd75788 00007f3ebf07f2a0 ò.¿>...
00007ffc:8dd75790 0000000000000000 .....
00007ffc:8dd75798 00007f3ebed51415 .. Ö¿>... return to 00007f3ebed51415
00007ffc:8dd757a0 00007f3ebf07f2a0 ò.¿>...
00007ffc:8dd757a8 0000000000000000 .....
00007ffc:8dd757b0 0000000000000000 .....
00007ffc:8dd757b8 00007f3ebed4ea79 yèÖ¿>... return to 00007f3ebed4ea79
00007ffc:8dd757c0 00007f3ebf07f2a0 ò.¿>...
00007ffc:8dd757c8 f7cb43f951a98e00 .. @QùCÈ=
00007ffc:8dd757d0 0000000000000000 .....
00007ffc:8dd757d8 0000000000000000 .....
00007ffc:8dd757e0 0000000000000000 .....
00007ffc:8dd757e8 0000000004006e7 c.@..... return to 0000000004006e7
```

红框中即为0x4007E0的栈空间，0x4006E7是其返回父函数的返回地址。由图可知，要想刚好覆盖父函数返回地址，用户名长度必须为 (0x7f0-0x6c0-1)，即303，减去的1为字符串结束符。

那么，我们是不是就是要覆盖到函数返回地址呢？一般的栈溢出确实是溢出到返回地址后使程序流程转向shellcode，但通过阅读writeup可知，这里的目标是继续往下溢出，直到溢出到argv[0]指针，即将argv[0]指针覆盖成flag的地址，使得程序在报错时，将flag泄漏出来。那么，argv[0]指针在哪里呢？我们在edb中继续下拉栈窗口的滚动条：

```
Stack
00007fff:32839508 00000000004008b0 *.@..... return to 00000000004008b0
00007fff:32839510 00007fff32839568 h..2y...
00007fff:32839518 0000000000000001 .....
00007fff:32839520 0000000000000000 .....
00007fff:32839528 0000000000000000 .....
00007fff:32839530 0000000004006ee i.@..... <entry point>
00007fff:32839538 00007fff32839560 ..2y...
00007fff:32839540 0000000000000000 .....
00007fff:32839548 000000000400717 *.@..... return to 000000000400717
00007fff:32839550 00007fff32839558 X..2y...
00007fff:32839558 00000000000001c .....
00007fff:32839568 0000000000000001 .....
00007fff:32839568 00007fff3283abe7 c<..2y... ASCII "/root/Desktop/readme.bin"
00007fff:32839570 0000000000000000 .....
00007fff:32839578 00007fff3283ac00 ..2y... ASCII "USER=root"
00007fff:32839580 00007fff3283ac0a ..2y... ASCII "XDG_SEAT=seat0"
00007fff:32839588 00007fff3283ac19 ..2y... ASCII "HOME=/root"
00007fff:32839590 00007fff3283ac24 $..2y... ASCII "DESKTOP_SESSION=default"
00007fff:32839598 00007fff3283ac3c <..2y... ASCII "LOGNAME=root"
00007fff:328395a0 00007fff3283ac49 I..2y... ASCII "USERNAME=root"
00007fff:328395a8 00007fff3283ac57 W..2y... ASCII "XDG_SESSION_ID=2"
00007fff:328395b0 00007fff3283ac68 h..2y... ASCII "WINDOWPATH=7"
00007fff:328395b8 00007fff3283ac75 u..2y... ASCII "PATH=/usr/local/sbin:/usr/local/
00007fff:328395c0 00007fff3283ac7 ..2y... ASCII "GDM_LANG=en_US.UTF-8"
00007fff:328395c8 00007fff3283accc I..2y... ASCII "XDG_RUNTIME_DIR=/run/user/0"
00007fff:328395d0 00007fff3283ace8 è..2y... ASCII "DISPLAY=0"
00007fff:328395d8 00007fff3283acf3 ó..2y... ASCII "LANG=en_US.UTF-8"
00007fff:328395e0 00007fff3283ad04 ..2y... ASCII "XDG_SESSION_DESKTOP=default"
00007fff:328395e8 00007fff3283ad20 ..2y... ASCII "XAUTHORITY=/var/run/gdm3/auth-fc
```

如上图所示的红框中就是argv[0]指针，地址是0x7fff32839568。由于每次运行程序的绝对地址是不一样的（但相对地址一样），我在这次运行时，user输入的是1111...字符串（而不是上文的12345678），此时的位置如下图所示：

Stack			
00007fff:32839328	00007f7eaad6021e	..0~...	return to 00007f7eaad6021e
00007fff:32839330	00007f7eab08e4e0	ää.←~...	
00007fff:32839338	00007f7eaad5ba8e	.0~...	return to 00007f7eaad5ba8e
00007fff:32839340	0000000000000000	.....	
00007fff:32839348	0000000000400844	D.@.....	return to 0000000000400844
00007fff:32839350	3131313131313131	11111111	
00007fff:32839358	3131313131313131	11111111	
00007fff:32839360	3131313131313131	11111111	
00007fff:32839368	3131313131313131	11111111	
00007fff:32839370	3131313131313131	11111111	
00007fff:32839378	3131313131313131	11111111	
00007fff:32839380	0000000031313131	1111....	
00007fff:32839388	00007f7eab09c77d	}Ç ←~...	return to 00007f7eab09c77d
00007fff:32839390	0000000000000000	.....	
00007fff:32839398	00007f7eab2b2a10	*+←~...	
00007fff:328393a0	0000000000000001	.....	
00007fff:328393a8	0000000000000000	.....	
00007fff:328393b0	00007fff00000001	...ÿ...	
00007fff:328393b8	0000000000600cc0	Ä.~.....	
00007fff:328393c0	0000000000000000	.....	
00007fff:328393c8	00007fff32839560	...2ÿ...	
00007fff:328393d0	0000000000000000	.....	
00007fff:328393d8	0000000000000000	.....	
00007fff:328393e0	0000000000000000	.....	
00007fff:328393e8	00007f7eab0a0e07	...←~...	return to 00007f7eab0a0e07
00007fff:328393f0	00007fff00000001	...ÿ...	
00007fff:328393f8	0000000000000000	.....	
00007fff:32839400	00000000f63d4e2e	.N=0....	
00007fff:32839408	00007f7eacfa028	( I~...	
00007fff:32839410	0000000000000000	.....	

显然，user的地址是0x7fff32839350，则

$0x7fff32839568 - 0x7fff32839350 = 0x218$ ，也就是说，我们在user字符串中输入0x218个字节后，正好到达argv[0]指针地址。第1个问题解决了。

## 2、怎样通过栈溢出将原本的argv[0]指针由文件名变成flag？

通过前面的代码分析可知，在我们输入第二个字符串str时，flag就会被覆盖，即使将argv[0]指向0x600D20，输出的错误信息中也不会包含flag。

这里，网上的writeup指出，除了0x600D20处存放的flag外，在0x400D20处也存放着同样的flag，所以我们可以将argv[0]指向0x400D20。网上的writeup是用gdb查找的flag，但我这个新手对gdb还是不熟，那就用edb的binary string插件吧，结果如下：





